6

DATABASE APPLICATION DEVELOPMENT

- How do application programs connect to a DBMS?
- How can applications manipulate data retrieved from a DBMS?
- How can applications modify data in a DBMS?
- What are cursors?
- What is JDBC and how is it used?
- What is SQLJ and how is it used?
- What are stored procedures?
- **Key concepts:** Embedded SQL, Dynamic SQL, cursors; JDBC, connections, drivers, ResultSets, java.sql, SQLJ; stored procedures, SQL/PSM

He profits most who serves best.

------Ivlotto for Rotary International

In Chapter 5, we looked at a wide range of SQL query constructs, treating SQL as an independent language in its own right. A relational DBMS supports an *interactive SQL* interface, and users can directly enter SQL commands. This simple approach is fine as long as the task at hand can be accomplished entirely with SQL cormnands. In practice, we often encounter situations in which we need the greater flexibility of a general-purpose programming language in addition to the data manipulation facilities provided by SQL. For example, we rnay want to integrate a database application with a nice graphical user interface, or we may want to integrate with other existing applications.

Applications that rely on the DBMS to manage data run as separate processes that connect to the DBlvIS to interact with it. Once a connection is established, SQL commands can be used to insert, delete, and modify data. SQL queries can be used to retrieve desired data. but we need to bridge an important difference in how a database system sees data and how an application program in a language like Java or C sees data: The result of a database query is a set (or multiset) or records, hut Java has no set or multiset data type. This mismatch is resolved through additional SQL constructs that allow applications to obtain a handle on a collection and iterate over the records one at a time.

We introduce Embedded SQL, Dynamic SQL, and cursors in Section 6.1. Embedded SQL allows us to access data using static SQL queries in application code (Section 6.1.1); with Dynamic SQL, we can create the queries at run-time (Section 6.1.3). Cursors bridge the gap between set-valued query answers and programming languages that do not support set-values (Section 6.1.2).

The emergence of Java as a popular application development language, especially for Internet applications, has made accessing a DBMS from Java code a particularly important topic. Section 6.2 covers JDBC, a prograruming interface that allows us to execute SQL queries from a Java program and use the results in the Java program. JDBC provides greater portability than Embedded SQL or Dynamic SQL, and offers the ability to connect to several DBMSs without recompiling the code. Section 6.4 covers SQLJ, which does the same for static SQL queries, but is easier to program in than Java, with JDBC.

Often, it is useful to execute application code at the database server, rather than just retrieve data and execute application logic in a separate process. Section 6.5 covers stored procedures, which enable application logic to be stored and executed at the database server. We conclude the chapter by discussing our B&N case study in Section 6.6.

While writing database applications, we must also keep in mind that typically many application programs run concurrently. The transaction concept, introduced in Chapter 1, is used to encapsulate the effects of an application on the database. An application can select certain transaction properties through SQL cormnands to control the degree to which it is exposed to the changes of other concurrently running applications. We touch on the transaction concept at many points in this chapter, and, in particular, cover transaction-related aspects of JDBC. A full discussion of transaction properties and SQL's support for transactions is deferred until Chapter 16.

Exainples that appear in this chapter are available online at

http://www.cs.wisc.edu/-dbbook

6.1 ACCESSING DATABASES FROM APPLICATIONS

In this section, we cover how SQL commands can be executed from within a program in a host language such as C or Java. The use of SQL commands within a host language program is called Embedded SQL. Details of Embedded SQL also depend on the host language. Although similar capabilities are supported for a variety of host languages, the syntax sometimes varies.

We first cover the basics of Embedded SQL with static SQL queries in Section 6.1.1. We then introduce cursors in Section 6.1.2. We discuss Dynamic SQL, which allows us to construct SQL queries at runtime (and execute them) in Section 6.1.3.

6.1.1 Embedded SQL

Conceptually, embedding SQL commands in a host language program is straightforward. SQL statements (i.e., not declarations) can be used wherever a statement in the host language is allowed (with a few restrictions). SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language. Also, any host language variables used to pass arguments into an SQL command must be declared in SQL. In particular, some special host language variables *must* be declared in SQL (so that, for example, any error conditions arising during SQL execution can be communicated back to the main application program in the host language).

There are, however, two complications to bear in mind. First, the data types recognized by SQL may not be recognized by the host language and vice versa. This mismatch is typically addressed by casting data values appropriately before passing them to or frorn SQL commands. (SQL, like other programming languages, provides an operator to cast values of alle type into values of another type.) The second complication has to do with SQL being set-oriented, and is addressed using cursors (see Section 6.1.2. Commands operate on and produce tables, which are sets

In our discussion of Embedded SQL, we assume that the host language is C for concreteness. because minor differcnces exist in how SQL statements are embedded in differcnt host languages.

Declaring Variables and Exceptions

SQL statements can refer to variables defined in the host program. Such host-language variables must be prefixed by a colon (:) in SQL statements and be declared between the commands EXEC SQL BEGIN DECLARE SECTION and EXEC

SQL END DECLARE SECTION. The declarations are similar to how they would look in a C program and, as usual in C. are separated by semicolons. For example. we can declare variables *c_sname*, *c_sid*, *c_rating*, and *c_age* (with the initial *c* used as a naming convention to emphasize that these are host language variables) as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

The first question that arises is which SQL types correspond to the various C types, since we have just declared a collection of C variables whose values are intended to be read (and possibly set) in an SQL run-time environment when an SQL statement that refers to them is executed. The SQL-92 standard defines such a correspondence between the host language types and SQL types for a number of host languages. In our example, *c_snamc* has the type CHARACTER(20) when referred to in an SQL statement, *c_sid* has the type INTEGER, *c_rating* has the type SMALLINT, and *c_age* has the type REAL.

We also need some way for SQL to report what went wrong if an error condition arises when executing an SQL statement. The SQL-92 standard recognizes two special variables for reporting errors, SQLCODE and SQLSTATE. SQLCODE is the older of the two and is defined to return some negative value when an error condition arises, without specifying further just what error a particular negative integer denotes. SQLSTATE, introduced in the SQL-92 standard for the first time, associates predefined values with several common error conditions, thereby introducing some uniformity to how errors are reported. One of these two variables *must* be declared. The appropriate C type for SQLCODE is long and the appropriate C type for SQLSTATE is char[6J , that is, a character string five characters long. (Recall the null-terminator in C strings.) In this chapter, we assume that SQLSTATE is declared.

Embedding SQL Statements

All SQL staternents embedded within a host program must be clearly marked, with the details dependent on the host language; in C, SQL statements must be prefixed by EXEC SQL. An SQL statement can essentially appear in any place in the host language program where a host language statement can appear.

As a simple example, the following Embedded SQL statement inserts a row, whose column values are based on the values of the host language variables contained in it, into the Sailors relation:

```
EXEC SQL
INSERT INTO Sailors VALUES (:c_sname, :c_sid, :c_rating, :c_age);
```

Observe that a semicolon terminates the command, as per the convention for terminating statements in C.

The SQLSTATE variable should be checked for errors and exceptions after each Embedded SQL statement. SQL provides the WHENEVER command to simplify this tedious task:

```
EXEC SQL WHENEVER [SQLERROR | NOT FOUND] [ CONTINUE | GOTO st'mt ]
```

The intent is that the value of SQLSTATE should be checked after each Embedded SQL statement is executed. If SQLERROR is specified and the value of SQLSTATE indicates an exception, control is transferred to *stmt*, which is presumably responsible for error and exception handling. Control is also transferred to *stmt* if NOT FOUND is specified and the value of SQLSTATE is 02000, which denotes NO DATA.

6.1.2 Cursors

A major problem in embedding SQL statements in a host language like C is that an *impedance mismatch* occurs because SQL operates on *sets* of records, whereas languages like C do not cleanly support a set-of-records abstraction. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation.

This mechanism is called a cursor. We can declare a cursor on any relation or on any SQL query (because every query returns a set of rows). Once a cursor is declared, we can open it (which positions the cursor just before the first row); fetch the next row; move the cursor (to the next row, to the row after the next *n*, to the first row, or to the previous row, etc., by specifying additional parameters for the FETCH command); or close the cursor. Thus, a cursor essentially allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

Basic Cursor Definition and Usage

Cursors enable us to examine, in the host language program, a collection of rows computed by an Embedded SQL statement:

- \Ve usually need to open a cursor if the embedded statement is a SELECT (i.e.) a query). However, we can avoid opening a cursor if the answer contains a single row, as we see shortly.
- INSERT, DELETE, and UPDATE staternents typically require no cursor, although some variants of DELETE and UPDATE use a cursor.

As an example, we can find the name and age of a sailor, specified by assigning a value to the host variable *c_sid*, declared earlier, as follows:

```
EXEC SQL SELECT S.sname, S.age
         INTO    :c_sname, :c_age
         FROM    Sailors S
         WHERE   S.sid = :c_sid;
```

The INTO clause allows us to assign the columns of the single answer row to the host variables *c_sname* and *c_age*. Therefore, we do not need a cursor to embed this query in a host language program. But what about the following query, which computes the names and ages of all sailors with a rating greater than the current value of the host variable *c_minrating*?

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating
```

This query returns a collection of rows, not just one row. 'When executed interactively, the answers are printed on the screen. If we embed this query in a C program by prefixing the cOlnmand with EXEC SQL, how can the answers be bound to host language variables? The INTO clause is inadequate because we must deal with several rows. The solution is to use a cursor:

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating;
```

This code can be included in a C program, and once it is executed, the cursor *sinfo* is defined. Subsequently, we can open the cursor:

```
OPEN sinfo:
```

The value of *c_minrating* in the SQL query associated with the cursor is the value of this variable when we open the cursor. (The cursor declaration is processed at compile-time, and the OPEN command is executed at run-time.)

A cursor can be thought of as 'pointing' to a row in the collection of answers to the query associated with it. When a cursor is opened, it is positioned just before the first row. We can use the FETCH command to read the first row of cursor *sinfo* into host language variables:

```
FETCH sinfo INTO :c_sname, :c_age;
```

When the FETCH statement is executed, the cursor is positioned to point at the next row (which is the first row in the table when FETCH is executed for the first time after opening the cursor) and the column values in the row are copied into the corresponding host variables. By repeatedly executing this FETCH statement (say, in a while-loop in the C program), we can read all the rows computed by the query, one row at a time. Additional parameters to the FETCH command allow us to position a cursor in very flexible ways, but we do not discuss them.

How do we know when we have looked at all the rows associated with the cursor? By looking at the special variables SQLCODE or SQLSTATE, of course. SQLSTATE, for example, is set to the value 02000, which denotes NO DATA, to indicate that there are no more rows if the FETCH statement positions the cursor after the last row.

When we are done with a cursor, we can close it:

```
CLOSE sinfo;
```

It can be opened again if needed, and the value of : *c_minrating* in the SQL query associated with the cursor would be the value of the host variable *c_minrating* at that time.

Properties of Cursors

The general form of a cursor declaration is:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR
        [WITH HOLD]
        FOR some query
        [ ORDER BY order-item-list ]
        [ FOR READ ONLY | FOR UPDATE ]
```

A cursor can be declared to be a read-only cursor (FOR READ ONLY) or, if it is a cursor on a base relation or an updatable view, to be an updatable cursor (FOR UPDATE). If it is IIpdatable, simple variants of the UPDATE and

DELETE commands allow us to update or delete the row on which the cursor is positioned. For example, if *sinfa* is an updatable cursor and open, we can execute the following statement:

```
UPDATE  Sailors S
SET     S.rating = S.rating - 1
WHERE   CURRENT of sinfo;
```

This Embedded SQL statement modifies the *rating* value of the row currently pointed to by cursor *sinfa;* similarly, we can delete this row by executing the next statement:

```
DELETE  Sailors S
WHERE   CURRENT of sinfo;
```

A cursor is updatable by default unless it is a scrollable or insensitive cursor (see below), in which case it is read-only by default.

If the keyword SCROLL is specified, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways; otherwise, only the basic FETCH command, which retrieves the next row, is allowed.

If the keyword INSENSITIVE is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows. Otherwise, and by default, other actions of some transaction could modify these rows, creating unpredictable behavior. For example, while we are fetching rows using the *sinfa* cursor, we might modify *rating* values in Sailor rows by concurrently executing the command:

```
UPDATE  Sailors S
SET     S.rating = S.rating -
```

Consider a Sailor row such that (1) it has not yet been fetched, and (2) its original *rating* value would have met the condition in the WHERE clause of the query associated with *sinfa,* but the new *rating* value does not. Do we fetch such a Sailor row? If INSENSITIVE is specified, the behavior is as if all answers were computed and stored when *sinfo* was opened; thus, the update command has no effect on the rows fetched by *sinfa* if it is executed after *sinfo* is opened. If INSENSITIVE is not specified, the behavior is implementation dependent in this situation.

A holdable cursor is specified using the WITH HOLD clause, and is not closed when the transaction is conunitted. The motivation for this cornes from long

transactions in which we access (and possibly change) a large number of rows of a table. If the transaction is aborted for any reason, the system potentially has to redo a lot of work when the transaction is restarted. Even if the transaction is not aborted, its locks are held for a long time and reduce the concurrency of the system. The alternative is to break the transaction into several smaller transactions, but remembering our position in the table between transactions (and other similar details) is complicated and error-prone. Allowing the application program to commit the transaction it initiated, while retaining its handle on the active table (i.e., the cursor) solves this problem: The application can commit its transaction and start a new transaction and thereby save the changes it has made thus far.

Finally, in what order do FETCH commands retrieve rows? In general this order is unspecified, but the optional ORDER BY clause can be used to specify a sort order. Note that columns mentioned in the ORDER BY clause cannot be updated through the cursor!

The order-item-list is a list of order-items; an order-item is a column name, optionally followed by one of the keywords ASC or DESC. Every column mentioned in the ORDER BY clause must also appear in the select-list of the query associated with the cursor; otherwise it is not clear what columns we should sort on. The keywords ASC or DESC that follow a column control whether the result should be sorted-with respect to that column-in ascending or descending order; the default is ASC. This clause is applied as the last step in evaluating the query.

Consider the query discussed in Section 5.5.1, and the answer shown in Figure 5.13. Suppose that a cursor is opened on this query, with the clause:

```
ORDER BY minage ASC, rating DESC
```

The answer is sorted first in ascending order by *minage,* and if several rows have the same *minage* value, these rows are sorted further in descending order by *rating.* The cursor would fetch the rows in the order shown in Figure 6.1.

rating	*minage*
8	25.5
3	25.5
7	35.0

Figure 6.1 Order in which Tuples Are Fetched

6.1.3 Dynamic SQL

Consider an application such as a spreadsheet or a graphical front-end that needs to access data from a DBMS. Such an application must accept commands from a user and, based on what the user needs, generate appropriate SQL statements to retrieve the necessary data. In such situations, we may not be able to predict in advance just what SQL statements need to be executed, even though there is (presumably) some algorithm by which the application can construct the necessary SQL statements once a user's command is issued.

SQL provides some facilities to deal with such situations; these are referred to as Dynamic SQL. We illustrate the two main commands, PREPARE and EXECUTE, through a simple example:

```
char c_sqlstring[] = {"DELETE FROM Sailors WHERE rating>5"};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

The first statement declares the C variable *c_sqlstring* and initializes its value to the string representation of an SQL command. The second statement results in this string being parsed and compiled as an SQL command, with the resulting executable bound to the SQL variable *readytogo*. (Since *readytogo* is an SQL variable, just like a cursor name, it is not prefixed by a colon.) The third statement executes the command.

Many situations require the use of Dynamic SQL. However, note that the preparation of a Dynamic SQL command occurs at run-time and is run-time overhead. Interactive and Embedded SQL commands can be prepared once at compile-time and then re-executecl as often as desired. Consequently you should limit the use of Dynamic SQL to situations in which it is essential.

There are many more things to know about Dynamic SQL---how we can pass parameters from the host language program to the SQL statement being pre-parcel, for example--but we do not discuss it further.

6.2 AN INTRODUCTION TO JDBC

Embedded SQL enables the integration of SQL with a general-purpose programming language. As described in Section 6.1.1, a DBMS-specific preprocessor transforms the Embedded SQL statements into function calls in the host language. The details of this translation vary across DBMSs, and therefore even though the source code can be cOlnpiled to work with different DBMSs, the final executable works only with one specific DBMS.

ODBC and JDBC, short for Open DataBase Connectivity and Java DataBase Connectivity, also enable the integration of SQL with a general-purpose programming language. Both ODBC and JDBC expose database capabilities in a standardized way to the application programmer through an application programming interface (API). In contrast to Embedded SQL, ODBC and JDBC allow a single executable to access different DBMSs *without recompilation.* Thus, while Embedded SQL is DBMS-independent only at the source code level, applications using ODBC or JDBC are DBMS-independent at the source code level and at the level of the executable. In addition, using ODBC or JDBC, an application can access not just one DBMS but several different ones simultaneously.

ODBC and JDBC achieve portability at the level of the executable by introducing an extra level of indirection. All direct interaction with a specific DBMS happens through a DBMS-specific driver. A driver is a software program that translates the ODBC or JDBC calls into DBMS-specific calls. Drivers are loaded dynamically on demand since the DBMSs the application is going to access are known only at run-time. Available drivers are registered with a driver manager.

One interesting point to note is that a driver does not necessarily need to interact with a DBMS that understands SQL. It is sufficient that the driver translates the SQL commands from the application into equivalent commands that the DBMS understands. Therefore, in the remainder of this section, we refer to a data storage subsystem with which a driver interacts as a data source.

An application that interacts with a data source through ODBC or JDBC selects a data source, dynamically loads the corresponding driver, and establishes a connection with the data source. There is no limit on the number of open connections, and an application can have several open connections to different data sources. Each connection has transaction semantics; that is, changes from one connection are visible to other connections only after the connection has committed its changes. While a connection is opcn, transactions are executed by submitting SQL statements, retrieving results, processing errors, and finally committing or rolling back. The application disconnects from the data source to terminate the interaction.

In the remainder of this chapter, we concentrate on JDBC.

JDBC Drivers: The most up-to-date source of .IDBC drivers is the Sun JDBC Driver page at
http://industry.java.sun.com/products/jdbc/drivers
JDBC drivers are available for all major database sytems.

6.2.1 Architecture

The architecture of JDBC has four main components: the *application,* the *driver manager,* several data source specific *drivers,* and the corresponding *data SOUTces.*

The *application* initiates and terminates the connection with a data source. It sets transaction boundaries, submits SQL statements, and retrieves the results-----all through a well-defined interface as specified by the JDBC API. The primary goal of the *driver manager* is to load JDBC drivers and pass JDBC function calls from the application to the correct driver. The driver manager also handles JDBC initialization and information calls from the applications and can log all function calls. In addition, the driver manager performs·some rudimentary error checking. The *driver* establishes the connection with the data source. In addition to submitting requests and returning request results, the driver translates data, error formats, and error codes from a form that is specific to the data source into the JDBC standard. The *data source* processes commands from the driver and returns the results.

Depending on the relative location of the data source and the application, several architectural scenarios are possible. Drivers in JDBC are cla.ssified into four types depending on the architectural relationship between the application and the data source:

- Type I Bridges: This type of driver translates JDBC function calls into function calls of another API that is not native to the DBMS. An example is a JDBC-ODBC bridge; an application can use JDBC calls to access an ODBC compliant data source. The application loads only one driver, the bridge. Bridges have the advantage that it is easy to piggy-back the applica.tion onto an existing installation, and no new drivers have to be installed. But using bridges has several drawbacks. The increased number of layers between data source and application affects performance. In addition, the user is limited to the functionality that the ODBC driver supports.
- Type II Direct Translation to the Native API via Non-Java Driver: This type of driver translates JDBC function calls directly into method invocations of the API of one specific data source. The driver is

usually written using a combination of C++ and Java; it is dynamically linked and specific to the data source. This architecture performs significantly better than a JDBC-ODBC bridge. One disadvantage is that the database driver that implements the API needs to be installed on each computer that runs the application.

- Type III—Network Bridges: The driver talks over a network to a middleware server that translates the JDBC requests into DBMS-specific method invocations. In this case, the driver on the client site (Le., the network bridge) is not DBMS-specific. The JDBC driver loaded by the application can be quite small, as the only functionality it needs to implement is sending of SQL statements to the middleware server. The middleware server can then use a Type II JDBC driver to connect to the data source.
- Type IV-Direct Translation to the Native API via Java Driver: Instead of calling the DBMS API directly, the driver communicates with the DBMS through Java sockets. In this case, the driver on the client side is written in Java, but it is DBMS-specific. It translates JDBC calls into the native API of the database system. This solution does not require an intermediate layer, and since the implementation is all Java, its performance is usually quite good.

6.3 JDBC CLASSES AND INTERFACES

JDBC is a collection of Java classes and interfaces that enables database access from prograrl1s written in the Java language. It contains methods for connecting to a remote data source, executing SQL statements, examining sets of results from SQL statements, transaction management, and exception handling. The classes and interfaces are part of the `java.sql` package. Thus, all code fragments in the remainder of this section should include the statement `import java.sql.*` at the beginning of the code; we omit this statement in the remainder of this section. JDBC 2.0 also includes the `javax.sql` package, the JDBC Optional Package. The package `javax.sql` adds, among other things, the capability of connection pooling and the `RowSet` interface. We discuss connection pooling in Section 6.3.2, and the `ResultSet` interface in Section 6.3.4.

We now illustrate the individual steps that are required to submit a database query to a data source and to retrieve the results.

6.3.1 JDBC Driver Management

In JDBC, data source drivers are managed by the Drivermanager class, which maintains a list of all currently loaded drivers. The Drivermanager class has

methods registerDriver, deregisterDriver, and getDrivers to enable dynamic addition and deletion of drivers.

The first step in connecting to a data source is to load the corresponding JDBC driver. This is accomplished by using the Java mechanism for dynamically loading classes. The static method forName in the Class class returns the Java class as specified in the argument string and executes its static constructor. The static constructor of the dynamically loaded class loads an instance of the Driver class, and this Driver object registers itself with the DriverManager class.

The following Java example code explicitly loads a JDBC driver:

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

There are two other ways ofregistering a driver. We can include the driver with -Djdbc. drivers=oracle/jdbc. driver at the command line when we start the Java application. Alternatively, we can explicitly instantiate a driver, but this method is used only rarely, as the name of the driver has to be specified in the application code, and thus the application becomes sensitive to changes at the driver level.

After registering the driver, we connect to the data source.

6.3.2 Connections

A session with a data source is started through creation of a Connection object; A connection identifies a logical session with a data source; multiple connections within the same Java program can refer to different data sources or the same data source. Connections are specified through a **JDBC URL**, a URL that uses the jdbc protocol. Such a URL has the form

```
jdbc:<subprotocol>:<otherParameters>
```

The code example shown in Figure 6.2 establishes a connection to an Oracle database assuming that the strings userId and password are set to valid values.

In JDBC, connections can have different properties. For example, a connection can specify the granularity of transactions. If autocommit is set for a connection, then each SQL statement is considered to be its own transaction. If autocommit is off, then a series of statements that compose a transaction can be committed using the commit 0 method of the Connection class, or aborted using the rollbackO method. The Connection class has methods to set the

```
String uri = "jdbc:oracle:www.bookstore.com:3083"
Connection connection;
try {
    Connection connection =
        DriverManager.getConnection(urI,userId,password);
}
catch(SQLException excpt) {
    System.out.println(excpt.getMessageO);
    return;
}
```

Figure 6.2 Establishing a Connection with JDBC

JDBC Connections: Remember to close connections to data sources and return shared connections to the connection pool. Database systems have a limited number of resources available for connections, and orphan connections can often only be detected through time-outs-and while the database system is waiting for the connection to time-out, the resources used by the orphan connection are wasted.

autocommit mode (Connection. setAutoCommit) and to retrieve the current autocommit mode (getAutoCommit). The following methods are part of the Connection interface and permit setting and getting other properties:

- public int getTransactionIsolation() throws SQLExceptionand public void setTransactionlsolation(int 1) throws SQLException. These two functions get and set the current level of isolation for transactions handled in the current connection. All five SQL levels of isolation (see Section 16.6 for a full discussion) are possible, and argument *1* can be set as follows:
  - TRANSACTIONJNONE
  - TRANSACTIONJREAD.UNCOMMITTED
  - TRANSACTIONJREAD.COMMITTED
  - TRANSACTIONJREPEATABLEJREAD
  - TRANSACTION.BERIALIZABLE
- public boolean getReadOnlyO throws SQLException and public void setReadOnly(boolean readOnly) throws SQLException. These two functions allow the user to specify whether the transactions executecl through this connection are rcad only.

- public boolean isClosed() throws SQLException.
  Checks whether the current connection has already been closed.
- setAutoCommit and getAutoCommit.
  We already discussed these two functions.

Establishing a connection to a data source is a costly operation since it involves several steps, such as establishing a network connection to the data source, authentication, and allocation of resources such as memory. In case an application establishes many different connections from different parties (such as a Web server), connections are often **pooled** to avoid this overhead. A **connection pool** is a set of established connections to a data source. Whenever a new connection is needed, one of the connections from the pool is used, instead of creating a new connection to the data source.

Connection pooling can be handled either by specialized code in the application, or the optional javax.sql package, which provides functionality for connection pooling and allows us to set different parameters, such as the capacity of the pool, and shrinkage and growth rates. Most application servers (see Section 7.7.2) implement the javax.sql package or a proprietary variant.

6.3.3 Executing SQL Statements

We now discuss how to create and execute SQL statements using JDBC. In the JDBC code examples in this section, we assume that we have a Connection object named con. JDBC supports three different ways of executing statements: Statement, PreparedStatement, and CallableStatement. The Statement class is the base class for the other two statment classes. It allows us to query the data source with any static or dynamically generated SQL query. We cover the PreparedStatement class here and the CallableStatement class in Section 6.5, when we discuss stored procedures.

The PreparedStatement class dynamicaJly generates precompiled SQL statements that can be used several times; these SQL statements can have parameters, but their structure is fixed when the PreparedStatement object (representing the SQL statement) is created.

Consider the sample code using a PreparedStatment object shown in Figure 6.3. The SQL query specifies the query string, but uses '?' for the values of the parameters, which are set later using methods setString, setFloat, and setInt. The "1' placeholders can be used anywhere in SQL statements where they can be replaced with a value. Examples of places where they can appear include the WHERE clause (e.g., 'WHERE author=?'), or in SQL UPDATE and INSERT statements, as in Figure 6.3. The method setString is one way

```
// initial quantity is always zero
String sql = "INSERT INTO Books VALUES(?, 7, ?, ?, 0, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);

// now instantiate the parameters with values
// assume that isbn, title, etc. are Java variables that
// contain the values to be inserted
pstmt.clearParameters();
pstmt.setString(l, isbn);
pstmt.setString(2, title);
pstmt.setString(3, author);
pstmt.setFloat(5, price);
pstmt.setInt(6, year);

int numRows = pstmt.executeUpdate();
```

Figure 6.3 SQL Update Using a PreparedStatement Object

to set a parameter value; analogous methods are available for int, float, and date. It is good style to always use clearParameters 0 before setting parameter values in order to remove any old data.

There are different ways of submitting the query string to the data source. In the example, we used the executeUpdate command, which is used if we know that the SQL statement does not return any records (SQL UPDATE, INSERT, ALTER, and DELETE statements). The executeUpdate method returns an integer indicating the number of rows the SQL statement modified; it returns 0 for successful execution without modifying any rows.

The executeQuery method is used if the SQL statement returns data, such as in a regular SELECT query. JDBC has its own cursor mechanism in the form of a ResultSet object, which we discuss next. The execute method is more general than executeQuery and executeUpdate; the references at the end of the chapter provide pointers with more details.

6.3.4 ResultSets

As discussed in the previous section, the statement executeQuery returns a ResultSet object, which is similar to a cursor. ResultSet cursors in JDBC 2.0 are very powerful; they allow forward and reverse scrolling and in-place editing and insertions.

In its most basic form, the ResultSet object allows us to read one row of the output of the query at a time. Initially, the ResultSet is positioned before the first row, and we have to retrieve the first row with an explicit call to the next() method. The next method returns false if there are no more rows in the query answer, and true other\vise. The code fragment shown in Figure 6.4 illustrates the basic usage of a ResultSet object.

```
ResultSet rs=stmt.executeQuery(sqlQuery);
// rs is now a cursor
// first call to rs.nextO moves to the first record
// rs.nextO moves to the next row
String sqlQuery;
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next()) {
    // process the data
}
```

Figure 6.4 Using a ResultSet Object

While next() allows us to retrieve the logically next row in the query answer, we can move about in the query answer in other ways too:

- previous() moves back one row.
- absolute(int num) moves to the row with the specified number.
- relative(int num) moves forward or backward (if num is negative) relative to the current position. relative(-1) has the same effect as previous.
- first() moves to the first row, and last() moves to the last row.

Matching Java and SQL Data Types

In considering the interaction of an application with a data source, the issues we encountered in the context of Embedded SQL (e.g., passing information between the application and the data source through shared variables) arise again. To deal with such issues, JDBC provides special data types and specifies their relationship to corresponding SQL data types. Figure 6.5 shows the accessor methods in a ResultSet object for the most common SQL datatypes. With these accessor methods, we can retrieve values from the current row of the query result referenced by the ResultSet object. There are two forms for each accessor method: One method retrieves values by column index, starting at one, and the other retrieves values by column name. The following example shows how to access fields of the current ResultSet row using accesssor methods.

SQL Type	Java class	ResultSet get method
BIT	Boolean	getBooleanO
CHAR	String	getStringO
VARCHAR	String	getStringO
DOUBLE	Double	getDoubleO
FLOAT	Double	getDoubleO
INTEGER	Integer	getIntO
REAL	Double	getFloatO
DATE	java.sql.Date	getDateO
TIME	java.sql.Time	getTimeO
TIMESTAMP	java.sql.TimeStamp	getTimestamp()

Figure 6.5 Reading SQL Datatypes from a ResultSet Object

```
ResultSet rs=stmt.executeQuery(sqIQuery);
String sqlQuerYi
ResultSet rs = stmt.executeQuery(sqIQuery)
while (rs.nextO) {
    isbn = rs.getString(l);
    title = rs.getString("TITLE");
    // process isbn and title
}
```

6.3.5 Exceptions and Warnings

Similar to the SQLSTATE variable, most of the methods in java. sql can throw an exception of the type SQLException if an error occurs. The information includes SQLState, a string that describes the error (e.g., whether the statement contained an SQL syntax error). In addition to the standard getMessage 0 method inherited from Throwable, SQLException has two additional methods that provide further information, and a method to get (or chain) additional exceptions:

- public String getSQLState 0 returns an SQLState identifier based on the SQL:1999 specification, as discussed in Section 6.1.1.
- public int getErrorCode () retrieves a vendor-specific error code.
- public SQLException getNextExceptionO gets the next exception in a chain of exceptions associated with the current SQLException object.

An SQL\¥arning is a subclass of SQLException. Warnings are not as severe as errors and the program can usually proceed without special handling of warnings. \Varnings are not thrown like other exceptions, and they are not caught as

part of the try"-catch block around a java.sql statement. We Heed to specifically test whether warnings exist. Connection, Statement, and ResultSet objects all have a getWarnings 0 method with which we can retrieve SQL warnings if they exist. Duplicate retrieval of warnings can be avoided through clearWarnings O. Statement objects clear warnings automatically on execution of the next statement; ResultSet objects clear warnings every time a new tuple is accessed.

Typical code for obtaining SQLWarnings looks similar to the code shown in Figure 6.6.

```
try {
    stmt = con.createStatement();
    warning = con.getWarnings();
    while( warning != null) {
        // handleSQLWarnings          //code to process warning
        warning = warning.getNextWarningO;      //get next warning
    }
    con.clear\Varnings();

    stmt.executeUpdate( queryString );
    warning = stmt.getWarnings();
    while( warning != null) {
        // handleSQLWarnings          //code to process warning
        warning = warning.getNextWarningO;      //get next warning
    }
} // end try
catch ( SQLException SQLe) {
    // code to handle exception
} // end catch
```

Figure 6.6 Processing JDBC Warnings and Exceptions

6.3.6 Examining Database Metadata

We can use the DatabaseMetaData object to obtain information about the database system itself, as well as information frorn the database catalog. For example, the following code fragment shows how to obtain the name and driver version of the JDBC driver:

```
DatabaseMetaData md = con.getMetaD<Lta():

System.out.println("Driver Information:");
```

```
System.out.println("Name:" + md.getDriverNameO
            + "; version:" + mcl.getDriverVersion());
```

The DatabaseMetaData object has many more methods (in JDBC 2.0, exactly 134); we list some methods here:

- public ResultSet getCatalogs0 throws SqLException. This function returns a ResultSet that can be used to iterate over all the catalog relations. The functions getIndexInfo0 and getTables0 work analogously.
- pUblic int getMaxConnections0 throws SqLException. This function returns the ma.ximum number of connections possible.

We will conclude our discussion of JDBC with an example code fragment that examines all database metadata shown in Figure 6.7.

```
DatabaseMetaData dmd = con.getMetaDataO;
ResultSet tablesRS = dmd.getTables(null,null,null,null);
string tableName;

while(tablesRS.next()) {
    tableNarne = tablesRS.getString("TABLE_NAME");

    // print out the attributes of this table
    System.out.println("The attributes of table"
            + tableName + " are:");
    ResultSet columnsRS = dmd.getColums(null,null,tableName, null);
    while (columnsRS.next()) {
        System.out.print(columnsRS.getString("COLUMN_NAME")
                +" ");
    }

    // print out the primary keys of this table
    System.out.println("The keys of table" + tableName + " are:");
    ResultSet keysRS = dmd.getPrimaryKeys(null,null,tableName);
    while (keysRS.next()) {
        'System.out.print(keysRS.getStringC'COLUMN_NAME") +" ");
    }
}
```

Figure 6.7 Obtaining Infon-nation about a Data Source

6.4 SQLJ

SQLJ (short for 'SQL-Java') was developed by the SQLJ Group, a group of database vendors and Sun. SQLJ was developed to complement the dynamic way of creating queries in JDBC with a static model. It is therefore very close to Embedded SQL. Unlike JDBC, having semi-static SQL queries allows the compiler to perform SQL syntax checks, strong type checks of the compatibility of the host variables with the respective SQL attributes, and consistency of the query with the database schema-tables, attributes, views, and stored procedures—all at compilation time. For example, in both SQLJ and Embedded SQL, variables in the host language always are bound statically to the same arguments, whereas in JDBC, we need separate statements to bind each variable to an argument and to retrieve the result. For example, the following SQLJ statement binds host language variables title, price, and author to the return values of the cursor books.

```
#sql books = {
    SELECT title, price INTO :title, :price
    FROM Books WHERE author = :author
};
```

In JDBC, we can dynamically decide which host language variables will hold the query result. In the following example, we read the title of the book into variable ftitle if the book was written by Feynman, and into variable otitle otherwise:

```
// assume we have a ResultSet cursor rs
author = rs.getString(3);

if (author=="Feynman") {
    ftitle = rs.getString(2):
}
else {
    otitle = rs.getString(2);
}
```

When writing SQLJ applications, we just write regular Java code and embed SQL statements according to a set of rules. SQLJ applications are pre-processed through an SQLJ translation program that replaces the embedded SQLJ code with calls to an SQLJ Java library. The modified program code can then be compiled by any Java compiler. Usually the SQLJ Java library makes calls to a JDBC driver, which handles the connection to the database system.

An important philosophical difference exists between Embedded SQL and SQLJ and JDBC. Since vendors provide their own proprietary versions of SQL, it is advisable to write SQL queries according to the SQL-92 or SQL:1999 standard. However, when using Embedded SQL, it is tempting to use vendor-specific SQL constructs that offer functionality beyond the SQL-92 or SQL:1999 standards. SQLJ and JDBC force adherence to the standards, and the resulting code is much more portable across different database systems.

In the remainder of this section, we give a short introduction to SQLJ.

6.4.1 Writing SQLJ Code

We will introduce SQLJ by means of examples. Let us start with an SQLJ code fragment that selects records from the Books table that match a given author.

```
String title; Float price; String atithor;
#sql iterator Books (String title, Float price);
Books books;

// the application sets the author
// execute the query and open the cursor
#sql books = {
    SELECT title, price INTO :titIe, :price
    FROM Books WHERE author = :author
};

// retrieve results
while (books.next()) {
    System.out.println(books.titleO + ", " + books.price());
}
books.close();
```

The corresponding JDBC code fragment looks as follows (assuming we also declared price, name, and author:

```
PrcparcdStatcment stmt = connection.prepareStatement(
"SELECT title, price FROM Books WHERE author = ?");

// set the parameter in the query ancl execute it
stmt.setString(1, author);
ResultSet 1'8 = stmt.executeQuery();

// retrieve the results
while (rs.next()) {
```

```
System.out.println(rs.getString(l) + ", " + rs.getFloat(2));
}
```

Comparing the JDBC and SQLJ code, we see that the SQLJ code is much easier to read than the JDBC code. Thus, SQLJ reduces software development and maintenance costs.

Let us consider the individual components of the SQLJ code in more detail. All SQLJ statements have the special prefix #sql. In SQLJ, we retrieve the results of SQL queries with iterator objects, which are basically cursors. An iterator is an instance of an iterator class. Usage of an iterator in SQLJ goes through five steps:

- Declare the Iterator Class: In the preceding code, this happened through the statement
  #sql iterator Books (String title, Float price);
  This statement creates a new Java class that we can use to instantiate objects.
- Instantiate an Iterator Object from the New Iterator Class: We instantiated our iterator in the statement Books books;.
- Initialize the Iterator Using a SQL Statement: In our example, this happens through the statement #sql books = ...
- Iteratively, Read the Rows From the Iterator Object: This step is very similar to reading rows through a ResultSet object in JDBC.
- Close the Iterator Object.

There are two types of iterator classes: named iterators and positional iterators. For named iterators, we specify both the variable type and the name of each column of the iterator. This allows us to retrieve individual columns by name as in our previous example where we could retrieve the title colunm from the Books table using the expression books.titIe(). For positional iterators, we need to specify only the variable type for each column of the iterator. To access the individual columns of the iterator, we use a FETCH ... INTO eonstruct, similar to Embedded SQL. Both iterator types have the same performance; which iterator to use depends on the programmer's taste.

Let us revisit our example. \Ve can make the iterator a positional iterator through the following statement:

```
#sql iterator Books (String, Float);
```

We then retrieve the individual rows from the iterator as follows:

```
while (true) {
    #sql { FETCH :books INTO :title, :price, };
    if (books.endFetch()) {
        break:
    }

    // process the book
}
```

6.5 STORED PROCEDURES

It is often important to execute some parts of the application logic directly in the process space of the database system. Running application logic directly at the database has the advantage that the amount of data that is transferred between the database server and the client issuing the SQL statement can be minimized, while at the same time utilizing the full power of the database server.

When SQL statements are issued from a remote application, the records in the result of the query need to be transferred from the database system back to the application. If we use a cursor to remotely access the results of an SQL statement, the DBMS has resources such as locks and memory tied up while the application is processing the records retrieved through the cursor. In contrast, a stored procedure is a program that is executed through a single SQL statement that can be locally executed and completed within the process space of the database server. The results can be packaged into one big result and returned to the application, or the application logic can be performed directly at the server, without having to transmit the results to the client at all.

Stored procedures are also beneficial for software engineering reasons. Once a stored procedure is registered with the database server, different users can re-use the stored procedure, eliminating duplication of efforts in writing SQL queries or application logic, and making code maintenance easy. In addition, application programmers do not need to know the the database schema if we encapsulate all database access into stored procedures.

Although they are called stored *procedures*, they do not have to be procedures in a programming language sense; they can be functions.

6.5.1 Creating a Simple Stored Procedure

Let us look at the example stored procedure written in SQL shown in Figure 6.8. We see that stored procedures must have a name; this stored procedure

has the name 'ShowNumberOfOrders.' Otherwise, it just contains an SQL statement that is precompiled and stored at the server.

```
CREATE PROCEDURE ShowNumberOfOrders
SELECT C.cid, C.cname, COUNT(*)
       FROM    Customers C, Orders a
       WHERE   C.cid = O.cid
       GROUP BY C.cid, C.cname
```

Figure 6.8 A Stored Procedure in SQL

Stored procedures can also have parameters. These parameters have to be valid SQL types, and have one of three different modes: IN, OUT, or INOUT. IN parameters are arguments to' the stored procedure. OUT parameters are returned from the stored procedure; it assigns values to all OUT parameters that the user can process. INOUT parameters combine the properties of IN and OUT parameters: They contain values to be passed to the stored procedures, and the stored procedure can set their values as return values. Stored procedures enforce strict type conformance: If a parameter is of type INTEGER, it cannot be called with an argument of type VARCHAR.

Let us look at an example of a stored procedure with arguments. The stored procedure shown in Figure 6.9 has two arguments: book_isbn and addedQty. It updates the available number of copies of a book with the quantity from a new shipment.

```
CREATE PROCEDURE AddInventory (
                 IN book_isbn CHAR(lO),
                 IN addedQty INTEGER)
UPDATE Books
       SET     qty_in_stock = qtyjn_stock + addedQty
       WHERE   bookjsbn = isbn
```

Figure 6.9 A Stored Procedure with Arguments

Stored procedures do not have to be written in SQL; they can be written in any host language. As an example, the stored procedure shown in Figure 6.10 is a Java function that is dynamically executed by the database server whenever it is called by the dient:

6.5.2 Calling Stored Procedures

Stored procedures can be called in interactive SQL with the CALL statement:

```
CREATE PROCEDURE RallkCustomers(IN number INTEGER)
LANGUAGE Java
EXTERNAL NAME 'file:///c:/storedProcedures/rank.jar'
```

Figure 6.10 A Stored Procedure in Java

```
CALL storedProcedureName(argumentl, argument2, ... , argumentN);
```

In Embedded SQL, the arguments to a stored procedure are usually variables in the host language. For example, the stored procedure AddInventory would be called as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char isbn[lO];
long qty;
EXEC SQL END DECLARE SECTION

// set isbn and qty to some values
EXEC SQL CALL AddInventory(:isbn,:qty);
```

Calling Stored Procedures from JDBC

We can call stored procedures from JDBC using the CallableStatment class. CallableStatement is a subclass of PreparedStatement and provides the same functionality. A stored procedure could contain multiple SQL staternents or a series of SQL statements-thus, the result could be many different ResultSet objects. We illustrate the case when the stored procedure result is a single ResultSet.

```
CallableStatement cstmt=
            con.prepareCall("{call ShowNumberOfOrders}");
ResultSet rs = cstmt.executeQueryO
while (rs.next())
```

Calling Stored Procedures from SQLJ

The stored procedure 'ShowNumberOfOrders' is called as follows using SQLJ:

```
// create the cursor class
#sql !terator CustomerInfo(int cid, String cname, int count);

// create the cursor
```

```
CustomerInfo customerinfo;

// call the stored procedure
#sql customerinfo = {CALL ShowNumberOfOrders};
while (customerinfo.nextO) {
    System.out.println(customerinfo.cid() + "," +
                customerinfo.count());
}
```

6.5.3 SQLIPSM

All major database systems provide ways for users to write stored procedures in a simple, general purpose language closely aligned with SQL. In this section, we briefly discuss the SQL/PSM standard, which is representative of most vendor-specific languages. In PSM, we define modules, which are collections of stored procedures, temporary relations, and other declarations.

In SQL/PSM, we declare a stored procedure as follows:

```
CREATE PROCEDURE name (parameter1,..., parameterN)
    local variable declarations
    procedure code;
```

We can declare a function similarly as follows:

```
CREATE FUNCTION name (parameterl,..., parameterN)
            RETURNS sqIDataType
    local variable declarations
    function code;
```

Each parameter is a triple consisting of the mode (IN, OUT, or INOUT as discussed in the previous section), the parameter name, and the SQL datatype of the parameter. We can seen very simple SQL/PSM procedures in Section 6.5.1. In this case, the local variable declarations were empty, and the procedure code consisted of an SQL query.

We start out with an example of a SQL/PSM function that illustrates the main SQL/PSM constructs. The function takes as input a customer identified by her *cid* and a year. The function returns the rating of the customer, which is defined as follows: Customers who have bought more than ten books during the year are rated 'two'; customer who have purchased between 5 and 10 books are rated 'one', otherwise the customer is rated 'zero'. The following SQL/PSM code computes the rating for a given customer and year.

```
CREATE PROCEDURE RateCustomer
```

```
                    (IN custId INTEGER, IN year INTEGER)
                    RETURNS INTEGER
DECLARE rating INTEGER;
DECLARE numOrders INTEGER;
SET numOrders =
        (SELECT COUNT(*) FROM Orders 0 WHERE O.tid = custId);
IF (numOrders> 10) THEN rating=2;
ELSEIF (numOrders>5) THEN rating=1;
ELSE rating=O;
END IF;
RETURN rating;
```

Let us use this example to give a short overview of some SQL/PSM constructs:

- We can declare local variables using the DECLARE statement. In our example, we declare two local variables: 'rating', and 'numOrders'.
- PSM/SQL functions return values via the RETURN statement. In our example, we return the value of the local variable 'rating'.
- We can assign values to variables with the SET statement. In our example, we assigned the return value of a query to the variable 'numOrders'.
- SQL/PSM has branches and loops. Branches have the following form:

```
IF (condition) THEN statements;
ELSEIF statements;

ELSEIF statements;
ELSE statements; END IF
```

  Loops are of the form

```
LOOP
        staternents:
END LOOP
```

- Queries can be used as part of expressions in branches; queries that return a single value can be assigned to variables as in our example above.
- 'We can use the same cursor statements as in Embedded SQL (OPEN, FETCH, CLOSE), but we do not need the EXEC SQL constructs, and variables do not have to be prefixed by a colon ':'.

We only gave a very short overview of SQL/PSM; the references at the end of the chapter provide more information.

6.6 CASE STUDY: THE INTERNET BOOK SHOP

DBDudes finished logical database design, as discussed in Section 3.8, and now consider the queries that they have to support. They expect that the application logic will be implemented in Java, and so they consider JDBC and SQLJ as possible candidates for interfacing the database system with application code.

Recall that DBDudes settled on the following schema:

Books(*isbn:* CHAR(10), *title:* CHAR(8), *author:* CHAR(80),
qty_in_stock: INTEGER, *price:* REAL, *year_published:* INTEGER)
Customers(*cid:* INTEGER, *cname:* CHAR(80), *address:* CHAR(200))
Orders(*ordernum:* INTEGER, *isbn:* CHAR(lO), *cid:* INTEGER,
cardnum: CHAR(l6), *qty:* INTEGER, *order_date:* DATE, *ship_date:* DATE)

Now, DBDudes considers the types of queries and updates that will arise. They first create a list of tasks that will be performed in the application. Tasks performed by customers include the following.

- Customers search books by author name, title, or ISBN.
- Customers register with the website. Registered customers might want to change their contact information. DBDudes realize that they have to augment the Customers table with additional information to capture login and password information for each customer; we do not discuss this aspect any further.
- Customers check out a final shopping basket to complete a sale.
- Customers add and delete books from a 'shopping basket' at the website.
- Customers check the status of existing orders and look at old orders.

Administrative tasks performed by employees of B&N are listed next.

- Employees look up customer contact information.
- Employees add new books to the inventory.
- Employees fulfill orders, and need to update the shipping date of individual books.
- Employees analyze the data to find profitable customers and customers likely to respond to special marketing campaigns.

Next, DBDudes consider the types of queries that will arise out of these tasks. To support searching for books by name, author, title, or ISBN, DBDudes decide to write a stored procedure as follows:

```
CREATE PROCEDURE SearchByISBN (IN book.isbn CHAR(10))
    SELECT B.title, B.author, B.qty_in_stock, B.price, B.yeaLpublished
    FROM   Books B
    WHERE  B.isbn = book.isbn
```

Placing an order involves inserting one or more records into the Orders table. Since DBDudes has not yet chosen the Java-based technology to program the application logic, they assume for now that the individual books in the order are stored at the application layer in a Java array. To finalize the order, they write the following JDBC code shown in Figure 6.11, which inserts the elements from the array into the Orders table. Note that this code fragment assumes several Java variables have been set beforehand.

```
String sql = "INSERT INTO Orders VALUES(7, 7, 7, 7, 7, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);
con.setAutoCommit(false);

try {
    // orderList is a vector of Order objects
    // ordernum is the current order number
    // dd is the ID of the customer, cardnum is the credit card number
    for (int i=O; iiorderList.lengthO; i++)
        // now instantiate the parameters with values
        Order currentOrder = orderList[i];
        pstmt.clearParameters();
        pstmt.setInt(l, ordernum);
        pstmt.setString(2, Order.getlsbnO);
        pstmt.setInt(3, dd);
        pstmt.setString(4, creditCardNum);
        pstmt.setlnt(5, Order.getQtyO);
        pstmt.setDate(6, null);

        pstmt.executeUpdate();
    }
    con.commit();
catch (SqLException e){
    con.rollbackO;
    System.out.println(e.getMessage());
}
```

Figure 6.11 Inserting a Completed Order into the Database

DBDudes writes other JDBC code and stored procedures for all of the remaining tasks. They use code similar to some of the fragments that we have seen in this chapter.

- Establishing a connection to a database, as shown in Figure 6.2.
- Adding new books to the inventory, as shown in Figure 6.3.
- Processing results from SQL queries as shown in Figure 6.4-
- For each customer, showing how many orders he or she has placed. We showed a sample stored procedure for this query in Figure 6.8.
- Increasing the available number of copies of a book by adding inventory, as shown in Figure 6.9.
- Ranking customers according to their purchases, as shown in Figure 6.10.

DBDudcs takes care to make the application robust by processing exceptions and warnings, as shown in Figure 6.6.

DBDudes also decide to write a trigger, which is shown in Figure 6.12. Whenever a new order is entered into the Orders table, it is inserted with ship_date set to NULL. The trigger processes each row in the order and calls the stored procedure 'UpdateShipDate'. This stored procedure (whose code is not shown here) updates the (anticipated) ship_date of the new order to 'tomorrow', in case qtyjlLstock of the corresponding book in the Books table is greater than zero. Otherwise, the stored procedme sets the ship_date to two weeks.

```
CREATE TRIGGER update_ShipDate
        AFTER INSERT ON Orders                    /* Event */
  FOR EACH ROW
  BEGIN CALL UpdatcShipDate(new); END             /* Action */
```

Figure 6.12 Trigger to Update the Shipping Date of New Orders

6.7 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- Why is it not straightforward to integrate SQL queries with a host programming language? (Section 6.1.1)
- How do we declare variables in Ernbcdded SQL? (Section 6.1.1)

7.5 THE THREE-TIER APPLICATION ARCHITECTURE

In this section, we discuss the overall architecture of data-intensive Internet applications. Data-intensive Internet applications can be understood in terms of three different functional components: *data management, application logic,* and *pTesentation.* The component that handles data mallgement usually utilizes a DBMS for data storage, but application logic and presentation involve much more than just the DBMS itself.

We start with a short overview of the history of database-backed application architectures, and introduce single-tier and client-server architectures in Section 7.5.1. We explain the three-tier architecture in detail in Section 7.5.2, and show its advantages in Section 7.5.3.

7.5.1 Single-Tier and Client-Server Architectures

In this section, we provide some perspective on the three-tier architecture by discussing single-tier and client-server architectures, the predecessors of the three-tier architecture. Initially, data-intensive applications were combined into a single tier, including the DBMS, application logic, and user interface, as illustrated in Figure 7.5. The application typically ran on a mainframe, and users accessed it through *dumb teT'minals* that could perform only data input and display. This approach has the benefit of being easily maintained by a central administrator.

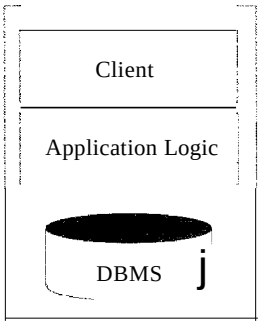

Figure 7.5 A Single-Tier Architecture

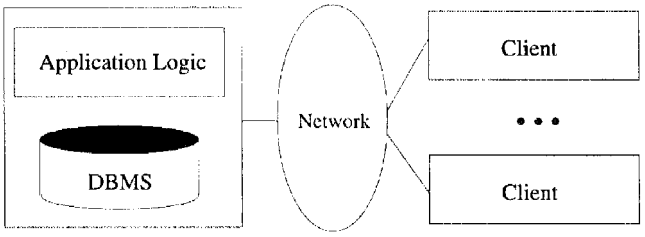

Figure 7.6 A Two-Server Architecture: Thin Clients

Single-tier architectures have a,n important drawback: Users expect graphical interfaces that require much more computational power than simple dumb terminals. Centralized computation of the graphical displays of such interfaces requires much more computational power than a single server has available, and thus single-tier architectures do not scale to thousands of users. The commoditization of the PC and the availability of cheap client computers led to the developlnent of the two-tier architecture.

Two-tier architectures, often also referred to as client-server architectures, consist of a client computer and a server computer, which interact through a well-defined protocol. What part of the functionality the client implements, and what part is left to the server, can vary. In the traditional client-server architecture, the client implements just the graphical user interface, and the server. implements both the business logic and the data management; such clients are often called thin clients, and this architecture is illustra,ted in Figure 7.6.

Other divisions are possible, such as more powerful clients that implement both user interface and business logic, or clients that implement user interface and part of the business logic, with the remaining part being implemented at the

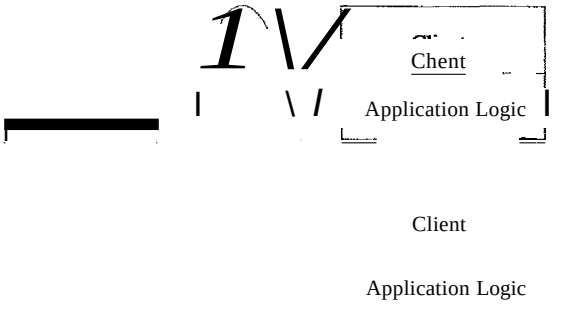

Figure 7.7 A Two-Tier Architecture: Thick Clients

server level; such clients are often called **thick** clients, and this architecture is illustrated in Figure 7.7.

Compared to the single-tier architecture, two-tier architectures physically separate the user interface from the data management layer. To implement two-tier architectures, we can no longer have dumb terminals on the client side; we require computers that run sophisticated presentation code (and possibly, application logic).

Over the last ten years, a large number of client-server development tools such Microsoft Visual Basic and Sybase Powerbuilder have been developed. These tools permit rapid development of client-server software, contributing to the success of the client-server model, especially the thin-client version.

The thick-client model has several disadvantages when compared to the thin-client model. First, there is no central place to update and maintain the business logic, since the application code runs at many client sites. Second, a large amount of trust is required between the server and the clients. As an exam-- pIe, the DBMS of a bank has to trust the (application executing at an) ATM machine to leave the database in a consistent state. (One way to address this problem is through *stored procedures*, trusted application code that is registered with the DBMS and can be called from SQL statelnents. We discuss stored procedures in detail in Section 6.5.)

A third disadvantage of the thick-client architecture is that it does not scale with the number of clients; it typically cannot handle more than a few hundred clients. The application logic at the client issues SQL queries to the server and the server returns the query result to the client, where further processing takes place. Large query results might be transferred between client and server.

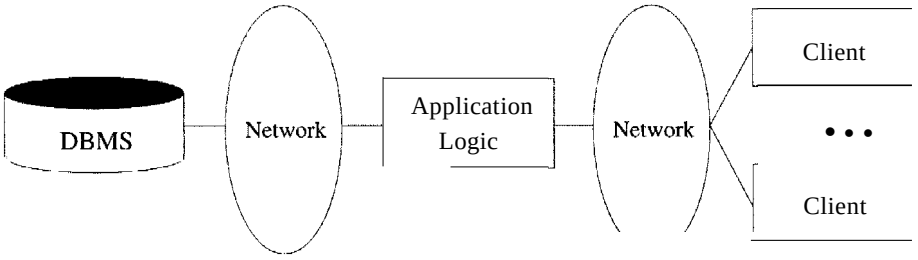

Figure 7.8 A Standard Three-Tier Architecture

(Stored procedures can mitigate this bottleneck.) Fourth, thick-client systems do not scale as the application accesses more and more database systems. Assume there are x different database systems that are accessed by y clients, then there are $x \cdot y$ different connections open at any time, clearly not a scalable solution.

These disadvantages of thick-client systems and the widespread adoption of standard, very thin clients—notably, Web browsers—have led to the widespread use thin-client architectures.

7.5.2 Three-Tier Architectures

The thin-client two-tier architecture essentially separates presentation issues from the rest of the application. The three-tier architecture goes one step further, and also separates application logic from data management:

- Presentation Tier: Users require a natural interface to make requests, provide input, and to see results. The widespread use of the Internet has made Web-based interfaces increasingly popular.
- Middle Tier: The application logic executes here. An enterprise-class application reflects complex business processes, and is coded in a general purpose language such as C++ or Java.
- Data Management Tier: Data-intensive Web applications involve DBMSs, which are the subject of this book.

Figure 7.8 shows a basic three-tier architecture. Different technologies have been developed to enable distribution of the three tiers of an application across multiple hardware platforms and different physical sites. Figure 7.9 shows the technologies relevant to each tier.

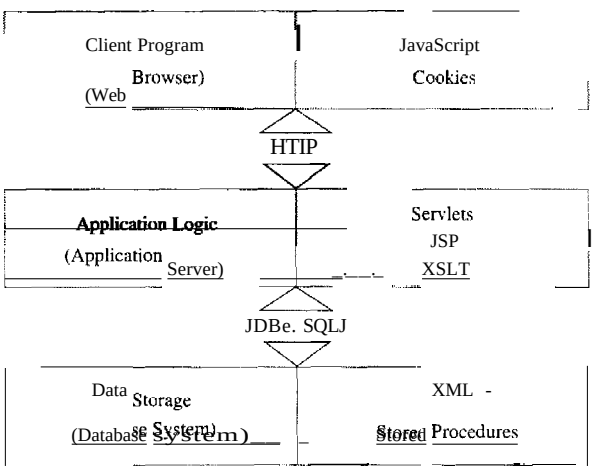

Figure 7.9 Technologies for the Three Tiers

Overview of the Presentation Tier

At the presentation layer, we need to provide forms through which the user can issue requests, and display responses that the middle tier generates. The hypertext markup language (HTML) discussed in Section 7.3 is the basic data presentation language.

It is important that this layer of code be easy to adapt to different display devices and formats; for example, regular desktops versus handheld devices versus cell phones. This adaptivity can be achieved either at the middle tier through generation of different pages for different types of client, or directly at the client through style sheets that specify how the data should be presented. In the latter case, the middle tier is responsible for producing the appropriate data in response to user requests, whereas the presentation layer decides *how* to display that information.

We cover presentation tier technologies, including style sheets, in Section 7.6.

Overview of the Middle Tier

The middle layer runs code that implements the business logic of the application: It controls what data needs to be input before an action can be executed, determines the control flow between multi-action steps, controls access to the database layer, and often assembles dynamically generated HTML pages from database query results.

The middle tier code is responsible for supporting all the different roles involved in the application. For example, in an Internet shopping site implementation, we would like customers to be able to browse the catalog and make purchases, administrators to be able to inspect current inventory, and possibly data analysts to ask summary queries about purchase histories. Each of these roles can require support for several complex actions.

For example, consider the a customer who wants to buy an item (after browsing or searching the site to find it). Before a sale can happen, the customer has to go through a series of steps: She has to add items to her shopping basket, she has to provide her shipping address and credit card number (unless she has an account at the site), and she has to finally confirm the sale with tax and shipping costs added. Controlling the flow among these steps and remembering already executed steps is done at the middle tier of the application. The data carried along during this series of steps might involve database accesses, but usually it is not yet permanent (for example, a shopping basket is not stored in the database until the sale is confirmed).

We cover the middle tier in detail in Section 7.7.

7.5.3 Advantages of the Three-Tier Architecture

The three-tier architecture has the following advantages:

- Heterogeneous Systems: Applications can utilize the strengths of different platforms and different software components at the different tiers. It is easy to modify or replace the code at any tier without affecting the other tiers.
- **Thin** Clients: Clients only need enough computation power for the presentation layer. Typically, clients are Web browsers.
- Integrated Data Access: In many applications, the data must be accessed from several sources. This can be handled transparently at the middle tier, where we can centrally manage connections to all database systems involved.
- Scalabilit,y to Many Clients: Each client is lightweight and all access to the system is through the middle tier. The middle tier can share database connections across clients, and if the middle tier becomes the bottle-neck, we can deploy several servers executing the middle tier code; clients can connect to anyone of these servers, if the logic is designed appropriately. This is illustrated in Figure 7.10, which also shows how the middle tier accesses multiple data sources. Of course, we rely upon the DBMS for each

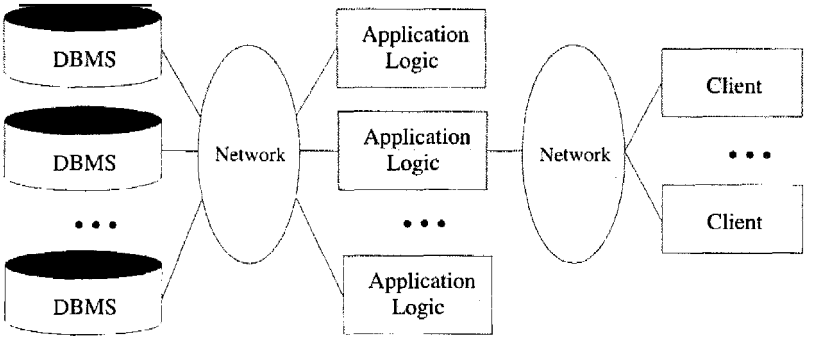

Figure 7.10 Middle-Tier Replication and Access to Multiple Data Sources

data source to be scalable (and this might involve additional parallelization or replication, as discussed in Chapter 22).

- Software Development Benefits: By dividing the application cleanly into parts that address presentation, data access, and business logic, we gain many advantages. The business logic is centralized, and is therefore easy to maintain, debug, and change. Interaction between tiers occurs through well-defined, standardized APIs. Therefore, each application tier can be built out of reusable components that can be individually developed, debugged, and tested.

7.6 THE PRESENTATION LAYER

In this section, we describe technologies for the client side of the three-tier architecture. We discuss HTML forms as a special means of passing arguments from the client to the middle tier (i.e., from the presentation tier to the middle tier) in Section 7.6.1. In Section 7.6.2, we introduce JavaScript, a Java-based scripting language that can be used for light-weight computation in the client tier (e.g., for simple animations). We conclude our discussion of client-side technologies by presenting style sheets in Section 7.6.3. Style sheets are languages that allow us to present the same webpage with different formatting for clients with different presentation capabilities; for example, Web browsers versus cell phones, or even a Netscape browser versus Microsoft's Internet Explorer.

7.6.1 HTML Forms

HTML forms are a common way of communicating data from the client tier to the middle tier. The general format of a form is the following:

```
<FORM ACTION="page.jsp" METHOD="GET" NAME="LoginForm">
```

```
</FORM>
```

A single HTML document can contain more than one form. Inside an HTML form, we can have any HTML tags except another FORM element.

The FORM tag has three important attributes:

- ACTION: Specifies the URI of the page to which the form contents are submitted; if the ACTION attribute is absent, then the URI of the current page is used. In the sample above, the form input would be submited to the page named page. j sp, which should provide logic for processing the input from the form. (We will explain methods for reading form data at the middle tier in Section 7.7.)
- METHOD: The HTTP/1.0 method used to submit the user input from the filled-out form to the webserver. There are two choices, GET and POST; we postpone their discussion to the next section.
- NAME: This attribute gives the form a name. Although not necessary, naming forms is good style. In Section 7.6.2, we discuss how to write client-side programs in JavaScript that refer to forms by name and perform checks on form fields.

Inside HTML forms, the INPUT, SELECT, and TEXTAREA tags are used to specify user input elements; a form can have many elements of each type. The simplest user input element is an INPUT field, a standalone tag with no terminating tag. An example of an INPUT tag is the following:

```
<INPUT TYPE=ltext" NAME="title">
```

The INPUT tag has several attributes. The three most important ones are TYPE, NAME, and VALUE. The TYPE attribute determines the type of the input field. If the TYPE attribute has value **text**, then the field is a text input field. If the TYPE attribute has value password, then the input field is a text field where the entered characters are displayed as stars on the screen. If the TYPE attribute has value **reset**, it is a simple button that resets all input fields within the form to their default values. If the TYPE attribute has value submit, then it is a button that sends the values of the different input fields in the form to the server. Note that **reset** and submit input fields affect the entire form.

The NAME attribute of the INPUT tag specifies the symbolic name for this field and is used to identify the value of this input fi.eld when it is sent to the server. NAME has to be set for INPUT tags of all types except submit and **reset**. In the preceding example, we specified **title** as the NAME of the input field.

The VALUE attribute of an input tag can be used for text or password fields to specify the default contents of the field. For submit or reset buttons, VALUE determines the label of the button.

The form in Figure 7.11 shows two text fields, one regular text input field and one password field. It also contains two buttons, a reset button labeled 'Reset Values' and a submit button labeled 'Log on.' Note that the two input fields are named, whereas the reset and submit button have no NAME attributes.

```
<FORM ACTION="page.jsp" METHoD="GET" NAME="LoginForm">
    <INPUT TYPE="text" NAME="username" VALUE=" Joe"><P>
    <INPUT TYPE="password" NAME="p&ssword"><P>
    <INPUT TYPE="reset" VALUE="Reset Values"><P>
    <INPUT TYPE="submit" VALUE="Log on">
</FoRM>
```

Figure 7.11 HTl'vlL Form with Two Text Fields and Two Buttons

HTML forms have other ways of specifying user input, such as the aforementioned TEXTAREA and SELECT tags; we do not discuss them.

Passing Arguments to Server-Side Scripts

As mentioned at the beginning of Section 7.6.1, there are two different ways to submit HTML Form data to the webserver. If the method GET is used, then the contents of the form are assembled into a query URI (as discussed next) and sent to the server. If the method POST is used, then the contents of the form are encoded as in the GET method, but the contents are sent in a separate data block instead of appending them directly to the URI. Thus, in the GET method the form contents are directly visible to the user as the constructed URI, whereas in the POST method, the form contents are sent inside the HTTP request message body and are not visible to the user.

Using the GET method gives users the opportunity to bookmark the page with the constructed URI and thus directly jump to it in subsequent sessions; this is not possible with the POST method. The choice of GET versus POST should be determined' by the application and its requirements.

Let us look at the encoding of the URI when the GET method is used. The encoded URI has the following form:

action?name1=value1&name2=value2&name3=value3

The action is the URI specified in the ACTION attribute to the FORM tag, or the current document URI if no ACTION attribute was specified. The 'name=value' pairs are the user inputs from the INPUT fields in the form. For form INPUT fields where the user did not input anything, the name is stil present with an empty value (name=). As a concrete example, consider the password submission form at the end of the previous section. Assume that the user inputs 'John Doe' as username, and 'secret' as password. Then the request URI is:

page.jsp?username=J01111+Doe&password=secret

The user input from forms can contain general ASCII characters, such as the space character, but URIs have to be single, consecutive strings with no spaces. Therefore, special characters such as spaces, '=', and other unprintable characters are encoded in a special way. To create a URI that has form fields encoded, we perform the following three steps:

1. Convert all special characters in the names and values to '%xyz,' where 'xyz' is the ASCII value of the character in hexadecimal. Special characters include =, &, %, +, and other unprintable characters. Note that we could encode *all* characters by their ASCII value.
2. Convert all space characters to the '+' character.
3. Glue corresponding names and values from an individual HTML INPUT tag together with '=' and then paste name-value pairs from different HTML INPUT tags together using'&' to create a request URI of the form: action?namel=value1&name2=value2&name3=value3

Note that in order to process the input elements from the HTML form at the middle tier, we need the ACTION attribute of the FORM tag to point to a page, script, or program that will process the values of the form fields the user entered. We discuss ways of receiving values from form fields in Sections 7.7.1 and 7.7.3.

7.6.2 JavaScript

JavaScript is a scripting language at the client tier with which we can add programs to webpages that run directly at the client (Le., at the machine running the Web browser). JavaScript is often used for the following types of computation at the client:

- Browser Detection: JavaScript can be used to detect the browser type and load a browser-specific page.
- Form Validation: JavaScript is used to perform simple consistency checks on form fields. For example, a JavaScript program might check whether a

form input that asks for an email address contains the character '@,' or if all required fields have been input by the user.

- **Browser Control:** This includes opening pages in customized windows; examples include the annoying pop-up advertisements that you see at many websites, which are programmed using JavaScript.

JavaScript is usually embedded into an HTML document with a special tag, the SCRIPT tag. The SCRIPT tag has the attribute LANGUAGE, which indicates the language in which the script is written. For JavaScript, we set the language attribute to JavaScript. Another attribute of the SCRIPT tag is the SRC attribute, which specifies an external file with JavaScript code that is automatically embedded into the HTML document. Usually JavaScript source code files use a '.js' extension. The following fragment shows a JavaScript file included in an HTML document:

```
<SCRIPT LANGUAGE=" JavaScript" SRC="validateForm.js"> </SCRIPT>
```

The SCRIPT tag can be placed inside HTML comments so that the JavaScript code is not displayed verbatim in Web browsers that do not recognize the SCRIPT tag. Here is another JavaScipt code example that creates a pop-up box with a welcoming message. We enclose the JavaScipt code inside HTML comments for the reasons just mentioned.

```
<SCRIPT LANGUAGE=" JavaScript" >
<!--
    alert("Welcome to our bookstore");
//-->
</SCRIPT>
```

JavaScript provides two different commenting styles: single-line comments that start with the '//' character, and multi-line comments starting with '/*' and ending with ,*/' characters.[1]

JavaScript has variables that can be numbers, boolean values (true or false), strings, and some other data types that we do not discuss. Global variables have to be declared in advance of their usage with the keyword var, and they can be used anywhere inside the HTML documents. Variables local to a JavaScript function (explained next) need not be declared. Variables do not have a fixed type, but implicitly have the type of the data to which they have been assigned.

---

1 Actually, '<!--' also marks the start of a single-line comment, which is why we did not have to mark the HTML starting cormnent '<!--' in the preceding example using JavaScript comment notation. In contrast, the HTML closing comment "-->" has to be commented out in JavaScript as it is interpreted otherwise.

JavaScript has the usual assignment operators (=, + =, etc.), the usual arithmetic operators (+, -, *, /, %), the usual comparison operators (==, ! =, >=, etc.), and the usual boolean operators (&& for logical AND, || for logical OR, and! for negation). Strings can be concatenated using the '+' character. The type of an object determines the behavior of operators; for example 1+1 is 2, since we are adding numbers, whereas "1"+"1" is "11," since we are concatenating strings. JavaScript contains the usual types of statements, such as assignments, conditional statements (if Ccondition) {statements;} else {statements; }), and loops (for-loop, do-while, and while-loop).

JavaScript allows us to create functions using the function keyword: function f Cargl, arg2) {statements;}. We can call functions from JavaScript code, and functions can return values using the keyword return.

We conclude this introduction to JavaScript with a larger example of a JavaScript function that tests whether the login and password fields of a HTML form are not empty. Figure 7.12 shows the JavaScript function and the HTML form. The JavaScript code is a function called testLoginEmptyO that tests whether either of the two input fields in the form named LoginForm is empty. In the function testLoginEmpty, we first use variable loginForm to refer to the form LoginForm using the implicitly defined variable document, which refers to the current HTML page. (JavaScript has a library of objects that are implicitly defined.) We then check whether either of the strings loginForm. userif. value or loginForm. password. value is empty.

The function testLoginEmpty is checked within a form event handler. An event **handler** is a function that is called if an event happens on an object in a webpage. The event handler we use is onSubmit, which is called if the submit button is pressed (or if the user presses return in a text field in the form). If the event handler returns true, then the form contents are submitted to the server, otherwise the form contents are not submitted to the server.

JavaScript has functionality that goes beyond the basics that we explained in this section; the interested reader is referred to the bibliographic notes at the end of this chapter.

7.6.3 Style Sheets

Different clients have different displays, and we need correspondingly different ways of displaying the same information. For example, in the simplest case, we might need to use different font sizes or colors that provide high-contrast on a black-and-white screen. As a more sophisticated example, we might need to re-arrange objects on the page to accommodate small screens in personal

```
<SCRIPT LANGUAGE="JavaScript">
<!--
function testLoginEmpty()
{
    10ginForm = document.LoginForm
    if ((loginForm.userid.value == "") ||
            (loginFonn.password.value == "")) {
        alert(,Please enter values for userid and password.');
        return false;
    }
    else
        return true;
}
//-->
</SCRIPT>
<Hi ALIGN = "CENTER">Barns and Nobble Internet Bookstore</Hi>
<H3 ALIGN = "CENTER">Please enter your userid and password:</H3>
<FORM NAME = "LoginForm" METHOD="POST"
        ACTION="TableOfContents.jsp"
        onSubmit="return testLoginEmptyO">
    Userid: <INPUT TYPE="TEXT" NAME="userid"><P>
    Password: <INPUT TYPE="PASSWORD" NAME="password"><P>
    <INPUT TYPE="SUBMIT" VALUE="Login" NAME="SUBMIT">
    <INPUT TYPE="RESET" VALUE="Clear Input" NAME="RESET">
</FORM>
```

Figure 7.12 Form Validation with JavaScript

digital assistants (PDAs). As another example, we might highlight different infonnation to focus on some important part of the page. A style sheet is a method to adapt the same document contents to different presentation formats. A style sheet contains instructions that tell a Web browser (or whatever the client uses to display the webpage) how to translate the data of a document into a presentation that is suitable for the client's display.

Style sheets separate the transformative aspect of the page from the rendering aspects of the page. During transformation, the objects in the XML document are rearranged to form a different structure, to omit parts of the XML document, or to merge two different XML documents into a single document. During rendering, we take the existing hierarchical structure of the XML document and format the document according to the user's display device.

```
BODY {BACKGROUND-COLOR: yellow}
Hi {FONT-SIZE: 36pt}
H3 {COLOR: blue}
P {MARGIN-LEFT: 50px; COLOR: red}
```

Figure 7.13 An Example Style sheet

The use of style sheets has many advantages. First, we can reuse the same document many times and display it differently depending on the context. Second, we can tailor the display to the reader's preference such as font size, color style, and even level of detail. Third, we can deal with different output formats, such as different output devices (laptops versus cell phones), different display sizes (letter versus legal paper), and different display media (paper versus digital display). Fourth, we can standardize the display format within a corporation and thus apply style sheet conventions to documents at any time. Further, changes and improvements to these display conventions can be managed at a central place.

There are two style sheet languages: XSL and ess. ess was created for HTML with the goal of separating the display characteristics of different formatting tags from the tags themselves. XSL is an extension of ess to arbitrary XML docurnents; besides allowing us to define ways of formatting objects, XSL contains a transformation language that enables us to rearrange objects. The target files for ess are HTML files, whereas the target files for XSL are XML files.

Cascading Style Sheets

A Cascading Style Sheet (CSS) defines how to display HTML elements. (In Section 7.13, we introduce a more general style sheet language designed for XML documents.) Styles are normally stored in style sheets, which are files that contain style definitions. Many different HTML documents, such as all documents in a website, can refer to the same ess. Thus, we can change the format of a website by changing a single file. This is a very convenient way of changing the layout of many webpages at the same time, and a first step toward the separation of content from presentation.

An example style sheet is shown in Figure 7.13. It is included into an HTML file with the following line:

```
<LINK REL="style sheet" TYPE="text/css" HREF="books.css" />
```

Each line in a CSS sheet consists of three parts; a selector, a property, and a value. They are syntactically arranged in the following way:

selector {property: value}

The selector is the element or tag whose format we are defining. The property indicates the tag's attribute whose value we want to set in the style sheet, and the property is the actual value of the attribute. As an example, consider the first line of the example style sheet shown in Figure 7.13:

BODY {BACKGROUND-COLOR: yellow}

This line has the same effect as changing the HTML code to the following:

<BODY BACKGROUND-COLOR="yellow">.

The value should always be quoted, as it could consist of several words. More than one property for the same selector can be separated by semicolons as shown in the last line of the example in Figure 7.13:

P {MARGIN-LEFT: 50px; COLOR: red}

Cascading style sheets have an extensive syntax; the bibliographic notes at the end of the chapter point to books and online resources on CSSs.

XSL

XSL is a language for expressing style sheets. An XSL style sheet is, like CSS, a file that describes how to display an XML document of a given type. XSL shares the functionality of CSS and is compatible with it (although it uses a different syntax).

The capabilities of XSL vastly exceed the functionality of CSS. XSL contains the XSL Transformation language, or XSLT, a language that allows us to transform the input XML document into a XML document with another structure. For example, with XSLT we can change the order of elements that we are displaying (e.g.; by sorting them), process elements more than once, suppress elements in one place and present them in another, and add generated text to the presentation.

XSL also contains the XML Path Language (XPath), a language that allows us to refer to parts of an XML document. We discuss XPath in Section

27. XSL also contains XSL Formatting Object, a way of formatting the output of an XSL transformation.

7.7 THE MIDDLE TIER

In this section, we discuss technologies for the middle tier. The first generation of middle-tier applications were stand-alone programs written in a general-purpose programming language such as C, C++, and Perl. Programmers quickly realized that interaction with a stand-alone application was quite costly; the overheads include starting the application every time it is invoked and switching processes between the webserver and the application. Therefore, such interactions do not scale to large numbers of concurrent users. This led to the development of the application server, which provides the run-time environment for several technologies that can be used to program middle-tier application components. Most of today's large-scale websites use an application server to run application code at the middle tier.

Our coverage of technologies for the middle tier mirrors this evolution. We start in Section 7.7.1 with the Common Gateway Interface, a protocol that is used to transmit arguments from HTML forms to application programs running at the middle tier. We introduce application servers in Section 7.7.2. We then describe technologies for writing application logic at the middle tier: Java servlets (Section 7.7.3) and Java Server Pages (Section 7.7.4). Another important functionality is the maintenance of state in the middle tier component of the application as the client component goes through a series of steps to complete a transaction (for example, the purchase of a market basket of items or the reservation of a flight). In Section 7.7.5, we discuss Cookies, one approach to maintaining state.

7.7.1 CGI: The Common Gateway Interface

The Common Gateway Interface connects HTML forms with application programs. It is a protocol that defines how arguments from forms are passed to programs at the server side. We do not go into the details of the actual CGI protocol since libraries enable application programs to get arguments from the HTML fonn; we shortly see an example in a CGI program. Programs that communicate with the webserver via CGI are often called CGI scripts, since many such application programs were written in a scripting language such as Perl.

As an example of a program that interfaces with an HTML form via CGI, consider the sample page shown in Figure 7.14. This webpage contains a form where a user can fill in the name of an author. If the user presses the 'Send

```
<HTML><HEAD><TITLE>The Database Bookstore</TITLE></HEAD>
<BODY>
<FORM ACTION="find_books.cgi" METHOD=POST>
    Type an author name:
    <INPUT TYPE="text" NAME=lauthorName"
           SIZE=30 MAXLENGTH=50>
    <INPUT TYPE="submitil value="Send it">
    <INPUT TYPE=lreset" VALUE="Clear form">
</FORM>
</BODY></HTML>
```

Figure 7.14 A Sample Web Page Where Form Input Is Sent to a CGI Script

it' button, the Perl script 'findBooks.cgi' shown in Figure 7.14 is executed as a separate process. The CGl protocol defines how the communication between the form and the script is performed. Figure 7.15 illustrates the processes created when using the CGl protocol.

Figure 7.16 shows the example CGl script, written in Perl. We omit error-checking code for simplicity. Perl is an interpreted language that is often used for CGl scripting and many Perl libraries, called **modules,** provide high-level interfaces to the CGl protocol. \Ve use one such library, called the **DBI library,** in our example. The CGI module is a convenient collection of functions for creating CGl scripts. In part 1 of the sample script, we extract the argument of the HTML form that is passed along from the client as follows:

```
$authorName = $dataIn- >paramCauthorName');
```

Note that the parameter name authorName was used in the form in Figure 7.14 to name the first input field. Conveniently, the CGl protocol abstracts the actual implementation of how the webpage is returned to the Web browser; the webpage consists simply of the output of our program, and we start assembling the output HTML page in part 2. Everything the script writes in print-statements is part of the dynamically constructed webpage returned to the browser. We finish in part 3 by appending the closing format tags to the resulting page.

7.7.2 Application Servers

Application logic can be enforced through server-side programs that are invoked using the CGl protocol. However, since each page request results in the creation of a new process, this solution does not scale well to a large number of simultaneous requests. This performance problem led to the development of

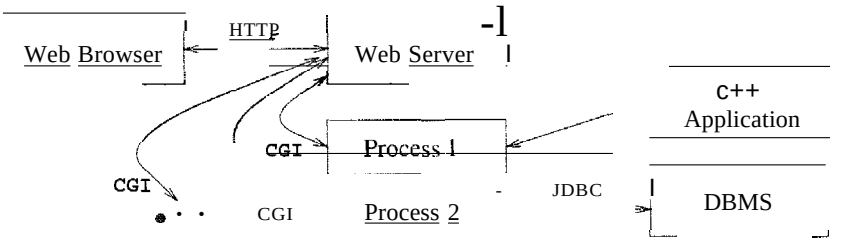

Figure 7.15 Process Structure with eGI Scripts

```
#!/usr/bin/perl
use CGI;

### part 1
$dataln = new CGI;
$dataIn-¿header();
$authorName = $dataln-l,param('authorName');

### part 2
print (II<HTML><TITLE>Argument passing test</TITLE> II) ;
print (II The user passed the following argument: II) ;
print (II authorName: ", $authorName);

### part 3
print ("</HTML>");
exit;
```

Figure 7.16 A Simple Perl Script

specialized programs called application servers. An application server maintains a pool of threads or processes and uses these to execute requests. Thus, it avoids the startup cost of creating a new process for each request.

Application servers have evolved into flexible middle-tier packages that provide many functions in addition to eliminating the process-creation overhead. They facilitate concurrent access to several heterogeneous data sources (e.g., by providing JDBC drivers), and provide session management services. Often, business processes involve several steps. Users expect the system to maintain continuity during such a multistep session. Several session identifiers such as cookies, URI extensions, and hidden fields in HTML forms can be used to identify a session. Application servers provide functionality to detect when a session starts and ends and keep track of the sessions of individual users. They

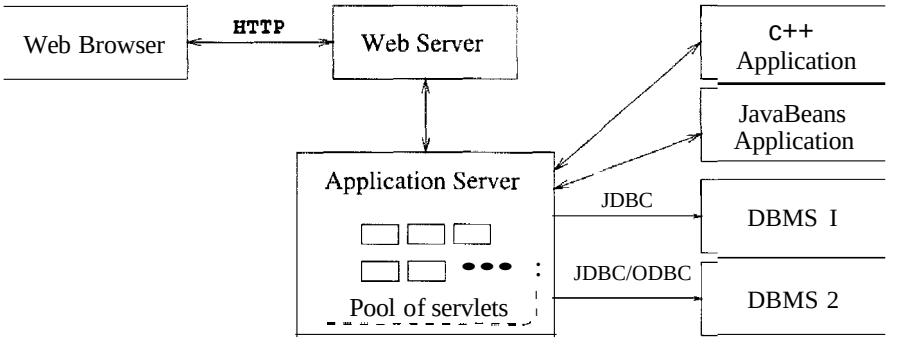

Figure 7.17 Process Structure in the Application Server Architecture

also help to ensure secure database access by supporting a general user-id mechanism. (For more on security, see Chapter 21.)

A possible architecture for a website with an application server is shown in Figure 7.17. The client (a Web browser) interacts with the webserver through the HTTP protocol. The webserver delivers static HTML or XML pages directly to the client. To assemble dynamic pages, the webserver sends a request to the application server. The application server contacts one or more data sources to retrieve necessary data or sends update requests to the data sources. After the interaction with the data sources is completed, the application server assembles the webpage and reports the result to the webserver, which retrieves the page and delivers it to the client.

The execution of business logic at the webserver's site, server-side processing, has become a standard model for implementing more complicated business processes on the Internet. There are many different technologies for server-side processing and we only mention a few in this section; the interested reader is referred to the bibliographic notes at the end of the chapter.

7.7.3 Servlets

Java servlets are pieces of Java code that run on the middle tier, in either webservers or application servers. There are special conventions on how to read the input from the user request and how to write output generated by the servlet. Servlets are truly platform-independent, and so they have become very popular with Web developers.

Since servlets are Java programs, they are very versatile. For example, servlets can build webpages, access databases, and maintain state. Servlets have access

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

pUblic class ServletTemplate extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        // Use 'out' to send content to browser
        out.println("Hello World");
    }
}
```

Figure 7.18 Servlet Template

to all Java APIs, including JDBC. All servlets must implement the Servlet interface. In most cases, servlets extend the specific HttpServlet class for servers that communicate with clients via HTTP. The HttpServlet class provides methods such as doGet and doPost to receive arguments from HTML forms, and it sends its output back to the elient via HTTP. Servlets that communicate through other protocols (such as ftp) need to extend the class GenericServlet.

Servlets are compiled Java classes executed and maintained by a servlet container. The servlet container manages the lifespan of individual servlets by creating and destroying them. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by webservers. For such applications, there is a useful library of HTTP-specific servlet classes.

Servlets usually handle requests from HTML forms and maintain state between the client and the server. We discuss how to maintain state in Section 7.7.5. A template of a generic servlet structure is shown in Figure 7.18. This simple servlet just outputs the two words "Hello World," but it shows the general structure of a full-fledged servlet. The request object is used to read HTML form data. The response object is used to specify the HTTP response status code and headers of the HTTP response. The object out is used to compose the content that is returned to the client.

Recall that HTTP sends back the status line, a header, a blank line, and then the context. Right now our servlet just returns plain text. We can extend our servlet by setting the content type to HTML, generating HTML as follows:

```
PrinfWriter out = response.get\Vriter();
String docType =
    "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
    "Transitional//EN"> \n";
out.println(docType +
    "<HTML>\n" +
    "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
    "<BODY>\n" +
    "<Hl>Hello WWW</Hl>\n" +
    "</BODY></HTML>");
```

What happens during the life of a servlet? Several methods are called at different stages in the development of a servlet. When a requested page is a servlet, the webserver forwards the request to the servlet container, which creates an instance of the servlet if necessary. At servlet creation time, the servlet container calls the init () method, and before deallocating the servlet, the servlet container calls the servlet's destroyO method.

When a servlet container calls a servlet because of a requested page, it starts with the service () method, whose default behavior is to call one of the following methods based on the HTTP transfer method: service () calls doGet 0 for a HTTP GET request, and it calls doPost () for a HTTP POST request. This automatic dispatching allows the servlet to perform different tasks on the request data depending on the HTTP transfer method. Usually, we do not override the service () method, unless we want to program a servlet that handles both HTTP POST and HTTP GET requests identically.

We conclude our discussion of servlets with an example, shown in Figure 7.19, that illustrates how to pass arguments from an HTML form to a servlet.

7.7.4 JavaServer Pages

In the previous section, we saw how to use Java programs in the middle tier to encode application logic and dynamically generate webpages. If we needed to generate HTML output, we wrote it to the out object. Thus, we can think about servlets as Java code embodying application logic, with embedded HTML for output.

JavaServer pages (JSPs) interchange the roles of output and application logic. JavaServer pages are written in HTML with servlet-like code embedded in special HT1VIL tags. Thus, in comparison to servlets, JavaServer pages are better suited to quickly building interfaces that have some logic inside, whereas servlets are better suited for complex application logic.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class ReadUserName extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType('j textjhtml'j);
        PrintWriter out = response.getWriter();

        out.println("<BODY>\n" +
            "<Hi ALIGN=CENTER> Username: </Hi>\n" +
            "<UL>\n" +
            " <LI>title: "
            + request.getParameter("userid") + "\n" +
            + request.getParameter("password'j) + "\n" +
            1</UL>\n" +
            1</BODY></HTML>")j
    }

    public void doPost(HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Figure 7.19 Extracting the User Name and Password From a Form

While there is a big difference for the programmer, the middle tier handles JavaServer pages in a very simple way: They are usually compiled into a servlet, which is then handled by a servlet container analogous to other servlets.

The code fragment in Figure 7.20 shows a simple JSP example. In the middle of the HTML code, we access information that was passed from a form.

```
<!DOCTYPE HTML PUBLIC 11_//W3C//DTD HTML 4.0
        Transitional//EN"">
<HTML>
<HEAD><TITLE>Welcome to Barnes and Nobble</TITLE></HEAD>
<BODY>
    <Hl>Welcome back!</Hl>
    <% String name="NewUser"";
        if (request.getParameter(lIusernamell) != null) {
            name=request.getParameter("username");
        }
    %>
    You are logged on as user <%=name%>
    <P>
    Regular HTML for all the rest of the on-line store's webpage.
</BODY>
</HTML>
```

Figure 7.20 Reading Form Parameters in JSP

7.7.5 **Maintaining State**

As discussed in previous sections, there is a need to maintain a user's state across different pages. As an example, consider a user who wants to make a purchase at the Barnes and Nobble website. The user must first add items into her shopping basket, which persists while she navigates through the site. Thus, we use the notion of state mainly to remember information as the user navigates through the site.

The HTTP protocol is stateless. We call an interaction with a webserver stateless if no inforination is retained from one request to the next request. We call an interaction with a webserver stateful, or we say that state is maintained, if some memory is stored between requests to the server, and different actions are taken depending on the contents stored.

In our example of Barnes and Nobble, we need to maintain the shopping basket of a user. Since state is not encapsulated in the HTTP protocol, it has to be maintained either at the server or at the client. Since the HTTP protocol is stateless by design, let us review the advantages and disadvantages of this design decision. First, a stateless protocol is easy to program and use, and it is great for applications that require just retrieval of static information. In addition, no extra memory is used to maintain state, and thus the protocol itself is very efficient. On the other hand, without some additional mechanism at the presentation tier and the middle tier, we have no record of previous requests, and we cannot program shopping baskets or user logins.

Since we cannot maintain state in the HTTP protocol, where should we mtain-tain state? There are basically two choices. We can maintain state in the middle tier, by storing information in the local main memory of the application logic, or even in a database system. Alternatively, we can maintain state on the client side by storing data in the form of a *cookie*. We discuss these two ways of maintaining state in the next two sections.

Maintaining State at the Middle Tier

At the middle tier, we have several choices as to *where* we maintain state. First, we could store the state at the bottom tier, in the database server. The state survives crashes of the system, but a database access is required to query or update the state, a potential performance bottleneck. An alternative is to store state in main memory at the middle tier. The drawbacks are that this information is volatile and that it might take up a lot of main memory. We can also store state in local files at the middle tier, as a compromise between the first two approaches.

A rule of thumb is to use state maintenance at the middle tier or database tier only for data that needs to persist over many different user sessions. Examples of such data are past customer orders, click-stream data recording a user's movement through the website, or other permanent choices that a user makes, such as decisions about personalized site layout, types of messages the user is willing to receive, and so on. As these examples illustrate, state information is often centered around users who interact with the website.

Maintaining State at the Presentation Tier: Cookies

Another possibility is to store state at the presentation tier and pass it to the middle tier with every HTTP request. We essentially work around around the statelessness of the HTTP protocol by sending additional information with every request. Such information is called a cookie.

A **cookie** is a collection of *(name, value)*-pairs that can be manipulated at the presentation and middle tiers. Cookies are easy to use in Java servlets and Java8erver Pages and provide a simple way to make non-essential data persistent at the client. They survive several client sessions because they persist in the browser cache even after the browser is closed.

One disadvantage of cookies is that they are often perceived as as being invasive, and many users disable cookies in their Web browser; browsers allow users to prevent cookies from being saved on their machines. Another disadvantage is that the data in a cookie is currently limited to 4KB, but for most applications this is not a bad limit.

We can use cookies to store information such as the user's shopping basket, login information, and other non-permanent choices made in the current session.

Next, we discuss how cookies can be manipulated from servlets at the middle tier.

The Servlet Cookie API

A cookie is stored. in a small text file at the client and. contains *(name, value)*-pairs, where both name and value are strings. We create a new cookie through the Java Cookie class in the middle tier application code:

```
Cookie cookie = new Cookie("username","guest");
cookie.setDomain("www.bookstore.com");
cookie.set8ecure(false);                    // no 88L required
cookie.setMaxAge(60*60*24*7*31);      // one month lifetime
response.addCookie(cookie);
```

Let us look at each part of this code. First, we create a new Cookie object with the specified *(name, value)*-pair. Then we set attributes of the cookie; we list some of the most common attributes below:

- setDomain and getDomain: The domain specifies the website that will receive the cookie. The default value for this attribute is the domain that created the cookie.
- setSecure and getSecure: If this flag is true, then the cookie is sent only if we are llsing a secure version of the HTTP protocol, such as 88L.
- setMaxAge and getMaxAge: The MaxAge attribute determines the lifetime of the cookie in seconds. If the value of MaxAge is less than or equal to zero, the cookie is deleted when the browser is closed.

- setName and getName: We did not use these functions in our code fragment; they allow us to Ilame the cookie.
- setValue and getValue: These functions allow us to set and read the value of the cookie.

The cookie is added to the request object within the Java servlet to be sent to the client. Once a cookie is received from a site (www.bookstore.comin this example), the client's Web browser appends it to all HTTP requests it sends to this site, until the cookie expires.

We can access the contents of a cookie in the middle-tier code through the request object getCookies 0 method, which returns an array of Cookie objects. The following code fragment reads the array and looks for the cookie with name 'username.'

```
Cookie[] cookies = request.getCookiesO;
String theUser;
for(int i=O; i < cookies.length; i++) {
    Cookie cookie = cookies[i];
    if (cookie.getNameO.equals("username"))
        theUser = cookie.getValueO;
}
```

A simple test can be used to check whether the user has turned off cookies: Send a cookie to the user, and then check whether the request object that is returned still contains the cookie. Note that a cookie should never contain an unencrypted password or other private, unencrypted data, as the user can easily inspect, modify, and erase any cookie at any time, including in the middle of a session. The application logic needs to have sufficient consistency checks to ensure that the data in the cookie is valid.