part 5

Database Programming Techniques

This page intentionally left blank

chapter 13

Introduction to SQL Programming Techniques

In Chapters 4 and 5, we described several aspects of the SQL language, which is the standard for relational databases. We described the SQL statements for data definition, schema modification, queries, views, and updates. We also described how various constraints on the database contents, such as key and referential integrity constraints, are specified.

In this chapter and the next, we discuss some of the methods that have been developed for accessing databases from programs. 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, or some other programming language. In addition, many scripting languages, such as PHP and JavaScript, are also being used for programming of database access within Web applications. In this chapter, we focus on how databases can be accessed from the traditional programming languages C/C++ and Java, whereas in the next chapter we introduce how databases are accessed from scripting languages such as PHP and JavaScript. Recall from Section 2.3.1 that when database statements are included in a program, the general-purpose programming language is called the *host language*, whereas the database language—SQL, in our case—is called the *data sublanguage*. In some cases, special *database programming languages* are developed specifically for writing database applications. Although many of these were developed as research prototypes, some notable database programming languages have widespread use, such as Oracle's PL/SQL (Programming Language/SQL).

It is important to note that database programming is a very broad topic. There are whole textbooks devoted to each database programming technique and how that technique is realized in a specific system. New techniques are developed all the time, and changes to existing techniques are incorporated into newer system versions and languages. An additional difficulty in presenting this topic is that although there are SQL standards, these standards themselves are continually evolving, and each DBMS vendor may have some variations from the standard. Because of this, we have chosen to give an introduction to some of the main types of database programming techniques and to compare these techniques, rather than study one particular method or system in detail. The examples we give serve to illustrate the main differences that a programmer would face when using each of these database programming techniques. We will try to use the SQL standards in our examples rather than describe a specific system. When using a specific system, the materials in this chapter can serve as an introduction, but should be augmented with the system manuals or with books describing the specific system.

We start our presentation of database programming in Section 13.1 with an overview of the different techniques developed for accessing a database from programs. Then, in Section 13.2, we discuss the rules for embedding SQL statements into a general-purpose programming language, generally known as *embedded SQL*. This section also briefly discusses *dynamic SQL*, in which queries can be dynamically constructed at runtime, and presents the basics of the SQLJ variation of embedded SQL that was developed specifically for the programming language Java. In Section 13.3, we discuss the technique known as *SQL/CLI* (Call Level Interface), in which a library of procedures and functions is provided for accessing the database. Various sets of library functions have been proposed. The SQL/CLI set of functions is the one given in the SQL standard. Another library of functions is *ODBC* (Open Data Base Connectivity). We do not describe ODBC because it is considered to be the predecessor to SQL/CLI. A third library of functions—which we do describe—is *JDBC*; this was developed specifically for accessing databases from Java. In Section 13.4 we discuss *SQL/PSM* (Persistent Stored Modules), which is a part of the SQL standard that allows program modules—procedures and functions—to be stored by the DBMS and accessed through SQL. We briefly compare the three approaches to database programming in Section 13.5, and provide a chapter summary in Section 13.6.

13.1 Database Programming: Techniques and Issues

We now turn our attention to the techniques that have been developed for accessing databases from programs and, in particular, to the issue of how to access SQL databases from application programs. Our presentation of SQL in Chapters 4 and 5 focused on the language constructs for various database operations—from schema definition and constraint specification to querying, updating, and specifying views. Most database systems have an **interactive interface** where these SQL commands can be typed directly into a monitor for execution by the database system. For example, in a computer system where the Oracle RDBMS is installed, the command SQLPLUS starts the interactive interface. The user can type SQL commands or queries directly over several lines, ended by a semicolon and the Enter key (that is,

"`; <cr>"`). Alternatively, a **file of commands** can be created and executed through the interactive interface by typing `@`<*filename*>. The system will execute the commands written in the file and display the results, if any.

The interactive interface is quite convenient for schema and constraint creation or for occasional ad hoc queries. However, in practice, the majority of database interactions are executed through programs that have been carefully designed and tested. These programs are generally known as **application programs** or **database applications**, and are used as *canned transactions* by the end users, as discussed in Section 1.4.3. Another common use of database programming is to access a database through an application program that implements a **Web interface**, for example, when making airline reservations or online purchases. In fact, the vast majority of Web electronic commerce applications include some database access commands. Chapter 14 gives an overview of Web database programming using PHP, a scripting language that has recently become widely used.

In this section, first we give an overview of the main approaches to database programming. Then we discuss some of the problems that occur when trying to access a database from a general-purpose programming language, and the typical sequence of commands for interacting with a database from a software program.

13.1.1 Approaches to Database Programming

Several techniques exist for including database interactions in application programs. The main approaches for database programming are the following:

1. **Embedding database commands in a general-purpose programming language.** In this approach, database statements are **embedded** into the host programming language, but they are identified by a special prefix. For example, the prefix for embedded SQL is the string EXEC SQL, which precedes all SQL commands in a host language program.[1] A **precompiler** or **preproccessor** scans the source program code to identify database statements and extract them for processing by the DBMS. They are replaced in the program by function calls to the DBMS-generated code. This technique is generally referred to as **embedded SQL**.
2. **Using a library of database functions.** A **library of functions** is made available to the host programming language for database calls. For example, there could be functions to connect to a database, execute a query, execute an update, and so on. The actual database query and update commands and any other necessary information are included as parameters in the function calls. This approach provides what is known as an **application programming interface** (**API**) for accessing a database from application programs.
3. **Designing a brand-new language.** A **database programming language** is designed from scratch to be compatible with the database model and query language. Additional programming structures such as loops and conditional

[1]Other prefixes are sometimes used, but this is the most common.

statements are added to the database language to convert it into a full-fledged programming language. An example of this approach is Oracle's PL/SQL.

In practice, the first two approaches are more common, since many applications are already written in general-purpose programming languages but require some database access. The third approach is more appropriate for applications that have intensive database interaction. One of the main problems with the first two approaches is *impedance mismatch*, which does not occur in the third approach.

13.1.2 Impedance Mismatch

Impedance mismatch is the term used to refer to the problems that occur because of differences between the database model and the programming language model. For example, the practical relational model has three main constructs: columns (attributes) and their data types, rows (also referred to as tuples or records), and tables (sets or multisets of records). The first problem that may occur is that the *data types of the programming language* differ from the *attribute data types* that are available in the data model. Hence, it is necessary to have a **binding** for each host programming language that specifies for each attribute type the compatible programming language types. A different binding is needed *for each programming language* because different languages have different data types. For example, the data types available in C/C++ and Java are different, and both differ from the SQL data types, which are the standard data types for relational databases.

Another problem occurs because the results of most queries are sets or multisets of tuples (rows), and each tuple is formed of a sequence of attribute values. In the program, it is often necessary to access the individual data values within individual tuples for printing or processing. Hence, a binding is needed to map the *query result data structure*, which is a table, to an appropriate data structure in the programming language. A mechanism is needed to loop over the tuples in a **query result** in order to access a single tuple at a time and to extract individual values from the tuple. The extracted attribute values are typically copied to appropriate program variables for further processing by the program. A **cursor** or **iterator variable** is typically used to loop over the tuples in a query result. Individual values within each tuple are then extracted into distinct program variables of the appropriate type.

Impedance mismatch is less of a problem when a special database programming language is designed that uses the same data model and data types as the database model. One example of such a language is Oracle's PL/SQL. The SQL standard also has a proposal for such a database programming language, known as *SQL/PSM*. For object databases, the object data model (see Chapter 11) is quite similar to the data model of the Java programming language, so the impedance mismatch is greatly reduced when Java is used as the host language for accessing a Java-compatible object database. Several database programming languages have been implemented as research prototypes (see the Selected Bibliography).

13.1.3 Typical Sequence of Interaction in Database Programming

When a programmer or software engineer writes a program that requires access to a database, it is quite common for the program to be running on one computer system while the database is installed on another. Recall from Section 2.5 that a common architecture for database access is the client/server model, where a **client program** handles the logic of a software application, but includes some calls to one or more **database servers** to access or update the data.[2] When writing such a program, a common sequence of interaction is the following:

1. When the client program requires access to a particular database, the program must first *establish* or *open* a **connection** to the database server. Typically, this involves specifying the Internet address (URL) of the machine where the database server is located, plus providing a login account name and password for database access.
2. Once the connection is established, the program can interact with the database by submitting queries, updates, and other database commands. In general, most types of SQL statements can be included in an application program.
3. When the program no longer needs access to a particular database, it should *terminate* or *close* the connection to the database.

A program can access multiple databases if needed. In some database programming approaches, only one connection can be active at a time, whereas in other approaches multiple connections can be established simultaneously.

In the next three sections, we discuss examples of each of the three main approaches to database programming. Section 13.2 describes how SQL is *embedded* into a programming language. Section 13.3 discusses how *function calls* are used to access the database, and Section 13.4 discusses an extension to SQL called SQL/PSM that allows *general-purpose programming constructs* for defining modules (procedures and functions) that are stored within the database system.[3] Section 13.5 compares these approaches.

13.2 Embedded SQL, Dynamic SQL, and SQLJ

In this section, we give an overview of the technique for how SQL statements can be embedded in a general-purpose programming language. We focus on two languages: C and Java. The examples used with the C language, known as **embedded**

[2]As we discussed in Section 2.5, there are two-tier and three-tier architectures; to keep our discussion simple, we will assume a two-tier client/server architecture here.

[3]SQL/PSM illustrates how typical general-purpose programming language constructs—such as loops and conditional structures—can be incorporated into SQL.

SQL, are presented in Sections 13.2.1 through 13.2.3, and can be adapted to other programming languages. The examples using Java, known as **SQLJ**, are presented in Sections 13.2.4 and 13.2.5. In this 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.

13.2.1 Retrieving Single Tuples with Embedded SQL

To illustrate the concepts of embedded SQL, we will use C as the host programming language.[4] When using C as the host language, an embedded SQL statement is distinguished from programming language statements by prefixing it with the keywords EXEC SQL so that a **preprocessor** (or **precompiler**) can separate embedded SQL statements from the host language code. The SQL statements within a program are terminated by a matching END-EXEC or by a semicolon (;). Similar rules apply to embedding SQL in other programming languages.

Within an embedded SQL command, we may 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 are prefixed by a colon (:) *when they appear in an 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. Names of database schema constructs—such as attributes and relations—can only be used within the SQL commands, but shared program variables can be used elsewhere in the C program without the colon (:) prefix.

Suppose that we want to write C programs to process the COMPANY database in Figure 3.5. We need to declare program variables to match the types of the database attributes that the program will process. The programmer can choose the names of the program variables; they may or may not have names that are identical to their corresponding database attributes. We will use the C program variables declared in Figure 13.1 for all our examples and show C program segments without variable declarations. Shared variables are declared within a declare section in the program, as shown in Figure 13.1 (lines 1 through 7).[5] A few of the common bindings of C types to SQL types are as follows. The SQL types INTEGER, SMALLINT, REAL, and DOUBLE are mapped to the C types `long`, `short`, `float`, and `double`, respectively. Fixed-length and varying-length strings (CHAR[*i*], VARCHAR[*i*]) in SQL can be mapped to arrays of characters (`char [i+1], varchar [i+1]`) in C that are one character longer than the SQL type because strings in C are terminated by a NULL

[4]Our discussion here also applies to the C++ programming language, since we do not use any of the object-oriented features, but focus on the database programming mechanism.

[5]We use line numbers in our code segments for easy reference; these numbers are not part of the actual code.

```
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 ;
```

Figure 13.1
C program variables used in the embedded SQL examples E1 and E2.

character (`\0`), which is not part of the character string itself.[6] Although `varchar` is not a standard C data type, it is permitted when C is used for SQL database programming.

Notice that the only embedded SQL commands in Figure 13.1 are lines 1 and 7, which tell the precompiler to take note of the C variable names between BEGIN DECLARE and END DECLARE because they can be included in embedded SQL statements—as long as they are preceded by a colon (:). Lines 2 through 5 are regular C program declarations. The C program variables declared in lines 2 through 5 correspond to the attributes of the EMPLOYEE and DEPARTMENT tables from the COMPANY database in Figure 3.5 that was declared by the SQL DDL in Figure 4.1. The variables declared in line 6—SQLCODE and SQLSTATE—are used to communicate errors and exception conditions between the database system and the executing program. Line 0 shows a program variable loop that will not be used in any embedded SQL statement, so it is declared outside the SQL declare section.

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> ;

In the examples in this chapter, we assume that the appropriate connection has already been established to the COMPANY database, and that it is the currently active connection.

[6]SQL strings can also be mapped to `char*` types in C.

Communicating between the Program and the DBMS Using SQLCODE and SQLSTATE. The two special **communication variables** that are used by the DBMS to communicate exception or error conditions to the program are SQLCODE and SQLSTATE. The **SQLCODE** variable shown in Figure 13.1 is an integer variable. After each database command is executed, the DBMS returns a value in SQLCODE. A value of 0 indicates that the statement was executed successfully by the DBMS. If SQLCODE > 0 (or, more specifically, if SQLCODE = 100), this indicates that no more data (records) are available in a query result. If SQLCODE < 0, this indicates some error has occurred. In some systems—for example, in the Oracle RDBMS—SQLCODE is a field in a record structure called SQLCA (SQL communication area), so it is referenced as SQLCA.SQLCODE. In this case, the definition of SQLCA must be included in the C program by including the following line:

```
EXEC SQL include SQLCA ;
```

In later versions of the SQL standard, a communication variable called **SQLSTATE** was added, which is a string of five characters. A value of '00000' in SQLSTATE indicates no error or exception; other values indicate various errors or exceptions. For example, '02000' indicates 'no more data' when using SQLSTATE. Currently, both SQLSTATE and SQLCODE are available in the SQL standard. Many of the error and exception codes returned in SQLSTATE are supposed to be standardized for all SQL vendors and platforms,[7] whereas the codes returned in SQLCODE are not standardized but are defined by the DBMS vendor. Hence, it is generally better to use SQLSTATE because this makes error handling in the application programs independent of a particular DBMS. As an exercise, the reader should rewrite the examples given later in this chapter using SQLSTATE instead of SQLCODE.

Example of Embedded SQL Programming. Our first example to illustrate embedded SQL programming is a repeating program segment (loop) that takes as input a Social Security number of an employee and prints some information from the corresponding EMPLOYEE record in the database. The C program code is shown as program segment E1 in Figure 13.2. The program reads (inputs) an Ssn value and then retrieves the EMPLOYEE tuple with that Ssn from the database via the embedded SQL command. The **INTO** clause (line 5) specifies the program variables into which attribute values from the database record are retrieved. C program variables in the INTO clause are prefixed with a colon (:), as we discussed earlier. The INTO clause can be used in this way only when the query result is a single record; if multiple records are retrieved, an error will be generated. We will see how multiple records are handled in Section 13.2.2.

Line 7 in E1 illustrates the communication between the database and the program through the special variable SQLCODE. If the value returned by the DBMS in SQLCODE is 0, the previous statement was executed without errors or exception conditions. Line 7 checks this and assumes that if an error occurred, it was because

[7]In particular, SQLSTATE codes starting with the characters 0 through 4 or A through H are supposed to be standardized, whereas other values can be implementation-defined.

```
    //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)   }
```

Figure 13.2
Program segment E1, a C program segment with embedded SQL.

no EMPLOYEE tuple existed with the given Ssn; therefore it outputs a message to that effect (line 8).

In E1 a *single record* is selected by the embedded SQL query (because Ssn is a key attribute of EMPLOYEE);. When a single record is retrieved, the programmer can assign its attribute values directly to C program variables in the INTO clause, as in line 5. In general, an SQL query can retrieve many tuples. In that case, the C program will typically go through the retrieved tuples and process them one at a time. The concept of a *cursor* is used to allow tuple-at-a-time processing of a query result by the host language program. We describe cursors next.

13.2.2 Retrieving Multiple Tuples with Embedded SQL Using Cursors

We can think of a **cursor** as a pointer that points to a *single tuple* (*row*) from the result of a query that retrieves multiple tuples. The cursor is declared when the SQL query command is declared in the program. Later in the program, an **OPEN CURSOR** command fetches the query result from the database and sets the cursor to a position *before the first row* in the result of the query. This becomes the **current row** for the cursor. Subsequently, **FETCH** commands are issued in the program; each FETCH moves the cursor to the *next row* in the result of the query, making it the current row and copying its attribute values into the C (host language) program variables specified in the FETCH command by an INTO clause. The cursor variable is basically an **iterator** that iterates (loops) over the tuples in the query result—one tuple at a time.

To determine when all the tuples in the result of the query have been processed, the communication variable SQLCODE (or, alternatively, SQLSTATE) is checked. If a FETCH command is issued that results in moving the cursor past the last tuple in the result of the query, a positive value (SQLCODE $>$ 0) is returned in SQLCODE, indicating that no data (tuple) was found (or the string '02000' is returned in SQLSTATE). The programmer uses this to terminate a loop over the tuples in the query result. In general, numerous cursors can be opened at the same time. A

CLOSE CURSOR command is issued to indicate that we are done with processing the result of the query associated with that cursor.

An example of using cursors to process a query result with multiple records is shown in Figure 13.3, where a cursor called EMP is declared in line 4. The EMP cursor is associated with the SQL query declared in lines 5 through 6, but the query is not executed until the OPEN EMP command (line 8) is processed. The OPEN <cursor name> command executes the query and fetches its result as a table into the program workspace, where the program can loop through the individual rows (tuples) by subsequent FETCH <cursor name> commands (line 9). We assume that appropriate C program variables have been declared as in Figure 13.1. The program segment in E2 reads (inputs) a department name (line 0), retrieves the matching department number from the database (lines 1 to 3), and then retrieves the employees who work in that department via the declared EMP cursor. A loop (lines 10 to 18) iterates over each record in the query result, one at a time, and prints the employee name. The program then reads (inputs) a raise amount for that employee (line 12) and updates the employee's salary in the database by the raise amount that was provided (lines 14 to 16).

This example also illustrates how the programmer can *update* database records. When a cursor is defined for rows that are to be modified (**updated**), we must add

Figure 13.3
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 ;
```

the clause **FOR UPDATE OF** in the cursor declaration and list the names of any attributes that will be updated by the program. This is illustrated in line 7 of code segment E2. If rows are to be **deleted**, the keywords **FOR UPDATE** must be added without specifying any attributes. In the embedded UPDATE (or DELETE) command, the condition **WHERE CURRENT OF**<cursor name> specifies that the current tuple referenced by the cursor is the one to be updated (or deleted), as in line 16 of E2.

Notice that declaring a cursor and associating it with a query (lines 4 through 7 in E2) does not execute the query; the query is executed only when the OPEN <cursor name> command (line 8) is executed. Also notice that there is no need to include the **FOR UPDATE OF** clause in line 7 of E2 if the results of the query are to be used *for retrieval purposes only* (no update or delete).

General Options for a Cursor Declaration. Several options can be specified when declaring a cursor. The general form of a cursor declaration is as follows:

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

We already briefly discussed the options listed in the last line. The default is that the query is for retrieval purposes (FOR READ ONLY). If some of the tuples in the query result are to be updated, we need to specify FOR UPDATE OF <attribute list> and list the attributes that may be updated. If some tuples are to be deleted, we need to specify FOR UPDATE without any attributes listed.

When the optional keyword SCROLL is specified in a cursor declaration, it is possible to position the cursor in other ways than for purely sequential access. A **fetch orientation** can be added to the FETCH command, whose value can be one of NEXT, PRIOR, FIRST, LAST, ABSOLUTE *i*, and RELATIVE *i*. In the latter two commands, *i* must evaluate to an integer value that specifies an absolute tuple position within the query result (for ABSOLUTE *i*), or a tuple position relative to the current cursor position (for RELATIVE *i*). The default fetch orientation, which we used in our examples, is NEXT. The fetch orientation allows the programmer to move the cursor around the tuples in the query result with greater flexibility, providing random access by position or access in reverse order. When SCROLL is specified on the cursor, the general form of a FETCH command is as follows, with the parts in square brackets being optional:

```
FETCH [ [ <fetch orientation> ] FROM ] <cursor name> INTO <fetch target
list> ;
```

The ORDER BY clause orders the tuples so that the FETCH command will fetch them in the specified order. It is specified in a similar manner to the corresponding clause for SQL queries (see Section 4.3.6). The last two options when declaring a cursor (INSENSITIVE and WITH HOLD) refer to transaction characteristics of database programs, which we will discuss in Chapter 21.

13.2.3 Specifying Queries at Runtime Using Dynamic SQL

In the previous examples, the embedded SQL queries were written as part of the host program source code. Hence, any time 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, such as the interactive interfaces available for most relational DBMSs. Another example is when a user-friendly interface generates SQL queries dynamically for the user based on point-and-click operations on a graphical schema (for example, a QBE-like interface; see Appendix C). In this section, we give a brief overview of **dynamic SQL**, which is one technique for writing this type of database program, by giving a simple example to illustrate how dynamic SQL can work. In Section 13.3, we will describe another approach for dealing with dynamic queries.

Program segment E3 in Figure 13.4 reads a string that is input by the user (that string should be an SQL update command) 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. Notice that in this case no syntax check or other types of checks on the command are possible *at compile time*, since the SQL command is not available until runtime. This contrasts with our previous examples of embedded SQL, where the query could be checked at compile time because its text was in the program source code.

Although including a dynamic update command is relatively straightforward in dynamic SQL, a dynamic query is much more complicated. This is because usually we do not know the types or the number of attributes to be retrieved by the SQL query when we are writing the program. A complex data structure is sometimes needed to allow for different numbers and types of attributes in the query result if no prior information is known about the dynamic query. Techniques similar to those that we discuss in Section 13.3 can be used to assign query results (and query parameters) to host program variables.

In E3, the reason for separating PREPARE and EXECUTE is that if the command is to be executed multiple times in a program, it can be prepared only once. Preparing the command generally involves syntax and other types of checks by the system, as

```
    //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 ;
    ...
```

Figure 13.4
Program segment E3, a C program segment that uses dynamic SQL for updating a table.

well as generating the code for executing it. It is possible to combine the PREPARE and EXECUTE commands (lines 4 and 5 in E3) into a single statement by writing

```
EXEC SQL EXECUTE IMMEDIATE :sqlupdatestring ;
```

This is useful if the command is to be executed only once. Alternatively, the programmer can separate the two statements to catch any errors after the PREPARE statement, if any.

13.2.4 SQLJ: Embedding SQL Commands in Java

In the previous subsections, we gave an overview of how SQL commands can be embedded in a traditional programming language, using the C language in our examples. We now turn our attention to how SQL can be embedded in an object-oriented programming language,[8] in particular, the Java language. SQLJ is a standard that has been adopted by several vendors for embedding SQL in Java. Historically, SQLJ was developed after JDBC, which is used for accessing SQL databases from Java using function calls. We discuss JDBC in Section 13.3.2. In this section, we focus on SQLJ as it is used in the Oracle RDBMS. An SQLJ translator will 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.[9] In this section, we focus on how to use SQLJ concepts to write embedded SQL in a Java program.

Before being able to process SQLJ with Java in Oracle, it is necessary to import several class libraries, shown in Figure 13.5. These include the JDBC and IO classes (lines 1 and 2), plus the additional classes listed in lines 3, 4, and 5. In addition, the program must first connect to the desired database using the function call `getConnection`, which is one of the methods of the `oracle` class in line 5 of Figure

```
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) ;
    ...
```

Figure 13.5
Importing classes needed for including SQLJ in Java programs in Oracle, and establishing a connection and default context.

[8]This section assumes familiarity with object-oriented concepts (see Chapter 11) and basic JAVA concepts.

[9]We discuss JDBC drivers in Section 13.3.2.

13.5. The format of this function call, which returns an object of type *default context*,[10] is as follows:

```
public static DefaultContext
getConnection(String url, String user, String password,
    Boolean autoCommit)
throws SQLException ;
```

For example, we can write the statements in lines 6 through 8 in Figure 13.5 to connect to an Oracle database located at the `url` <url name> using the login of <user name> and <password> with automatic commitment of each command,[11] and then set this connection as the **default context** for subsequent commands.

In the following examples, we will not show complete Java classes or programs since it is not our intention to teach Java. Rather, we will show program segments that illustrate the use of SQLJ. Figure 13.6 shows the Java program variables used in our examples. Program segment J1 in Figure 13.7 reads an employee's Ssn and prints some of the employee's information from the database.

Notice that because Java already uses the concept of **exceptions** for error handling, a special exception called `SQLException` is used to return errors or exception conditions after executing an SQL database command. This plays a similar role to SQLCODE and SQLSTATE in embedded SQL. Java has many types of predefined exceptions. Each Java operation (function) must specify the exceptions that can be **thrown**—that is, the exception conditions that may occur while executing the Java code of that operation. If a defined exception occurs, the system transfers control to the Java code specified for exception handling. In J1, exception handling for an `SQLException` is specified in lines 7 and 8. In Java, the following structure

```
try {<operation>} catch (<exception>) {<exception handling
   code>} <continuation code>
```

is used to deal with exceptions that occur during the execution of <`operation`>. If no exception occurs, the <`continuation code`> is processed directly. Exceptions

Figure 13.6
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 ;
```

[10]A *default context*, when set, applies to subsequent commands in the program until it is changed.

[11]*Automatic commitment* roughly means that each command is applied to the database after it is executed. The alternative is that the programmer wants to execute several related database commands and then commit them together. We discuss commit concepts in Chapter 21 when we describe database transactions.

```
    //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)
```

Figure 13.7
Program segment J1, a Java program segment with SQLJ.

that can be thrown by the code in a particular operation should be specified as part of the operation declaration or *interface*—for example, in the following format:

```
<operation return type> <operation name> (<parameters>)
   throws SQLException, IOException ;
```

In SQLJ, the embedded SQL commands within a Java program are preceded by `#sql`, as illustrated in J1 line 3, so that they can be identified by the preprocessor. The `#sql` is used instead of the keywords EXEC SQL that are used in embedded SQL with the C programming language (see Section 13.2.1). SQLJ uses an *INTO clause*—similar to that used in embedded SQL—to return the attribute values retrieved from the database by an SQL query into Java program variables. The program variables are preceded by colons (:) in the SQL statement, as in embedded SQL.

In J1 a *single tuple* is retrieved by the embedded SQLJ query; that is why we are able to assign its attribute values directly to Java program variables in the INTO clause in line 4 in Figure 13.7. For queries that retrieve many tuples, SQLJ uses the concept of an *iterator*, which is similar to a cursor in embedded SQL.

13.2.5 Retrieving Multiple Tuples in SQLJ Using Iterators

In SQLJ, an **iterator** is a type of object associated with a collection (set or multiset) of records in a query result.[12] The iterator is associated with the tuples and attributes that appear in a query result. There are two types of iterators:

1. A **named iterator** is associated with a query result by listing the attribute *names and types* that appear in the query result. The attribute names must correspond to appropriately declared Java program variables, as shown in Figure 13.6.
2. A **positional iterator** lists only the *attribute types* that appear in the query result.

[12]We discussed iterators in more detail in Chapter 11 when we presented object database concepts.

In both cases, the list should be *in the same order* as the attributes that are listed in the SELECT clause of the query. However, looping over a query result is different for the two types of iterators, as we shall see. First, we show an example of using a *named* iterator in Figure 13.8, program segment J2A. Line 9 in Figure 13.8 shows how a *named iterator type* `Emp` is declared. Notice that the names of the attributes in a named iterator type must match the names of the attributes in the SQL query result. Line 10 shows how an *iterator object* `e` of type `Emp` is created in the program and then associated with a query (lines 11 and 12).

When the iterator object is associated with a query (lines 11 and 12 in Figure 13.8), the program fetches the query result from the database and sets the iterator to a position *before the first row* in the result of the query. This becomes the **current row** for the iterator. Subsequently, **next** operations are issued on the iterator object; each `next` moves the iterator to the *next row* in the result of the query, making it the current row. If the row exists, the operation retrieves the attribute values for that row into the corresponding program variables. If no more rows exist, the `next` operation returns NULL, and can thus be used to control the looping. Notice that the named iterator does not need an INTO clause, because the program variables corresponding to the retrieved attributes are already specified when the iterator type is declared (line 9 in Figure 13.8).

Figure 13.8
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() ;
```

In Figure 13.8, the command `(e.next())` in line 13 performs two functions: It gets the next tuple in the query result and controls the `while` loop. Once the program is done with processing the query result, the command `e.close()` (line 16) closes the iterator.

Next, consider the same example using *positional* iterators as shown in Figure 13.9 (program segment J2B). Line 9 in Figure 13.9 shows how a *positional iterator type* `Emppos` is declared. The main difference between this and the named iterator is that there are no attribute names (corresponding to program variable names) in the positional iterator—only attribute types. This can provide more flexibility, but makes the processing of the query result slightly more complex. The attribute types must still must be compatible with the attribute types in the SQL query result and in the same order. Line 10 shows how a *positional iterator object* `e` of type `Emppos` is created in the program and then associated with a query (lines 11 and 12).

The positional iterator behaves in a manner that is more similar to embedded SQL (see Section 13.2.2). A **FETCH** <**iterator variable**> **INTO** <**program variables**> command is needed to get the next tuple in a query result. The first time `fetch` is executed, it gets the first tuple (line 13 in Figure 13.9). Line 16 gets the next tuple until no more tuples exist in the query result. To control the loop, a positional iterator function `e.endFetch()` is used. This function is set to a value of TRUE when the iterator is initially associated with an SQL query (line 11), and is set to FALSE

Figure 13.9
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() ;
```

each time a fetch command returns a valid tuple from the query result. It is set to TRUE again when a fetch command does not find any more tuples. Line 14 shows how the looping is controlled by negation.

13.3 Database Programming with Function Calls: SQL/CLI and JDBC

Embedded SQL (see Section 13.2) is sometimes referred to as a **static** database programming approach because the query text is written within the program source code and cannot be changed without recompiling or reprocessing the source code. The use of function calls is a more **dynamic** approach for database programming than embedded SQL. We already saw one dynamic database programming technique—dynamic SQL—in Section 13.2.3. The techniques discussed here provide another approach to dynamic database programming. A **library of functions**, also known as an **application programming interface** (**API**), is used to access the database. Although this provides more flexibility because no preprocessor is needed, one drawback is that syntax and other checks on SQL commands have to be done at runtime. Another drawback is that it sometimes requires more complex programming to access query results because the types and numbers of attributes in a query result may not be known in advance.

In this section, we give an overview of two function call interfaces. We first discuss the **SQL Call Level Interface** (**SQL/CLI**), which is part of the SQL standard. This was developed as a follow-up to the earlier technique known as ODBC (Open Database Connectivity). We use C as the host language in our SQL/CLI examples. Then we give an overview of **JDBC**, which is the call function interface for accessing databases from Java. Although it is commonly assumed that JDBC stands for Java Database Connectivity, JDBC is just a registered trademark of Sun Microsystems, *not* an acronym.

The main advantage of using a function call interface is that it makes it easier to access multiple databases within the same application program, even if they are stored under different DBMS packages. We discuss this further in Section 13.3.2 when we discuss Java database programming with JDBC, although this advantage also applies to database programming with SQL/CLI and ODBC (see Section 13.3.1).

13.3.1 Database Programming with SQL/CLI Using C as the Host Language

Before using the function calls in SQL/CLI, it is necessary to install the appropriate library packages on the database server. These packages are obtained from the vendor of the DBMS being used. We now give an overview of how SQL/CLI can be used in a C program.[13] We will illustrate our presentation with the sample program segment CLI1 shown in Figure 13.10.

[13]Our discussion here also applies to the C++ programming language, since we do not use any of the object-oriented features but focus on the database programming mechanism.

Figure 13.10
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) ;
  }
}
```

When using SQL/CLI, the SQL statements are dynamically created and passed as *string parameters* in the function calls. Hence, it is necessary to keep track of the information about host program interactions with the database in runtime data structures because the database commands are processed at runtime. The information is kept in four types of records, represented as *structs* in C data types. An **environment record** is used as a container to keep track of one or more database connections and to set environment information. A **connection record** keeps track of the information needed for a particular database connection. A **statement record** keeps track of the information needed for one SQL statement. A **description record** keeps track of the information about tuples or parameters—for example, the number of attributes and their types in a tuple, or the number and types of parameters in a function call. This is needed when the programmer does not know this information about the query when writing the program. In our examples, we assume that the programmer knows the exact query, so we do not show any description records.

Each record is accessible to the program through a C pointer variable—called a **handle** to the record. The handle is returned when a record is first created. To create a record and return its handle, the following SQL/CLI function is used:

```
SQLAllocHandle(<handle_type>, <handle_1>, <handle_2>)
```

In this function, the parameters are as follows:

- `<handle_type>` indicates the type of record being created. The possible values for this parameter are the keywords `SQL_HANDLE_ENV`, `SQL_HANDLE_DBC`, `SQL_HANDLE_STMT`, or `SQL_HANDLE_DESC`, for an environment, connection, statement, or description record, respectively.
- `<handle_1>` indicates the container within which the new handle is being created. For example, for a connection record this would be the environment within which the connection is being created, and for a statement record this would be the connection for that statement.
- `<handle_2>` is the pointer (handle) to the newly created record of type `<handle_type>`.

When writing a C program that will include database calls through SQL/CLI, the following are the typical steps that are taken. We illustrate the steps by referring to the example CLI1 in Figure 13.10, which reads a Social Security number of an employee and prints the employee's last name and salary.

1. The *library of functions* comprising SQL/CLI must be included in the C program. This is called `sqlcli.h`, and is included using line 0 in Figure 13.10.
2. Declare *handle variables* of types `SQLHSTMT`, `SQLHDBC`, `SQLHENV`, and `SQLHDESC` for the statements, connections, environments, and descriptions needed in the program, respectively (lines 2 to 4).[14] Also declare variables of type `SQLRETURN` (line 5) to hold the return codes from the SQL/CLI function calls. A return code of 0 (zero) indicates *successful execution* of the function call.
3. An *environment record* must be set up in the program using `SQLAllocHandle`. The function to do this is shown in line 6. Because an environment record is not contained in any other record, the parameter `<handle_1>` is the NULL handle `SQL_NULL_HANDLE` (NULL pointer) when creating an environment. The handle (pointer) to the newly created environment record is returned in variable `env1` in line 6.
4. A *connection record* is set up in the program using `SQLAllocHandle`. In line 7, the connection record created has the handle `con1` and is contained in the environment `env1`. A **connection** is then established in `con1` to a particular server database using the `SQLConnect` function of SQL/CLI (line 8). In our example, the database server name we are connecting to is *dbs* and the account name and password for login are *js* and *xyz*, respectively.
5. A *statement record* is set up in the program using `SQLAllocHandle`. In line 9, the statement record created has the handle `stmt1` and uses the connection `con1`.
6. The statement is *prepared* using the SQL/CLI function `SQLPrepare`. In line 10, this assigns the SQL **statement string** (the *query* in our example) to the

[14]To keep our presentation simple, we will not show description records here.

statement handle `stmt1`. The question mark (`?`) symbol in line 10 represents a **statement parameter**, which is a value to be determined at runtime—typically by binding it to a C program variable. In general, there could be several parameters in a statement string. They are distinguished by the order of appearance of the question marks in the statement string (the first `?` represents parameter 1, the second `?` represents parameter 2, and so on). The last parameter in `SQLPrepare` should give the length of the SQL statement string in bytes, but if we enter the keyword `SQL_NTS`, this indicates that the string holding the query is a *NULL-terminated string* so that SQL can calculate the string length automatically. This use of `SQL_NTS` also applies to *other string parameters* in the function calls in our examples.

7. Before executing the query, any parameters in the query string should be bound to program variables using the SQL/CLI function `SQLBindParameter`. In Figure 13.10, the parameter (indicated by `?`) to the prepared query referenced by `stmt1` is bound to the C program variable `ssn` in line 12. If there are *n* parameters in the SQL statement, we should have *n* `SQLBindParameter` function calls, each with a different *parameter position* (1, 2, …, *n*).
8. Following these preparations, we can now execute the SQL statement referenced by the handle `stmt1` using the function `SQLExecute` (line 13). Notice that although the query will be executed in line 13, the query results have not yet been assigned to any C program variables.
9. In order to determine where the result of the query is returned, one common technique is the **bound columns** approach. Here, each column in a query result is bound to a C program variable using the `SQLBindCol` function. The columns are distinguished by their order of appearance in the SQL query. In Figure 13.10 lines 15 and 16, the two columns in the query (`Lname` and `Salary`) are bound to the C program variables `lname` and `salary`, respectively.[15]
10. Finally, in order to retrieve the column values into the C program variables, the function `SQLFetch` is used (line 17). This function is similar to the FETCH command of embedded SQL. If a query result has a collection of tuples, each `SQLFetch` call gets the next tuple and returns its column values into the bound program variables. `SQLFetch` returns an exception (nonzero) code if there are no more tuples in the query result.[16]

[15]An alternative technique known as **unbound columns** uses different SQL/CLI functions, namely SQLGetCol or SQLGetData, to retrieve columns from the query result without previously binding them; these are applied after the SQLFetch command in line 17.

[16]If unbound program variables are used, SQLFetch returns the tuple into a temporary program area. Each subsequent SQLGetCol (or SQLGetData) returns one attribute value in order. Basically, for each row in the query result, the program should iterate over the attribute values (columns) in that row. This is useful if the number of columns in the query result is variable.

As we can see, using dynamic function calls requires a lot of preparation to set up the SQL statements and to bind statement parameters and query results to the appropriate program variables.

In CLI1 a *single tuple* is selected by the SQL query. Figure 13.11 shows an example of retrieving multiple tuples. We assume that appropriate C program variables have been declared as in Figure 13.1. The program segment in CLI2 reads (inputs) a department number and then retrieves the employees who work in that department. A loop then iterates over each employee record, one at a time, and prints the employee's last name and salary.

Figure 13.11
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) ;
    }
  }
}
```

13.3.2 JDBC: SQL Function Calls for Java Programming

We now turn our attention to how SQL can be called from the Java object-oriented programming language.[17] The function libraries for this access are known as **JDBC**.[18] The Java programming language was designed to be platform independent—that is, a program should be able to run on any type of computer system that has a Java interpreter installed. Because of this portability, many RDBMS vendors provide JDBC drivers so that it is possible to access their systems via Java programs. A **JDBC driver** is basically an implementation of the function calls specified in the JDBC application programming interface (API) for a particular vendor's RDBMS. Hence, a Java program with JDBC function calls can access any RDBMS that has a JDBC driver available.

Because Java is object-oriented, its function libraries are implemented as **classes**. Before being able to process JDBC function calls with Java, it is necessary to import the **JDBC class libraries**, which are called `java.sql.*`. These can be downloaded and installed via the Web.[19]

JDBC is designed to allow a single Java program to connect to several different databases. These are sometimes called the **data sources** accessed by the Java program. These data sources could be stored using RDBMSs from different vendors and could reside on different machines. Hence, different data source accesses within the same Java program may require JDBC drivers from different vendors. To achieve this flexibility, a special JDBC class called the **driver manager** class is employed, which keeps track of the installed drivers. A driver should be *registered* with the driver manager before it is used. The operations (methods) of the driver manager class include `getDriver`, `registerDriver`, and `deregisterDriver`. These can be used to add and remove drivers dynamically. Other functions set up and close connections to data sources, as we will see.

To load a JDBC driver explicitly, the generic Java function for loading a class can be used. For example, to load the JDBC driver for the Oracle RDBMS, the following command can be used:

```
    Class.forName("oracle.jdbc.driver.OracleDriver")
```

This will register the driver with the driver manager and make it available to the program. It is also possible to load and register the driver(s) needed in the command line that runs the program, for example, by including the following in the command line:

```
    -Djdbc.drivers = oracle.jdbc.driver
```

[17]This section assumes familiarity with object-oriented concepts (see Chapter 11) and basic Java concepts.

[18]As we mentioned earlier, JDBC is a registered trademark of Sun Microsystems, although it is commonly thought to be an acronym for Java Database Connectivity.

[19]These are available from several Web sites—for example, at http://industry.java.sun.com/products/jdbc/drivers.

The following are typical steps that are taken when writing a Java application program with database access through JDBC function calls. We illustrate the steps by referring to the example JDBC1 in Figure 13.12, which reads a Social Security number of an employee and prints the employee's last name and salary.

1. The JDBC *library of classes* must be imported into the Java program. These classes are called `java.sql.*`, and can be imported using line 1 in Figure 13.12. Any additional Java class libraries needed by the program must also be imported.
2. Load the JDBC driver as discussed previously (lines 4 to 7). The Java exception in line 5 occurs if the driver is not loaded successfully.
3. Create appropriate variables as needed in the Java program (lines 8 and 9).

Figure 13.12
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.printline(lname + salary) ;
24)   } }
25) }
```

4. **The `Connection` object.** A **connection object** is created using the `getConnection` function of the `DriverManager` class of JDBC. In lines 12 and 13, the `Connection` object is created by using the function call `getConnection(urlstring)`, where `urlstring` has the form

   ```
   jdbc:oracle:<driverType>:<dbaccount>/<password>
   ```

   An alternative form is

   ```
   getConnection(url, dbaccount, password)
   ```

   Various properties can be set for a connection object, but they are mainly related to transactional properties, which we discuss in Chapter 21.

5. **The `Statement` object.** A **statement object** is created in the program. In JDBC, there is a basic statement class, `Statement`, with two specialized subclasses: `PreparedStatement` and `CallableStatement`. The example in Figure 13.12 illustrates how `PreparedStatement` objects are created and used. The next example (Figure 13.13) illustrates the other type of

```
    //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.printline(lname + salary) ;
23)   } }
24) }
```

Figure 13.13
Program segment JDBC2, a Java program segment that uses JDBC for a query with a collection of tuples in its result.

Statement objects. In line 14 in Figure 13.12, a query string with a single parameter—indicated by the ? symbol—is created in the string variable stmt1. In line 15, an object p of type PreparedStatement is created based on the query string in stmt1 and using the connection object conn. In general, the programmer should use PreparedStatement objects if a query is to be executed *multiple times*, since it would be prepared, checked, and compiled only once, thus saving this cost for the additional executions of the query.

6. **Setting the statement parameters.** The question mark (?) symbol in line 14 represents a **statement parameter**, which is a value to be determined at runtime, typically by binding it to a Java program variable. In general, there could be several parameters, distinguished by the order of appearance of the question marks within the statement string (first ? represents parameter 1, second ? represents parameter 2, and so on), as we discussed previously.
7. Before executing a PreparedStatement query, any parameters should be bound to program variables. Depending on the type of the parameter, different functions such as setString, setInteger, setDouble, and so on are applied to the PreparedStatement object to set its parameters. The appropriate function should be used to correspond to the data type of the parameter being set. In Figure 13.12, the parameter (indicated by ?) in object p is bound to the Java program variable ssn in line 18. The function setString is used because ssn is a string variable. If there are *n* parameters in the SQL statement, we should have *n* set... functions, each with a different parameter position (1, 2, ..., *n*). Generally, it is advisable to clear all parameters before setting any new values (line 17).
8. Following these preparations, we can now execute the SQL statement referenced by the object p using the function executeQuery (line 19). There is a generic function execute in JDBC, plus two specialized functions: executeUpdate and executeQuery. executeUpdate is used for SQL insert, delete, or update statements, and returns an integer value indicating the number of tuples that were affected. executeQuery is used for SQL retrieval statements, and returns an object of type ResultSet, which we discuss next.
9. **The ResultSet object.** In line 19, the result of the query is returned in an ***object* r** of type **ResultSet**. This resembles a two-dimensional array or a table, where the tuples are the rows and the attributes returned are the columns. A ResultSet object is similar to a cursor in embedded SQL and an iterator in SQLJ. In our example, when the query is executed, r refers to a tuple before the first tuple in the query result. The r.next() function (line 20) moves to the next tuple (row) in the ResultSet object and returns NULL if there are no more objects. This is used to control the looping. The programmer can refer to the attributes in the current tuple using various get... functions that depend on the type of each attribute (for example, getString, getInteger, getDouble, and so on). The programmer can either use the attribute positions (1, 2) or the actual attribute names

(`"Lname"`, `"Salary"`) with the `get...` functions. In our examples, we used the positional notation in lines 21 and 22.

In general, the programmer can check for SQL exceptions after each JDBC function call. We did not do this to simplify the examples.

Notice that JDBC does not distinguish between queries that return single tuples and those that return multiple tuples, unlike some of the other techniques. This is justifiable because a single tuple result set is just a special case.

In example JDBC1, a *single tuple* is selected by the SQL query, so the loop in lines 20 to 24 is executed at most once. The example shown in Figure 13.13 illustrates the retrieval of multiple tuples. The program segment in JDBC2 reads (inputs) a department number and then retrieves the employees who work in that department. A loop then iterates over each employee record, one at a time, and prints the employee's last name and salary. This example also illustrates how we can execute a query directly, without having to prepare it as in the previous example. This technique is preferred for queries that will be executed only once, since it is simpler to program. In line 17 of Figure 13.13, the programmer creates a `Statement` object (instead of `PreparedStatement`, as in the previous example) without associating it with a particular query string. The query string `q` is *passed to the statement object* `s` when it is executed in line 18.

This concludes our brief introduction to JDBC. The interested reader is referred to the Web site http://java.sun.com/docs/books/tutorial/jdbc/, which contains many further details about JDBC.

13.4 Database Stored Procedures and SQL/PSM

This section introduces two additional topics related to database programming. In Section 13.4.1, we discuss the concept of stored procedures, which are program modules that are stored by the DBMS at the database server. Then in Section 13.4.2 we discuss the extensions to SQL that are specified in the standard to include general-purpose programming constructs in SQL. These extensions are known as SQL/PSM (SQL/Persistent Stored Modules) and can be used to write stored procedures. SQL/PSM also serves as an example of a database programming language that extends a database model and language—namely, SQL—with some programming constructs, such as conditional statements and loops.

13.4.1 Database Stored Procedures and Functions

In our presentation of database programming techniques so far, there was an implicit assumption that the database application program was running on a client machine, or more likely at the *application server computer* in the middle-tier of a three-tier client-server architecture (see Section 2.5.4 and Figure 2.7). In either case, the machine where the program is executing is different from the machine on which