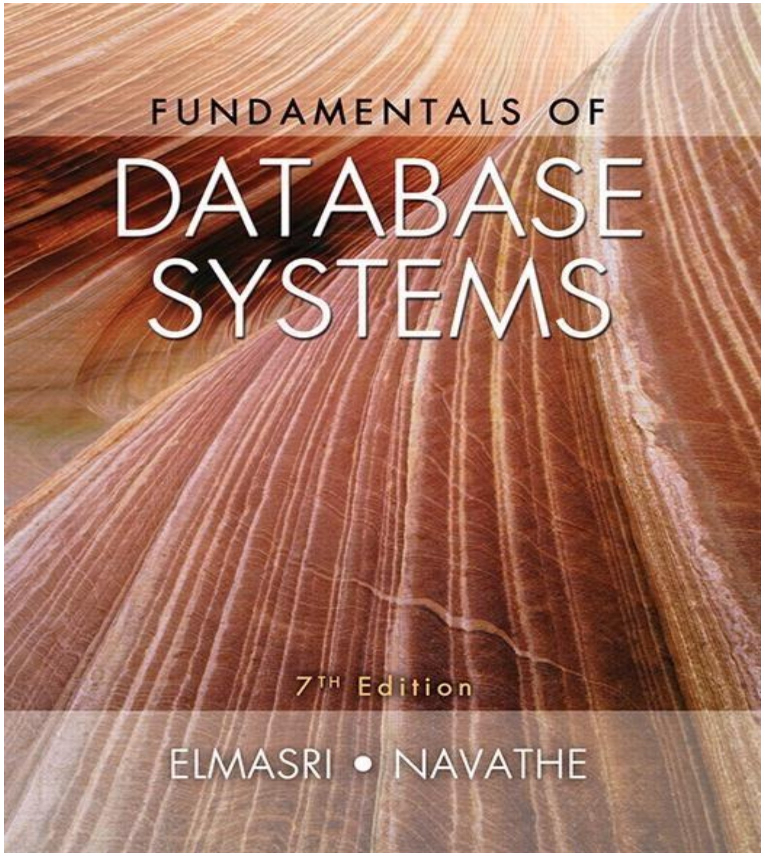
Comp-4150: Advanced and Practical Database Systems

- Ramez Elmasri , Shamkant B. Navathe(2016) Fundamentals of Database Systems (7th Edition), Pearson, isbn 10: 0-13-397077-9; isbn-13:978-0-13-397077-7.

Chapter 13:

Introduction to SQL Programming Techniques



Chapter 10: Introduction to SQL Programming Techniques: Outline

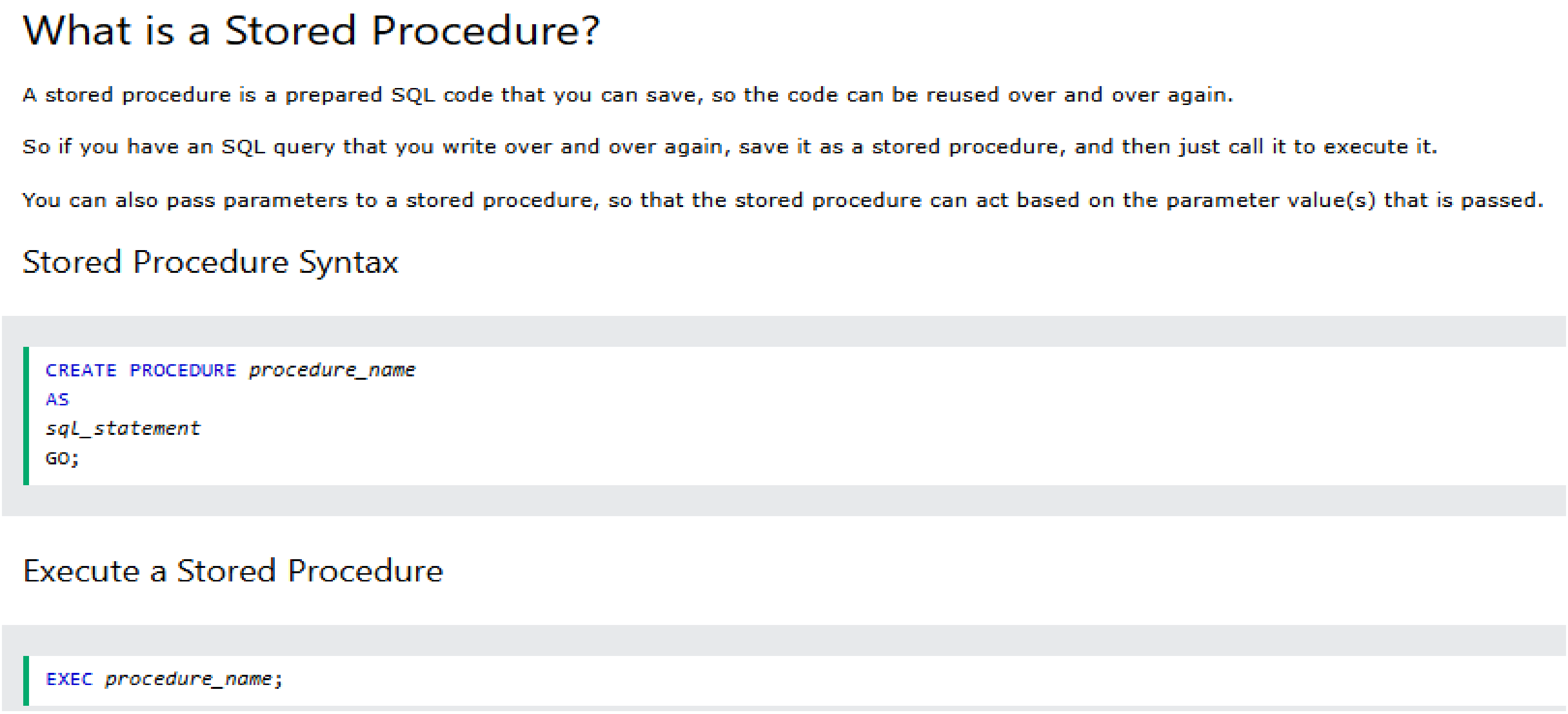
- 1. Database Programming: Techniques and Issues
- 2. Embedded SQL, Dynamic SQL, and SQLJ
- 3. Database Programming with Function Calls: SQL/CLI and JDBC

Introduction to SQL Programming Techniques

- **Database applications: Techniques and Issues**
- **What are some of the techniques developed for accessing databases from application programs?**
- **Most database access in practical applications (e.g., student information system) is accomplished through software programs that implement database applications through the following three main approaches:**
- 1. Embedding Database Commands in a general-purpose programming language called Host language such as:
    - Java (embedded version is SQLJ), C/C++/C# (embedded SQL with C is Pro*C), COBOL, or some other programming language, (e.g., Dynamic SQL).

Introduction to SQL Programming Techniques

- 2. Database Programming with Function Calls and Class Libraries: SQL/CLI (eg., ODBC), JDBC connectivity
- 3. Database Stored Procedures and SQL/PSM (**SQL/PSM** (SQL/Persistent Stored Modules) is an ISO standard mainly defining an extension of SQL with a procedural language for use in stored procedures)



What is a Stored Procedure?

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Stored Procedure Syntax

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

Execute a Stored Procedure

```
EXEC procedure_name;
```

10.1. Database Programming: Techniques and Issues

- SQL standards are
  - Continually evolving
  - Each DBMS vendor may have some variations from standard
- **Interactive interface (e.g., through a query processor as Oracle SQLPlus or through a DBMS desktop front-end tool such as MS Access or SQL Developer, consists of**
  - SQL commands typed directly into a monitor
- To Execute a **filename.sql of SQL commands, use:**
  - *@filename.sql*
- **Application programs** or **database applications**
  - Used as canned transactions(predefined set of operations) by the end users access a database and may have web interface.

10.1.1 Three Approaches to Database Programming (Discussed in Chapter 10)

- **1. Embedding** database commands in a general-purpose host programming language such as C or Java.
  - Here, Database statements are identified by a special prefix. For example, in an embedded SQL program with C host programming language, an embedded SQL statement is prefixed with the keywords: EXEC SQL
  - **Precompiler** or **preprocessor** scans the source program code
    - Identifies database statements and extracts them for processing by the DBMS
  - This first Approach is Called **embedded SQL**

10.1.1 Three Approaches to Database Programming (Discussed in Chapter 10)

- 2. Using a library of database functions (API) provided by the host language to connect to the DBMS.
  - **Library of functions** available to the host programming language (e.g. JDBC with Java)
  - **These functions in this second approach are called Application programming interface (API)**
- 3. Designing a brand-new database programming language
  - **Database programming language** designed from scratch by adding programming constructs of decision, repetition, functions and others to SQL.
  - These are called stored procedures (eg. Oracle PL/SQL)
- The First two approaches are more commonly used.

10.1.2 Impedance Mismatch

- Impedance mismatch discusses the problems that may occur when databases are queried using any of the three database programming approaches (e.g., Oracle PL/SQL and programming language model of methods 1 and 2)
- **1. Data model Binding** for each host programming language with the data model of the database can be challenging.
  - Thus, each such embedded C language program specifies for each attribute relational database (e.g., Varchar) type the compatible programming language types (e.g., char[ ])
- 2. Cursor or iterator variable has to be used to
  - Loop over the tuples in a query result that are typically presented as a relation in a relational database and these operations can appear complex.

10.1.3 Typical Sequence of Interaction in Database Programming

- When writing a database application program, a common sequence of interaction is:
- 1. The program must first Open a connection to database server usually by specifying the internet address (URL) of the machine where the server is located, plus providing a login account name and password for the database access.
- 2. Once the connection is established, the program can Interact with database by submitting queries, updates, and other database commands.
- 3. When the program no longer needs access to a particular database, it should terminate or close connection to the database

10.2. Embedded SQL, Dynamic SQL, and SQLJ

- **Examples are:**
- **1. Embedded SQL**
  - C language
- **2. SQLJ**
  - Java language
- Programming language called **host language**

Static or Embedded SQL are SQL statements in an application that do not change at runtime and, therefore, can be hard-coded into the application. Dynamic SQL is SQL statements that are constructed at runtime; for example, the application may allow users to enter their own queries.

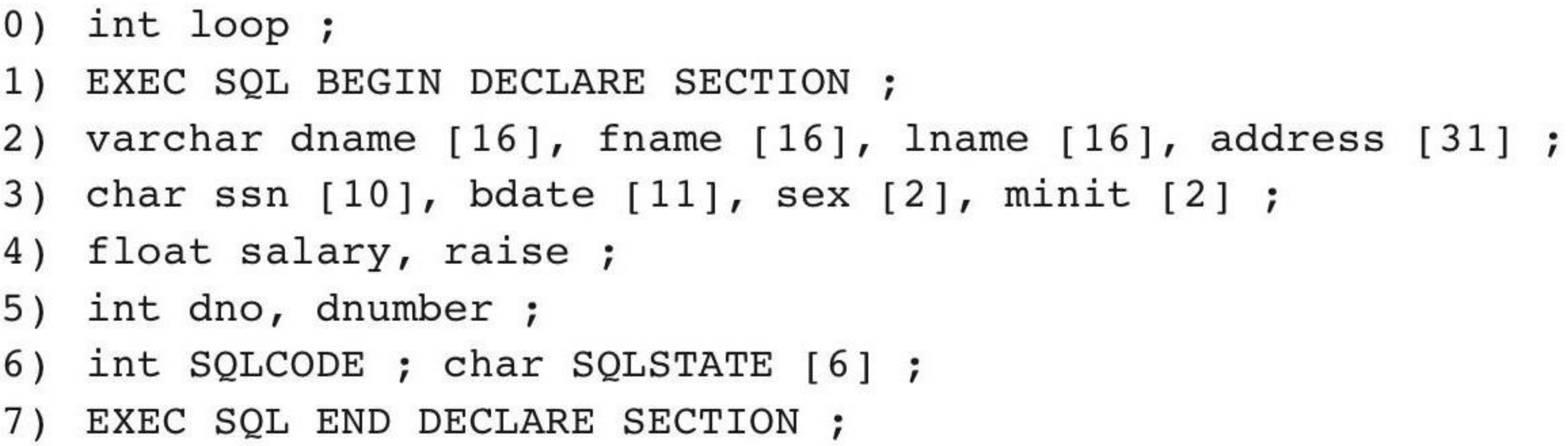
Dynamic SQL is a programming technique that enables you to build SQL statements dynamically at runtime

10.2.1. Retrieving Single Tuples with Embedded SQL

- EXEC SQL
  - Prefix
  - **Preprocessor** separates embedded SQL statements from host language code
  - Terminated by a matching END-EXEC
    - Or by a semicolon (;)
- **Shared variables**
  - Used in both the C program and the embedded SQL statements
  - Prefixed by a colon (:) in SQL statement

Host Variable Syntax
Any variable declared inside the EXEC SQL BEGIN DECLARE SECTION is a **host variable**.
It can be used to pass data **to** or **from** the SQL query.
The colon (:) is mandatory to differentiate host variables from SQL keywords or table/column names.

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



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

10.2.1. Retrieving Single Tuples with Embedded SQL (cont'd.)

- Connecting to the database

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

- Change connection

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

- Terminate connection

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

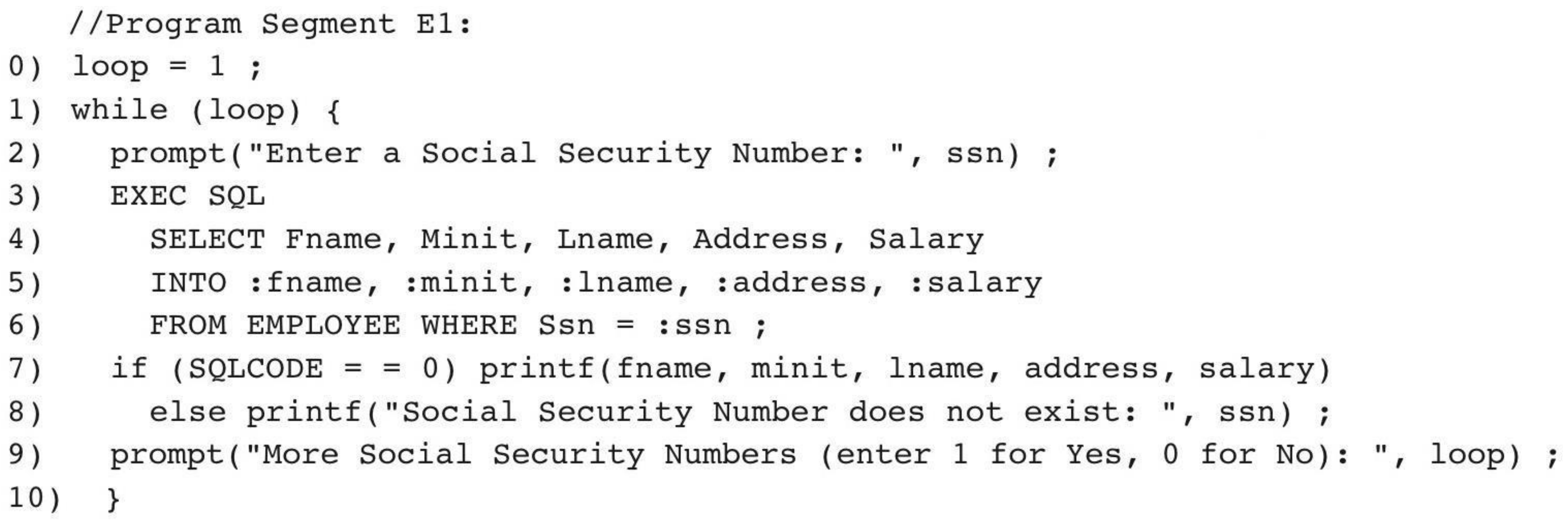
10.2.1 Retrieving Single Tuples with Embedded SQL (cont'd.)

- **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

10.2.1. Retrieving Single Tuples with Embedded SQL (cont'd.)

- 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
  - Example 10.1: Given the Company Database example table Employee already in your database, write an embedded SQL program in C/C++ that will accept as input a social security number of an employee and prints some information from the Employee record.

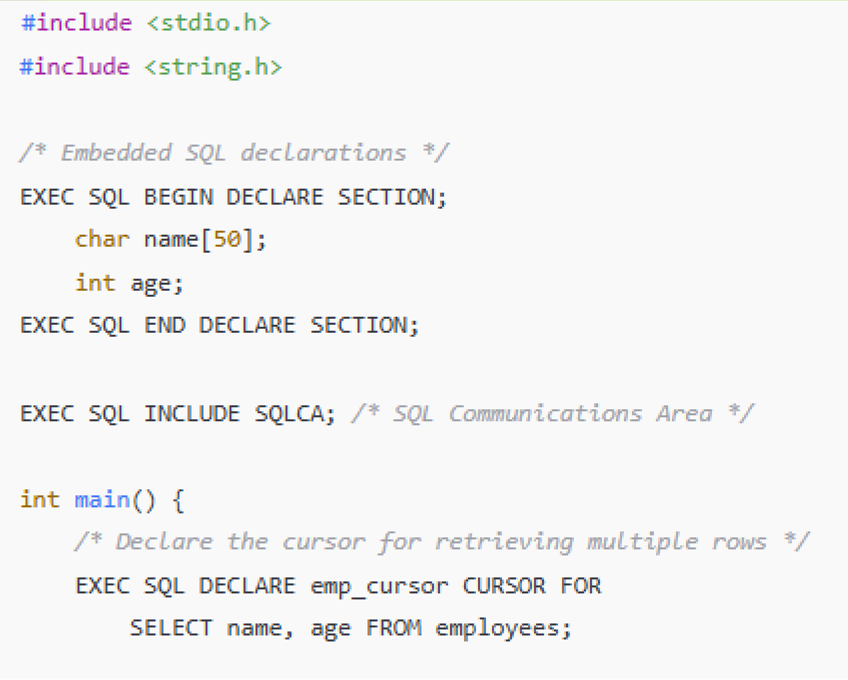
Figure 10.2 Program segment E1, a C program segment with embedded SQL.



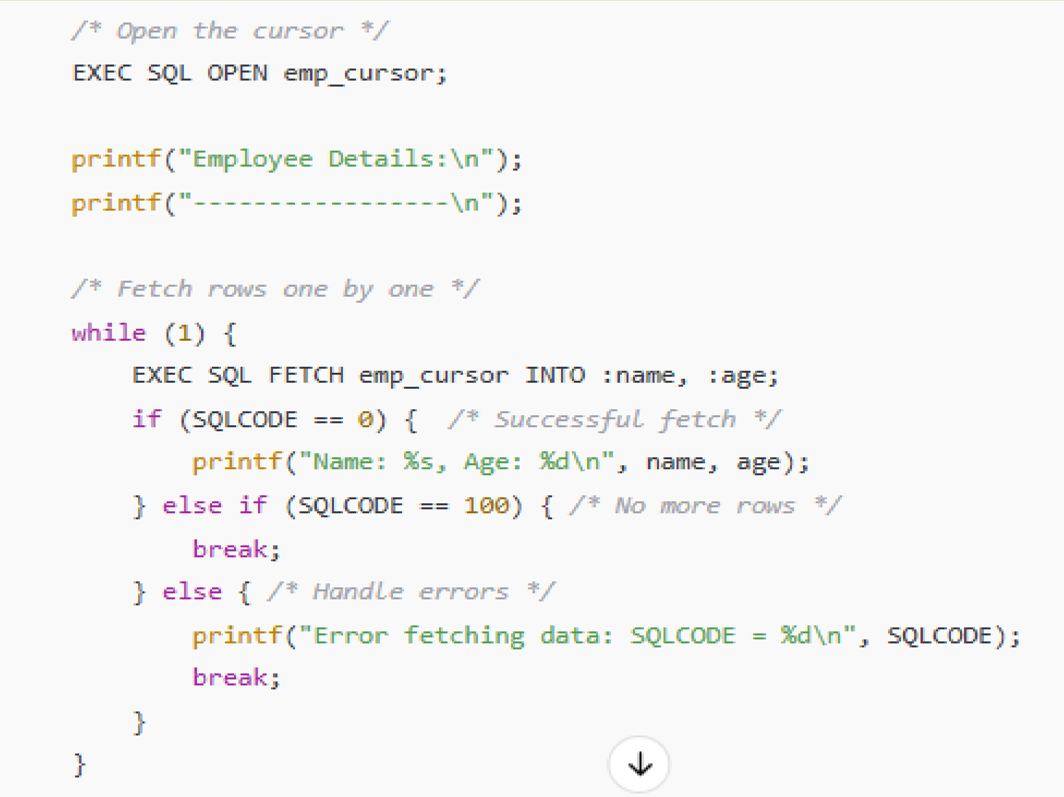
```
//Program Segment E1:
loop = 1 ;
while (loop) {
  prompt("Enter a Social Security Number: ", ssn) ;
  EXEC SQL
    SELECT Fname, Minit, Lname, Address, Salary
    INTO :fname, :minit, :lname, :address, :salary
    FROM EMPLOYEE WHERE Ssn = :ssn ;
  if (SQLCODE = = 0) printf(fname, minit, lname, address,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

10.2.2. Retrieving Multiple Tuples with Embedded SQL Using Cursors

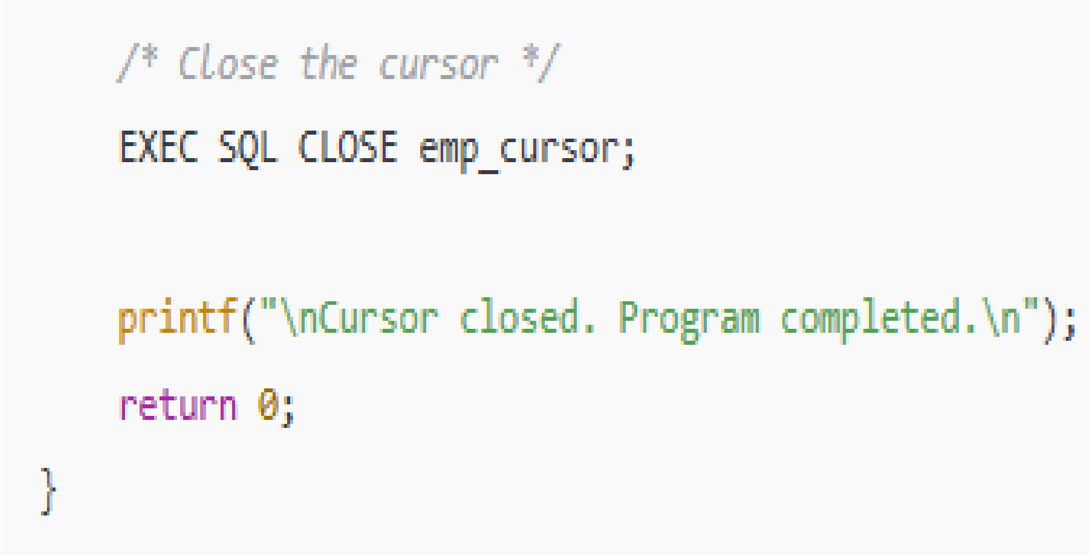
- Cursor
  - Points to a single tuple (row) from result of query
- **OPEN CURSOR** command
  - Fetches query result and sets cursor to a position before first row in result
  - Becomes current row for cursor
- **FETCH** commands
  - Moves cursor to next row in result of query



```
#include <stdio.h>
#include <string.h>

/* Embedded SQL declarations */
EXEC SQL BEGIN DECLARE SECTION;
    char name[50];
    int age;
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA; /* SQL Communications Area */

int main() {
    /* Declare the cursor for retrieving multiple rows */
    EXEC SQL DECLARE emp_cursor CURSOR FOR
        SELECT name, age FROM employees;
```



```
/* Open the cursor */
EXEC SQL OPEN emp_cursor;

printf("Employee Details:\n");
printf("-----------------\n");

/* Fetch rows one by one */
while (1) {
    EXEC SQL FETCH emp_cursor INTO :name, :age;
    if (SQLCODE == 0) {  /* Successful fetch */
        printf("Name: %s, Age: %d\n", name, age);
    } else if (SQLCODE == 100) { /* No more rows */
        break;
    } else { /* Handle errors */
        printf("Error fetching data: SQLCODE = %d\n", SQLCODE);
        break;
    }
}
```



```
    /* Close the cursor */
    EXEC SQL CLOSE emp_cursor;


    printf("\nCursor closed. Program completed.\n");
    return 0;
}
```

Explanation:

1. **Cursor Declaration**:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR SELECT name, age FROM employees;
```

   - A cursor named emp_cursor is declared for selecting name and age columns from the employees table.

2. **Opening the Cursor**:

```
EXEC SQL OPEN emp_cursor;
```

   - Opens the cursor and prepares to retrieve rows.

3. **Fetching Data**:

```
EXEC SQL FETCH emp_cursor INTO :name, :age;
```

   - Fetches one row at a time into host variables name and age.

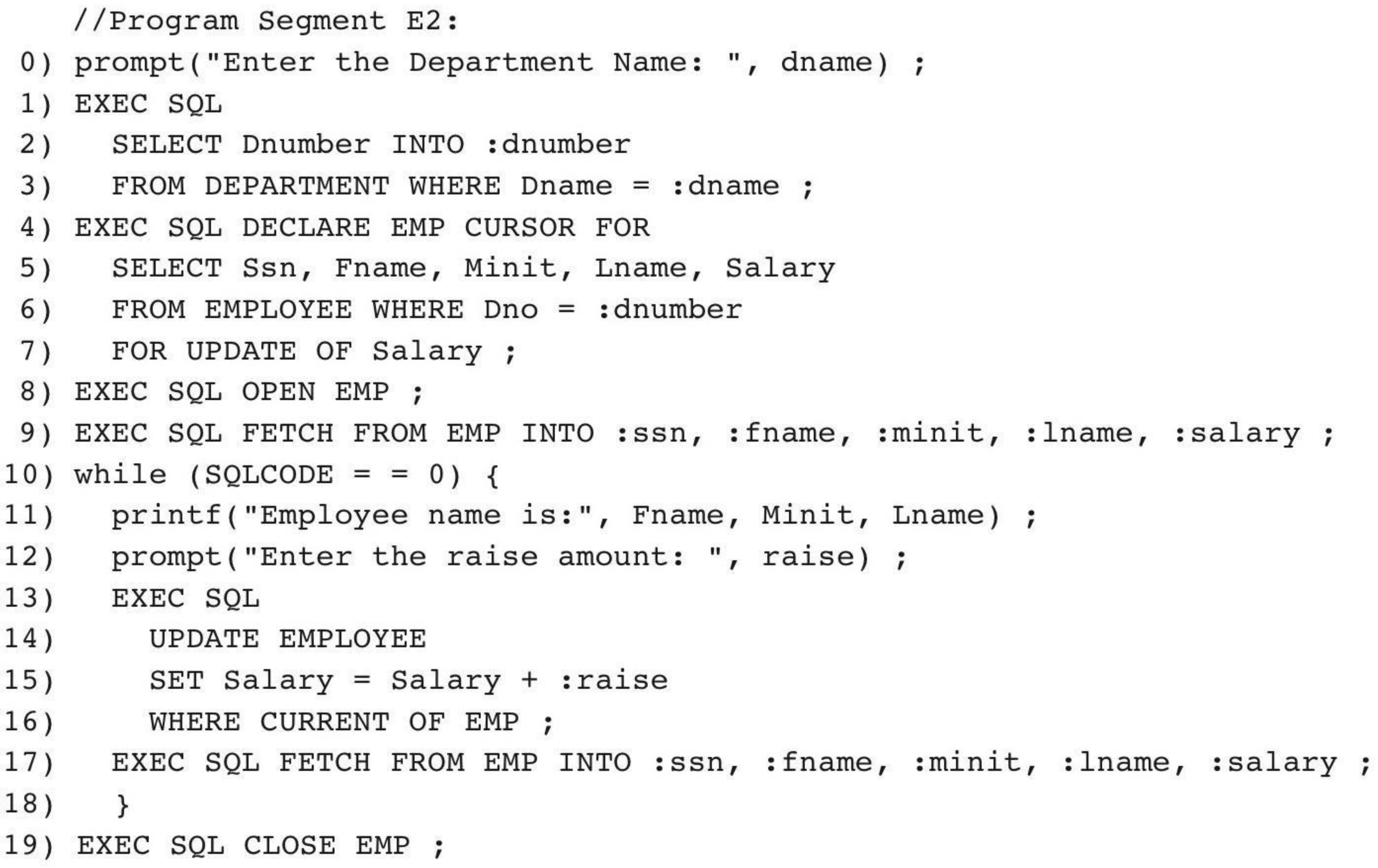
4. **SQLCODE Handling**:
   - **SQLCODE == 0**: Successful row fetch.
   - **SQLCODE == 100**: No more rows to fetch.
   - **Other SQLCODE values**: Handle errors.

5. **Closing the Cursor**:

```
EXEC SQL CLOSE emp_cursor;
```

   - Closes the cursor when done.

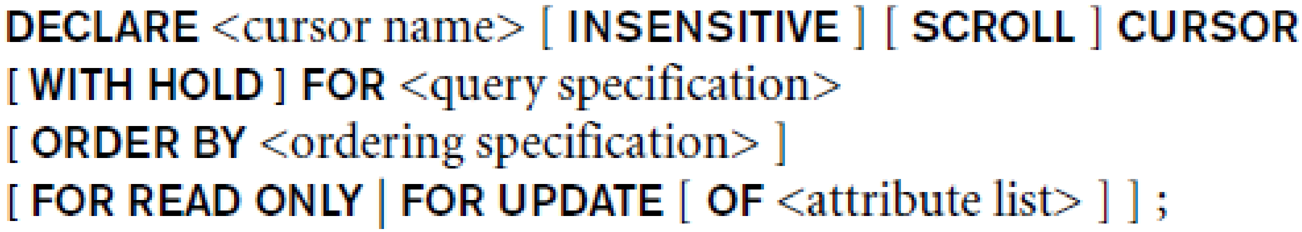
Figure 10.3 Program segment E2, a C program segment that uses cursors with embedded SQL for update purposes.



```
//Program Segment E2:
prompt("Enter the Department Name: ", dname) ;
EXEC SQL
  SELECT Dnumber INTO :dnumber
  FROM DEPARTMENT WHERE Dname = :dname ;
EXEC SQL DECLARE EMP CURSOR FOR
  SELECT Ssn, Fname, Minit, Lname, Salary
  FROM EMPLOYEE WHERE Dno = :dnumber
  FOR UPDATE OF Salary ;
EXEC SQL OPEN EMP ;
EXEC SQL FETCH FROM EMP INTO :ssn, :fname, :minit, :lname, :salary ;
while (SQLCODE = = 0) {
  printf("Employee name is:", Fname, Minit, Lname) ;
  prompt("Enter the raise amount: ", raise) ;
  EXEC SQL
    UPDATE EMPLOYEE
    SET Salary = Salary + :raise
    WHERE CURRENT OF EMP ;
  EXEC SQL FETCH FROM EMP INTO :ssn, :fname, :minit, :lname, :salary ;
  }
EXEC SQL CLOSE EMP ;
```

10.2.2. Retrieving Multiple Tuples with Embedded SQL Using Cursors (cont'd.)

- **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*

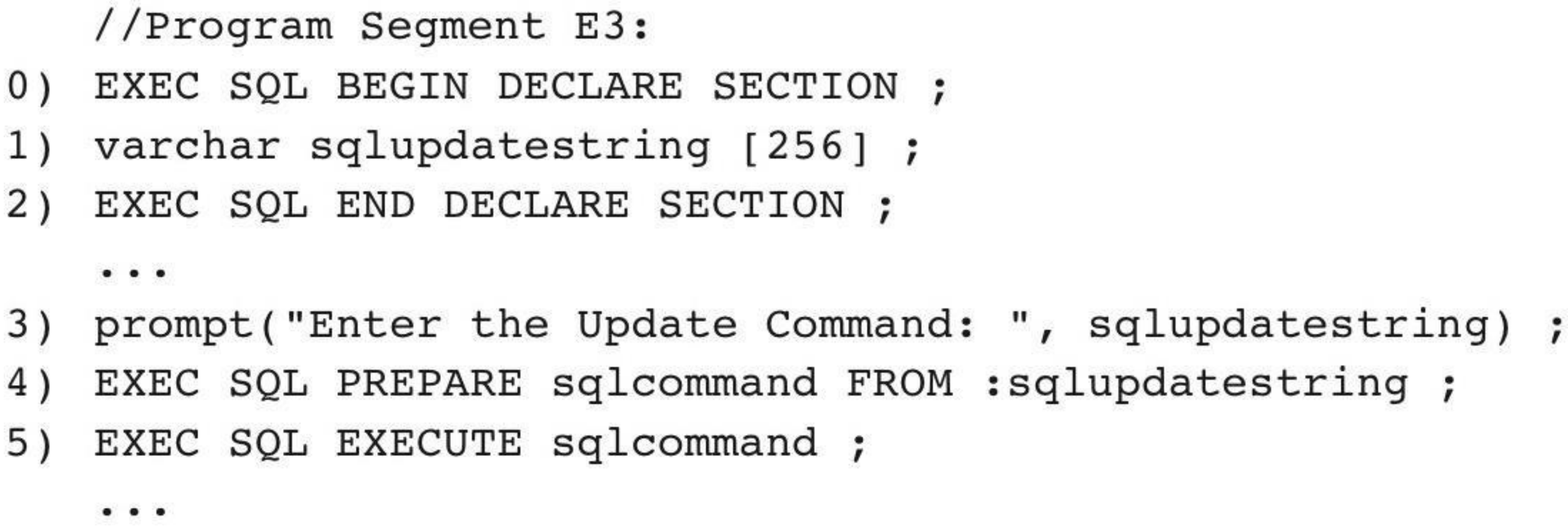


```
DECLARE <cursor name> [ INSENSITIVE ] [ SCROLL ] CURSOR
[ WITH HOLD ] FOR <query specification>
[ ORDER BY <ordering specification> ]
[ FOR READ ONLY | FOR UPDATE [ OF <attribute list> ] ] ;
```

10.2.3. Specifying Queries at Runtime Using Dynamic SQL

- **Dynamic SQL**
    - Execute different SQL queries or updates dynamically at runtime
- Dynamic update
- Dynamic query

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

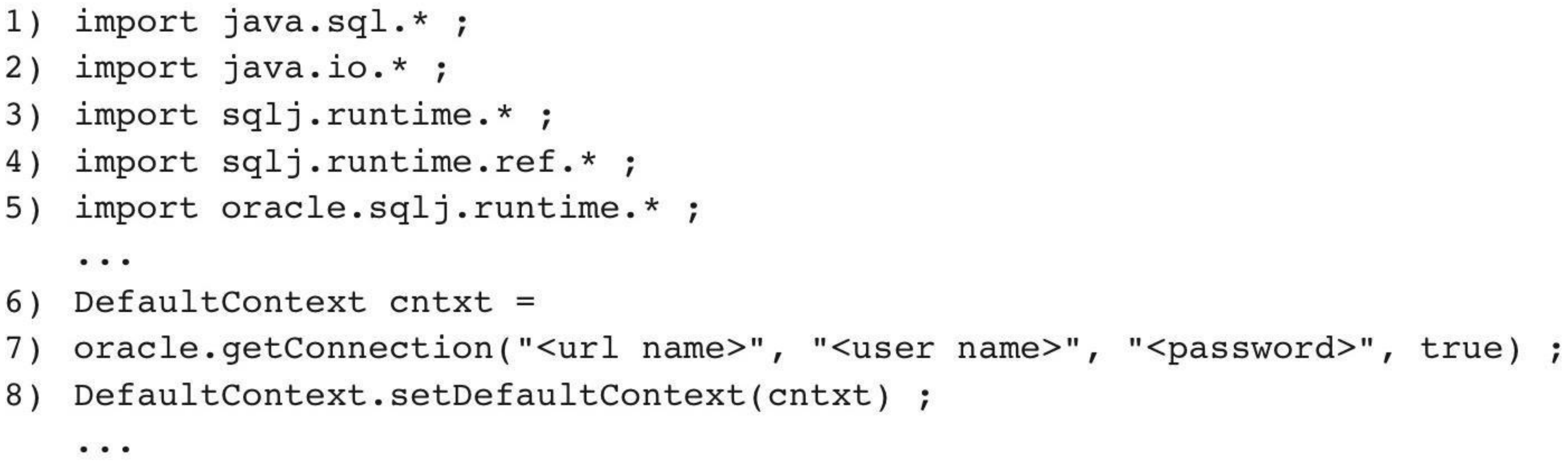


```
//Program Segment E3:
EXEC SQL BEGIN DECLARE SECTION ;
varchar sqlupdatestring [256] ;
EXEC SQL END DECLARE SECTION ;
...
prompt("Enter the Update Command: ", sqlupdatestring) ;
EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;
EXEC SQL EXECUTE sqlcommand ;
...
```

10.2.4. SQLJ: Embedding SQL Commands in Java

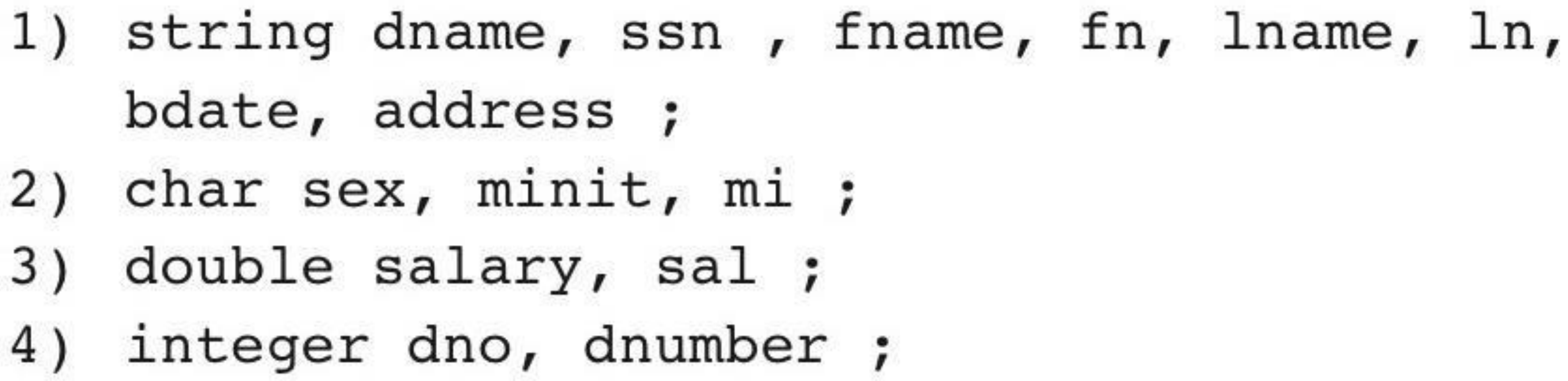
- Standard adopted by several vendors for embedding SQL in Java
- Import several class libraries
- **Default context**
- Uses **exceptions** for error handling
  - `SQLException` is used to return errors or exception conditions

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



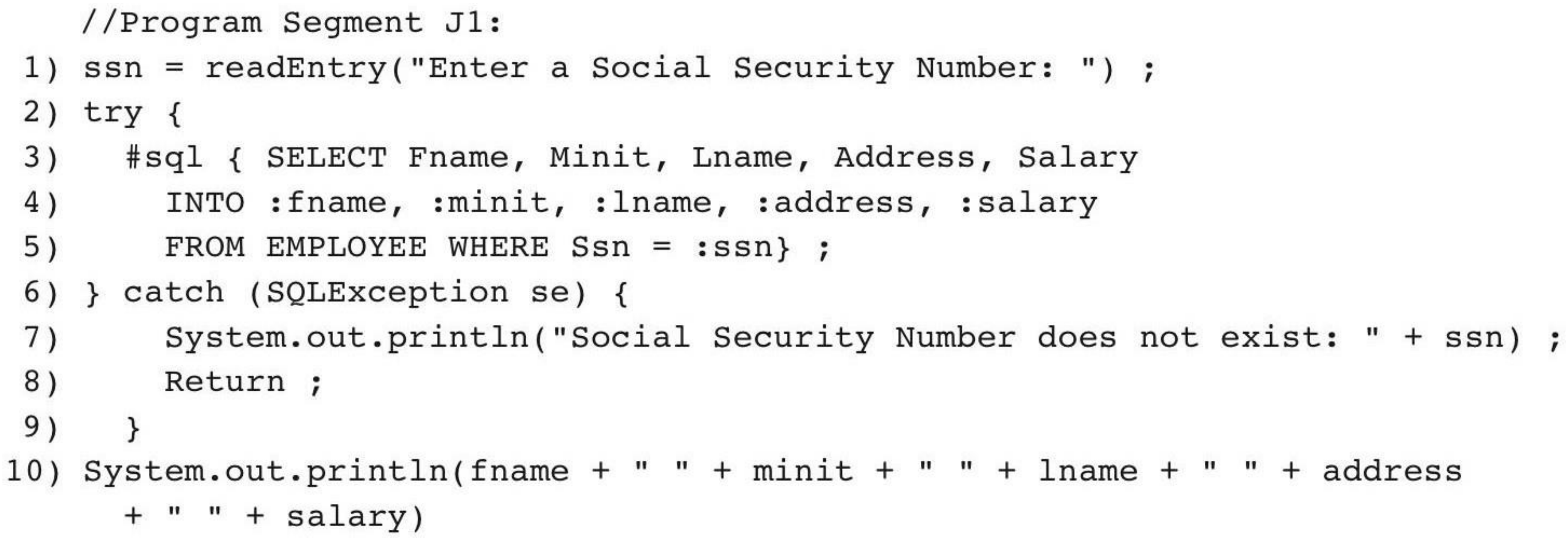
```
import java.sql.* ;
import java.io.* ;
import sqlj.runtime.* ;
import sqlj.runtime.ref.* ;
import oracle.sqlj.runtime.* ;
...
DefaultContext cntxt =
oracle.getConnection("<url name>", "<user name>", "<password>", true) ;
DefaultContext.setDefaultContext(cntxt) ;
...
```

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.

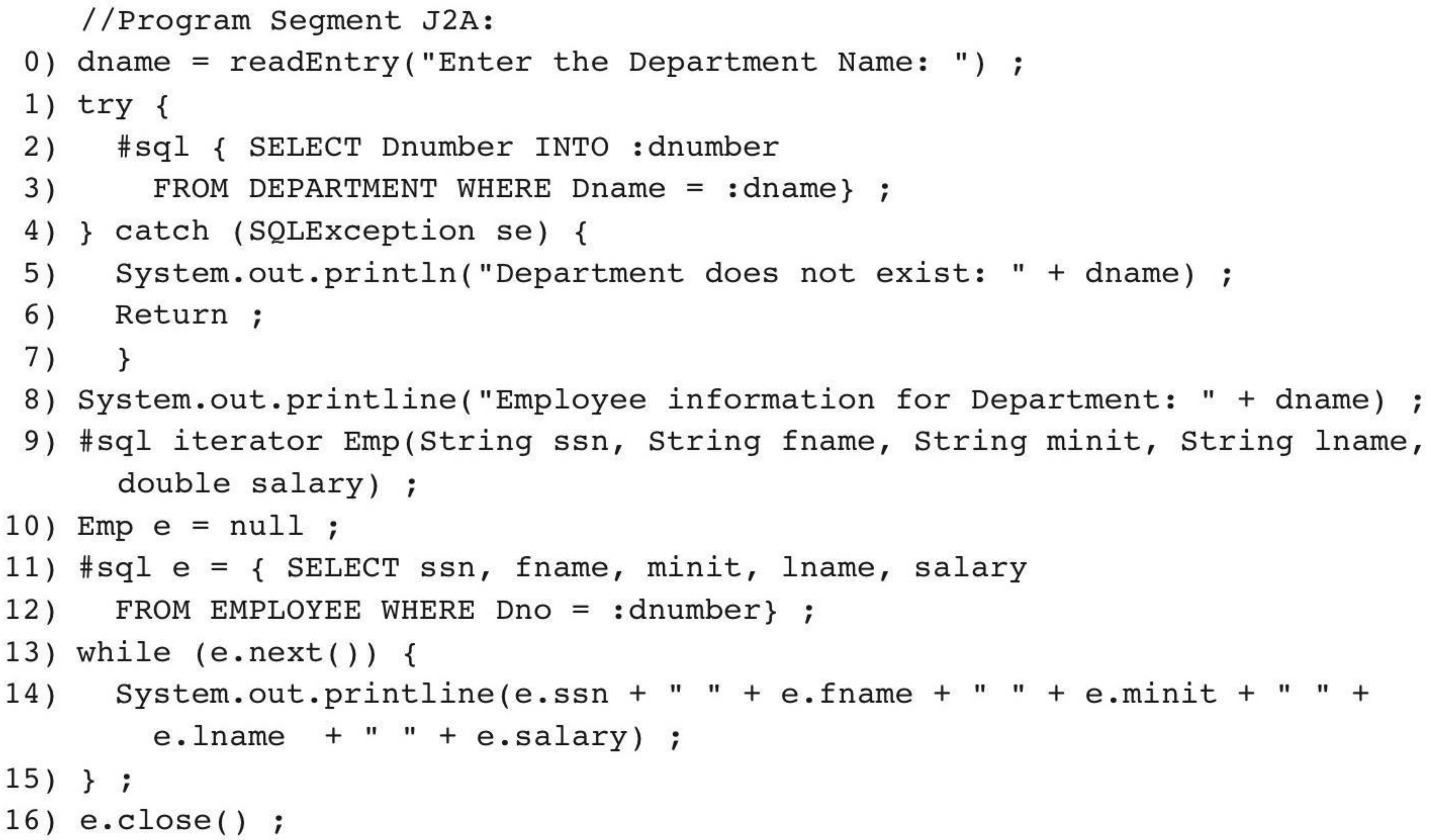


```
//Program Segment J1:
ssn = readEntry("Enter a Social Security Number: ") ;
try {
  #sql { SELECT Fname, Minit, Lname, Address, Salary
    INTO :fname, :minit, :lname, :address, :salary
    FROM EMPLOYEE WHERE Ssn = :ssn} ;
} catch (SQLException se) {
    System.out.println("Social Security Number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

10.2.5. Retrieving Multiple Tuples in SQLJ Using Iterators

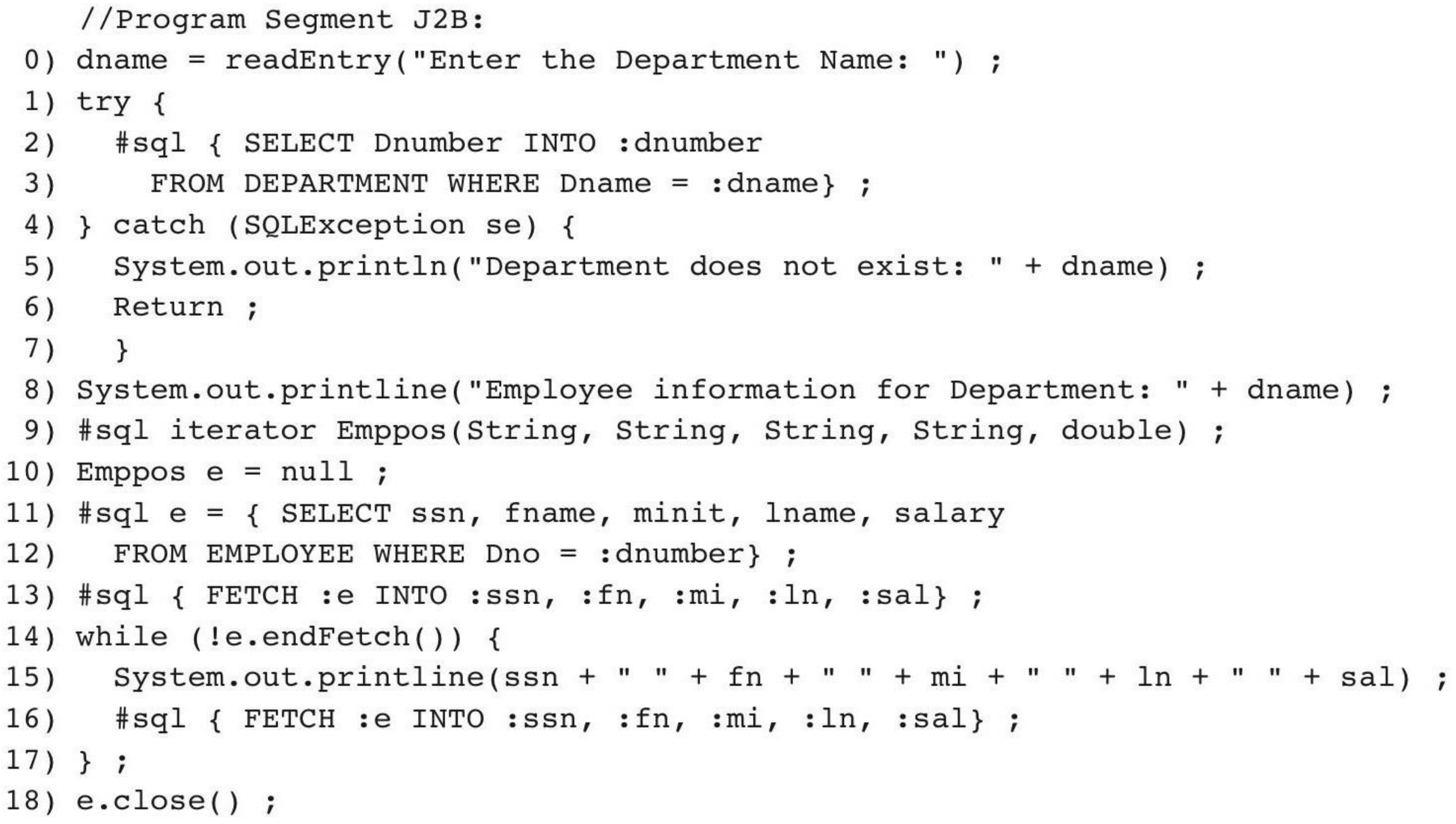
- **Iterator**
  - Object associated with a collection (set or multiset) of records in a query result
- **Named iterator**
  - Associated with a query result by listing attribute names and types in query result
- **Positional iterator**
  - Lists only attribute types in query result

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:
dname = readEntry("Enter the Department Name: ") ;
try {
  #sql { SELECT Dnumber INTO :dnumber
    FROM DEPARTMENT WHERE Dname = :dname} ;
} catch (SQLException se) {
  System.out.println("Department does not exist: " + dname) ;
  Return ;
  }
System.out.printline("Employee information for Department: " + dname) ;
#sql iterator Emp(String ssn, String fname, String minit, String lname,
  double salary) ;
Emp e = null ;
#sql e = { SELECT ssn, fname, minit, lname, salary
  FROM EMPLOYEE WHERE Dno = :dnumber} ;
while (e.next()) {
  System.out.printline(e.ssn + " " + e.fname + " " + e.minit + " " +
    e.lname  + " " + e.salary) ;
} ;
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:
dname = readEntry("Enter the Department Name: ") ;
try {
  #sql { SELECT Dnumber INTO :dnumber
    FROM DEPARTMENT WHERE Dname = :dname} ;
} catch (SQLException se) {
  System.out.println("Department does not exist: " + dname) ;
  Return ;
  }
System.out.printline("Employee information for Department: " + dname) ;
#sql iterator Emppos(String, String, String, String, double) ;
Emppos e = null ;
#sql e = { SELECT ssn, fname, minit, lname, salary
  FROM EMPLOYEE WHERE Dno = :dnumber} ;
#sql { FETCH :e INTO :ssn, :fn, :mi, :ln, :sal} ;
while (!e.endFetch()) {
  System.out.printline(ssn + " " + fn + " " + mi + " " + ln + " " + sal) ;
  #sql { FETCH :e INTO :ssn, :fn, :mi, :ln, :sal} ;
} ;
e.close() ;
```

10.3. 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

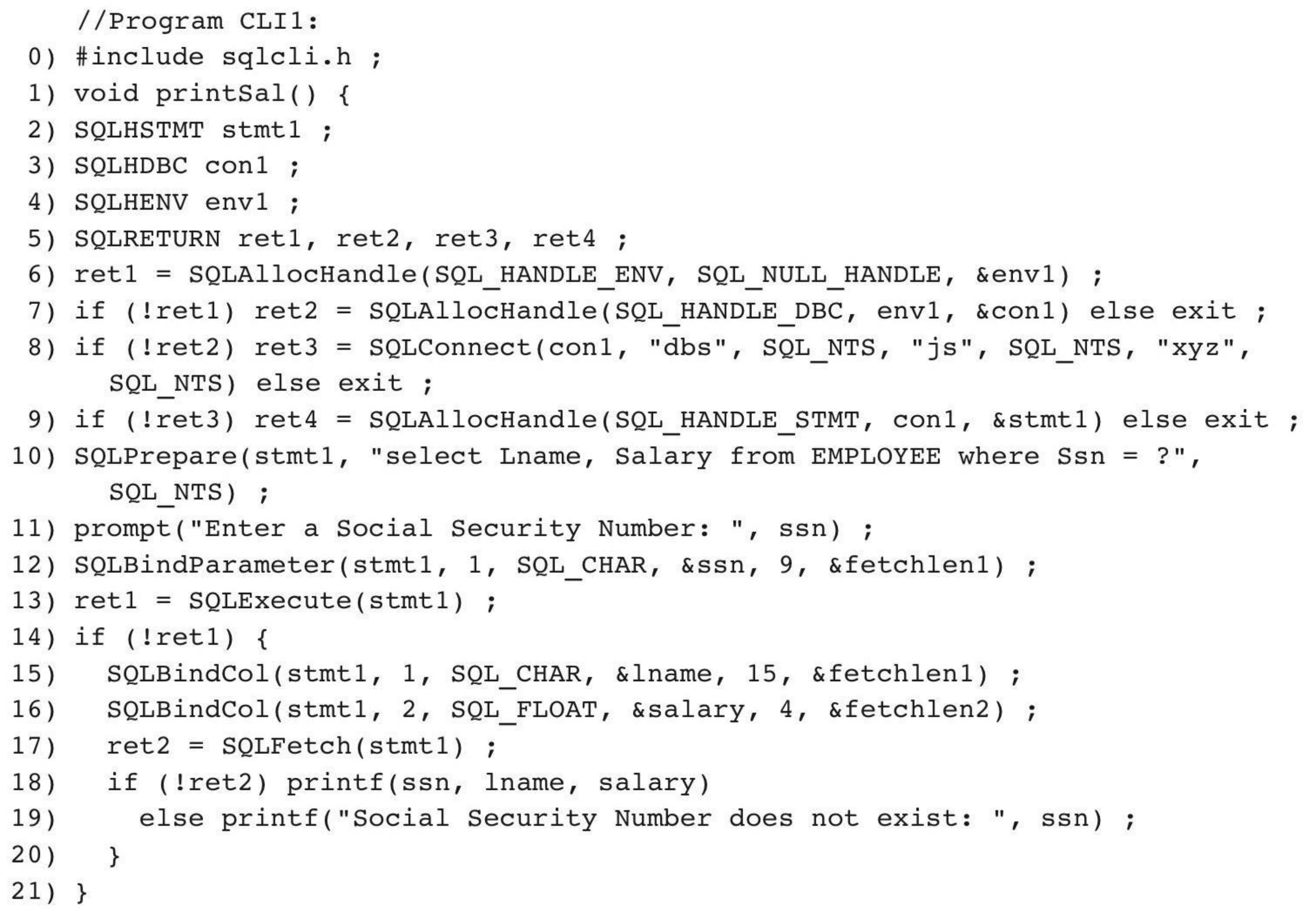
10.3.1. SQL/CLI: Using Cas the Host Language

- **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

10.3.1. SQL/CLI: Using C as the Host Language (cont'd.)

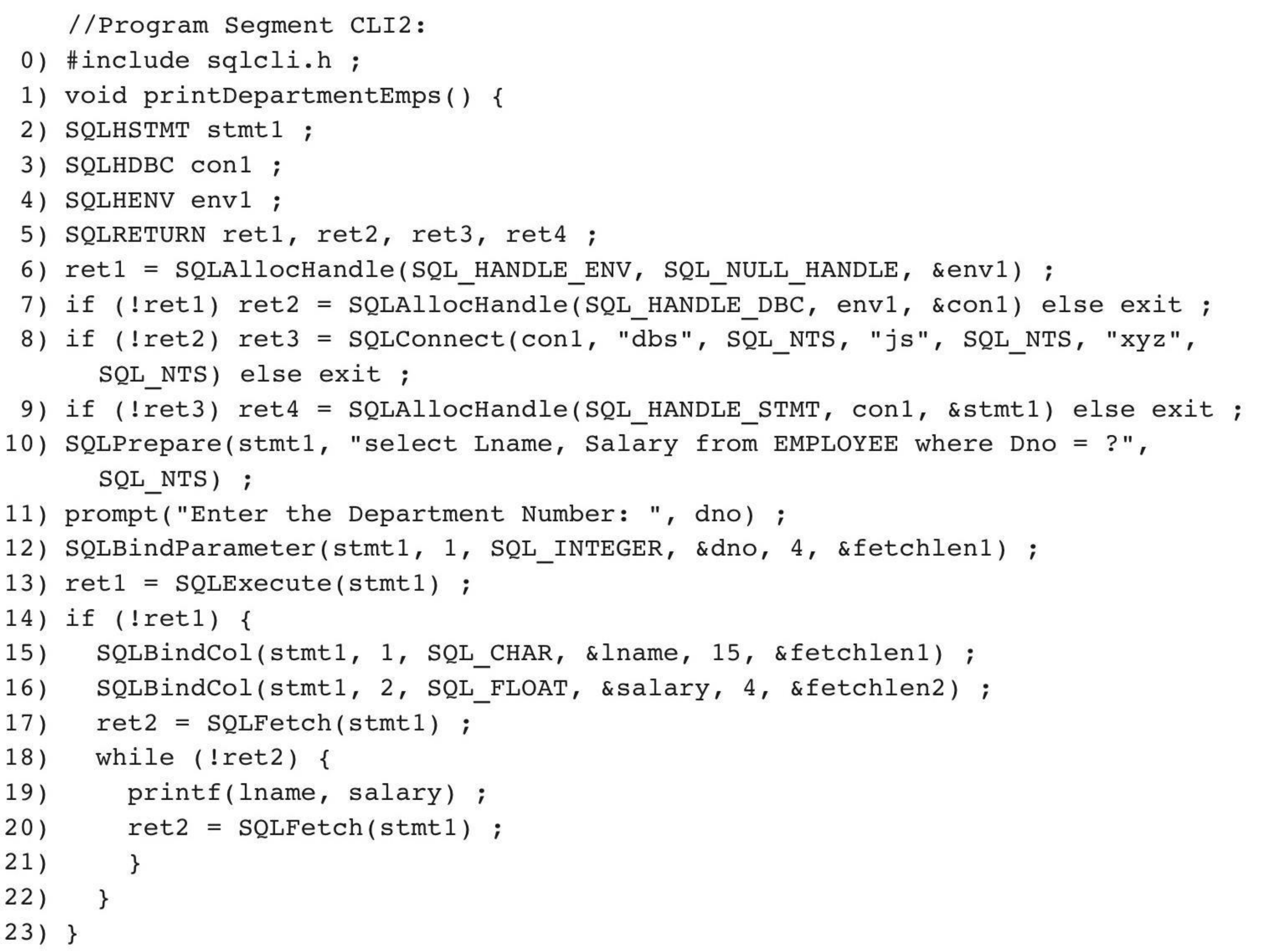
- **Description record**
  - Keeps track of information about tuples or parameters
- **Handle** to the record
  - C pointer variable makes record accessible to program

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



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

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:
#include sqlcli.h ;
void printDepartmentEmps() {
SQLHSTMT stmt1 ;
SQLHDBC con1 ;
SQLHENV env1 ;
SQLRETURN ret1, ret2, ret3, ret4 ;
ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz",
  SQL_NTS) else exit ;
if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
SQLPrepare(stmt1, "select Lname, Salary from EMPLOYEE where Dno = ?",
  SQL_NTS) ;
prompt("Enter the Department Number: ", dno) ;
SQLBindParameter(stmt1, 1, SQL_INTEGER, &dno, 4, &fetchlen1) ;
ret1 = SQLExecute(stmt1) ;
if (!ret1) {
  SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
  SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
  ret2 = SQLFetch(stmt1) ;
  while (!ret2) {
    printf(lname, salary) ;
    ret2 = SQLFetch(stmt1) ;
    }
  }
}
```

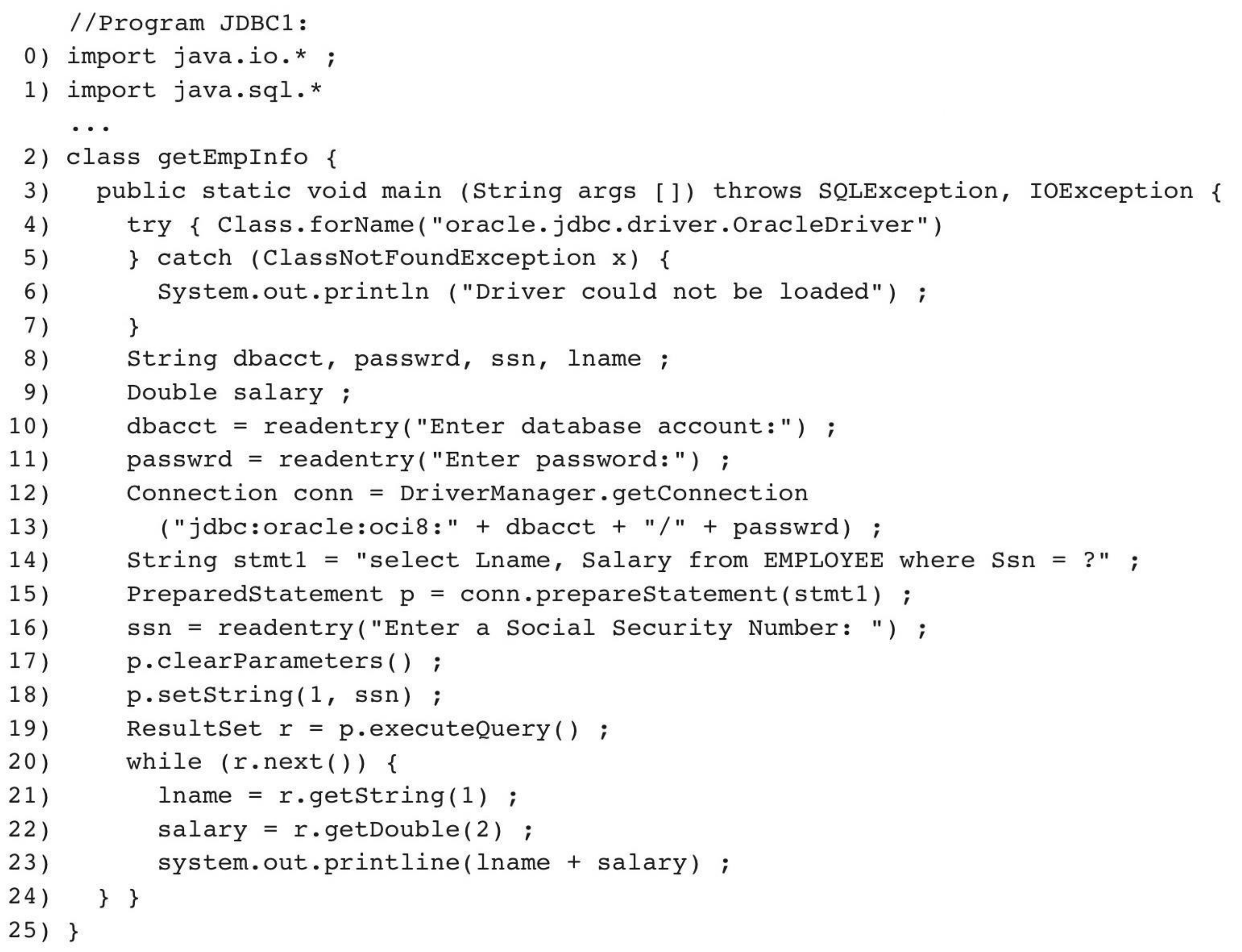
10.3.2. JDBC: SQL Function Calls for Java Programming

- 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

10.3.2. JDBC: SQL Function Calls for Java Programming

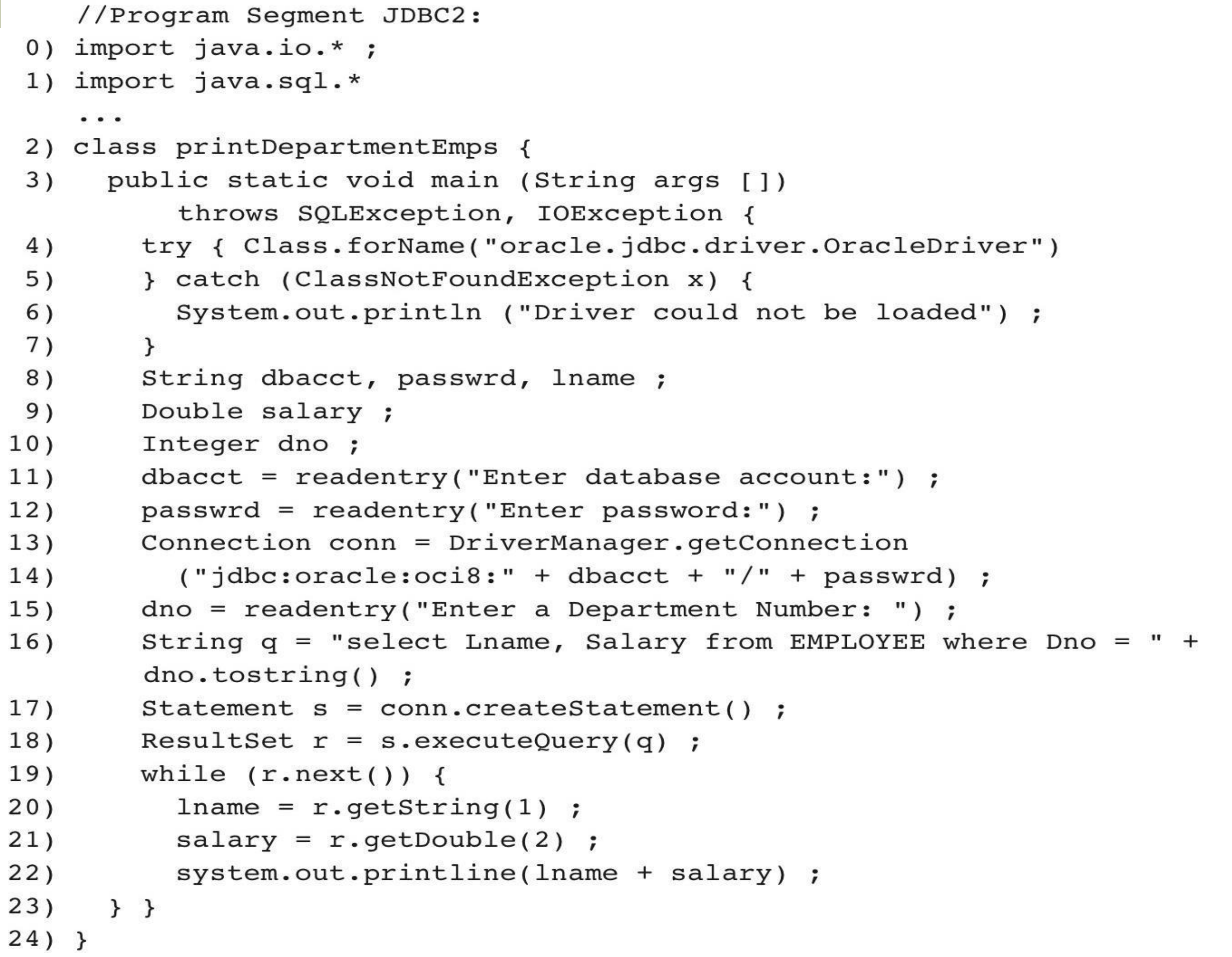
- **`Connection` object**
- **`Statement` object** has two subclasses:
  - `PreparedStatement` and `CallableStatement`
- Question mark (`?`) symbol
  - Represents a statement parameter
  - Determined at runtime
- **`ResultSet` object**
  - Holds results of query

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



```
//Program JDBC1:
import java.io.* ;
import java.sql.*
...
class getEmpInfo {
  public static void main (String args []) throws SQLException, IOException {
    try { Class.forName("oracle.jdbc.driver.OracleDriver")
    } catch (ClassNotFoundException x) {
      System.out.println ("Driver could not be loaded") ;
    }
    String dbacct, passwrd, ssn, lname ;
    Double salary ;
    dbacct = readentry("Enter database account:") ;
    passwrd = readentry("Enter password:") ;
    Connection conn = DriverManager.getConnection
      ("jdbc:oracle:oci8:" + dbacct + "/" + passwrd) ;
    String stmt1 = "select Lname, Salary from EMPLOYEE where Ssn = ?" ;
    PreparedStatement p = conn.prepareStatement(stmt1) ;
    ssn = readentry("Enter a Social Security Number: ") ;
    p.clearParameters() ;
    p.setString(1, ssn) ;
    ResultSet r = p.executeQuery() ;
    while (r.next()) {
      lname = r.getString(1) ;
      salary = r.getDouble(2) ;
      system.out.printline(lname + salary) ;
  } }
}
```

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:
import java.io.* ;
import java.sql.*
...
class printDepartmentEmps {
 public static void main (String args [])
   throws SQLException, IOException {
  try { Class.forName("oracle.jdbc.driver.OracleDriver")
  } catch (ClassNotFoundException x) {
   System.out.println ("Driver could not be loaded") ;
  }
  String dbacct, passwrd, lname ;
  Double salary ;
  Integer dno ;
  dbacct = readentry("Enter database account:") ;
  passwrd = readentry("Enter password:") ;
  Connection conn = DriverManager.getConnection
   ("jdbc:oracle:oci8:" + dbacct + "/" + passwrd) ;
  dno = readentry("Enter a Department Number: ") ;
  String q = "select Lname, Salary from EMPLOYEE where Dno = " +
  dno.tostring() ;
  Statement s = conn.createStatement() ;
  ResultSet r = s.executeQuery(q) ;
  while (r.next()) {
   lname = r.getString(1) ;
   salary = r.getDouble(2) ;
   system.out.printline(lname + salary) ;
 } }
}
```

10.4. Database Stored Procedures and SQL/PSM (More Discussion of Oracle PL/SQL in Part B)

- **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

10.4. Database Stored Procedures and SQL/PSM

- **Persistent stored modules**
  - Stored persistently by the DBMS
- 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

10.4. Database Stored Procedures and SQL/PSM

- Declaring stored procedures:

```
CREATE PROCEDURE <procedure name> (<parameters>)
<local declarations>
<procedure body> ;
declaring a function, a return type is necessary,
  so the declaration form is
CREATE FUNCTION <function name> (<parameters>)
RETURNS <return type>
<local declarations>
<function body> ;
```

10.4. Database Stored Procedures and SQL/PSM

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

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

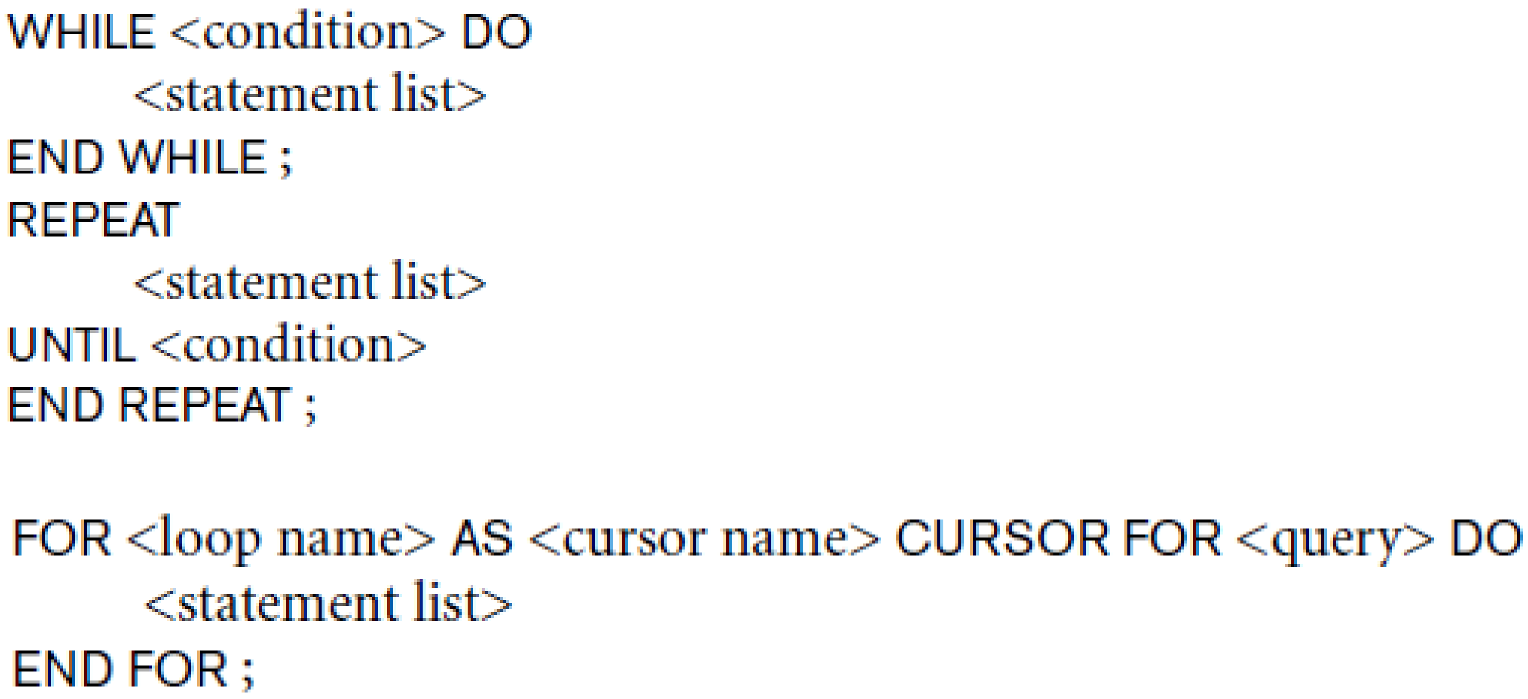
SQL/PSM: Extending SQL for Specifying Persistent Stored Modules

- Conditional branching statement:

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

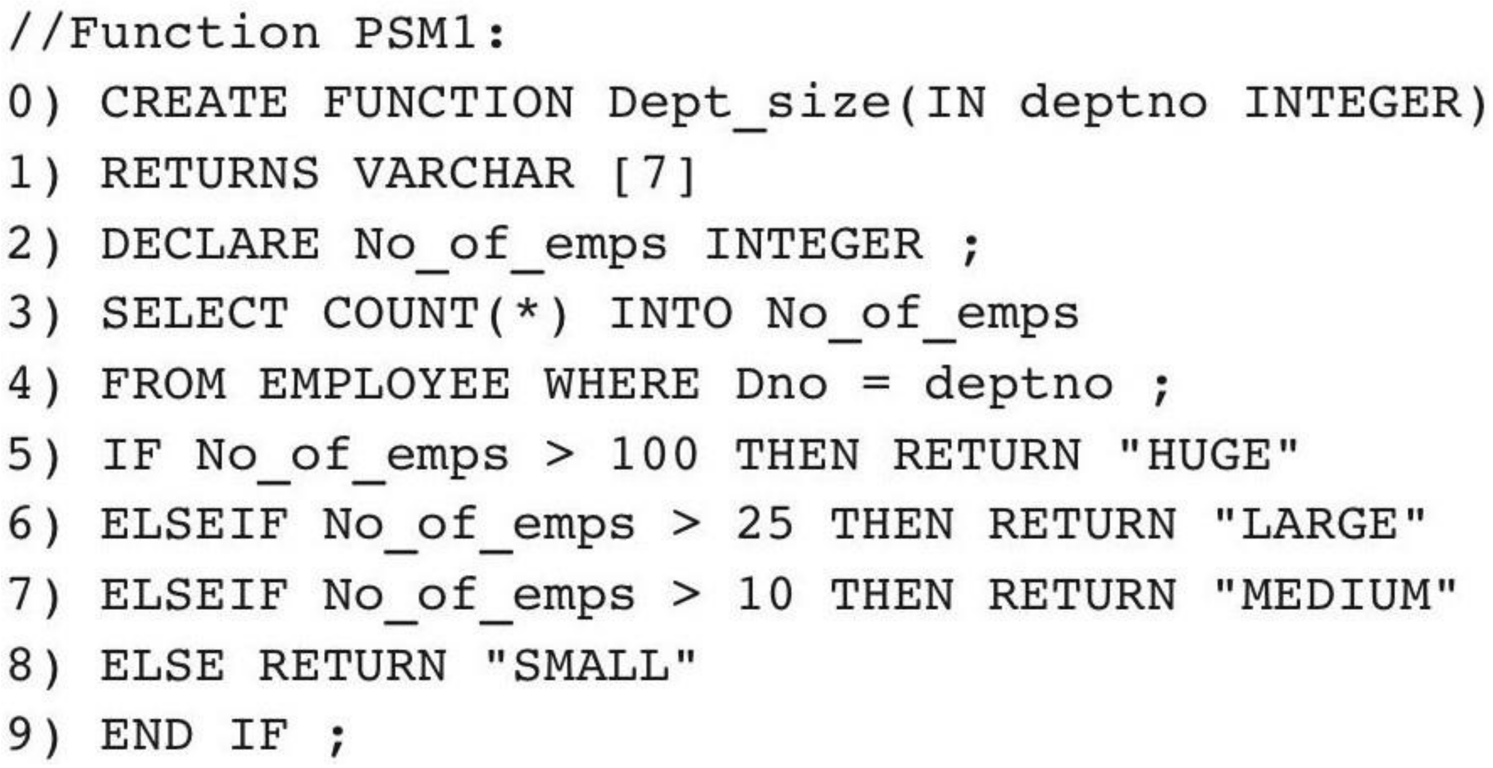
SQL/PSM (cont'd.)

- Constructs for looping



```
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 ;
```

Figure 10.14 Declaring a function in SQL/PSM.



```
//Function PSM1:
CREATE FUNCTION Dept_size(IN deptno INTEGER)
RETURNS VARCHAR [7]
DECLARE No_of_emps INTEGER ;
SELECT COUNT(*) INTO No_of_emps
FROM EMPLOYEE WHERE Dno = deptno ;
IF No_of_emps > 100 THEN RETURN "HUGE"
ELSEIF No_of_emps > 25 THEN RETURN "LARGE"
ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM"
ELSE RETURN "SMALL"
END IF ;
```

Comparing the Three Approaches

- Embedded SQL Approach
  - Query text checked for syntax errors and validated against database schema at compile time
  - For complex applications where queries have to be generated at runtime
    - Function call approach more suitable

Comparing the Three Approaches (cont'd.)

- Library of Function Calls Approach
  - More flexibility
  - More complex programming
  - No checking of syntax done at compile time
- Database Programming Language Approach
  - Does not suffer from the impedance mismatch problem
  - Programmers must learn a new language

Summary

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