

## Week 6: Embedded SQL

*Impedance Mismatch*  
*Statement-Level vs Call-Level*  
*Interfaces*  
*Static SQL*  
*Transactions and Cursors*  
*Dynamic SQL*  
*JDBC*



## Embedded SQL

- Traditional applications often need to  SQL statements inside the instructions of a procedural programming language (C, COBOL, etc.)
- There is a severe mismatch between the computational model of a programming language (PL) and that of a DBMS:
  - ✓ A PL is Turing-complete, SQL is not;
  - ✓ The variables of a PL take as values single records, those of SQL whole tables;
  - ✓ PL computations are generally on a single data structure, SQL ones on bulk data structures.

## Interactive vs. Non-Interactive SQL

- **Non-interactive SQL**: Statements are included in an application program written in a host language — such as C, Java, COBOL
- **Interactive SQL**: Statements input from terminal; DBMS outputs to screen
- Interactive SQL is inadequate for most uses:
  - ✓ It may be necessary to process the data before output;
  - ✓ Amount of data returned not known in advance;

## Basic Elements of Embedded SQL

- Programs with embedded SQL use a  to manage SQL statements. Embedded statements are preceded by
- Program variables may be used as parameters in the SQL statements (preceded by ':')
- **select** statements producing a single row and update statements can be embedded easily; but not **select** statements producing sets of rows.
- The SQL environment offers a predefined variable **sqlstate** which describes the execution status of a statement (= "00000" if executed successfully).

## Application Program

- **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.
- **Using the facilities of both:** the application program can act as an intermediary between the user at a terminal and the DBMS.

## Introducing SQL to an Application

SQL statements can be incorporated into an application program in two different ways.

- [ ] Application program is a mixture of host language statements and SQL statements and directives.
- [ ] 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.

## Preparation

- Before any SQL statement is executed, it must be [ ] 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 [ ].
- Preparation is a complex activity, usually done at run time, justified by the complexity of query processing.

## Statement Level Interface

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

## Statement Level Interface

→ SQL constructs in an application take two forms:

✓  Useful when SQL portion of program is known at **compile time**.

✓  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.

## Static SQL

```
EXEC SQL BEGIN DECLARE S;  
  unsigned long num_enrolled;  
  char crs_code;  
  char SQLSTATE [5];  
EXEC SQL END DECLARE SE;  
  
.....  
EXEC SQL SELECT C.NumEnrolled  
  INTO :num_enrolled  
  FROM Course C  
  WHERE C.CrsCode = :crs_code;
```

→ Declaration section for host/SQL communication.

→ Colon convention for value (**WHERE**) and result (**INTO**) parameters.

## Call Level Interface

→ Application program written entirely in host language (no precompiler)

Examples: JDBC, ODBC

→ SQL statements are values of string variables constructed  using host language — similar to dynamic SQL

→ Application uses string variables as arguments of library routines that communicate with DBMS

e.g.

## 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
```

## Example: Course Deregistration

```
EXEC SQL [ ] 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 [ ]  
else {  
  EXEC SQL UPDATE Course C  
    SET C.Numenrolled = C.Numenrolled - 1  
    WHERE C.CrsCode = :crscode;  
  if ( ! strcmp (SQLSTATE, "00000") ) EXEC SQL [ ]  
  else EXEC SQL [ ]  
}
```

## 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 [ ] or [ ] statements.

## Impedance Mismatch Problem

- Fundamental problem with database technology: [ ] — traditional programming languages process records one-at-a-time (tuple-oriented); SQL processes tuple sets (set-oriented).
- [ ] solve this problem: A cursor returns tuples from a result set, to be process one-by-one.
- Syntax of cursor definition:

```
declare CursorName [ scroll ]  
cursor for SelectSQL  
  [ for < read only | update [ of Attribute {,  
  Attribute } ] ] >
```

## Operations on Cursors

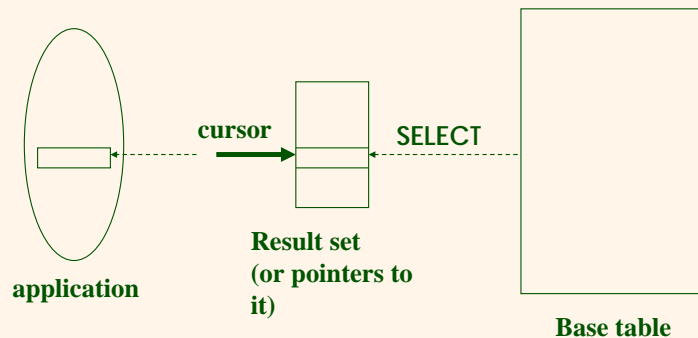
- **Result set** – rows returned by a SELECT statement
- To execute the query associated with a cursor:
- To extract one tuple from the query result:
- To free the cursor, discarding the query result:
- To access the current tuple (when a cursor reads a relation, in order to update it):

## Example of Cursor Use

```
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

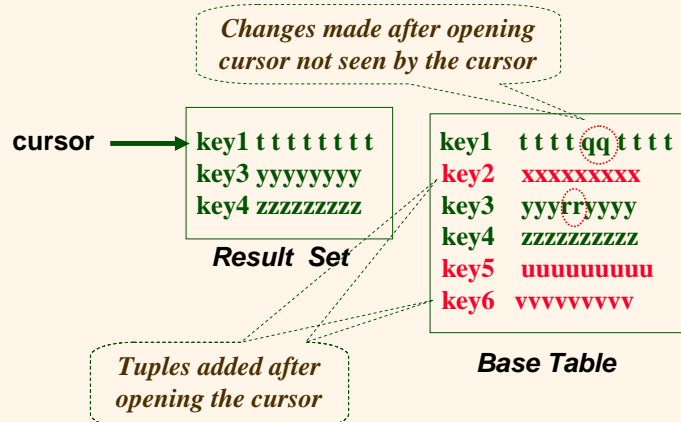
## How Cursors Work



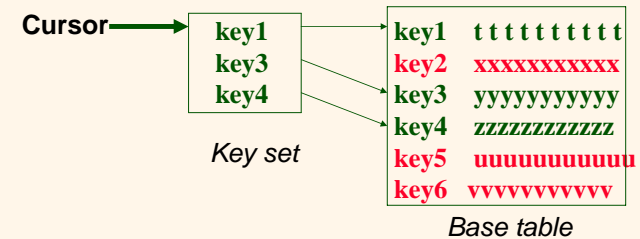
## Cursor Types

- : Result set (effectively) 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
- : 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



## Keyset-Driven Cursor



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

## Keyset-Driven Cursor

- Example of a cursor that is not insensitive.
- 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.

## Syntax for 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 views

## 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:

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

*Get previous tuple*

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

## Procedure in Oracle PL/SQL

```
Procedure Debit(ClientAcct char(5),Withdr int) is
  OldAmount integer; NewAmount integer;
  Threshold integer;
begin
  select Amount,Overdraft into OldAmount, Threshold
  from BankAcct where AcctNo = ClientAcct
  for update of Amount;
  NewAmount := OldAmount - WithDr;
  if NewAmount > Threshold
  then update BankAcct
    set Amount = NewAmount
    where AcctNo = ClientAcct;
  else insert into OverDraftExceeded
    values(ClientAcct,Withdr,sysdate);
  end if;
end Debit;
```

## Stored Procedures

- **(Stored) Procedure** – written in a PL, included as schema element (stored in DBMS), invoked by the application.
- For example,

```
procedure AssignCity
  (:Dep char(20),:City char(20))
update Department
set City = :City
where Name = :Dep
```
- SQL-2 does not support the definition of complex procedures
- Most systems offer SQL extensions that support complex procedures (e.g., Oracle PL/SQL).

## 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 – all database access is encapsulated within the procedure

## Dynamic SQL

- When applications do not know at compile-time the statement to execute, they need **dynamic SQL**.
- Major problem: managing the transfer of parameters between program and SQL environment.
- For direct execution:  
`execute immediate SQLStatement`
- For execution preceded by the analysis of the statement:  
`prepare CommandName from SQLStatement`  
followed by:  
`execute CommandName [ into TargetList ]`  
`[ using ParameterList ]`

## Parameters for Static SQL

### For Static SQL:

- Names of (host language) parameters are contained in SQL statement and **available to pre-compiler**.
- 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;`

## 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;
```

*placeholder*

- `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`

## Parameters for Dynamic SQL

- **Dynamic SQL**: SQL statement constructed at run time when symbol table is no longer present.
- Two cases to deal with:
  - ✓ Case I: Parameters **are** known at compile time;
  - ✓ Case II: Parameters not known at compile time.



## Case I

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

```
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;
```

→ EXECUTE statement is compiled using symbol table; *fetch()* and *store()* routines generated.

## Case II

→ 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

## Dealing with Case I

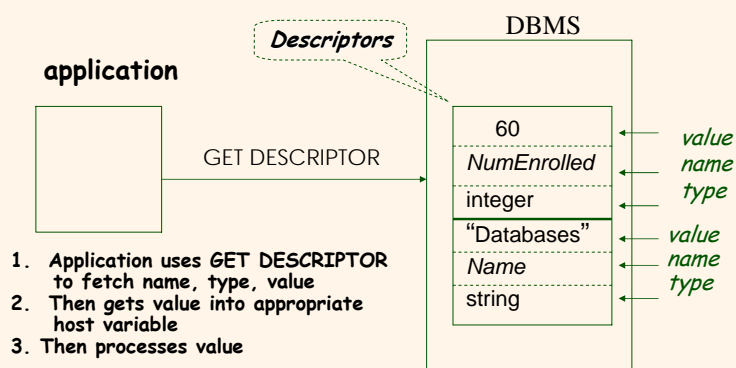
- ✓ Fetch and store routines are executed at client when EXECUTE is executed to communicate argument values with DBMS
- ✓ EXECUTE can be invoked multiple times with **different values** of *in* parameters
  - Each invocation uses same query execution plan
- ✓ Values substituted for placeholders by DBMS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

## Dealing with Case II

- 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' "
```



1. Application uses **GET DESCRIPTOR** to fetch name, type, value
2. Then gets value into appropriate host variable
3. Then processes value

CSC343 Introduction to Databases — University of Toronto

Embedded SQL — 37

## 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';
```

CSC343 Introduction to Databases — University of Toronto

Embedded SQL — 39

## Dynamic SQL Calls with Descriptors

```
... .. 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 ... ..
```

CSC343 Introduction to Databases — University of Toronto

Embedded SQL — 38

## 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;
```

CSC343 Introduction to Databases — University of Toronto

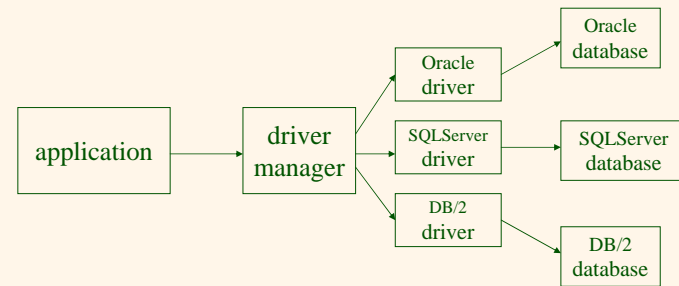
Embedded SQL — 40

## 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 Run-Time Architecture



## JDBC

- 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

## Executing a Query

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

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

## Executing a Query

```
String crs_code, semester;
```

```
.....
```

```
ps.setString(1, crs_code); // set value of 1st in parameter  
ps.setString(2, semester); // set value of 2nd 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...  
}
```

## 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

## Result Sets and Cursors

→ Three types of result sets in JDBC:

- ✓ : not scrollable
- ✓ : scrollable; changes made to underlying tables after the creation of the result set are not visible through that result set
- ✓ : 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
```

## 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

## 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.

## 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

## Example of SQLJ Iterator

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

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

```
GetEnrolledIter iter1;
```

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

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

## SQLJ Example

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

## 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

## 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)
```

## 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
```

## Example of Embedded SQL

```
void DisplayDepartmentSalaries(char DeptName[])  
{ char FirstName[20], Surname[20];  
  long int Salary;  
  $ declare DeptEmp cursor for  
    select FirstName, Surname, Salary  
    from Employee  
    where Dept = :DeptName;  
  $ open DeptEmp;  
  $ fetch DeptEmp into :FirstName, :Surname,  
  :Salary;  
    printf("Department %s\n", DeptName);  
    while (sqlcode == 0)  
    { printf("Name: %s %s ", FirstName, Surname);  
      printf("Salary: %d\n", Salary);  
    }  
  $ fetch DeptEmp into :FirstName, :Surname,  
  :Salary; }  
  $ close DeptEmp; }
```