# CHAPTER 10
# Database Programming

Note: Slides, content, web links and end chapter questions are prepared from Pearson textbook and other Internet resources.
Additionally, many information added from another Pearson book of Kroenke, Auer and others of 2019 – 15th edition.

# Topics of Discussion

A. Database Programming: Techniques and Issues.
B. Approaches to Database Programming
    A. Database Stored Procedures and SQL/PSM
    B. Embedded SQL, Dynamic SQL, and SQLJ
    C. Database Programming with Function Calls: SQL/CLI and JDBC
C. Comparing the 3 Approaches

# A. Database Programming: Techniques and Issues

- Most database access in practical applications is accomplished through software programs that implement database applications.
- This software is usually developed in a general-purpose programming language such as Java, C/C++/C#, COBOL (historically), or some other programming language.
- In addition, many scripting languages, such as PHP, Python, and JavaScript, are also being used for programming of database access within Web applications.
  - Host language : Java, C, C++, C#, COBOL, Python etc..
  - Data sublanguage: SQL

# Database Programming: Techniques and Issues

- Since the beginning, special database programming languages are also developed, and have widespread use, such as Oracle's PL/SQL (Programming Language/SQL), and Transect-SQL (T-SQL) of MS SQL Server.
- Most databases have an Interactive interface
  - SQL commands typed directly into a terminal
  - Execute file of commands @<*filename*>
- Application programs or database applications
  - Used as canned transactions by the end users accessing a database. May have a **Web interface.**

# B. Approaches to Database Programming

1. Database programming language of DBMS itself.
2. Embedding database commands in a general-purpose programming language.
3. Using a library of database functions or classes.

# Approaches to Database Programming

- 1) Database Programming Language (stored procedure)
  - **Database programming language** designed from scratch to be compatible with the database model and query language.
  - Additional programming structures such as loops, and conditional statements are added to the database language to convert it into a full-fledged programming language.
  - In SQL standard it is known as SQL/PSM (SQL/ persistent stored module).
  - Language for specifying stored procedures.

# Approaches to Database Programming

- **2) Embedding** database commands in a general-purpose programming language
  - Database statements identified by a special prefix. (for example EXEC SQL).
  - Pre-compiler or preprocessor scans the source program code
    - Identify database statements and extract them for processing by the DBMS
  - Called **embedded SQL**

# Approaches to Database Programming

- **3**) Using a library of database functions
  - **Library of functions** available to the host programming language to connect to a database.
  - **Application programming interface (API)** used.
  - For object-oriented programming languages (OOPLs), a **class library** is used.
  - For example, Java has the JDBC class library, which can generate various types of objects.
  - Objects like connection objects, query objects, and query result objects.
  - Each type of object has a set of operations associated with the class corresponding to the object.

# B-1 SQL/Persistent Stored Modules (SQL/PSM)

- **Stored procedures**
  - Program modules stored by the DBMS at the database server.
  - Can be functions or procedures.
- SQL/PSM (**SQL/Persistent Stored Modules**)
  - Extensions to SQL
  - Include general-purpose programming constructs in SQL
  - Stored persistently by the DBMS
  - SQL/PSM is an ANSI/ISO standard for embedding procedural programming functionality into SQL.

# SQL/Persistent Stored Modules (SQL/PSM)

- Useful:
  - When database program is needed by several applications.
  - To reduce data transfer and communication cost between client and server in certain situations.
  - To enhance modeling power provided by views.
- Each DBMS product implements SQL/PSM in a different way, with some closer to the standard than others.
    - Microsoft SQL Server calls its version Transact-SQL (**T-SQL**)
    - Oracle Database calls its variant Procedural Language/SQL (**PL/SQL**)
    - MySQL 5.7 implements SQL/PSM, but has no special name for its variant of SQL
    - **PL/pgSQL** in PostgreSQL

# Database Stored Procedures and Functions

- Declaring stored procedures:

```
CREATE PROCEDURE <procedure name> (<parameters>)
<local declarations>
<procedure body> ;

CREATE FUNCTION <function name> (<parameters>)
RETURNS <return type>
<local declarations>
<function body> ;
```

- Each parameter has parameter type
  - **Parameter type**: one of the SQL data types – int, varchar etc..
  - **Parameter mode**: `IN`, `OUT`, or `INOUT`
- Calling a stored procedure:

```
CALL <procedure name or function name>
     (<argument list>) ;
```

# Database Stored Procedures and Functions

- Conditional branching statement:

```
IF <condition> THEN <statement list>
ELSEIF <condition> THEN <statement list>
...
ELSEIF <condition> THEN <statement list>
ELSE <statement list>
END IF ;
```

- Constructs for looping
  1. While
  2. For

```
WHILE <condition> DO
        <statement list>
END WHILE ;
REPEAT
        <statement list>
UNTIL <condition>
END REPEAT ;
```

```
FOR <loop name> AS <cursor name> CURSOR FOR <query> DO
        <statement list>
END FOR ;
```

# Database Stored Procedures and Functions

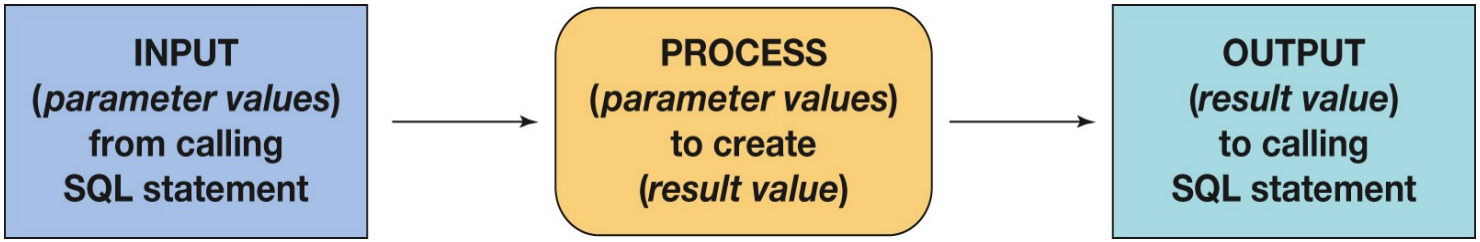
- All user-defined variables must use the @ as the first character. (In MSSQL server and Oracle)
  For example @rowcount
  - SELECT @rowcount =count(*) FROM Employee;
- A typical cursor code (a pointer to a particular row) pattern is:
  - /********* EXAMPLE CODE – DO NOT RUN – ****It is a pseudo code**********/
  - DECLARE SQLcursor CURSOR FOR (SELECT * FROM Employee);
  - /******** Opening SQLcursor executes (SELECT * FROM Employee) *******/
  - OPEN SQLcursor;
  - MOVE SQLcursor to first row of (SELECT * FROM Employee);
  - WHILE (SQLcursor not past the last row) LOOP
  - SET Employee_LastName = Lname;
  - ...other statements...
  - REPEAT LOOP UNTIL DONE;
  - CLOSE SQLcursor
  - ……other processing...

# Transect SQL commands

- Transact-SQL **USE [ {DatabaseName} ]** command tells the stored procedure to use the <DatabaseName> database when it is called. This is the SQL command equivalent of selecting the database name in the drop-down list of Object Explorer in SSMS Studio.
- Transact-SQ**L SET ANSI_NULLS ON** command specifies how SQL server do comparisons of NULL using equal (=) and not equal (<>).
- Transact-SQL **SET QUOTED_IDENTIFIER ON** command specifies that object identifiers (table names, column names, etc.) can be enclosed in double quotes ("), which allows the use of SQL reserved words as object names. For ex: we could run the following statement:
  - **SELECT "Select"**
  - **FROM "FROM"**
  - **WHERE "Where" = 'San Francisco';**
- **GO** command is not a Transact-SQL statement but rather a command used by the SSMS Studio. To mark the end of a batch of commands so the utility can process sections of the code (as marked by the GO commands) separately instead of all at once.

# User-Defined Functions

- A **user-defined function (stored function)** is a stored set of SQL statements that:
  - is *called by name* from another SQL statement.
  - may have *input parameters* passed to it by the calling SQL statement.
  - *returns an output value* to the SQL statement that is called the function.



# User Defined Function to Concatenate FirstName and LastName of Employee table of COMPANY db

```
Create Function [dbo].[NameJoinFullname]
-- These are the input parametres
( @Fname Char(50), @Lname Char(50)
)
Returns Varchar(100)
as
Begin
    -- This is the variable that will hold the value to be returned
    Declare @FullName varchar(100);
    -- SQL statements to join first and last name
    Select @FullName = RTRIM(@Lname) + ', ' + RTRIM(@Fname);
    -- Return the full name
    Return @FullName;
END;
```

# Using the *NameJoinFullName* Function

```
select dbo.NameJoinFullname(Fname, Lname) AS Emp_Full_Name,
        bdate, ssn, address
from employee
order by  Emp_Full_Name;
```

Results | Messages

| Emp_Full_Name | bdate | ssn | address |
|---|---|---|---|
| Bacher, Red | 1980-05-21 | 666666613 | 196 Elm Street, Miami, FL |
| Ball, Nandita | 1969-04-16 | 555555501 | 222 Howard, Sacramento, CA |
| Bays, Bonnie | 1956-06-19 | 444444401 | 111 Hollow, Milwaukee, WI |
| Bender, Bob | 1968-04-17 | 666666600 | 8794 Garfield, Chicago, IL |
| Best, Alec | 1966-06-18 | 444444402 | 233 Solid, Milwaukee, WI |
| Borg, James | 2027-11-10 | 888665551 | 450 Stone, Houston, TX |
| Borg, James | 2027-11-10 | 888665555 | 450 Stone, Houston, TX |
| Brand, Tom | 1966-12-16 | 222222203 | 112 Third St, Milwaukee, WI |
| Carter, Chris | 1960-03-21 | 222222205 | 565 Jordan, Milwaukee, WI |
| Chase, Jeff | 1970-01-07 | 333333301 | 145 Bradbury, Sacramento, CA |
| Drew, Naveen | 1970-05-23 | 666666610 | 198 Elm St, Philadelphia, PA |
| English, Joyce | 1962-07-31 | 453453453 | 5631 Rice Oak, Houston, TX |
| Freed, Alex | 1950-10-09 | 444444400 | 4333 Pillsbury, Milwaukee, WI |

# Using SQL Triggers

- A **trigger** is a stored program that is executed by the DBMS whenever a specified event occurs.
- Three trigger types:
  BEFORE, INSTEAD OF, and AFTER
  - Each type can be declared for Insert, Update, and Delete
  - Resulting in a total of nine trigger types
- Oracle supports all nine trigger types.
- SQL Server supports six* trigger types (INSTEAD OF and AFTER).
- MySQL supports six* trigger types (BEFORE and AFTER).

# Summary of SQL Triggers by DBMS Product*

| Trigger Type / DML Action | BEFORE | INSTEAD OF | AFTER |
|---|---|---|---|
| INSERT | Oracle Database<br>MySQL | Oracle Database<br>SQL Server | Oracle Database<br>SQL Server<br>My SQL |
| UPDATE | Oracle Database<br>My SQL | Oracle Database<br>SQL Server | Oracle Database<br>SQL Server<br>MySQL |
| DELETE | Oracle Database<br>MySQL | Oracle Database<br>SQL Server | Oracle Database<br>SQL Server<br>MySQL |

* Note: This table details might have changed with different versions and respective product. Need to verify by respective DBMS vendor.

# Uses for SQL Triggers

- When a trigger is fired, the DBMS supplies:
  - Old and new values for the update
  - New values for insert
  - Old values for deletion
- The way the values are supplied depends on the DBMS product.

Uses of SQL Triggers

Provide default values.

Enforce data constraints.

Update views.

Perform referential integrity actions.

# Trigger Code to Insert a row in Employee table

```
USE [COMPANY]
GO

/****** Object:  Trigger [dbo].[Insert_invalid_dept_number]    Script Date: 9/16/2020 11:47:45 AM ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO


-- =============================================
-- Author:      <Author,,Name>
-- Create date: <Create Date,,>
-- Description: <Description,,>
-- =============================================
CREATE or ALTER  TRIGGER [dbo].[Insert_invalid_dept_number]
    ON [dbo].[employee] AFTER INSERT

AS
BEGIN
SET NOCOUNT ON;
DECLARE     @ssn                    AS  Char(9),
            @fname                  AS  Char(25),
            @lname                  AS  Char(25),
            @dno                    As  numeric,
            @dept_rowcount          As  numeric
```

# Trigger Code to Insert a row in Employee table

```
/*  Get the values from insereted by INSERT  */
    SELECT  @ssn = Ssn,
            @fname = Fname,
            @lname = Lname,
            @dno = Dno
    FROM    INSERTED;

    /*  First find that entered new employee department exits? */
    SELECT  @dept_rowcount = COUNT(*)
        FROM    dbo.department AS T
        WHERE   T.Dnumber = @dno;
-- Since this is an after trigger, @@Rowcount includes the new row.
 --      SET @PriorRowCount = (@@ROWCOUNT -1 );
      IF  @dept_rowcount = 1
        /*  The newly emter Employee`s department exit in the Company */
        BEGIN
        PRINT '*************************************************************'
        PRINT ''
        PRINT '   Insert Comlete'
        PRINT ''
        PRINT '   New Employee ssn        =  '+@ssn
        PRINT '   New Employee name       =  '+@fname +' '+@lname
        PRINT '   New Employee Dept       =  '+@dno
        PRINT '*************************************************************'
      END
```

# Trigger Code to Insert a row in Employee table

```
    ELSE
        /*  Disallow the tranaction and send an error message.  */
        BEGIN
            /*  Disallow the insert of the new Employee  */
            ROLLBACK TRANSACTION;
            /*  Print the error message  */
        PRINT '************************************************************************'
        PRINT ''
        PRINT '   You have attempted to assign the department that does not exist in company '
        PRINT '************************************************************************'
        PRINT '   Company policy - Please reassign departemnt to newly entered employee.'
        PRINT ''
        PRINT '************************************************************************'
        END
END;
GO
```

# Using trigger

- When we try to insert a row for new employee assigning department number 9, trigger fired and not allow to insert because department 9 dose not exist in Department table

```
use COMPANY
go

INSERT INTO employee VALUES
  ('Ashok','B','Patel','123456777','09-Jan-55','731 Fondren, Houston, TX','M',30000,'333445555',9);
```

Messages

```
**********************************************************************

   You have attempted to assign the department that does not exit in company
**********************************************************************
   Company policy - Please reassign departemnt to newly entered employee.

**********************************************************************
Msg 3609, Level 16, State 1, Line 4
The transaction ended in the trigger. The batch has been aborted.

Completion time: 2020-09-16T11:45:26.1939398-04:00
```

# Using Stored Procedures

- A **stored procedure** is a program that is stored within the database and is compiled when used.
  - In Oracle, it can be written in PL/SQL
  - In SQL Server 2017, it can be written in T-SQL
- Stored procedures can receive input parameters and they can return results
- Stored procedures can be called from:
  - Programs written in standard languages, e.g., Java, C#
  - Scripting languages, e.g., JavaScript, VBScript
  - SQL command prompt, e.g., SQL Plus, Query Analyzer

# Procedure code for “Print_Emp_Data”

```
USE [COMPANY]
GO

/****** Object:  StoredProcedure [dbo].[Print_Emp_Data]    Script Date: 9/12/2020 6:18:29 PM ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

-- =============================================
-- Author:      <Ashok Patel>
-- Create date: <Sept 11, 2020>
-- =============================================

CREATE   PROCEDURE [dbo].[Print_Emp_Data]
    @emp_LastName       Char(25),
    @emp_FirstName      Char(25)
AS
    DECLARE     @ssn            AS Char(9)
    DECLARE     @lname          AS Char(25)
    DECLARE     @fname          AS Char(25)


    DECLARE     emp_data_cursor     CURSOR FOR
                    SELECT  ssn, lname, fname
                    FROM        dbo.employee
                    WHERE       lname = @emp_LastName
                        AND     fname = @emp_FirstName
```

# Procedure code for “Print_Emp_Data”

```
PRINT '*************************************************************'
PRINT ''
Print '                    EMPLOYEE Data '
PRINT '*************************************************************'

OPEN emp_data_cursor;
FETCH NEXT FROM emp_data_cursor
    INTO @ssn, @lname, @fname

    WHILE @@FETCH_STATUS = 0
        BEGIN

PRINT ''
PRINT '    Emp_SSN       = '+(CONVERT (Char(9), @ssn))
PRINT '    Emp Name      = '+(RTRIM(@fname))+' '+(RTRIM(@lname))
PRINT '*************************************************************'

        FETCH NEXT FROM emp_data_cursor
        INTO @ssn, @lname, @fname

        END;

CLOSE emp_data_cursor;
DEALLOCATE emp_data_cursor;
```

# Execute Procedure “Print_Emp_Data”

```
-- command for executing procedure - print_emp_data

execute Print_Emp_Data
    @emp_LastName = Smith, @emp_FirstName = John;
```

50 %

Messages

```
**********************************************************

                    EMPLOYEE Data
**********************************************************

   Emp_SSN       = 123456789
   Emp Name      = John Smith
**********************************************************

Completion time: 2020-09-16T11:19:18.9322967-04:00
```

# Triggers Versus Stored Procedures

**Trigger**

Module of code that is called by the DBMS when INSERT, UPDATE, or DELETE commands are issued.

Assigned to a table or view.

Depending on the DBMS, may have more than one trigger per table or view.

Triggers may issue INSERT, UPDATE, and DELETE commands and thereby may cause the invocation of other triggers.

**Stored Procedures**

Module of code that is called by a user or database administrator.

Assigned to a database, but not to a table or a view.

Can issue INSERT, UPDATE, DELETE, and MERGE commands.

Used for repetitive administration tasks or as part of an application.

# Advantages of Stored Procedures

Advantages of Stored Procedures

- Greater security.
- Decreased network traffic.
- SQL can be optimized.
- Code sharing.
- Less work.
- Standardized processing.
- Specialization among developers.

# Comparison of User-Defined Functions, Triggers, and Stored Procedures

| | User-Defined Functions | Triggers | Stored Procedures |
|---|---|---|---|
| Can accept parameters | Yes | No | Yes |
| Can return a result value or values | Yes | No | Yes |
| Can be used in SELECT statements | Yes | No | No |
| Can use SELECT statements | Yes | Yes | Yes |
| Can use INSERT statements | No | Yes | Yes |
| Can use UPDATE statements | No | Yes | Yes |
| Can use DELETE statements | No | Yes | Yes |
| Can call a User-Defined Function | Yes | Yes | Yes |
| Can invoke a Trigger | No | Yes<br>Indirectly via INSERT, UPDATE, or DELETE) | Yes<br>(Indirectly via INSERT, UPDATE, or DELETE) |
| Can invoke a Stored Procedure | No | Yes | Yes |
| Is stored as a database-wide object | Yes | No | Yes |
| Is stored as a table-specific object | No | Yes | No |

# B-2. Embedded SQL, Dynamic SQL, and SQLJ

- **Embedded SQL:** In the embedded approach, the programming language is called the host language. Most SQL statements—including data or constraint definitions, queries, updates, or view definitions—can be embedded in a host language program.
  - Like C language in our example.
- `EXEC SQL`
  - Prefix
  - **Preprocessor** separates embedded SQL statements from host language code
  - Terminated by a matching `END-EXEC` Or by a semicolon (;)

# Embedded SQL, Dynamic SQL, and SQLJ

- Within an embedded SQL command, the programmer can refer to specially declared C program variables; these are called shared variables because they are used in both the C program and the embedded SQL statements.
- **Shared variables**
  - Prefixed by a colon (:) in SQL statement
  - This distinguishes program variable names from the names of database schema constructs such as attributes (column names) and relations (table names).
  - It also allows program variables to have the same names as attribute names, since they are distinguishable by the colon (:) prefix in the SQL statement.

# Embedded SQL, Dynamic SQL, and SQLJ

- Suppose that we want to write C programs to process the COMPANY database in Figure 5.5. We need to declare program variables to match the types of the database attributes that the program will process.

**Figure 10.1** C program variables used in the embedded SQL examples E1 and E2.

```
0) int loop ;
1) EXEC SQL BEGIN DECLARE SECTION ;
2) varchar dname [16], fname [16], lname [16], address [31] ;
3) char ssn [10], bdate [11], sex [2], minit [2] ;
4) float salary, raise ;
5) int dno, dnumber ;
6) int SQLCODE ; char SQLSTATE [6] ;
7) EXEC SQL END DECLARE SECTION ;
```

# Embedded SQL, Dynamic SQL, and SQLJ

- Example of embedded SQL program (figure 10.2) is a repeating program segment (loop) that takes as input the SSN of an employee and prints some information from the corresponding EMPLOYEE record in the database.

**Figure 10.2** Program segment E1, a C program segment with embedded SQL.

```
   //Program Segment E1:
0) loop = 1 ;
1) while (loop) {
2)   prompt("Enter a Social Security Number: ", ssn) ;
3)   EXEC SQL
4)     SELECT Fname, Minit, Lname, Address, Salary
5)     INTO :fname, :minit, :lname, :address, :salary
6)     FROM EMPLOYEE WHERE Ssn = :ssn ;
7)   if (SQLCODE = = 0) printf(fname, minit, lname, address, salary)
8)     else printf("Social Security Number does not exist: ", ssn) ;
9)   prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10)  }
```

# Retrieving Single Tuples with Embedded SQL

- SQLCODE and SQLSTATE communication variables
  - Used by DBMS to communicate exception or error conditions
  - SQLCODE variable
    - 0 = statement executed successfully
    - 100 = no more data available in query result
    - < 0 = indicates some error has occurred
  - SQLSTATE
    - String of five characters. '00000' = no error or exception
    - Other values indicate various errors or exceptions
    - For example, '02000' indicates 'no more data' when using SQLSTATE

# Embedded SQL, Dynamic SQL, and SQLJ

- Connecting to the Database: The SQL command for establishing a connection to a database has the following form:

```
CONNECT TO <server name> AS <connection name>
AUTHORIZATION <user account name and password>;
```

- In general, since a user or program can access several database servers, several connections can be established, but only one connection can be active at any point in time. The programmer or user can use the `<connection name>` to change from the currently active connection to a different one by using the following command.

```
SET CONNECTION <connection name> ;
```

- Once a connection is no longer needed, it can be terminated by the following command.

```
DISCONNECT <connection name> ;
```

Note: In the examples which we discussing, we assume that the appropriate connection has already been established to the COMPANY database, and that it is the currently active connection.

# Retrieving Multiple Rows Using Cursors

- Cursor: Points to a single tuple (row) from result of query
- **OPEN CURSOR** command
  - Fetches query result and sets cursor to a position before first row in result
  - Becomes current row for cursor
- **FETCH** commands
  - Moves cursor to next row in result of query
- **FOR UPDATE OF**
  - List the names of any attributes that will be updated by the program
- Fetch orientation
  - Added using value: `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE` *i*, and `RELATIVE` *i*

**DECLARE** <cursor name> [ **INSENSITIVE** ] [ **SCROLL** ] **CURSOR**
[ **WITH HOLD** ] **FOR** <query specification>
[ **ORDER BY** <ordering specification> ]
[ **FOR READ ONLY** | **FOR UPDATE** [ **OF** <attribute list> ] ] ;

**Figure 10.3** Program segment E2, a C program segment that uses cursors with embedded SQL for update purposes.

```
    //Program Segment E2:
 0) prompt("Enter the Department Name: ", dname) ;
 1) EXEC SQL
 2)   SELECT Dnumber INTO :dnumber
 3)   FROM DEPARTMENT WHERE Dname = :dname ;
 4) EXEC SQL DECLARE EMP CURSOR FOR
 5)   SELECT Ssn, Fname, Minit, Lname, Salary
 6)   FROM EMPLOYEE WHERE Dno = :dnumber
 7)   FOR UPDATE OF Salary ;
 8) EXEC SQL OPEN EMP ;
 9) EXEC SQL FETCH FROM EMP INTO :ssn, :fname, :minit, :lname, :salary ;
10) while (SQLCODE = = 0) {
11)   printf("Employee name is:", Fname, Minit, Lname) ;
12)   prompt("Enter the raise amount: ", raise) ;
13)   EXEC SQL
14)     UPDATE EMPLOYEE
15)     SET Salary = Salary + :raise
16)     WHERE CURRENT OF EMP ;
17)   EXEC SQL FETCH FROM EMP INTO :ssn, :fname, :minit, :lname, :salary ;
18)   }
19) EXEC SQL CLOSE EMP ;
```

# Using Dynamic SQL

- **Dynamic SQL:** In the previous example of embedded SQL queries were written as part of the host program, hence, anytime we want to write a different query, we must modify the program code and go through all the steps involved (compiling, debugging, testing, and so on).
- In some cases, it is convenient to write a program that can execute different SQL queries or updates (or other operations) *dynamically at runtime*.
- For example, we may want to write a program that accepts an SQL query typed from the monitor, executes it, and displays its result,
- Another example, when a user-friendly interface generates SQL queries dynamically for the user based on user input through a Web interface or Mobile App.

# Using Dynamic SQL

- Program segment E3 in Figure 10.4 reads a string that is input by the user into the string program variable **sqlupdatestring** in line 3. It then prepares this as an SQL command in line 4 by associating it with the SQL variable sqlcommand. Line 5 then executes the command.

**Figure 10.4** Program segment E3, a C program segment that uses dynamic SQL for updating a table.

```
    //Program Segment E3:
0)  EXEC SQL BEGIN DECLARE SECTION ;
1)  varchar sqlupdatestring [256] ;
2)  EXEC SQL END DECLARE SECTION ;
    ...
3)  prompt("Enter the Update Command: ", sqlupdatestring) ;
4)  EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;
5)  EXEC SQL EXECUTE sqlcommand ;
    ...
```

# SQLJ: Embedding SQL in Java

- How SQL can be embedded in an object-oriented programming language, like Java. SQLJ is a standard that has been adopted by several vendors for embedding SQL in Java or embedding SQL in Java.
- An SQLJ translator generally convert SQL statements into Java, which can then be executed through the JDBC interface. Hence, it is necessary to install a JDBC driver when using SQLJ. (That Imports several class libraries)

**Figure 10.5** Importing classes needed for including SQLJ in Java programs in Oracle, and establishing a connection and default context.

```
1) import java.sql.* ;
2) import java.io.* ;
3) import sqlj.runtime.* ;
4) import sqlj.runtime.ref.* ;
5) import oracle.sqlj.runtime.* ;
   ...
6) DefaultContext cntxt =
7) oracle.getConnection("<url name>", "<user name>", "<password>", true) ;
8) DefaultContext.setDefaultContext(cntxt) ;
   ...
```

# SQLJ: Embedding SQL in Java

- **Default context:** Uses **exceptions** for error handling
  - `SQLException` is used to return errors or exception conditions

**Figure 10.6** Java program variables used in SQLJ examples J1 and J2.

```
1)  string dname, ssn , fname, fn, lname, ln,
    bdate, address ;
2)  char sex, minit, mi ;
3)  double salary, sal ;
4)  integer dno, dnumber ;
```

**Figure 10.7** Program segment J1, a Java program segment with SQLJ.

```
   //Program Segment J1:
1) ssn = readEntry("Enter a Social Security Number: ") ;
2) try {
3)   #sql { SELECT Fname, Minit, Lname, Address, Salary
4)     INTO :fname, :minit, :lname, :address, :salary
5)     FROM EMPLOYEE WHERE Ssn = :ssn} ;
6) } catch (SQLException se) {
7)     System.out.println("Social Security Number does not exist: " + ssn) ;
8)     Return ;
9)   }
10) System.out.println(fname + " " + minit + " " + lname + " " + address
     + " " + salary)
```

# Retrieving Multiple Tuples in SQLJ Using Iterators

- **Iterator**
  - Object associated with a collection (set or multiset) of records in a query result
- **Named iterator**
  - Associated with a query result by listing attribute names and types in query result
- **Positional iterator**
  - Lists only attribute types in query result

**Figure 10.8** Program segment J2A, a Java program segment that uses a **named iterator** to print employee information in a particular department.

```
    //Program Segment J2A:
 0) dname = readEntry("Enter the Department Name: ") ;
 1) try {
 2)   #sql { SELECT Dnumber INTO :dnumber
 3)     FROM DEPARTMENT WHERE Dname = :dname} ;
 4) } catch (SQLException se) {
 5)   System.out.println("Department does not exist: " + dname) ;
 6)   Return ;
 7)   }
 8) System.out.printline("Employee information for Department: " + dname) ;
 9) #sql iterator Emp(String ssn, String fname, String minit, String lname,
      double salary) ;
10) Emp e = null ;
11) #sql e = { SELECT ssn, fname, minit, lname, salary
12)   FROM EMPLOYEE WHERE Dno = :dnumber} ;
13) while (e.next()) {
14)   System.out.printline(e.ssn + " " + e.fname + " " + e.minit + " " +
        e.lname  + " " + e.salary) ;
15) } ;
16) e.close() ;
```

**Figure 10.9** Program segment J2B, a Java program segment that uses a **positional iterator** to print employee information in a particular department.

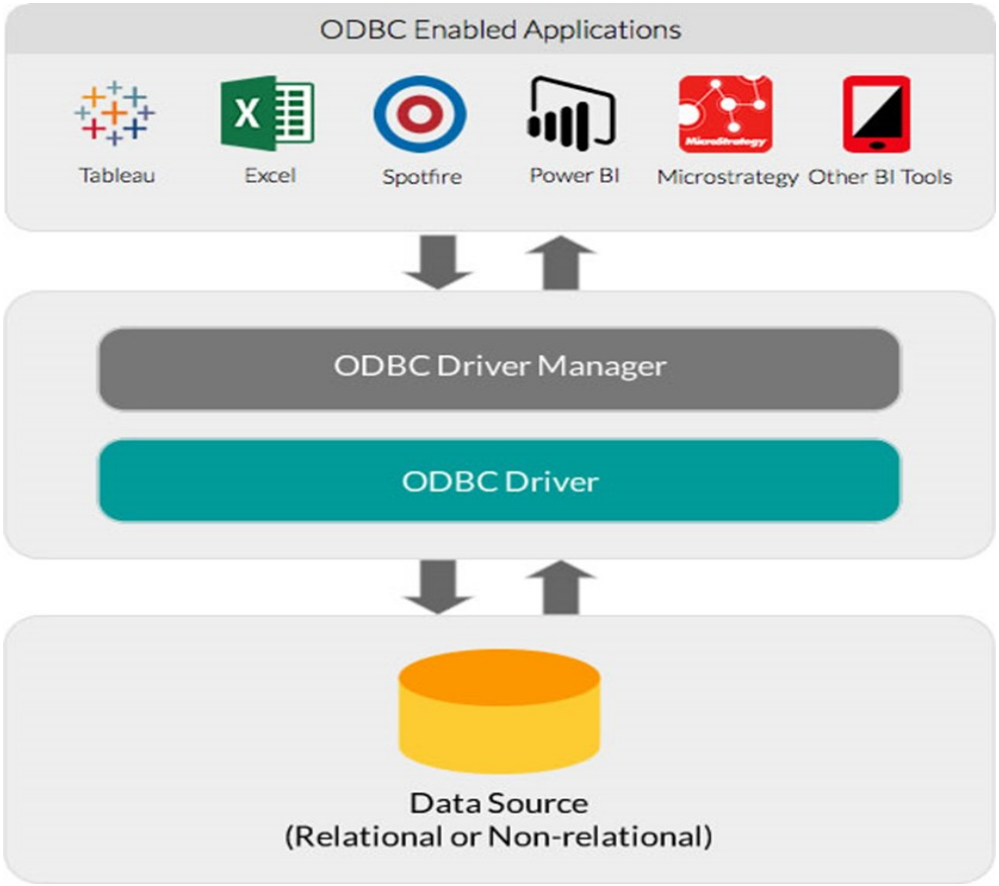
```
    //Program Segment J2B:
 0) dname = readEntry("Enter the Department Name: ") ;
 1) try {
 2)   #sql { SELECT Dnumber INTO :dnumber
 3)     FROM DEPARTMENT WHERE Dname = :dname} ;
 4) } catch (SQLException se) {
 5)   System.out.println("Department does not exist: " + dname) ;
 6)   Return ;
 7)   }
 8) System.out.printline("Employee information for Department: " + dname) ;
 9) #sql iterator Emppos(String, String, String, String, double) ;
10) Emppos e = null ;
11) #sql e = { SELECT ssn, fname, minit, lname, salary
12)   FROM EMPLOYEE WHERE Dno = :dnumber} ;
13) #sql { FETCH :e INTO :ssn, :fn, :mi, :ln, :sal} ;
14) while (!e.endFetch()) {
15)   System.out.printline(ssn + " " + fn + " " + mi + " " + ln + " " + sal) ;
16)   #sql { FETCH :e INTO :ssn, :fn, :mi, :ln, :sal} ;
17) } ;
18) e.close() ;
```

# B3. Database Programming with Function Calls: SQL/CLI & JDBC

- Use of function calls
  - **Dynamic** approach for database programming
- Library of functions
  - Also known as **application programming interface (API)**
  - Used to access database
- **SQL Call Level Interface (SQL/CLI)**
  - Part of SQL standard
  - This was developed as a standardization of the popular library of functions known as **ODBC** (Open Database Connectivity).

# Database Programming with Function Calls: SQL/CLI & JDBC

The ODBC driver processes ODBC function calls, submits SQL requests to a specific data source and returns results to the application.



# SQL/CLI: Using C as the Host Language

- **Environment record**
  - Track one or more database connections
  - Set environment information
- **Connection record**
  - Keeps track of information needed for a particular database connection
- **Statement record**
  - Keeps track of the information needed for one SQL statement
- **Description record**
  - Keeps track of information about tuples or parameters
- **Handle** to the record
  - C pointer variable makes record accessible to program

**Figure 10.10** Program segment CLI1, a C program segment with SQL/CLI.

```
   //Program CLI1:
 0) #include sqlcli.h ;
 1) void printSal() {
 2) SQLHSTMT stmt1 ;
 3) SQLHDBC con1 ;
 4) SQLHENV env1 ;
 5) SQLRETURN ret1, ret2, ret3, ret4 ;
 6) ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
 7) if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
 8) if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz",
      SQL_NTS) else exit ;
 9) if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select Lname, Salary from EMPLOYEE where Ssn = ?",
      SQL_NTS) ;
11) prompt("Enter a Social Security Number: ", ssn) ;
12) SQLBindParameter(stmt1, 1, SQL_CHAR, &ssn, 9, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15)   SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16)   SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17)   ret2 = SQLFetch(stmt1) ;
18)   if (!ret2) printf(ssn, lname, salary)
19)     else printf("Social Security Number does not exist: ", ssn) ;
20)   }
21) }
```

**Figure 10.11** Program segment CLI2, a C program segment that uses SQL/CLI for a query with a **collection of tuples** in its result.

```
   //Program Segment CLI2:
 0) #include sqlcli.h ;
 1) void printDepartmentEmps() {
 2) SQLHSTMT stmt1 ;
 3) SQLHDBC con1 ;
 4) SQLHENV env1 ;
 5) SQLRETURN ret1, ret2, ret3, ret4 ;
 6) ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
 7) if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
 8) if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz",
      SQL_NTS) else exit ;
 9) if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select Lname, Salary from EMPLOYEE where Dno = ?",
      SQL_NTS) ;
11) prompt("Enter the Department Number: ", dno) ;
12) SQLBindParameter(stmt1, 1, SQL_INTEGER, &dno, 4, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15)   SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16)   SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17)   ret2 = SQLFetch(stmt1) ;
18)   while (!ret2) {
19)     printf(lname, salary) ;
20)     ret2 = SQLFetch(stmt1) ;
21)     }
22)   }
23) }
```

# JDBC: SQL Function Calls for Java Programming

- JDBC
  - Java function libraries
- Single Java program can connect to several different databases
  - Called data sources accessed by the Java program
- `Class.forName("oracle.jdbc.driver.OracleDriver")`
  - Load a **JDBC driver** explicitly
- **`Connection` object**
- **`Statement` object** has two subclasses:
  - `PreparedStatement` and `CallableStatement`
- Question mark (`?`) symbol
  - Represents a statement parameter
  - Determined at runtime
- **`ResultSet` object**
  - Holds results of query

**Figure 10.12** Program segment JDBC1, a Java program segment with JDBC.

```
   //Program JDBC1:
 0) import java.io.* ;
 1) import java.sql.*
   ...
 2) class getEmpInfo {
 3)   public static void main (String args []) throws SQLException, IOException {
 4)     try { Class.forName("oracle.jdbc.driver.OracleDriver")
 5)     } catch (ClassNotFoundException x) {
 6)       System.out.println ("Driver could not be loaded") ;
 7)     }
 8)     String dbacct, passwrd, ssn, lname ;
 9)     Double salary ;
10)     dbacct = readentry("Enter database account:") ;
11)     passwrd = readentry("Enter password:") ;
12)     Connection conn = DriverManager.getConnection
13)       ("jdbc:oracle:oci8:" + dbacct + "/" + passwrd) ;
14)     String stmt1 = "select Lname, Salary from EMPLOYEE where Ssn = ?" ;
15)     PreparedStatement p = conn.prepareStatement(stmt1) ;
16)     ssn = readentry("Enter a Social Security Number: ") ;
17)     p.clearParameters() ;
18)     p.setString(1, ssn) ;
19)     ResultSet r = p.executeQuery() ;
20)     while (r.next()) {
21)       lname = r.getString(1) ;
22)       salary = r.getDouble(2) ;
23)       system.out.println(lname + salary) ;
24)   } }
25) }
```

**Figure 10.13** Program segment JDBC2, a Java program segment that uses JDBC for a query with a **collection of tuples** in its result.

```
    //Program Segment JDBC2:
 0) import java.io.* ;
 1) import java.sql.*
    ...
 2) class printDepartmentEmps {
 3)   public static void main (String args [])
          throws SQLException, IOException {
 4)     try { Class.forName("oracle.jdbc.driver.OracleDriver")
 5)     } catch (ClassNotFoundException x) {
 6)       System.out.println ("Driver could not be loaded") ;
 7)     }
 8)     String dbacct, passwrd, lname ;
 9)     Double salary ;
10)     Integer dno ;
11)     dbacct = readentry("Enter database account:") ;
12)     passwrd = readentry("Enter password:") ;
13)     Connection conn = DriverManager.getConnection
14)       ("jdbc:oracle:oci8:" + dbacct + "/" + passwrd) ;
15)     dno = readentry("Enter a Department Number: ") ;
16)     String q = "select Lname, Salary from EMPLOYEE where Dno = " +
        dno.tostring() ;
17)     Statement s = conn.createStatement() ;
18)     ResultSet r = s.executeQuery(q) ;
19)     while (r.next()) {
20)       lname = r.getString(1) ;
21)       salary = r.getDouble(2) ;
22)       system.out.println(lname + salary) ;
23)   } }
24) }
```

# C. Comparing the Three Approaches

1. Database Programming Language Approach
   - Does not suffer from the impedance mismatch problem
   - Programmers must learn a new language
2. Embedded SQL Approach
   - Query text checked for syntax errors and validated against database schema at compile time
   - For applications where queries have to be generated at runtime, 'dynamic SQL' is more suitable.
3. Library of Function Calls Approach
   - More flexibility
   - For complex programming
   - No checking of syntax at compile time

# Summary

- Techniques for database programming
  - Embedded SQL
  - SQLJ
  - Function call libraries
  - SQL/CLI standard
  - JDBC class library
  - Stored procedures
  - SQL/PSM

# End Chapter Questions

- 10.1. What is ODBC? How is it related to SQL/CLI?
- 10.2. What is JDBC? Is it an example of embedded SQL or of using function calls?
- 10.3. List the three main approaches to database programming. What are the advantages and disadvantages of each approach?
- 10.5. Describe the concept of a cursor and how it is used in embedded SQL.
- 10.6. What is SQLJ used for? Describe the two types of iterators available in SQLJ.