****Chapter in the Advanced Version*

Part

VIII

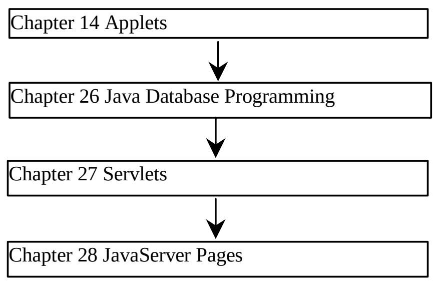
Web Programming

This part is devoted to develop Web applications using Java. Chapter 26 introduces how to use Java to develop database projects, and Chapters 27 and 28 introduce how to use Java servlets and JSP to generate dynamic contents from Web servers.

Chapter 26
Java Database Programming
Chapter 27
Servlets
Chapter 28
JavaServer Pages

***PD: Put this in the center of the back of the Part-opening page. AU*

Prerequisites for Part VIII


Chapter 14 Applets
Chapter 26 Java Database Programming
Chapter 27 Servlets
Chapter 28 JavaServer Pages

Chapter 26 Java Database Programming

Objectives

- To understand the concept of database and database management systems.
- To understand the relational data model: relational data structures, constraints, and languages.
- To use SQL to create and drop tables, retrieve and modify data.
- To become familiar with the JDBC API.
- To learn how to load a driver, connect to a database, execute statements, and process result sets using JDBC.
- To use the prepared statements to execute precompiled SQL statements.
- To handle transactions in the Connection interface
- To explore database metadata using the DatabaseMetaData

and ResultSetMetaData interfaces.

- To execute SQL statements in a batch mode.
- To process updateable and scrollable result sets.
- To store and retrieve images in JDBC.

26.1 Introduction

You may have heard a lot about database systems. Database systems are everywhere. Your social security information is stored in a database by the government. If you shop online, your purchase information is stored in a database by the company. If you attend a university, your academic information is stored in a database by the university. Database systems not only store data, they also provide means of accessing, updating, manipulating, and analyzing data. Your social security information is updated periodically, and you can register in courses online. Database systems play an important role for the society and for the commerce.

This chapter introduces database systems, SQL, and how to develop database applications using Java. If you already know SQL, you may skip Sections 26.2 and 26.3.

26.2 Relational Database Systems

A database system consists of a database, the software that stores and manages data in the database, and the application programs that present data and enable the user to interact with the database system, as shown in Figure 26.1.

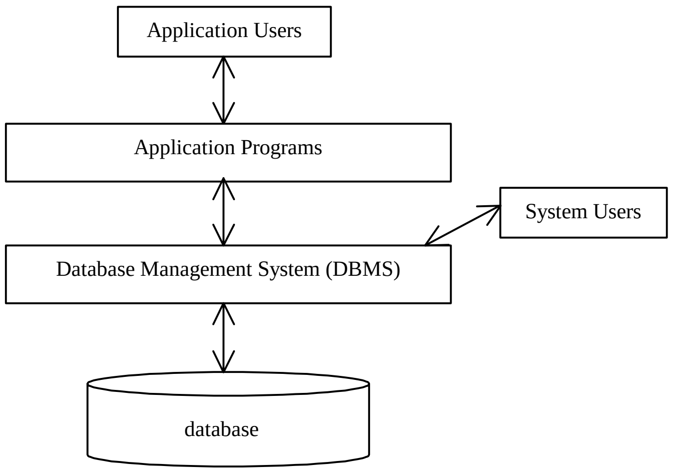


Figure 26.1

A database system consists of data, database management software, and application programs.

A database is a repository of data that form information. When you purchase a database system from a software vendor such as MySQL, Oracle, IBM, Microsoft, or Sybase, you actually purchase the software comprising a *database management system* (DBMS) from the vendor. Database management systems are designed for use by professional programmers and are not suitable for ordinary customers. Application programs are built on top of the DBMS for customers to access and update the database. Thus application programs can be viewed as the interfaces between the database system and its users. The application programs may be standalone GUI applications or Web applications, and may access several different database systems in the network, as shown in Figure 26.2.

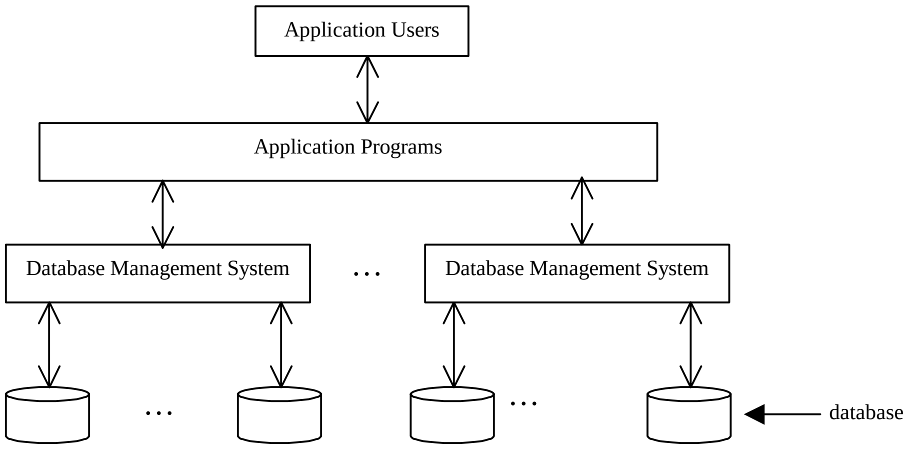


Figure 26.2

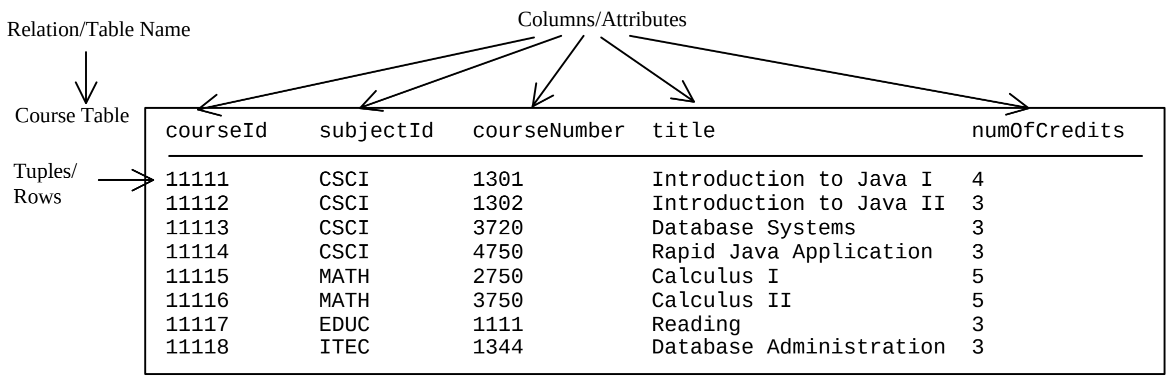
An application program may access multiple database systems.

Most of today's database systems are *relational database systems*, based on the relational data model. A relational data model has three key components: structure, integrity, and language. *Structure* defines the representation of the data. *Integrity* imposes constraints on the data. *Language* provides the means for accessing and manipulating data.

26.2.1 Relational Structures

The relational model is built around a simple and natural structure. A relation is actually a table that consists of non-duplicate rows. Tables are easy to understand and easy to use. The relational model provides a simple yet powerful way to represent data.

A row of a table represents a record, and a column of a table represents the value of a single attribute of the record. In the relational database theory, a row is called a *tuple* and a column is called an *attribute*. Figure 26.3 shows a sample table that stores information about the courses in a university. The table has eight tuples, and each tuple has five attributes.



courseId	subjectId	courseNumber	title	numOfCredits
11111	CSCI	1301	Introduction to Java I	4
11112	CSCI	1302	Introduction to Java II	3
11113	CSCI	3720	Database Systems	3
11114	CSCI	4750	Rapid Java Application	3
11115	MATH	2750	Calculus I	5
11116	MATH	3750	Calculus II	5
11117	EDUC	1111	Reading	3
11118	ITEC	1344	Database Administration	3

Figure 26.3

A table has a table name, column names, and rows.

Tables describe the relationship among data. Each row in a table represents a record of related data. For example, “11111”, “CSCI”, “1301”, “Introduction to Java I”, and “4” are related to form a record (the first row in Figure 26.3) in the Course table. Just as data in the same row are related, so too data in different tables may be related through common attributes. Suppose the database has two other tables named Student and Enrollment, as shown in Figures 26.4 and 26.5. The Course table and the Enrollment table are related through their common attribute courseId, and the Enrollment table and the Student table are related through ssn.

Student Table

ssn	firstName	mi	lastName	phone	birthDate	street	zipCode	deptID
444111110	Jacob	R	Smith	9129219434	1985-04-09	99 Kingston Street	31435	BIOL
444111111	John	K	Stevenson	9129219434	null	100 Main Street	31411	BIOL
444111112	George	K	Smith	9129213454	1974-10-10	1200 Abercorn St.	31419	CS
444111113	Frank	E	Jones	9125919434	1970-09-09	100 Main Street	31411	BIOL
444111114	Jean	K	Smith	9129219434	1970-02-09	100 Main Street	31411	CHEM
444111115	Josh	R	Woo	7075989434	1970-02-09	555 Franklin St.	31411	CHEM
444111116	Josh	R	Smith	9129219434	1973-02-09	100 Main Street	31411	BIOL
444111117	Joy	P	Kennedy	9129229434	1974-03-19	103 Bay Street	31412	CS
444111118	Toni	R	Peterson	9129229434	1964-04-29	103 Bay Street	31412	MATH
444111119	Patrick	R	Stoneman	9129229434	1969-04-29	101 Washington St.	31435	MATH
444111120	Rick	R	Carter	9125919434	1986-04-09	19 West Ford St.	31411	BIOL

Figure 26.4

A Student table stores student information.

Enrollment Table

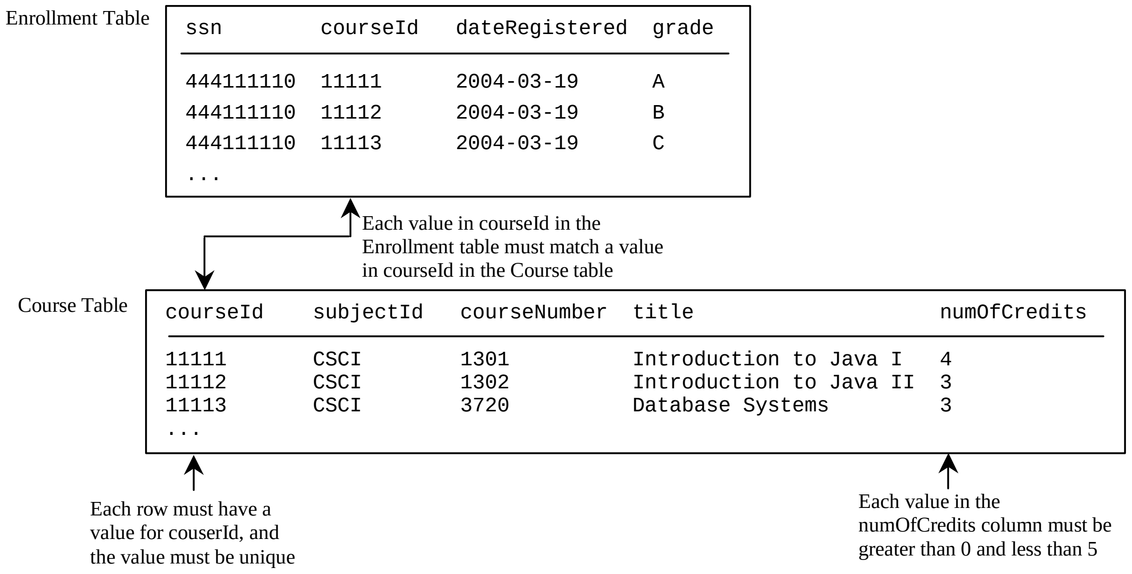
ssn	courseId	dateRegistered	grade
444111110	11111	2004-03-19	A
444111110	11112	2004-03-19	B
444111110	11113	2004-03-19	C
444111111	11111	2004-03-19	D
444111111	11112	2004-03-19	F
444111111	11113	2004-03-19	A
444111112	11114	2004-03-19	B
444111112	11115	2004-03-19	C
444111112	11116	2004-03-19	D
444111113	11111	2004-03-19	A
444111113	11113	2004-03-19	A
444111114	11115	2004-03-19	B
444111115	11115	2004-03-19	F
444111115	11116	2004-03-19	F
444111116	11111	2004-03-19	D
444111117	11111	2004-03-19	D
444111118	11111	2004-03-19	A
444111118	11112	2004-03-19	D
444111118	11113	2004-03-19	B

Figure 26.5

An <u>Enrollment</u> table stores student enrollment information.

26.2.2 Integrity Constraints

An integrity constraint imposes a condition that all the legal values of a table must satisfy. Figure 26.6 shows an example of some integrity constraints in the <u>Subject</u> and <u>Course</u> tables.


Enrollment Table
Each value in courseId in the Enrollment table must match a value in courseId in the Course table
Course Table
Each row must have a value for couserId, and the value must be unique
Each value in the numOfCredits column must be greater than 0 and less than 5

Enrollment Table

ssn	courseId	dateRegistered	grade
444111110	11111	2004-03-19	A
444111110	11112	2004-03-19	B
444111110	11113	2004-03-19	C
...			

Course Table

courseId	subjectId	courseNumber	title	numOfCredits
11111	CSCI	1301	Introduction to Java I	4
11112	CSCI	1302	Introduction to Java II	3
11113	CSCI	3720	Database Systems	3
...				

Figure 26.6

The Enrollment table and the Course table have integrity constraints.

In general, there are three types of constraints: *domain constraints, primary key constraints*, and *foreign key constraints*. Domain constraints and primary key constraints are known as *intra-relational constraints*, meaning that a constraint involves only one relation. The foreign key constraint is *inter-relational*, meaning that a constraint involves more than one relation.

26.2.2.1 Domain Constraints

Domain constraints specify the permissible values for an attribute. Domains can be specified using standard data types, such as integers, floating-point numbers, fixed-length strings, and variant-length strings. The standard data type specifies a broad range of values. Additional constraints can be specified to narrow the ranges. For example, you can specify that the numOfCredits attribute must be greater than 0 and less than 5. You can also specify whether an attribute can be null, which is a special value meaning unknown or not applicable.

26.2.2.2 Primary Key Constraints

To understand primary keys, it is helpful to know superkeys, keys, and candidate keys. A *superkey* is an attribute or a set of attributes that uniquely identifies the relation. That is, no two tuples have the same values on the superkey. By definition, a relation consists of a set of distinct tuples. The set of all attributes in the relation forms a superkey.

A *key* K is a minimal superkey, meaning that any proper subset of K is not a superkey. A relation can have several keys. In this case, each of the keys is called a *candidate key*. The *primary* key is one of the candidate keys designated by the database designer. The primary key is often used to identify tuples in a relation.

26.2.2.3 Foreign Key Constraints

In a relational database, data are related. Tuples in a relation are related and tuples in different relations are related through their common attributes. Informally speaking, the common attributes are foreign keys. The *foreign key constraints* define the relationships among relations.

Formally, a set of attributes *FK* is a *foreign key* in a relation *R* that references relation *T* if it satisfies the following two rules:

- The attributes in *FK* have the same domain as the primary key in *T*.
- A non-null value on *FK* in *R* must match a primary key value in *T*.

As shown in Figure 26.6, courseId is the foreign key in Enrollment that references the primary key courseId in Course. Every courseId value must match a courseId value in Course.

26.2.2.4 Enforcing Integrity Constraints

The database management system enforces integrity constraints and rejects operations that would violate them. For example, if you attempt to insert a new record ('11113', '3272', 'Database Systems', 0) into the Course table, it would fail because the credit hours must be greater than or equal to 0; if you attempt to insert a record with the same primary key as an existing record in the table, the DBMS would report an error and reject the operation; if you attempt to delete a record in the

Course table whose primary key value is referenced by the records in the Enrollment table, the DBMS would reject this operation.

> NOTE: All relational database systems support the primary key constraints and the foreign key constraints. Not all database systems support the domain constraints. For example, you cannot specify the constraint that numOfCredits is greater than 0 and less than 5 on Microsoft Access database.

26.3 SQL

Structured Query Language (SQL) is the language for defining tables and integrity constraints and for accessing and manipulating data. SQL (pronounced "S-Q-L" or "sequel") is the universal language for accessing relational database systems. Application programs may allow users to access a database without directly using SQL, but these applications themselves must use SQL to access the database. This section introduces some basic SQL commands.

> NOTE: There are hundreds of relational database management systems. They share the common SQL language but do not all support every feature of SQL. Some systems have their own extensions to SQL. This section introduces standard SQL supported by all systems.

26.3.1 Creating and Dropping Tables

Tables are the essential objects in a database. To create a table, use the create table statement to specify a table name, attributes, and types, as in the following example:

```
create table Course (
  courseId char(5),
  subjectId char(4) not null,
  courseNumber integer,
  title varchar(50) not null,
  numOfCredits integer,
  primary key (courseId)
);
```

This statement creates the Course table with attributes courseId, subjectId, courseNumber, title, and numOfCredits. Each attribute has a data type that specifies the type of data stored in the attribute. char(5) specifies that courseId consists of five characters. varchar(50) specifies that title is a variant-length string with a maximum of fifty characters. integer specifies that courseNumber is an integer. The primary key is courseId.

The tables Student and Enrollment can be created as follows:

```
create table Student (
  ssn char(9),
  firstName varchar(25),
  mi char(1),
  lastName varchar(25),
  birthDate date,
  street varchar(25),
  phone char(11),
  zipCode char(5),
  deptId char(4),
  primary key (ssn)
);

create table Enrollment (
  ssn char(9),
  courseId char(5),
  dateRegistered date,
  grade char(1),
  primary key (ssn, courseId),
  foreign key (ssn) references Student,
  foreign key (courseId) references Course
);
```

NOTE: SQL keywords are not case-sensitive. This book adopts the following naming conventions. Tables are named in the same way as Java classes, and attributes are named in the same way as Java variables. SQL keywords are named in the same way as Java keywords.

If a table is no longer needed, it can be dropped permanently using the drop table command. For example, the following statement drops the Course table.

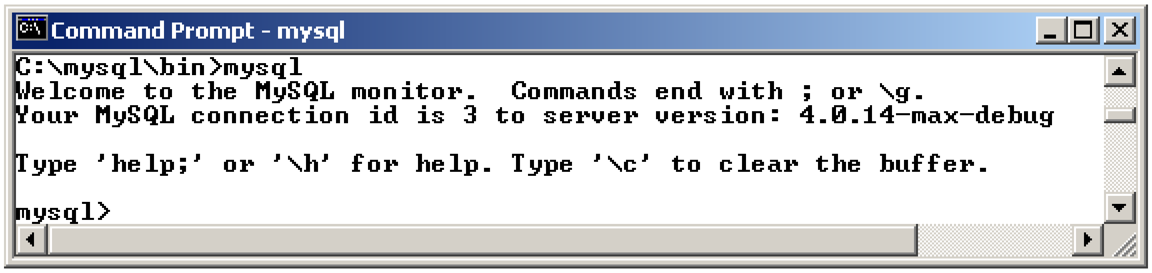
```
drop table Course;
```

If a table to be dropped is referenced by other tables, you have to drop the other tables first. For example, if you have created the tables Course, Student, and Enrollment and want to drop Course, you have to first drop Enrollment, because Course is referenced by Enrollment.

26.3.2 Using SQL on a Relational Database

SQL can be used on MySQL, Oracle, Sybase, IBM DB2, IBM Informix, Borland Interbase, and MS Access, or any other relational database system. The companion Website (www.prenhall.com/liang/intro5e.html) contains the supplements on how to install and use SQL on three most popular databases MySQL, Access, and Oracle. Since MySQL is free and easy to use, this chapter uses MySQL to demonstrate SQL.

Assume that you have installed MySQL with the default configuration; you can access MySQL from the DOS command prompt using the command mysql from the c:\mysql\bin directory, as shown in Figure 26.7.



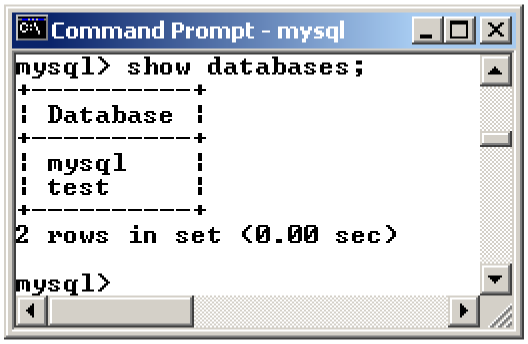
```
C:\mysql\bin>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 4.0.14-max-debug

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Figure 26.7

You can access a MySQL database server from the command window.

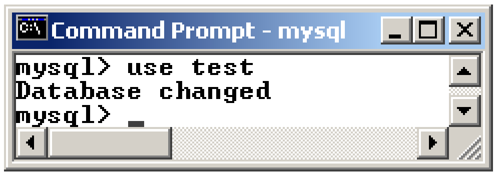
NOTE: On Windows, your MySQL database server starts every time your computer starts. You can stop it by typing the command net stop mysql and restart it by typing the command net start mysql.

By default, the server contains two databases named mysql and test. You can see these two databases displayed in Figure 26.8 (A) using the command show databases.



```
mysql> show databases;
+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.00 sec)

mysql>
```

(A)



```
mysql> use test
Database changed
mysql> _
```

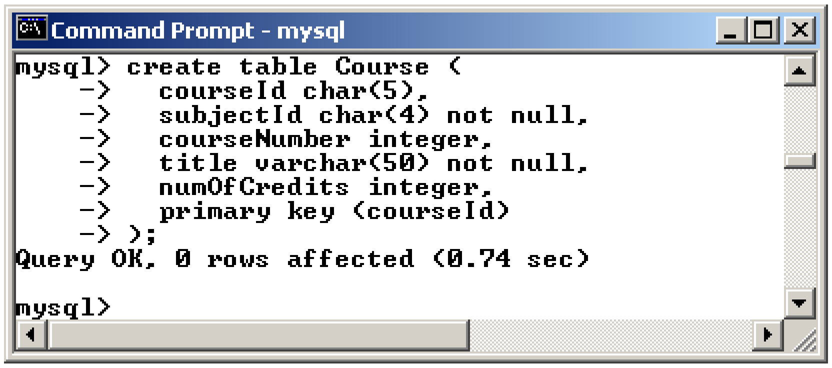
(B)

Figure 26.8

(A) The show databases command displays all available databases in the MySQL database server. (B) The use test command selects the test database.

The mysql database contains the tables that store the information about the server and its users. This database is intended for the server administrator to use. For example, the administrator can use it to create users and grant or revoke user privileges. Since you are the owner of the server installed on your system, you have full access to the mysql database. However, you should not create user tables in the mysql database. You can use the test database to store data or create new databases. You can also create a new database using the command create database *databasename* or drop an existing database using the command drop database *databasename*.

To select a database for use, type the use databasename command. Since the test database is created by default in every MySQL database, let us use it to demonstrate SQL commands. As shown in Figure 26.8 (B), the test database is selected. Enter the statement to create the Course table, as shown in Figure 26.9.

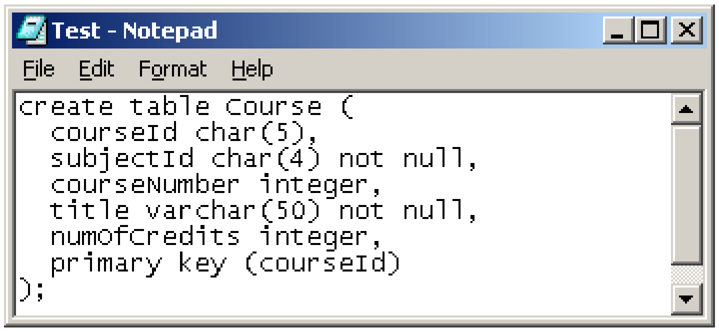


```
mysql> create table Course (
    ->   courseId char(5),
    ->   subjectId char(4) not null,
    ->   courseNumber integer,
    ->   title varchar(50) not null,
    ->   numOfCredits integer,
    ->   primary key (courseId)
    -> );
Query OK, 0 rows affected (0.74 sec)

mysql>
```

Figure 26.9

The execution result of the SQL statements is displayed in the MySQL monitor.

If you have typing errors, you have to retype the whole command. To avoid retyping the whole command, you can save the command in a file, and then run the command from the file. To do so, create a text file, e.g., named test.sql, which contains the commands. You can create the text file using any text editor, e.g., NotePad, as shown in Figure 26.10. To comment a line, precede with two dashes. You can now run the script file by typing <u>source test.sql</u> from the SQL command prompt, as shown in Figure 26.11.

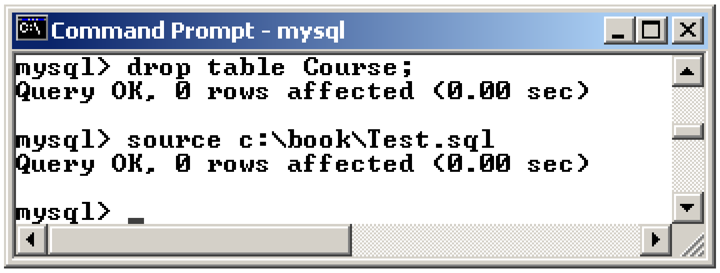


```
create table Course (
  courseId char(5),
  subjectId char(4) not null,
  courseNumber integer,
  title varchar(50) not null,
  numOfCredits integer,
  primary key (courseId)
);
```

Figure 26.10

You can use Notepad to create a text file for SQL commands.

******AU: Put a label to note that no ; for the source statement***



```
mysql> drop table Course;
Query OK, 0 rows affected (0.00 sec)

mysql> source c:\book\Test.sql
Query OK, 0 rows affected (0.00 sec)

mysql> _
```

Figure 26.11

You can run the SQL commands in a script file from MySQL.

26.3.3 Simple Insert, Update, and Delete

Once a table is created, you can insert data into it. You can also update and delete records. This section introduces simple insert, update, and delete statements.

The general syntax to insert a record into a table is

```
insert into tableName [(column1, column2, …, column]]
values (value1, value2, …, valuen);
```

For example, the following statement inserts a record into the Course table. The new record has the courseId '11113', subjectId 'CSCI', courseNumber 3720, title 'Database Systems', and creditHours 3.

```
insert into Course (courseId, subjectId, courseNumber, title)
values ('11113', 'CSCI', '3720', 'Database Systems', 3);
```

The column names are optional. If the column names are omitted, all column values for the record must be entered even though the columns have default values. String values are case-sensitive and enclosed inside single quotation marks.

The general syntax to update a table is

```
update tableName
set column1 = newValue1 [, column2 = newValue2, ...]
[where condition];
```

For example, the following statement changes the numOfCredits for the course whose title is Database Systems to 4.

```
update Course
set numOfCredits = 4
where title = 'Database Systems';
```

The general syntax to delete the records in a table is

```
delete [from] tableName
[where condition];
```

For example, the following statement deletes the Database Systems course from the Course table:

```
delete Course
where title = 'Database System';
```

The following statement deletes all records from the Course table:

```
delete Course;
```

26.3.4 Simple Queries

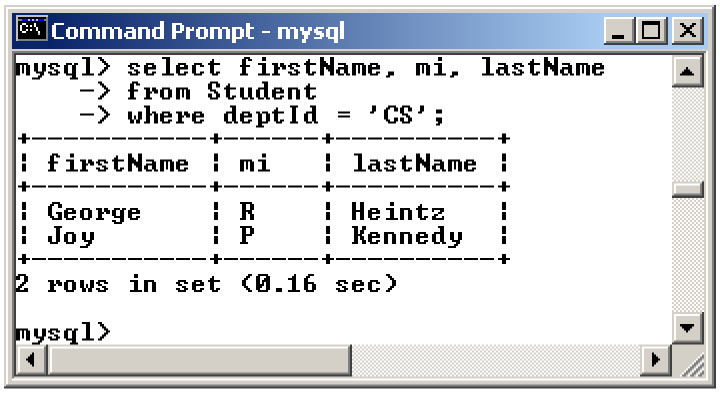
To retrieve information from tables, use a select statement with the following syntax:

```
select column-list
from table-list
where condition];
```

The select clause lists the columns to be selected. The from clause refers to the tables involved in the query. The where clause specifies the conditions for the selected rows.

Query 1: Select all students in the CS department, as shown in Figure 26.12.

```
select firstName, mi, lastName
from Student
where deptId = 'CS';
```



```
mysql> select firstName, mi, lastName
    -> from Student
    -> where deptId = 'CS';
+-----------+------+----------+
| firstName | mi   | lastName |
+-----------+------+----------+
| George    | R    | Heintz   |
| Joy       | P    | Kennedy  |
+-----------+------+----------+
2 rows in set (0.16 sec)

mysql>
```

Figure 26.12

The result of the select statement is displayed in a window.

26.3.5 Comparison and Boolean Operators

SQL has six comparison operators, as shown in Table 26.1, and three Boolean operators, as shown in Table 26.2.

Table 26.1

Comparison Operators

Operator	**Description**
=	Equal to
<> or !=	Not equal to
<	Less than
<=	Less or equal to
>	Greater than
>=	Greater than

Table 26.2

Boolean Operators

Operator	**Description**
not	logical negation
and	logical conjunction
or	logical disjunction

NOTE: The comparison and Boolean operators in SQL have the same meaning as in Java. In SQL the equals to operator is =, but it is == in Java. In SQL the not equal to operator is <> or !=, but it is != in Java. The not, and, and or operators are !, && (&), and || (|) in Java.

Query 2: Get the names of the students who are in the CS dept and live in the zip code 31411.

```
select firstName, mi, lastName
from Student
where deptId = 'CS' and zipCode = '31411';
```

NOTE: To select all the attributes from a table, you don't have to list all the attribute names in the select clause. Instead you can just specify an *asterisk* (*), which stands for all the attributes. For example, the following query displays all attributes of the students who are in the CS dept and live in the zip code 31411:

```
select *
from Student
where deptId = 'CS' and zipCode = '31411';
```

******End of NOTE***

26.3.6 The like, between-and, and is null Operators

SQL has a like operator that can be used for pattern matching. The syntax to check whether a string s has a pattern p is

s like p or s not like p

You can use the wild card characters % (percent symbol) and _ (underline symbol) in the pattern p. % matches zero or more characters, and _ matches any single character in s. For example, lastName like '_mi%' matches any string whose second and third letters are m and i. lastName not like '_mi%' excludes any string whose second and third letters are m and i.

NOTE: On the earlier version of MS Access, the wild card character is *, and the character ? matches any single character.

The between-and operator checks whether a value v is between two other values, v1 and v2, using the following syntax:

v between v1 and v2 or v not between v1 and v2

v between v1 and v2 is equivalent to v >= v1 and v <= v2, and v not between v1 and v2 is equivalent to v < v1 and v > v2.

The is null operator checks whether a value v is null using the following syntax:

v is null or v is not null or

Query 3: Get the social security numbers of the students whose grades are between 'C' and 'A'.

```
select ssn
from Enrollment
where grade between 'C' and 'A';
```

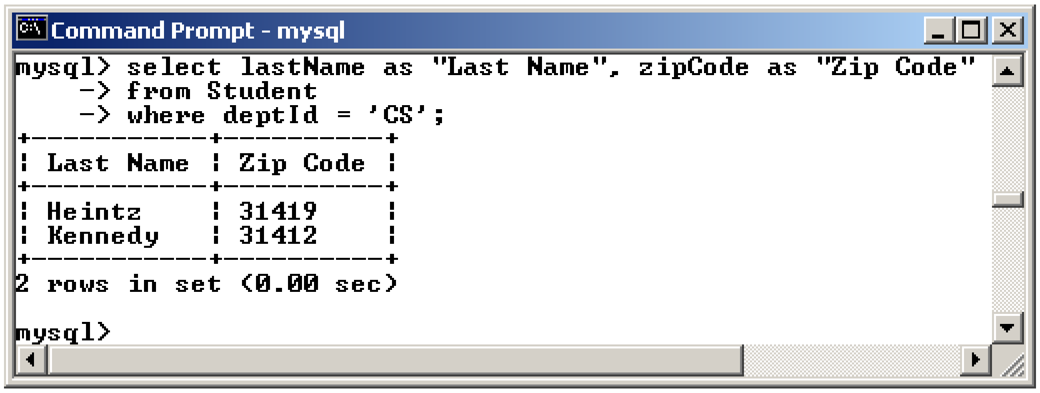
26.3.7 Column Alias

When a query result is displayed, SQL uses the column names as column headings. Usually the user gives abbreviated names for the columns, and the columns cannot have spaces when the table is created. Sometime it is desirable to give more descriptive names in the result heading. You can use the column aliases with the following syntax:

```
select columnName [as] alias
```

Query 4: Get the last name and zip code of the students in the CS department. Display the column headings as Last Name for lastName and Zip Code for zipCode. The query result is shown in Figure 26.13.

```
select lastName as "Last Name", zipCode as "Zip Code"
from Student
where deptId = 'CS';
```



```
mysql> select lastName as "Last Name", zipCode as "Zip Code"
    -> from Student
    -> where deptId = 'CS';
+-----------+----------+
| Last Name | Zip Code |
+-----------+----------+
| Heintz    | 31419    |
| Kennedy   | 31412    |
+-----------+----------+
2 rows in set (0.00 sec)

mysql>
```

Figure 26.13

You can use a column alias in the display.

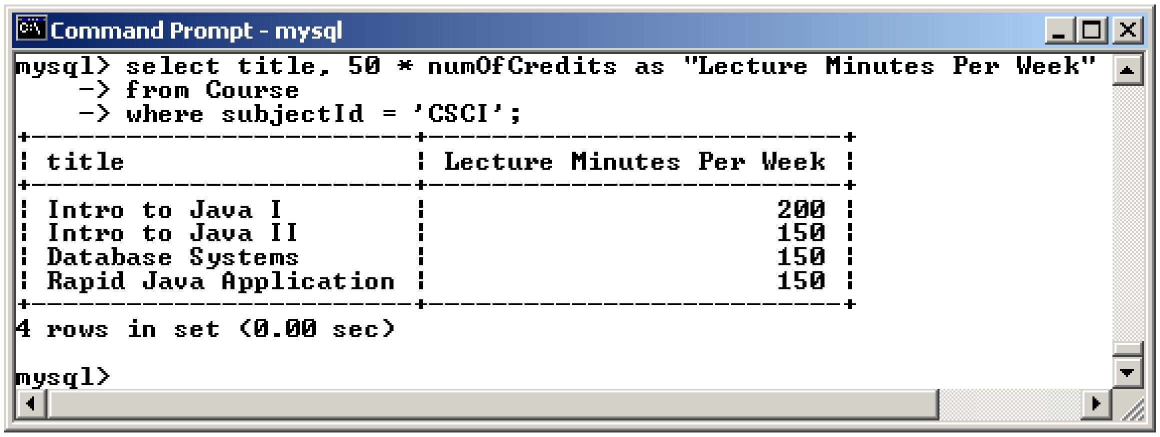
> NOTE: The as keyword is optional in MySQL and Oracle but is required in MS Access.

26.3.8 The Arithmetic Operators

You can use the arithmetic operators * (multiplication), / (division), + (addition), and - (subtraction) in SQL.

Query 5: Assume each credit hour is fifty minutes of lectures, get the total minutes for each course with the subject CSCI. The query result is shown in Figure 26.14.

```
select title, 50 * numOfCredits as "Lecture Minutes Per Week"
from Course
where subjectId = 'CSCI';
```



```
mysql> select title, 50 * numOfCredits as "Lecture Minutes Per Week"
    -> from Course
    -> where subjectId = 'CSCI';
+-------------------------+--------------------------+
| title                   | Lecture Minutes Per Week |
+-------------------------+--------------------------+
| Intro to Java I         |                      200 |
| Intro to Java II        |                      150 |
| Database Systems        |                      150 |
| Rapid Java Application  |                      150 |
+-------------------------+--------------------------+
4 rows in set (0.00 sec)

mysql>
```

Figure 26.14

You can use arithmetic operators in SQL.

26.3.9 Displaying Distinct Tuples

SQL provides the distinct keyword, which can be used to suppress duplicate tuples in the output. For example, the following statement displays all the subject IDs used by the courses:

```
select subjectId as "Subject ID"
from Course;
```

This statement displays all the subject IDs. To display distinct tuples, add the distinct keyword in the select clause, as follows:

```
select distinct subjectId as "Subject ID"
from Course;
```

When there is more than one item in the select clause, the distinct keyword applies to all the items that find distinct tuples.

26.3.10 Displaying Sorted Tuples

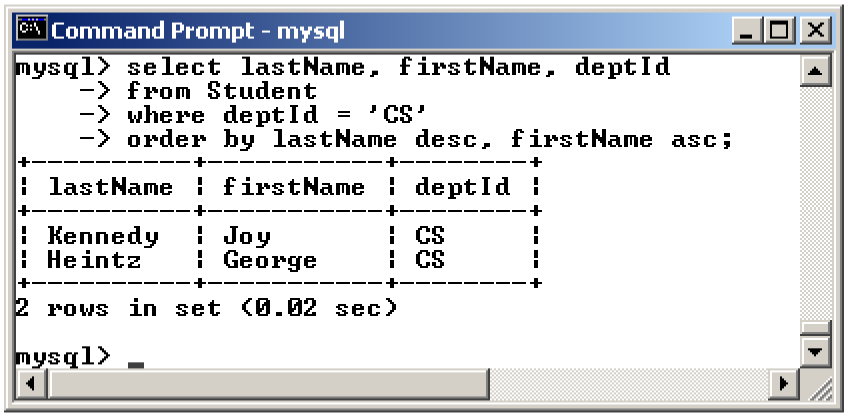
SQL provides the order by clause to sort the output using the following general syntax:

```
select column-list
from table-list
[where condition]
[order by columns-to-be-sorted];
```

In the syntax, columns-to-be-sorted specifies a column or a list of columns to be sorted. By default, the order is ascending. To sort in descending order, append the desc keyword. You could also append the asc keyword, but it is not necessary. When multiple columns are specified, the rows are sorted based on the first column, then the rows with the same values on the second column are sorted based on the second column, and so on.

Query 6: List the full names of the students in the CS department, ordered primarily on their last names in ascending order and secondarily on their first names in ascending order. The query result is shown in Figure 26.15.

```
select lastName, firstName, deptId
from Student
where deptId = 'CS'
order by lastName desc, firstName asc;
```



```
mysql> select lastName, firstName, deptId
    -> from Student
    -> where deptId = 'CS'
    -> order by lastName desc, firstName asc;
+-----------+-----------+--------+
| lastName  | firstName | deptId |
+-----------+-----------+--------+
| Kennedy   | Joy       | CS     |
| Heintz    | George    | CS     |
+-----------+-----------+--------+
2 rows in set (0.02 sec)

mysql> _
```

Figure 26.15

You can sort results using the order by clause.

26.3.11 Joining Tables

Often you need to get information from multiple tables, as demonstrated in the next query.

Query 7: List the courses taken by student Jacob Smith. To solve this query, you need to join tables Student and Enrollment, as shown in Figure 26.16.

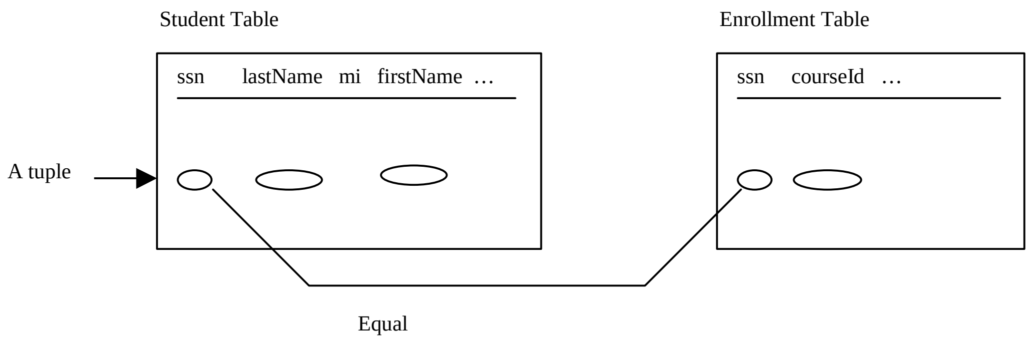


Figure 26.16

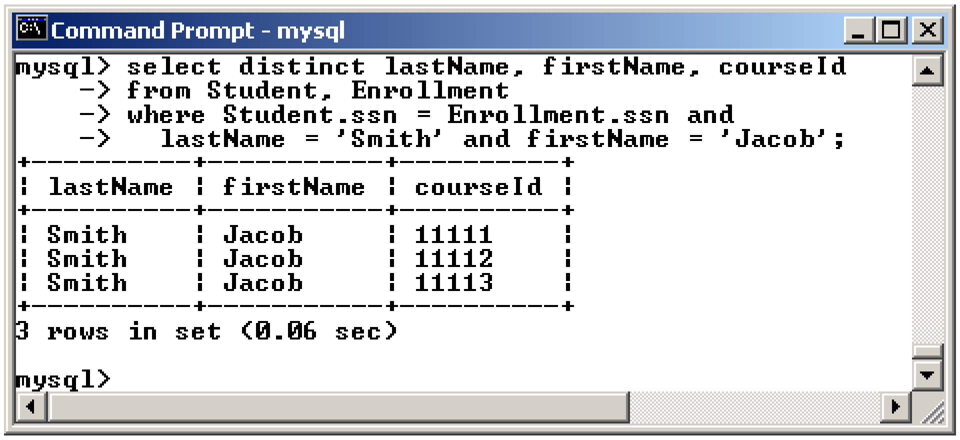
Student and Enrollment are joined on ssn.

You can write the query in SQL:

```
select distinct lastName, firstName, courseId
from Student, Enrollment
where Student.ssn = Enrollment.ssn and
  lastName = 'Smith' and firstName = 'Jacob';
```

The tables Student and Enrollment are listed in the from clause. The query examines every pair of rows, each made of item one from Student and another from Enrollment, and selects the pairs that satisfy the condition in the where clause. The rows in Student have the last name

Smith and the first name Jacob, and both rows from Student and Enrollment have the same ssn values. For each pair selected, lastName and firstName from Student and courseId from Enrollment are used to produce the result, as shown in Figure 26.17. Student and Enrollment have the same attribute ssn. To distinguish them in a query, use Student.ssn and Enrollment.ssn.



```
mysql> select distinct lastName, firstName, courseId
    -> from Student, Enrollment
    -> where Student.ssn = Enrollment.ssn and
    ->   lastName = 'Smith' and firstName = 'Jacob';
+----------+-----------+----------+
| lastName | firstName | courseId |
+----------+-----------+----------+
| Smith    | Jacob     | 11111    |
| Smith    | Jacob     | 11112    |
| Smith    | Jacob     | 11113    |
+----------+-----------+----------+
3 rows in set (0.06 sec)

mysql>
```

Figure 26.17

Query 7 demonstrates quires involving multiple tables.

26.4 JDBC

The Java API for developing Java database applications is called *JDBC*. JDBC is the trademarked name of a Java API that supports Java programs that access relational databases. JDBC is not an acronym, but it is often thought to stand for Java Database Connectivity.

JDBC provides Java programmers with a uniform interface for accessing and manipulating a wide range of relational databases. Using the JDBC API, applications written in the Java programming language can execute SQL statements, retrieve results, present data in a user-friendly interface, and propagate changes back to the database. The JDBC API can also be used to interact with multiple data sources in a distributed, heterogeneous environment.

The relationships between Java programs, JDBC API, JDBC drivers, and relational databases are shown in Figure 26.18. The JDBC API is a set of Java interfaces and classes used to write Java programs for accessing and manipulating relational databases. Since a JDBC driver serves as the interface to facilitate communications between JDBC and a proprietary database, JDBC drivers are database-specific. You need MySQL JDBC drivers to access MySQL database, and Oracle JDBC drivers to access Oracle database. Even for the same vendor, the drivers may be different for different versions of a database. For instance, the JDBC driver for Oracle 8/9 is different from the one for Oracle 8. A JDBC-ODBC bridge driver is included in JDK to support Java programs that access databases through ODBC drivers. An ODBC driver is preinstalled on Microsoft Windows 98, NT, 2000, and XP. You can use the JDBC-ODBC driver to access Microsoft Access database.

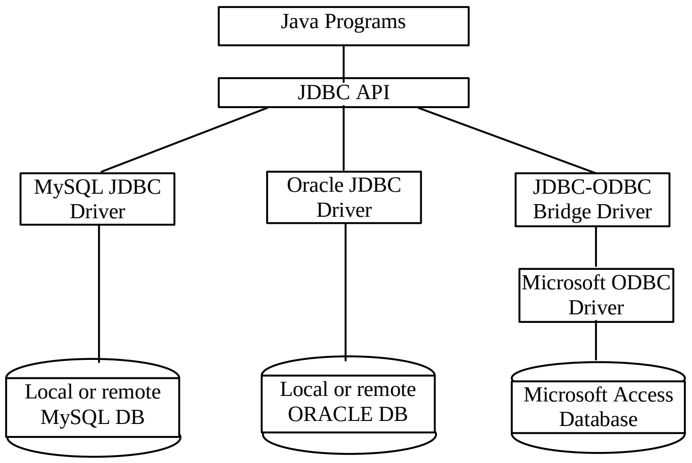


Figure 26.18

Java programs access and manipulate databases through JDBC drivers.

26.4.1 Developing Database Applications Using JDBC

The JDBC API is a Java application program interface to generic SQL databases that enables Java developers to develop DBMS-independent Java applications using a uniform interface.

The JDBC API consists of classes and interfaces for establishing connections with databases, sending SQL statements to databases, processing the results of the SQL statements, and obtaining database metadata. Four key interfaces are needed to develop any database application using Java: Driver, Connection, Statement, and ResultSet. These interfaces define a framework for generic SQL database access. The JDBC driver vendors provide implementation for them. The relationship of these interfaces is shown in Figure 26.19. A JDBC application loads an appropriate driver using the Driver interface, connects to the database using the Connection interface, creates and executes SQL statements using the Statement interface, and processes the result using the ResultSet interface if the statements return results. Note that some statements, such as SQL data definition statements and SQL data modification statements, do not return results.

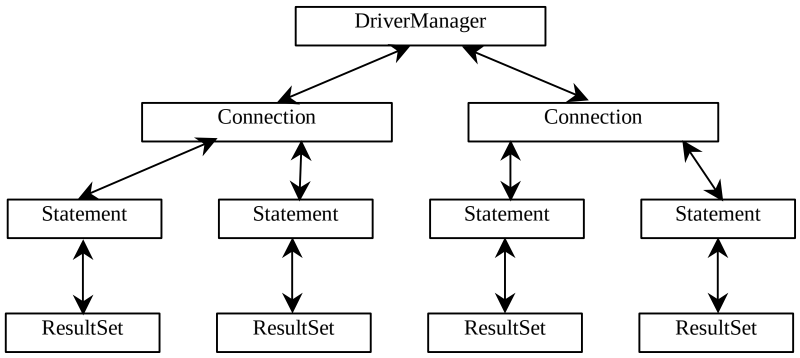


Figure 26.19

JDBC classes enable Java programs to connect to the database, send SQL statements, and process results.

The JDBC interfaces and classes are the building blocks in the development of Java database programs. A typical Java program takes the steps outlined below to access the database.

1. Loading drivers.

An appropriate driver must be loaded using the following statement before connecting to a database.

```
Class.forName("JDBCDriverClass");
```

A driver is a class. The drivers for Access, MySQL, and Oracle are listed in Table 26.3.

Table 26.3

JDBC Drivers

Database	**Driver Class**	**Source**
Access	sun.jdbc.odbc.JdbcOdbcDriver	already in JDK
MySQL	com.mysql.jdbc.Driver	Companion CD-ROM
Oracle	oracle.jdbc.driver.OracleDriver	Oracle Website

The JDBC-ODBC driver for Access is bundled in JDK. MySQL JDBC driver is contained in mysqljdbc.jar in the companion CD-ROM of this book. Oracle JDBC driver is contained in classes12.jar, which can be obtained from www.oracle.com. To use the MySQL and Oracle drivers, you have to add mysqljdbc.jar and classes12.jar in the classpath using the following DOS command on Windows:

```
classpath=%classpath%;c:\book\mysqljdbc.jar;c:\book\classes12.jar
```

If your program accesses several different databases, all their respective drivers must be loaded.

2. Establishing connections.

To connect to a database, use the static method getConnection(databaseURL) in the DriverManager class as follows:

```
Connection connection = DriverManager.getConnection(databaseURL);
```

where databaseURL is the unique identifier of the database on the Internet. Table 26.y lists the URL for MySQL, Oracle, and Access databases.

Table 26.3

JDBC URLs

Database	**URL Pattern**
Access	jdbc:odbc:dataSource
MySQL	jdbc:mysql://hostname/dbname
Oracle	jdbc:oracle:thin:@hostname:port#:oracleDBSID

For an ODBC data source, its databaseURL is jdbc:odbc:dataSource. An ODBC data source can be created using the ODBC Data Source Administrator on Windows. See Supplement N, “Tutorial for Microsoft Access,” on how to create an ODBC data source for an Access database. Suppose a data source named ExampleMDBDataSource has been created for an Access database. The following statement creates a Connection object:

```
Connection connection = DriverManager.getConnection
  ("jdbc:odbc:ExampleMDBDataSource");
```

The databaseURL for a MySQL database specifies the hostname and database name to locate a database. For example, the following statement creates a Connection object for the local MySQL database test:

```
Connection connection = DriverManager.getConnection
  ("jdbc:mysql://localhost/test");
```

Recalled that by default MySQL contains two databases named *mysql* and *test*. You can create custom database using the MySQL SQL command create database *databasename*.

The databaseURL for an Oracle database specifies the *hostname*, the *port#* where the database listens for incoming connection request, and *oracleDBSID* database name to locate a database. For example, the following statement creates a Connection object for the Oracle database on liang.armstrong.edu with username scott and password tiger:

```
Connection connection = DriverManager.getConnection
  ("jdbc:oracle:thin:@liang.armstrong.edu:1521:ora9i",
   "scott", "tiger");
```

3. Creating statements.

If a Connection object can be envisioned as a cable linking your program to a database, an object of Statement or its subclass can be viewed as a cart that delivers SQL statements for execution by the database and brings the result back to the program. Once a Connection object is created, you can create statements for executing SQL statements as follows:

```
Statement statement = connection.createStatement();
```

4. Executing statements.

A SQL DDL or update statement can be executed using executeUpdate(String sql) and a SQL query statement can be executed using executeQuery(String sql). The result of the query is returned in ResultSet. For example, the following code executes the SQL statement *create table Temp (col1 char(5), col2 char(5))*:

```
statement.executeUpdate
  ("create table Temp (col1 char(5), col2 char(5))");
```

The follow code executes the SQL query *select firstName, mi, lastName from Student where lastName = 'Smith'*:

```
// Select the columns from the Student table
ResultSet resultSet = stmt.executeQuery
  ("select firstName, mi, lastName from Student where lastName "
    + " = 'Smith'");
```

5. Processing ResultSet.

The ResultSet maintains a table whose current row can be retrieved. The initial row position is null. You can use the next method to move to the next row and use the various get methods to retrieve values from a current row. For example, the following code displays all result from the preceding SQL query.

```
// Iterate through the result and print the student names
while (resultSet.next())
  System.out.println(rset.getString(1) + " " + rset.getString(2)
    + ". " + rset.getString(3));
```

The getString(1), getString(2), and getString(3) methods retrieve the column values for firstName, mi, and lastName, respectively. Alternatively, you can use getString("firstName"), getString("mi"), and getString("lastName") to retrieve the same three column values. The first execution of the next() method sets the current row to the first row in the result set, and subsequent invocations of the next() method set the current row to the second, third, and so on, to the last row.

Here is the complete example that demonstrates connecting to a database, executing a simple query, and processing the query result with JDBC. The program connects to a local MySQL database and displays the students whose last name is Smith.

******PD: Please add line numbers in the following code******

```
import java.sql.*;

public class SimpleJdbc {
  public static void main(String[] args)
    throws SQLException, ClassNotFoundException {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

    // Establish a connection
    Connection connection = DriverManager.getConnection
      ("jdbc:mysql://localhost/test");
    System.out.println("Database connected");

    // Create a statement
    Statement statement = connection.createStatement();

    // Execute a statement
    ResultSet resultSet = statement.executeQuery
      ("select firstName, mi, lastName from Student where lastName "
        + " = 'Smith'");

    // Iterate through the result and print the student names
    while (resultSet.next())
      System.out.println(resultSet.getString(1) + "\t" +
        resultSet.getString(2) + "\t" + resultSet.getString(3));

    // Close the connection
    connection.close();
  }
}
```

The statement in Line 7 loads a JDBC driver for MySQL and the statement in Lines 11-12 connects to a local MySQL database. You may change them to connect to an Access or an Oracle database. The last statement (Line 29) closes the connection and releases resource related to the connection.

NOTE: Do not use a semicolon (;) to end the Oracle SQL command in a Java program. The semicolon does not work with the Oracle JDBC drivers. It does, however, work with the other drivers used in the book.

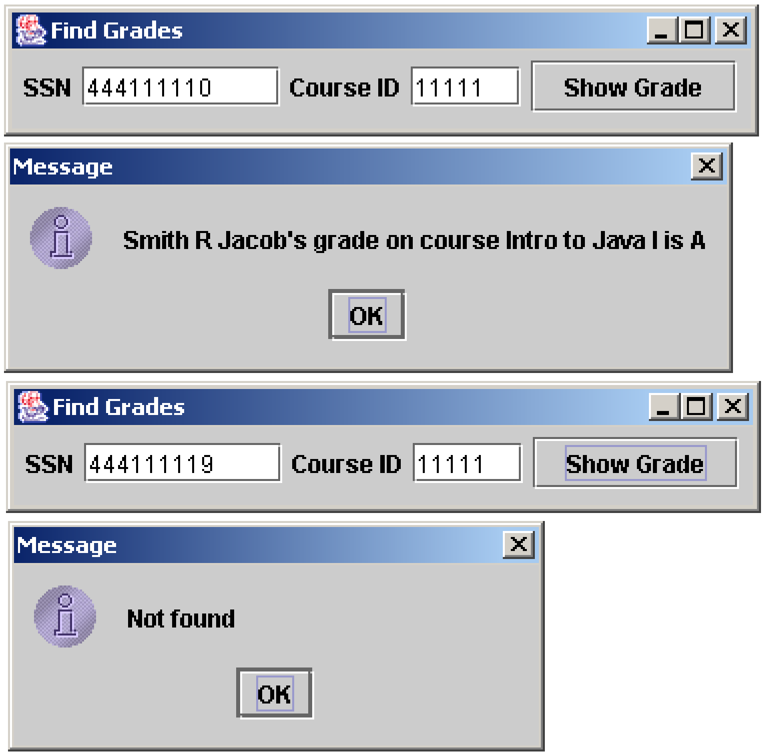
NOTE: The Connection interface handles transactions and specifies how they are processed. By default, a new connection is in auto-commit mode, and all its SQL statements are executed and committed as individual transactions. The commit occurs when the statement completes or the next execute occurs, whichever comes first. In the case of statements returning a result set, the statement completes when the last row of the result set has been retrieved or the result set has been closed. If a single statement returns multiple results, the commit occurs when all results have been retrieved. You can use the setAutoCommit(false) method to disable auto-commit, so that all SQL statements are grouped into one transaction that is terminated by a call to either the commit() or the rollback() method. The rollback() method undoes all changes made by the transaction.

Example 26.1

Accessing a Database from a Java Applet

Problem

This example demonstrates connecting to a database from a Java applet. The applet lets the user enter the SSN and the course ID to find a student's grade, as shown in Figure 26.20.



SSN 444111110 Course ID 11111

Smith R Jacob's grade on course Intro to Java I is A

SSN 444111119 Course ID 11111

Not found

Figure 26.20

A Java applet can access the database on the server.

Solution

Using the JDBC-ODBC bridge driver, your program cannot run as an applet from a Web browser because the ODBC driver contains non-Java native code. The JDBC drivers for MySQL and Oracle are written in Java and can run from the JVM in a Web browser. The code below uses the MySQL database on the host liang.armstrong.edu:

PD: Please add line numbers in the following code

```
import javax.swing.*;
import java.sql.*;
import java.awt.*;
import java.awt.event.*;

public class FindGrade extends JApplet {
  private JTextField jtfSSN = new JTextField(9);
  private JTextField jtfCourseId = new JTextField(5);
  private JButton jbtShowGrade = new JButton("Show Grade");

  // Statement for executing queries
  private Statement stmt;

  /** Initialize the applet */
  public void init() {
    // Initialize database connection and create a Statemet object
    initializeDB();

    jbtShowGrade.addActionListener(
      new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jbtShowGrade_actionPerformed(e);
      }
    });

    JPanel jPanel1 = new JPanel();
    jPanel1.add(new JLabel("SSN"));
    jPanel1.add(jtfSSN);
    jPanel1.add(new JLabel("Course ID"));
    jPanel1.add(jtfCourseId);
    jPanel1.add(jbtShowGrade);

    this.getContentPane().add(jPanel1, BorderLayout.NORTH);
  }

  private void initializeDB() {
    try {
      // Load the JDBC driver
      Class.forName("com.mysql.jdbc.Driver");
//        Class.forName("oracle.jdbc.driver.OracleDriver");
      System.out.println("Driver loaded");

      // Establish a connection
      Connection connection = DriverManager.getConnection
        ("jdbc:mysql://liang.armstrong.edu/test");
//      ("jdbc:oracle:thin:@liang.armstrong.edu:1521:ora9i",
//       "scott", "tiger");
      System.out.println("Database connected");

      // Create a statement
      stmt = connection.createStatement();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  private void jbtShowGrade_actionPerformed(ActionEvent e) {
    String ssn = jtfSSN.getText();
    String courseId = jtfCourseId.getText();
    try {
      String queryString = "select firstName, mi, " +
        "lastName, title, grade from Student, Enrollment, Course " +
        "where Student.ssn = '" + ssn + "' and Enrollment.courseId "
        + "= '" + courseId +
        "' and Enrollment.courseId = Course.courseId " +
        " and Enrollment.ssn = Student.ssn";

      ResultSet rset = stmt.executeQuery(queryString);

      if (rset.next()) {
        String lastName = rset.getString(1);
        String mi = rset.getString(2);
        String firstName = rset.getString(3);
        String title = rset.getString(4);
        String grade = rset.getString(5);

        // Display result in a dialog box
        JOptionPane.showMessageDialog(null, firstName + " " + mi +
          " " + lastName + "'s grade on course " + title + " is " +
          grade);
      } else {
        // Display result in a dialog box
        JOptionPane.showMessageDialog(null, "Not found");
      }
    }
    catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

Review

The initializeDB() method (Lines 36-53) loads the MySQL driver (Line 39), connects to the MySQL database on host liang.armstrong.edu (Lines 43-44), and creates a statement (Line 48).

You can run the applet standalone from the main method or test the applet using the appletviewer utility, as shown in Figure 26.20. If this applet is deployed on the server where the database is located, any client on the Internet can run it from a Web browser. Since the client may not have an Oracle driver, you should make the driver available along with the applet in one archive file. This archive file can be created as follows:

1. Copy c:\book\mysqljdbc.jar to FindGrade.zip.
2. Add FindGrade.class into FindGrade.zip using the WinZip utility.
3. Add FindGrade$1.class into FindGrade.zip using the WinZip utility. FindGrade$1.class is for the anonymous inner event adapter class for listening to the button action.

You need to deploy FindGrade.zip and FindGrade.html on the server. FindGrade.html should use the applet tag with a reference to the Zip file, as follows:

```
<applet
  code="FindGrade"
  archive="FindGrade.zip"
  width=380
  height=80
>
</applet>
```

NOTE: To access the database from an applet, it is necessary, because of security restrictions, for the applet to be downloaded from the server where the database is located. Hence, you have to deploy the applet on the server.

26.5 PreparedStatement

Once a connection to a particular database is established, it can be used to send SQL statements from your program to the database. The Statement interface is used to execute static SQL statements that contain no parameters. The PreparedStatement interface, extending Statement, is used to execute a precompiled SQL statement with or without IN parameters. Since the SQL statements are precompiled, they are efficient for repeated executions.

A PreparedStatement object is created using the preparedStatement method in the Connection interface. For example, the following code creates a PreparedStatement pstmt on a particular Connection connection for an SQL insert statement.

```
Statement pstmt = connection.prepareStatement
  ("insert into Student (firstName, mi, lastName) +
    values (?, ?, ?)");
```

This insert statement has three question marks as placeholders for parameters representing values for firstName, mi, and lastName in a record of the Student table.

As a subinterface of Statement, the PreparedStatement interface inherits all the methods defined in Statement. Additionally, it provides the methods for setting parameters in the object of PreparedStatement. These methods are used to set the values for the parameters before executing statements or procedures. In general, the set methods have the following name and signature:

```
setX(int parameterIndex, X value);
```

where *X* is the type of the parameter, and parameterIndex is the index of the parameter in the statement. The index starts from 1. For example, the method setString(int parameterIndex, String value) sets a String value to the specified parameter.

The following statements pass the parameters "Jack", "A", "Ryan" to the placeholders for firstName, mi, and lastName in PreparedStatement pstmt.

```
pstmt.setString(1, "Jack");
pstmt.setString(2, "A");
pstmt.setString(3, "Ryan");
```

After setting the parameters, you can execute the prepared statement by invoking executeQuery() for a SELECT statement and invoking executeUpdate() for a DDL or update statement. The executeQuery() and executeUpdate() methods are similar to the ones defined in the Statement interface except that they have no parameters, since the SQL statements are already specified in the preparedStatement method when the object of PreparedStatement is created.

Example 26.2

Using PreparedStatement to Execute Dynamic SQL Statements

Problem

This example rewrites the preceding example using PreparedStatement.

Solution

The program is given as follows:

PD: Please add line numbers in the following code

```
import javax.swing.*;
import java.sql.*;
import java.awt.*;
import java.awt.event.*;

public class FindGradeUsingPreparedStatement extends JApplet {
  boolean isStandalone = false;
  private JTextField jtfSSN = new JTextField(9);
  private JTextField jtfCourseId = new JTextField(5);
  private JButton jbtShowGrade = new JButton("Show Grade");

  // PreparedStatement for executing queries
  private PreparedStatement pstmt;

  /** Initialize the applet */
  public void init() {
    // Initialize database connection and create a Statement object
    initializeDB();

    jbtShowGrade.addActionListener(
      new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jbtShowGrade_actionPerformed(e);
      }
    });

    JPanel jPanel1 = new JPanel();
    jPanel1.add(new JLabel("SSN"));
    jPanel1.add(jtfSSN);
    jPanel1.add(new JLabel("Course ID"));
    jPanel1.add(jtfCourseId);
    jPanel1.add(jbtShowGrade);

    this.getContentPane().add(jPanel1, BorderLayout.NORTH);
  }

  private void initializeDB() {
    try {
      // Load the JDBC driver
      Class.forName("com.mysql.jdbc.Driver");
//        Class.forName("oracle.jdbc.driver.OracleDriver");
      System.out.println("Driver loaded");

      // Establish a connection
      Connection connection = DriverManager.getConnection
        ("jdbc:mysql://liang.armstrong.edu/test");
//      ("jdbc:oracle:thin:@liang.armstrong.edu:1521:ora9i",
//       "scott", "tiger");
      System.out.println("Database connected");

      String queryString = "select firstName, mi, " +
        "lastName, title, grade from Student, Enrollment, Course " +
        "where Student.ssn = ? and Enrollment.courseId = ? " +
        "and Enrollment.courseId = Course.courseId";

      // Create a statement
      pstmt = connection.prepareStatement(queryString);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  private void jbtShowGrade_actionPerformed(ActionEvent e) {
    String ssn = jtfSSN.getText();
    String courseId = jtfCourseId.getText();
    try {
      pstmt.setString(1, ssn);
      pstmt.setString(2, courseId);
      ResultSet rset = pstmt.executeQuery();

      if (rset.next()) {
        String lastName = rset.getString(1);
        String mi = rset.getString(2);
        String firstName = rset.getString(3);
        String title = rset.getString(4);
        String grade = rset.getString(5);
        // Display result in a dialog box
        JOptionPane.showMessageDialog(null, firstName + " " + mi +
          " " + lastName + "'s grade on course " + title + " is " +
          grade);
      }
      else {
        // Display result in a dialog box
        JOptionPane.showMessageDialog(null, "Not found");
      }
    }
    catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

Review

This example does exactly the same thing as Example 26.1, "Accessing a Database from a Java Applet," except that it uses the prepared statement to dynamically set the parameters. The code in this example is almost the same as in the preceding example. The new code is highlighted in bold.

A prepared query string is defined in Lines 59-62 with ssn and course ID as parameters. An SQL prepared statement is defined in Line 65. Before executing the query, the actual values of ssn and courseId are set to the parameters in Lines 93-94. Line 95 executes the prepared statement.

26.6 Retrieving Metadata

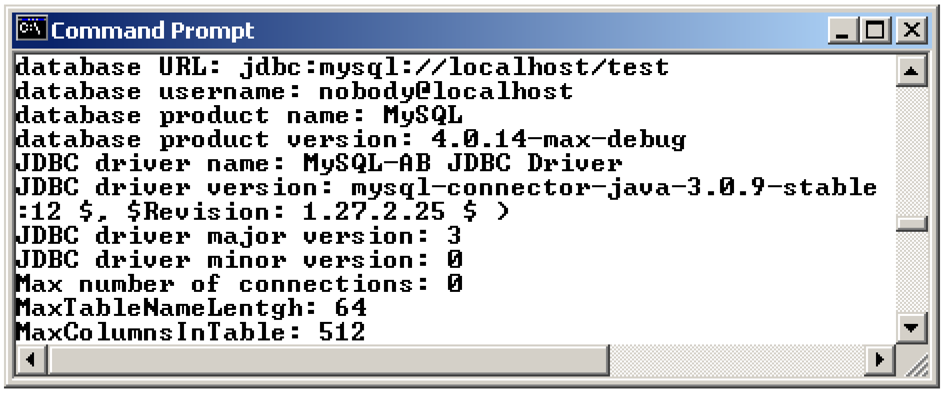
The Connection interface establishes a connection to a database. Within the context of a connection, SQL statements are executed and results are returned. A connection also provides access to database metadata information that describes database tables, supported SQL grammar, stored procedures, the capabilities of the database, and so on.

JDBC provides the DatabaseMetaData interface for obtaining database-wide information and the ResultSetMetaData interface for obtaining information on the specific ResultSet. To obtain an instance of DatabaseMetaData for a database, use the getMetaData method on a connection object like this:

```
DatabaseMetaData dbMetaData = connection.getMetaData();
```

If your program connects to a local MySQL database, the following statements display the database information, as shown in Figure 26.21.

```
DatabaseMetaData dbMetaData = connection.getMetaData();
System.out.println("database URL: " + dbMetaData.getURL());
System.out.println("database username: " +
  dbMetaData.getUserName());
System.out.println("database product name: " +
  dbMetaData.getDatabaseProductName());
System.out.println("database product version: " +
  dbMetaData.getDatabaseProductVersion());
System.out.println("JDBC driver name: " +
  dbMetaData.getDriverName());
System.out.println("JDBC driver version: " +
  dbMetaData.getDriverVersion());
System.out.println("JDBC driver major version: " +
  new Integer(dbMetaData.getDriverMajorVersion()));
System.out.println("JDBC driver minor version: " +
  new Integer(dbMetaData.getDriverMinorVersion()));
System.out.println("Max number of connections: " +
  new Integer(dbMetaData.getMaxConnections()));
System.out.println("MaxTableNameLentgh: " +
  new Integer(dbMetaData.getMaxTableNameLength()));
System.out.println("MaxColumnsInTable: " +
  new Integer(dbMetaData.getMaxColumnsInTable()));
```



```
database URL: jdbc:mysql://localhost/test
database username: nobody@localhost
database product name: MySQL
database product version: 4.0.14-max-debug
JDBC driver name: MySQL-AB JDBC Driver
JDBC driver version: mysql-connector-java-3.0.9-stable
:12 $, $Revision: 1.27.2.25 $ )
JDBC driver major version: 3
JDBC driver minor version: 0
Max number of connections: 0
MaxTableNameLentgh: 64
MaxColumnsInTable: 512
```

Figure 26.21

The DatabaseMetaData interface enables you to obtain database information.

The ResultSetMetaData interface describes information pertaining to the result set. A ResultSetMetaData object can be used to find out about the types and properties of the columns in a ResultSet. To obtain an instance of ResultSetMetaData, use the getMetaData method on a result set like this:

```
ResultSetMetaData rsMetaData = resultSet.getMetaData();
```

You can use the getColumnCount() method to find the number of columns in the result and the getColumnName(int) method to get the column names. For example, the following code displays all column names and the contents resulting from the SQL SELECT statement *select * from Student*.

```
// Execute a statement
ResultSet resultSet = statement.executeQuery
  ("select * from Student");

ResultSetMetaData rsMetaData = resultSet.getMetaData();
for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
  System.out.print(rsMetaData.getColumnName(i) + "\t");
System.out.println();

// Iterate through the result and print the student names
while (resultSet.next()) {
  for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
    System.out.print(resultSet.getString(i) + "\t");
  System.out.println();
}
```

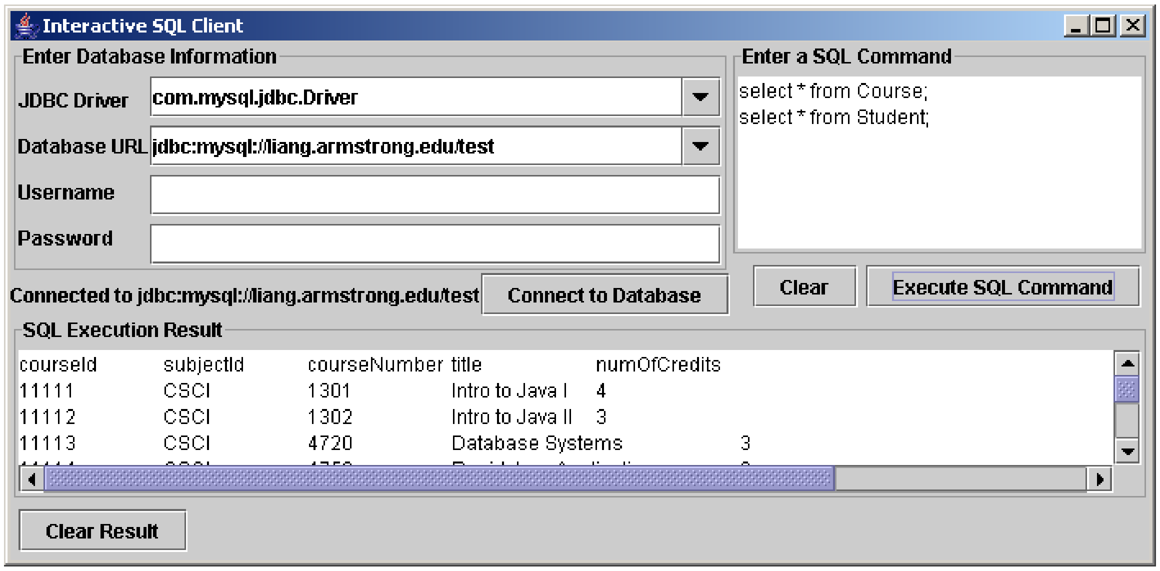
26.7 A Universal SQL Client (Optional)

You have learned how to use various drivers to connect to the database, create statements for executing SQL statements, and process the results from SQL queries. This section presents a universal SQL client that enables you to connect to any relational database and execute SQL commands.

Example 26.3 Creating an Interactive SQL Client

Problem

This example creates a Java applet for submitting and executing SQL commands interactively, as shown in Figure 26.22. The client can connect to any JDBC data source, and submit SQL SELECT commands and non-SELECT commands for execution. The execution result is displayed for the SELECT queries, and the execution status is displayed for the non-SELECT commands.



JDBC Driver: com.mysql.jdbc.Driver

Database URL: jdbc:mysql://liang.armstrong.edu/test

```
select * from Course;
select * from Student;
```

Connected to jdbc:mysql://liang.armstrong.edu/test

```
courseId    subjectId   courseNumber title           numOfCredits
11111       CSCI        1301         Intro to Java I    4
11112       CSCI        1302         Intro to Java II   3
11113       CSCI        4720         Database Systems           3
```

Figure 26.22

You can connect to any JDBC data source and execute SQL commands interactively.

Solution

The following code gives the solution to the problem.

******PD: Please add line numbers in the following code******

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.sql.*;
import java.util.*;
```

```
public class SQLClient extends JApplet {
  // Connection to the database
  private Connection connection;

  // Statement to execute SQL commands
  private Statement statement;

  // Text area to enter SQL commands
  private JTextArea jtasqlCommand = new JTextArea();

  // Text area to display results from SQL commands
  private JTextArea jtaSQLResult = new JTextArea();

  // JDBC info for a database connection
  JTextField jtfUsername = new JTextField();
  JPasswordField jpfPassword = new JPasswordField();
  JComboBox jcboURL = new JComboBox(new String[] {
    "jdbc:mysql://liang.armstrong.edu/test",
    "jdbc:odbc:exampleMDBDataSource",
    "jdbc:oracle:thin:@liang.armstrong.edu:1521:ora9i"});
  JComboBox jcboDriver = new JComboBox(new String[] {
    "com.mysql.jdbc.Driver", "sun.jdbc.odbc.JdbcOdbcDriver",
    "oracle.jdbc.driver.OracleDriver"});

  JButton jbtExecuteSQL = new JButton("Execute SQL Command");
  JButton jbtClearSQLCommand = new JButton("Clear");
  JButton jbtConnectDB1 = new JButton("Connect to Database");
  JButton jbtClearSQLResult = new JButton("Clear Result");

  // Create titled borders
  Border titledBorder1 = new TitledBorder("Enter a SQL Command");
  Border titledBorder2 = new TitledBorder("SQL Execution Result");
  Border titledBorder3 = new TitledBorder(
    "Enter Database Information");

  JLabel jlblConnectionStatus = new JLabel("No connection now");

  /** Initialize the applet */
  public void init() {
    JScrollPane jScrollPane1 = new JScrollPane(jtasqlCommand);
    jScrollPane1.setBorder(titledBorder1);
    JScrollPane jScrollPane2 = new JScrollPane(jtaSQLResult);
    jScrollPane2.setBorder(titledBorder2);

    JPanel jPanel1 = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    jPanel1.add(jbtClearSQLCommand);
    jPanel1.add(jbtExecuteSQL);

    JPanel jPanel2 = new JPanel();
    jPanel2.setLayout(new BorderLayout());
    jPanel2.add(jScrollPane1, BorderLayout.CENTER);
    jPanel2.add(jPanel1, BorderLayout.SOUTH);

    JPanel jPanel3 = new JPanel();
    jPanel3.setLayout(new BorderLayout());
    jPanel3.add(jlblConnectionStatus, BorderLayout.CENTER);
    jPanel3.add(jbtConnectDB1, BorderLayout.EAST);

    JPanel jPanel4 = new JPanel();
    jPanel4.setLayout(new GridLayout(4, 1, 10, 5));
    jPanel4.add(jcboDriver);
    jPanel4.add(jcboURL);
    jPanel4.add(jtfUsername);
    jPanel4.add(jpfPassword);

    JPanel jPanel5 = new JPanel();
    jPanel5.setLayout(new GridLayout(4, 1));
    jPanel5.add(new JLabel("JDBC Driver"));
    jPanel5.add(new JLabel("Database URL"));
    jPanel5.add(new JLabel("Username"));
    jPanel5.add(new JLabel("Password"));

    JPanel jPanel6 = new JPanel();
    jPanel6.setLayout(new BorderLayout());
    jPanel6.setBorder(titledBorder3);
    jPanel6.add(jPanel4, BorderLayout.CENTER);
    jPanel6.add(jPanel5, BorderLayout.WEST);

    JPanel jPanel7 = new JPanel();
    jPanel7.setLayout(new BorderLayout());
    jPanel7.add(jPanel3, BorderLayout.SOUTH);
    jPanel7.add(jPanel6, BorderLayout.CENTER);

    JPanel jPanel8 = new JPanel();
    jPanel8.setLayout(new BorderLayout());
    jPanel8.add(jPanel2, BorderLayout.CENTER);
    jPanel8.add(jPanel7, BorderLayout.WEST);

    JPanel jPanel9 = new JPanel(new FlowLayout(FlowLayout.LEFT));
    jPanel9.add(jbtClearSQLResult);

    jcboURL.setEditable(true);
    jcboDriver.setEditable(true);

    this.getContentPane().add(jPanel8, BorderLayout.NORTH);
    this.getContentPane().add(jScrollPane2, BorderLayout.CENTER);
    this.getContentPane().add(jPanel9, BorderLayout.SOUTH);

    jbtExecuteSQL.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          executeSQL();
        }
    });
    jbtConnectDB1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        connectToDB();
      }
    });
    jbtClearSQLCommand.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jtasqlCommand.setText(null);
      }
    });
    jbtClearSQLResult.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        jtaSQLResult.setText(null);
      }
    });
  }

  /** Connect to DB */
  private void connectToDB() {
    // Get database information from the user input
    String driver = (String)jcboDriver.getSelectedItem();
    String url = (String)jcboURL.getSelectedItem();
    String username = jtfUsername.getText().trim();
    String password = new String(jpfPassword.getPassword());

    // Connection to the database
    try {
      Class.forName(driver);
      connection = DriverManager.getConnection(
        url, username, password);
      jlblConnectionStatus.setText("Connected to " + url);
    }
    catch (java.lang.Exception ex) {
      ex.printStackTrace();
    }
  }

  /** Execute SQL commands */
  private void executeSQL() {
    if (connection == null) {
      jtaSQLResult.setText("Please connect to a database first");
      return;
    }
    else {
      String sqlCommands = jtasqlCommand.getText().trim();
      StringTokenizer commands =
        new StringTokenizer(sqlCommands.replace('\n', ' '), ";");

      while (commands.hasMoreTokens()) {
        String aCommand = commands.nextToken().trim();

        if (aCommand.toUpperCase().startsWith("SELECT")) {
          processSQLSelect(aCommand);
        }
        else {
          processSQLNonSelect(aCommand);
        }
      }
    }
  }

  /** Execute SQL SELECT commands */
  private void processSQLSelect(String sqlCommand) {
    try {
      // Get a new statement for the current connection
      statement = connection.createStatement();

      // Execute a SELECT SQL command
      ResultSet resultSet = statement.executeQuery(sqlCommand);

      // Find the number of columns in the result set
      int columnCount = resultSet.getMetaData().getColumnCount();
      String row = "";

      // Display column names
      for (int i = 1; i <= columnCount; i++) {
        row += resultSet.getMetaData().getColumnName(i) + "\t";
      }

      jtaSQLResult.append(row + '\n');

      while (resultSet.next()) {
        // Reset row to empty
        row = "";

        for (int i = 1; i <= columnCount; i++) {
          row += resultSet.getString(i) + "\t";
        }

        jtaSQLResult.append(row + '\n');
      }
    }
    catch (SQLException ex) {
      jtaSQLResult.setText(ex.toString());
    }
  }

  /** Execute SQL DDL, and modification commands */
  private void processSQLNonSelect(String sqlCommand) {
    try {
      // Get a new statement for the current connection
      statement = connection.createStatement();

      // Execute a non-SELECT SQL command
      statement.executeUpdate(sqlCommand);

      jtaSQLResult.setText("SQL command executed");
    }
    catch (SQLException ex) {
      jtaSQLResult.setText(ex.toString());
    }
  }
}
```

Review

The user selects or enters the JDBC driver, database URL, username, and password, and clicks the *Connect to Database* button to connect to the specified database using the connectToDB() method (Lines 129-146).

When the user clicks the *Execute SQL Command* button, the executeSQL() method is invoked (Lines 149-170) to gets the SQL commands from the text area (jtaSQLCommand), and extracts each command separated by semicolon (;). It then determines whether the command is a SELECT query or a DDL or data modification statement (Lines 162-167). If the command is a SELECT query, the processSQLSelect() method is invoked (Lines 173-206). This method uses the executeQuery method (Line 179) to obtain the query result. The result is displayed in the text area (jtaSQLResult). If the command is a non-SELECT query, the processSQLNonSelect() method is invoked (Lines 209-222). This method uses the executeUpdate method (Line 215) to execute the SQL command.

The getMetaData method (Lines 182, 187) in the ResultSet interface is used to obtain an instance of ResultSetMetaData. The getColumnCount method (Line 180) returns the number of columns in the result set and the getColumnName(i) method (Line 185) returns the column name for the ith column.

26.8 Batch Processing (Optional)

In all the preceding examples, SQL commands are submitted to the database for execution one at a time. This is inefficient for processing a large number of updates. For example, suppose you wanted to insert a thousand rows into a table. Submitting one INSERT command at a time would take nearly a thousand times longer than submitting all the INSERT commands in a batch at once. To improve performance, JDBC 2 introduced the batch update for processing nonselect SQL commands. A batch update consists of a sequence of nonselect SQL commands. These commands are collected in a batch and submitted to the database all together.
To use the batch update, you add nonselect commands to a batch using the addBatch method in the Statement interface. After all the SQL commands are added to the batch, use the executeBatch method to submit the batch to the database for execution.
For example, the following code adds a create table command, two insert statements in a batch, and executes the batch.

```
Statement statement = conn.createStatement();

// Add SQL commands to the batch
statement.addBatch("create table T (C1 integer, C2 varchar(15))");
statement.addBatch("insert into T values (100, 'Smith')");
statement.addBatch("insert into T values (200, 'Jones')");

// Execute the batch
int count[] = statement.executeBatch();
```

The executeBatch() method returns an array of counts, each of which counts the number of the rows affected by the SQL command. The first count returns 0 because it is a DDL command. The rest of the commands return 1 because only one row is affected.

NOTE: To find out whether a driver supports batch updates, invoke supportsBatchUpdates() on a DatabaseMetaData instance. If the driver supports batch updates, it will return true. The JDBC drivers for MySQL, Access, and Oracle all support batch updates.

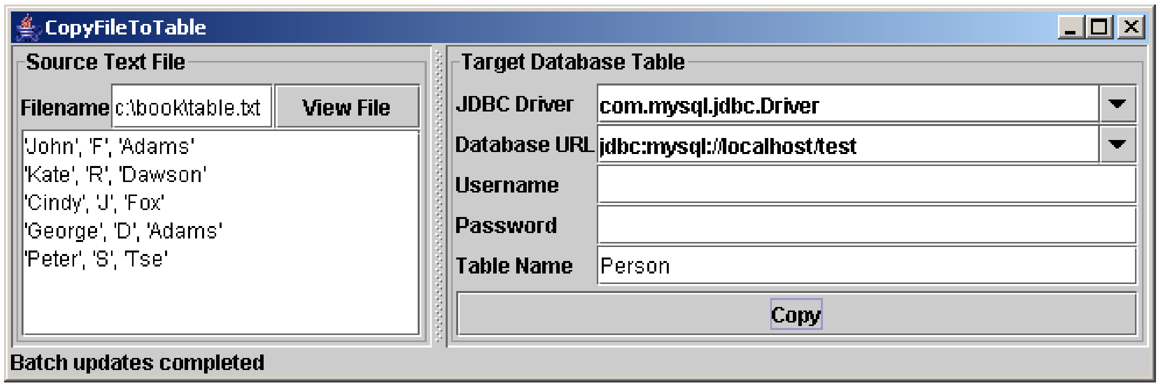
Example 26.4

Copying Text Files to Tables

Problem

In this example, you will write a program that gets data from a text file and copies the data to a table, as shown in Figure 26.23. The text file consists of the lines, each of which corresponds to a row in the table. The fields in a row are separated by commas. The string values in a row are enclosed in single quotes. You can view the text file by clicking the View File button and copy the text to the table by clicking the Copy button. The table must already be defined in the database. Figure 26.23 shows the text file table.txt copied to table Person. Person is created using the following statement:

```
create table Person (
  firstName varchar(20),
  mi char(1),
  lastName varchar(20)
)
```



Filename: c:\book\table.txt

```
'John', 'F', 'Adams'
'Kate', 'R', 'Dawson'
'Cindy', 'J', 'Fox'
'George', 'D', 'Adams'
'Peter', 'S', 'Tse'
```

JDBC Driver: com.mysql.jdbc.Driver
Database URL: jdbc:mysql://localhost/test
Table Name: Person

Batch updates completed

Figure 26.23

The CopyFileToTable utility copies text files to database tables.

Solution

The following code gives the solution to the problem.

******PD: Please add line numbers in the following code******

```
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.sql.*;
```

```
public class CopyFileToTable extends JFrame {
  // Text file info
  private JTextField jtfFilename = new JTextField();
  private JTextArea jtaFile = new JTextArea();

  // JDBC and table info
  private JComboBox jcboDriver = new JComboBox(new String[] {
    "com.mysql.jdbc.Driver", "sun.jdbc.odbc.JdbcOdbcDriver",
    "oracle.jdbc.driver.OracleDriver"});
  private JComboBox jcboURL = new JComboBox(new String[] {
    "jdbc:mysql://localhost/test", "jdbc:odbc:exampleMDBDataSource",
    "jdbc:oracle:thin:@liang.armstrong.edu:1521:ora9i"});
  private JTextField jtfUsername = new JTextField();
  private JPasswordField jtfPassword = new JPasswordField();
  private JTextField jtfTableName = new JTextField();

  private JButton jbtViewFile = new JButton("View File");
  private JButton jbtCopy = new JButton("Copy");
  private JLabel jlblStatus = new JLabel();

  public CopyFileToTable() {
    JPanel jPane1 = new JPanel();
    jPane1.setLayout(new BorderLayout());
    jPane1.add(new JLabel("Filename"), BorderLayout.WEST);
    jPane1.add(jbtViewFile, BorderLayout.EAST);
    jPane1.add(jtfFilename, BorderLayout.CENTER);

    JPanel jPane2 = new JPanel();
    jPane2.setLayout(new BorderLayout());
    jPane2.setBorder(new TitledBorder("Source Text File"));
    jPane2.add(jPane1, BorderLayout.NORTH);
    jPane2.add(new JScrollPane(jtaFile), BorderLayout.CENTER);

    JPanel jPane3 = new JPanel();
    jPane3.setLayout(new GridLayout(5, 0));
    jPane3.add(new JLabel("JDBC Driver"));
    jPane3.add(new JLabel("Database URL"));
    jPane3.add(new JLabel("Username"));
    jPane3.add(new JLabel("Password"));
    jPane3.add(new JLabel("Table Name"));

    JPanel jPane4 = new JPanel();
    jPane4.setLayout(new GridLayout(5, 0));
    jcboDriver.setEditable(true);
    jPane4.add(jcboDriver);
    jcboURL.setEditable(true);
    jPane4.add(jcboURL);
    jPane4.add(jtfUsername);
    jPane4.add(jtfPassword);
    jPane4.add(jtfTableName);

    JPanel jPane5 = new JPanel();
    jPane5.setLayout(new BorderLayout());
    jPane5.setBorder(new TitledBorder("Target Database Table"));
    jPane5.add(jbtCopy, BorderLayout.SOUTH);
    jPane5.add(jPane3, BorderLayout.WEST);
    jPane5.add(jPane4, BorderLayout.CENTER);

    getContentPane().add(jlblStatus, BorderLayout.SOUTH);
    getContentPane().add(new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
      jPane2, jPane5), BorderLayout.CENTER);

    jbtViewFile.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        showFile();
      }
    });

    jbtCopy.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          copyFile();
        }
        catch (Exception ex) {
          jlblStatus.setText(ex.toString());
        }
      }
    });
  }

  /** Display the file in the text area */
  private void showFile() {
    // Use a BufferedReader to read text from the file
    BufferedReader infile = null;

    // Get file name from the text field
    String filename = jtfFilename.getText().trim();

    String inLine;

    try {
      // Create a buffered stream
      infile = new BufferedReader(new FileReader(filename));

      // Read a line and append the line to the text area
      while ((inLine = infile.readLine()) != null) {
        jtaFile.append(inLine + '\n');
      }
    }
    catch (FileNotFoundException ex) {
      System.out.println("File not found: " + filename);
    }
    catch (IOException ex) {
      System.out.println(ex.getMessage());
    }
    finally {
      try {
        if (infile != null) infile.close();
      }
      catch (IOException ex) {
        System.out.println(ex.getMessage());
      }
    }
  }

  private void copyFile() throws Exception {
    // Load the JDBC driver
    System.out.println((String)jcboDriver.getSelectedItem());
    Class.forName(((String)jcboDriver.getSelectedItem()).trim());
    System.out.println("Driver loaded");

    // Establish a connection
    Connection conn = DriverManager.getConnection
      (((String)jcboURL.getSelectedItem()).trim(),
    jtfUsername.getText().trim(),
    String.valueOf(jtfPassword.getPassword()).trim());
    System.out.println("Database connected");

    // Read each line from the text file and insert it to the table
    insertRows(conn);
  }

  private void insertRows(Connection connection) {
    // Build the INSERT statement
    String sqlInsert = "insert into " + jtfTableName.getText()
      + " values (";

    // Use a BufferedReader to read text from the file
    BufferedReader infile = null;

    // Get file name from the text field
    String filename = jtfFilename.getText().trim();

    String inLine;

    try {
      // Create a buffered stream
      infile = new BufferedReader(new FileReader(filename));

      // Create a statement
      Statement statement = connection.createStatement();

      System.out.println("Driver major version? " +
      connection.getMetaData().getDriverMajorVersion());

      // Determine if the supportsBatchUpdates method supported in
      // DatabaseMetaData
      boolean batchUpdatesSupported = false;

      try {
        if (connection.getMetaData().supportsBatchUpdates()) {
          batchUpdatesSupported = true;
          System.out.println("batch updates supported");
        }
        else {
          System.out.println("The driver is of JDBC 2 type, but " +
          "does not support batch updates");
        }
      }
      catch (UnsupportedOperationException ex) {
        System.out.println("The driver does not support JDBC 2");
      }

      // Determine if the driver is capable of batch updates
      if (batchUpdatesSupported) {
        // Read a line and add the insert table command to the batch
        while ((inLine = infile.readLine()) != null) {
          statement.addBatch(sqlInsert + inLine + ")");
        }

        statement.executeBatch();

        jlblStatus.setText("Batch updates completed");
      }
      else {
        // Read a line and execute insert table command
        while ((inLine = infile.readLine()) != null) {
          statement.executeUpdate(sqlInsert + inLine + ")");
        }

        jlblStatus.setText("Single row update completed");
      }
    }
    catch (SQLException ex) {
      System.out.println(ex);
    }
    catch (FileNotFoundException ex) {
      System.out.println("File not found: " + filename);
    }
    catch (IOException ex) {
      System.out.println(ex.getMessage());
    }
    finally {
      try {
        if (infile != null) infile.close();
      }
      catch (IOException ex) {
        System.out.println(ex.getMessage());
      }
    }
  }
}
```

Review

The insertRows method (Lines 140-218) uses the batch updates to submit SQL INSERT commands to the database for execution, if the driver supports batch updates. Lines 178-175 check whether the driver supports batch updates. If the driver is not JDBC 2 compatible, an UnsupportedOperationException exception will be thrown (Line 177) when the supportsBatchUpdates() method is invoked.

The tables must be already created in the database. The file format and contents must match the database table specification. Otherwise, the SQL INSERT command would fail.

In Exercise 26.4, you will write a program to insert a thousand records to a database and compare the performance with and without batch updates.

26.9 Scrollable and Updateable Result Set (Optional)

The result sets used in the preceding examples are read sequentially. A result set maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next() method moves the cursor forward to the next row. This is known as *sequential forward reading*. It is only way of processing the rows in a result set that is supported by JDBC 1.

With JDBC 2, you can scroll the rows both forward and backward and move the cursor to a desired location using the first, last, next, previous, absolute, or relative method. Additionally, you can insert, delete, or update a row in the result set and have the changes automatically reflected in the database.

To obtain a scrollable or updateable result set, you must first create a statement with an appropriate type and concurrency mode. For a static statement, use

```
Statement statement = connection.createStatement
  (int resultSetType, int resultSetConcurrency);
```

For a prepared statement, use

```
PreparedStatement statement = connection.prepareStatement
  (String sql, int resultSetType, int resultSetConcurrency);
```

For a callable statement, use

```
CallableStatement statement = connection.prepareCall
  (String sql, int resultSetType, int resultSetConcurrency);
```

The possible values of resultSetType are the constants defined in the ResultSet:

[BL] TYPE_FORWARD_ONLY: The result set is accessed forward sequentially.

[BL] TYPE_SCROLL_INSENSITIVE: The result set is scrollable, but not sensitive to changes in the database.

[BX] TYPE_SCROLL_SENSITIVE: The result set is scrollable and sensitive to changes made by others. Use this type if you want the result set to be scrollable and updateable.

The possible values of resultSetConcurrency are the constants defined in the ResultSet:

[BL]CONCUR_READ_ONLY: The result set cannot be used to update the database.

[BX]CONCUR_UPDATABLE: The result set can be used to update the database.

For example, if you want the result set to be scrollable and updateable, you can create a static statement:

```
Statement statement = connection.createStatement
  (ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)
```

You use the executeQuery method in a Statement object to execute an SQL query that returns a result set as follows:

```
ResultSet resultSet = statement.executeQuery(query);
```

The methods first(), next(), previous(), and last() are used to move the cursor to the first row, next row, previous row, and last row; The absolute(int row) method moves the cursor to the specified row; and the get*Xxx*(int columnIndex) or get*Xxx*(String columnName) method is used to retrieve the value of a specified field at the current row. The methods insertRow(), deleteRow(), and updateRow() can also be used to insert, delete, and update the current row. Before applying insertRow or updateRow, you need to use the method update(int columnIndex, *Xxx* value) or update(String columnName, *Xxx* value) to write a new value to the field at the current row. The cancelRowUpdates() method cancels the updates made to a row. The close() method closes the result set and releases its resource. The boolean wasNull() method indicates whether the last column read had a value of SQL NULL.

Example 26.5

Scrolling and Updating Tables

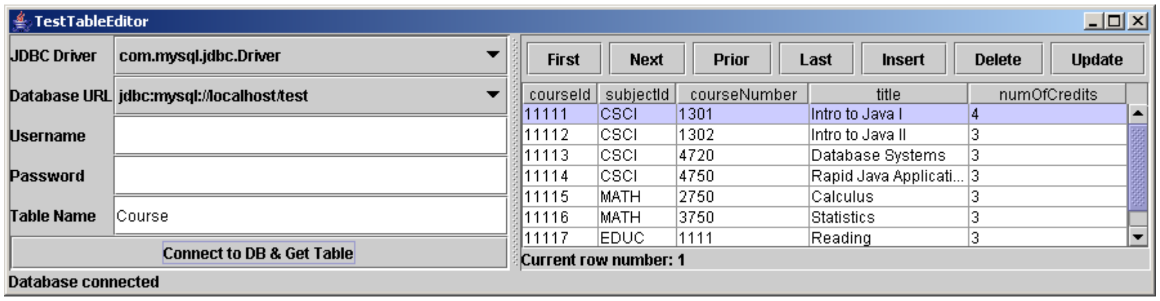
Problem

In this example, you will develop a useful utility that displays all the rows of a database table in a JTable and uses a scrollable and updateable result set to navigate the table and modify its contents.

As shown in Figure 26.24, you enter or select a JDBC Driver and Database, enter a username and a password, and specify a table name to connect the database and display the table contents in the JTable. You can then use the buttons First, Next, Prior, and Last to move the cursor to the first row, next row, previous row, and last row in the table, and use the buttons

Insert, Delete, and Update to modify the table contents. When clicking the Insert button, a dialog box is displayed to receive input, as shown in Figure 26.25.

The status bar at the bottom of the window shows the current row in the result set and the JTable. The cursor in the result set and the row in the JTable are synchronized. You can move the cursor by using the navigation buttons or by selecting a row in the JTable.


TestTableEditor

JDBC Driver: com.mysql.jdbc.Driver
Database URL: jdbc:mysql://localhost/test
Username:
Password:
Table Name: Course

Connect to DB & Get Table

First | Next | Prior | Last | Insert | Delete | Update

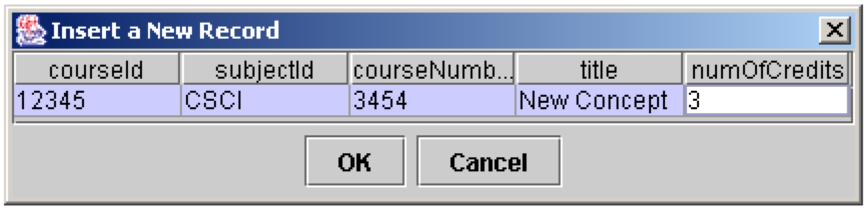
courseId	subjectId	courseNumber	title	numOfCredits
11111	CSCI	1301	Intro to Java I	4
11112	CSCI	1302	Intro to Java II	3
11113	CSCI	4720	Database Systems	3
11114	CSCI	4750	Rapid Java Applicati...	3
11115	MATH	2750	Calculus	3
11116	MATH	3750	Statistics	3
11117	EDUC	1111	Reading	3

Current row number: 1

Database connected

Figure 26.24

The program enables you to navigate and modify the table.


Insert a New Record

courseId	subjectId	courseNumb...	title	numOfCredits
12345	CSCI	3454	New Concept	3

OK | Cancel

Figure 26.25

The Insert a New Record dialog box lets the user to enter a new record.

Solution

Create three classes: TestTableEditor, TableEditor, and NewRecordDialog. TestTableEditor is the main class that enables the user to enter the database connection information and a table name. Once the database is connected, the table contents are displayed in an instance of TableEditor. The TableEditor class can be used to browse a table and modify a table. An instance of NewRecordDialog is displayed to let you enter a new record when you click the Insert button in TableEditor.

PD: Please add line numbers in the following code

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.sql.*;

public class TestTableEditor extends JApplet {
  private JComboBox jcboDriver = new JComboBox(new String[] {
    "com.mysql.jdbc.Driver", "oracle.jdbc.driver.OracleDriver",
    "sun.jdbc.odbc.JdbcOdbcDriver"});
  private JComboBox jcboURL = new JComboBox(new String[] {
    "jdbc:mysql://localhost/test", "jdbc:odbc:exampleMDBDataSource",
    "jdbc:oracle:thin:@liang.armstrong.edu:1521:ora9i"});

  private JButton jbtConnect =
    new JButton("Connect to DB & Get Table");
  private JTextField jtfUserName = new JTextField();
  private JPasswordField jpfPassword = new JPasswordField();
  private JTextField jtfTableName = new JTextField();
  private TableEditor tableEditor1 = new TableEditor();
  private JLabel jlblStatus = new JLabel();

  /** Creates new form TestTableEditor */
  public TestTableEditor() {
    JPanel jPane1 = new JPanel();
    jPane1.setLayout(new GridLayout(5, 0));
    jPane1.add(jcboDriver);
    jPane1.add(jcboURL);
    jPane1.add(jtfUserName);
    jPane1.add(jpfPassword);
    jPane1.add(jtfTableName);

    JPanel jPanel2 = new JPanel();
    jPanel2.setLayout(new GridLayout(5, 0));
    jPanel2.add(new JLabel("JDBC Driver"));
    jPanel2.add(new JLabel("Database URL"));
    jPanel2.add(new JLabel("Username"));
    jPanel2.add(new JLabel("Password"));
    jPanel2.add(new JLabel("Table Name"));

    JPanel jPane3 = new JPanel();
    jPane3.setLayout(new BorderLayout());
    jPane3.add(jbtConnect, BorderLayout.SOUTH);
    jPane3.add(jPanel2, BorderLayout.WEST);
    jPane3.add(jPane1, BorderLayout.CENTER);
    tableEditor1.setPreferredSize(new Dimension(400, 200));

    getContentPane().add(new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
      jPane3, tableEditor1), BorderLayout.CENTER);
    getContentPane().add(jlblStatus, BorderLayout.SOUTH);

    jbtConnect.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          // Connect to the database
          Connection connection = getConnection();
          tableEditor1.setConnectionAndTable(connection,
            jtfTableName.getText().trim());
        }
        catch (Exception ex) {
          jlblStatus.setText(ex.toString());
        }
      }
    });
  }

  /** Connect to a database */
  private Connection getConnection() throws Exception {
    // Load the JDBC driver
    System.out.println((String)jcboDriver.getSelectedItem());
    Class.forName(((String)jcboDriver.getSelectedItem()).trim());
    System.out.println("Driver loaded");

    // Establish a connection
    Connection connection = DriverManager.getConnection
      (((String)jcboURL.getSelectedItem()).trim(),
    jtfUserName.getText().trim(), jpfPassword.getText().trim());
    jlblStatus.setText("Database connected");

    return connection;
  }
}
```

PD: Please add line numbers in the following code

```
import java.util.*;
import java.sql.*;
import javax.swing.table.*;
import javax.swing.event.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TableEditor extends JPanel {
  // Dialog box for inserting a new record
  private NewRecordDialog newRecordDialog = new NewRecordDialog();

  // JDBC Connection
  private Connection connection;

  // Table name
  private String tableName;

  // JDBC Statement
  private Statement statement;

  // Result set for the table
  private ResultSet resultSet;

  // Table model
  private DefaultTableModel tableModel = new DefaultTableModel();

  // Table selection model
  private DefaultListSelectionModel listSelectionModel =
    new DefaultListSelectionModel();

  // New row vector
  private Vector rowVectors = new Vector();

  // columnHeaderVector to hold column names
  private Vector columnHeaderVector = new Vector();

  // Column count
  private int columnCount;

  private JButton jbtFirst = new JButton("First");
  private JButton jbtNext = new JButton("Next");
  private JButton jbtPrior = new JButton("Prior");
  private JButton jbtLast = new JButton("Last");
  private JButton jbtInsert = new JButton("Insert");
  private JButton jbtDelete = new JButton("Delete");
  private JButton jbtUpdate = new JButton("Update");

  private JLabel jlblStatus = new JLabel();
  private JTable jTable1 = new JTable();

  /** Creates new form TableEditor */
  public TableEditor() {
    jTable1.setModel(tableModel);
    jTable1.setSelectionModel(listSelectionModel);

    JPanel jPanel1 = new JPanel();
    setLayout(new BorderLayout());
    jPanel1.add(jbtFirst);
    jPanel1.add(jbtNext);
    jPanel1.add(jbtPrior);
    jPanel1.add(jbtLast);
    jPanel1.add(jbtInsert);
    jPanel1.add(jbtDelete);
    jPanel1.add(jbtUpdate);

    add(jPanel1, BorderLayout.NORTH);
    add(new JScrollPane(jTable1), BorderLayout.CENTER);
    add(jlblStatus, BorderLayout.SOUTH);

    jbtFirst.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        moveCursor("first");
      }
    });
    jbtNext.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        moveCursor("next");
      }
    });
    jbtPrior.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        moveCursor("previous");
      }
    });
    jbtLast.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        moveCursor("last");
      }
    });
    jbtInsert.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        insert();
      }
    });
    jbtDelete.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        delete();
      }
    });
    jbtUpdate.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        update();
      }
    });
    listSelectionModel.addListSelectionListener(
      new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent e) {
          listSelectionModel_valueChanged(e);
        }
      });
    }

  private void delete() {
    try {
      // Delete the record from the database
      resultSet.deleteRow();
      refreshResultSet();

      // Remove the row in the table
      tableModel.removeRow(
        listSelectionModel.getLeadSelectionIndex());
    }
    catch (Exception ex) {
      jlblStatus.setText(ex.toString());
    }
  }

  private void insert() {
    // Display the dialog box
    newRecordDialog.displayTable(columnHeaderVector);
    Vector newRecord = newRecordDialog.getNewRecord();

    if (newRecord == null) return;

    // Insert the record to the Swing table
    tableModel.addRow(newRecord);

    // Insert the record to the database table
    try {
      for (int i = 1; i <= columnCount; i++) {
        resultSet.updateObject(i, newRecord.elementAt(i - 1));
      }

      resultSet.insertRow();
      refreshResultSet();
    }
    catch (Exception ex) {
      jlblStatus.setText(ex.toString());
    }
  }

  /** Set cursor in the table and set the row number in the status */
  private void setTableCursor() throws Exception {
    int row = resultSet.getRow();
    listSelectionModel.setSelectionInterval(row - 1, row - 1);
    jlblStatus.setText("Current row number: " + row);
  }

  private void update() {
    try {
      // Get the current row
      int row = jTable1.getSelectedRow();

      // Gather data from the UI and update the database fields
      for (int i = 1;
        i <= resultSet.getMetaData().getColumnCount(); i++) {
        resultSet.updateObject(i, tableModel.getValueAt(row, i - 1));
      }

      // Invoke the update method in the result set
      resultSet.updateRow();
      refreshResultSet();
    }
    catch (Exception ex) {
      jlblStatus.setText(ex.toString());
    }
  }

  /** Move cursor to the next record */
  private void moveCursor(String whereToMove) {
    try {
      if (whereToMove.equals("first"))
        resultSet.first();
      else if (whereToMove.equals("next"))
        resultSet.next();
      else if (whereToMove.equals("previous"))
        resultSet.previous();
      else if (whereToMove.equals("last"))
        resultSet.last();
      setTableCursor();
    }
    catch (Exception ex) {
      jlblStatus.setText(ex.toString());
    }
  }

  /** Refresh the result set */
  private void refreshResultSet() {
    try {
      resultSet = statement.executeQuery(
        "SELECT * FROM " + tableName);
      // Set the cursor to the first record in the table
      moveCursor("first");
    }
    catch (SQLException ex) {
      ex.printStackTrace();
    }
  }

  /** Set database connection and table name in the TableEditor */
  public void setConnectionAndTable(Connection newConnection,
  String newTableName) {
    connection = newConnection;
    tableName = newTableName;
    try {
      statement = connection.createStatement(ResultSet.
        TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
      showTable();
      moveCursor("first");
    }
    catch (SQLException ex) {
      ex.printStackTrace();
    }
  }

  /** Display database table to a Swing table */
  private void showTable() throws SQLException {
    // Clear vectors to store data for a new table
    rowVectors.clear();
    columnHeaderVector.clear();

    // Obtain table contents
    resultSet = statement.executeQuery(
      "select * from " + tableName + ";");

    // Get column count
    columnCount = resultSet.getMetaData().getColumnCount();
    // Store rows to rowVectors
    while (resultSet.next()) {
      Vector singleRow = new Vector();
      for (int i = 0; i < columnCount; i++)
        // Store cells to a row
        singleRow.addElement(resultSet.getObject(i + 1));
      rowVectors.addElement(singleRow);
    }

    // Get column name and add to columnHeaderVector
    ResultSet rsColumns = connection.getMetaData().getColumns(
      null, null, tableName, null);
    while (rsColumns.next()) {
      columnHeaderVector.addElement(
        rsColumns.getString("COLUMN_NAME"));
    }

    // Set new data to the table model
    tableModel.setDataVector(rowVectors, columnHeaderVector);
  }

  /** Handle the selection in the table */
  void listSelectionModel_valueChanged(ListSelectionEvent e) {
    int selectedRow = jTable1.getSelectedRow();

    try {
      resultSet.absolute(selectedRow + 1);
      setTableCursor();
    }
    catch (Exception ex) {
      jlblStatus.setText(ex.toString());
    }
  }
}
```

PD: Please add line numbers in the following code

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;

public class NewRecordDialog extends JDialog {
  private JButton jbtOK = new JButton("OK");
  private JButton jbtCancel = new JButton("Cancel");

  private DefaultTableModel tableModel = new DefaultTableModel();
  private JTable jTable1 = new JTable(tableModel);
  private Vector newRecord;

  /** Creates new form NewRecordDialog */
  public NewRecordDialog(Frame parent, boolean modal) {
    super(parent, modal);
    setTitle("Insert a New Record");
    setModal(true);

    JPanel jPanel1 = new JPanel();
    jPanel1.add(jbtOK);
    jPanel1.add(jbtCancel);

    jbtOK.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        setVisible(false);
      }
    });
    jbtCancel.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        newRecord = null;
        setVisible(false);
      }
    });

    getContentPane().add(jPanel1, BorderLayout.SOUTH);
    getContentPane().add(new JScrollPane(jTable1), BorderLayout.CENTER);
  }

  public NewRecordDialog() {
    this(null, true);
  }

  public Vector getNewRecord() {
    return newRecord;
  }

  /** Display the table */
  void displayTable(Vector columnHeaderVector) {
    this.setSize(new Dimension(400, 100));

    tableModel.setColumnIdentifiers(columnHeaderVector);

    // Must create a new vector for a new record
    tableModel.addRow(newRecord = new Vector());
    setVisible(true);
  }
}
```

Review

The key class in this example is TableEditor, which can be used to navigate and modify the table contents. To use it, simply create an instance of TableEditor (Line 19 in TestTableEditor.java) set the database connection and the table name in the instance, and place it in a graphical user interface. The setConnectionAndTableName method (Lines 56-57 in TestTableEditor.java) involves creating a statement, obtaining a result set, and displaying the result set in the Swing table. The statement is created with the arguments TYPE_SCROLL_SENSITIVE and CONCUR_UPDATABLE for obtaining scrollable and updateable result sets (Lines 217-218 in TableEditor.java).

The showTable() method (Lines 228-256 in TableEditor.java) is responsible for transferring data from the database table to the Swing table. The column names and column count are obtained using the ResultSetMetaData interface. An instance of the ResultSetMetaData interface is obtained using the getMetaData method for the result set. Each record from the result set is copied to a row vector. The row vector is added to another vector that stores all the rows for the table model (tableModel) for the JTable.

The handling of the navigation buttons First, Next, Prior, and Last is simply to invoke the methods first(), next(), previous(), and last() to move the cursor in the result set and, at the same time, set the selected row in the Swing table.

The handling of the Insert button involves displaying the “Insert a New Record” dialog box (newRecordDialog1) for receiving the new record. Once the record is entered, clicking the OK button dismisses the dialog box. The new record is obtained by invoking the newRecordDialog1.getNewRecord() method. To insert the new record into the database, use the updateXxx method (Line 142 in TableEditor.java) to update the fields and then use the insertRow method to insert the record to the database table. Finally, you need to refresh the result set by re-executing the query. Theoretically, you should not have to refresh the result set (Line 146 in TableEditor.java). The driver should automatically reflect the changes in the database to the result set. However, none of the drivers I have tested supports this. So, it is safe to refresh it.

To implement the Delete button, invoke the deleteRow() method (Line 117 in TableEditor.java) in the result set to remove the record from the database, and use the removeRow method in TableModel to remove a row from the JTable.

To implement the Update button, invoke the updateXxx method (Line 168 in TableEditor.java) in the result set, and then invoke the updateRow method (Line 172 in TableEditor.java) to update the result set.

To implement the handler for list selection events on jTable1, set the cursor in the result set to match the row selected in jTable1.

NOTE: The TableEditor class in this example uses only the updateObject(columnIndex, object) method. This updates a string column. To update a column of double type, you have use updateDouble(columnIndex, doubleValue). See Exercise 26.7 to revise the program to handle all types of columns.

TIP: To ensure the effect of editing a field in the table, you need to press the Enter key or move the cursor to other fields.

NOTE: Many JDBC drivers including the MySQL and Oracle driver support the read-only scrollable result set but not the updateable scrollable result set. Thus you cannot modify the result set. You can use supportsResultSetType(int type) and supportsResultSetConcurrency(int type, int concurrency) in the DatabaseMetaData interface to find out which result type and currency modes are supported by the JDBC driver. But even if a driver supports the scrollable and updateable result set, a result set for a complex query might not be able to perform an update. For example, the result set for a query that involves several tables is likely not to support update operations.

26.10 Storing and Retrieving Images in JDBC (Optional)

Database can store not only numbers and strings, but also images. SQL3 introduced a new data type BLOB (Binary Large OBject) for storing binary data, which can be used to store images. Another new SQL3 type is CLOB (Character Large OBject) for storing a large text in the character format. JDBC 2 introduced the interfaces java.sql.Blob and java.sql.Clob to support mapping for these new SQL types. JBDC 2 also added new methods, such as getBlob, setBinaryStream, getClob, setBlob, and setClob, in the interfaces ResultSet and PreparedStatement to access SQL BLOB, and CLOB values.

To store an image into a cell in a table, the corresponding column for the cell must be of the BLOB type. For example, the following SQL statement creates a table whose type for the flag column is BLOB.

```
create table Country(name varchar(30), flag blob,
  description varchar(255));
```

In the preceding statement, the description column is limited to 255 characters, which is the upper limit for MySQL. For Orcale, the upper limit is 32,672 bytes for Oracle. For a large character field, you can use the CLOB type for Oracle, which can store up to two GB characters.

MySQL does not support CLOB. However, you can use BLOB to store a long string and convert binary data into characters.

NOTE: Access does not support the BLOB and CLOB types.

To insert a record with images to a table, define a prepared statement like this one:

```
PreparedStatement pstmt = connection.prepareStatement(
  "insert into Country values(?, ?, ?)");
```

Images are usually stored in files. You may first get an instance of InputStream for an image file and then use the setBinaryStream method to associate the input stream with a cell in the table, as follows:

```
// Store image to the table cell
File file = new File(imageFilenames[i]);
InputStream inputImage = new FileInputStream(file);
pstmt.setBinaryStream(2, inputImage, (int)(file.length()));
```

To retrieve an image from a table, use the getBlob method, as shown below:

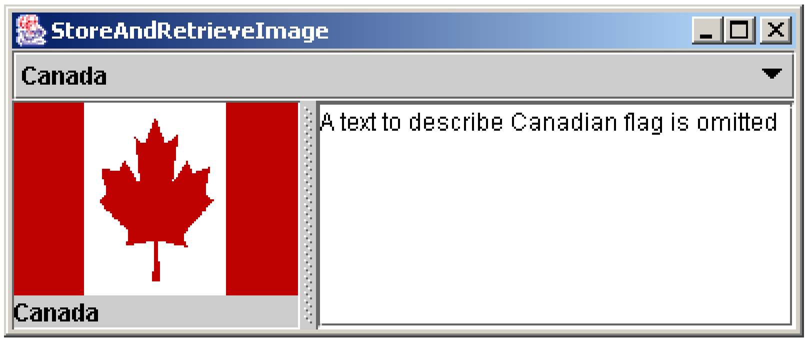
```
// Store image to the table cell
Blob blob = rs.getBlob(1);
ImageIcon imageIcon = new ImageIcon(
  blob.getBytes(1, (int)blob.length()));
```

Example 26.6

Storing and Retrieving Images in JDBC

Problem

In this example, you will create a table, populate it with data, including images, and retrieve and display images. The table is named Country. Each record in the table consists of three fields: name, flag, and description. Flag is an image field. The program first creates the table and stores data to it. Then the program retrieves the country names from the table and adds them to a combo box. When the user selects a name from the combo box, the country's flag and description are displayed, as shown in Figure 26.26.



A text to describe Canadian flag is omitted

Figure 26.26

The program enables you to retrieve data, including images, from a table and displays them.

Solution

First create the Country table using the following SQL statement:

```
create table Country(name varchar(30), flag blob,
  description varchar(255));
```

Create two classes: DescriptionPanel and StoreAndRetrieveImage. DescriptionPanel is a component for displaying a country (name, flag, and description).

Second create class StoreAndRetrieveImage to initialize the table and create the user interface using a DescriptionPanel and a combo box. DescriptionPanel is a component for displaying a country (name, flag, and description). This component was presented in Example 13.5, "Using Text Areas."

******PD: Please add line numbers in the following code******

```
import java.sql.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class StoreAndRetrieveImage extends JApplet {
  // Connection to the database
  private Connection connection;

  // Statement for static SQL statements
  private Statement stmt;

  // Prepared statement
  private PreparedStatement pstmt = null;
  private DescriptionPanel descriptionPanel1 = new DescriptionPanel();

  private JComboBox jcboCountry = new JComboBox();

  /** Creates new form StoreAndRetrieveImage */
  public StoreAndRetrieveImage() {
    try {
      connectDB(); // Connect to DB
      storeDataToTable(); //Store data to the table (including image)
      fillDataInComboBox(); // Fill in combo box
      retrieveFlagInfo((String)(jcboCountry.getSelectedItem()));
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }

    jcboCountry.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent evt) {
        retrieveFlagInfo((String)(evt.getItem()));
      }
    });

    getContentPane().add(jcboCountry, BorderLayout.NORTH);
    getContentPane().add(descriptionPanel1, BorderLayout.CENTER);
  }

  private void connectDB() throws Exception {
    // Load the driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");

    // Establish connection
    connection = DriverManager.getConnection
      ("jdbc:mysql://localhost/test");
    System.out.println("Database connected");

    // Create a statement for static SQL
    stmt = connection.createStatement();

    // Create a prepared statement to retrieve flag and description
    pstmt = connection.prepareStatement("select flag, description " +
      "from Country where name = ?");
  }

  private void storeDataToTable() {
    String[] countries = {"Canada", "UK", "USA", "Germany",
      "Indian", "China"};

    String[] imageFilenames = {"image/ca.gif", "image/uk.gif",
      "image/us.gif", "image/germany.gif", "image/india.gif",
      "image/china.gif"};

    String[] descriptions = {"A text to describe Canadian " +
      "flag is omitted", "British flag ...", "American flag ...",
      "German flag ...", "Indian flag ...", "Chinese flag ..."};

    try {
      // Create a prepared statement to insert records
      PreparedStatement pstmt = connection.prepareStatement(
        "insert into Country values(?, ?, ?)");

      // Store all predefined records
      for (int i = 0; i < countries.length; i++) {
        pstmt.setString(1, countries[i]);

        // Store image to the table cell
        java.net.URL url =
          this.getClass().getResource(imageFilenames[i]);
        InputStream inputImage = url.openStream();
        pstmt.setBinaryStream(2, inputImage,
          (int)(inputImage.available()));

        pstmt.setString(3, descriptions[i]);
        pstmt.executeUpdate();
      }

      System.out.println("Table Country populated");
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  private void fillDataInComboBox() throws Exception {
    ResultSet rs = stmt.executeQuery("select name from Country");
    while (rs.next()) {
      jcboCountry.addItem(rs.getString(1));
    }
  }

  private void retrieveFlagInfo(String name) {
    try {
      pstmt.setString(1, name);
      ResultSet rs = pstmt.executeQuery();
      if (rs.next()) {
        Blob blob = rs.getBlob(1);
        ImageIcon imageIcon = new ImageIcon(
        blob.getBytes(1, (int)blob.length()));
        descriptionPanel1.setImageIcon(imageIcon);
        descriptionPanel1.setName(name);
        String description = rs.getString(2);
        descriptionPanel1.setDescription(description);
      }
    }
    catch (Exception ex) {
      System.err.println(ex);
    }
  }
}
```

Review

Not all databases and their drivers support the SQL BLOB type. This example works on MySQL and Oracle, but not on Access.

The storeDataToTable method (Lines 60-97) first drops the Country table if it exists, creates the Country table, and populates the table with data. The fillDataInComboBox method retrieves the country names and adds them to the combo box. The `retrieveFlagInfo(name)` method retrieves the flag and description for the specified country name.

Key Classes and Methods

- **java.sql.DriverManager** A basic for managing JDBC drivers and connecting data sources. Use getConnection(databaseURL) or getConnection(databaseURL, username, password) to create a Connection.
- **java.sql.Connection** An interface that represents a connection to a database. Use createStatement() to create a static statement and prepareStatement(String sql) to create a prepared statement. You may obtain a scrollable or updateable using createStatement(int resultSetType, int resultSetConcurrency) and prepareStatement(String sql, int resultSetType, int resultSetConcurrency).
- **java.sql.Statement** An interface that represents a statement. Use executeUpdate() to execute a data declaration statement and use executeQuery(query) to execute a query. Use addBatch(sqlStatement) to add SQL statements for batch execution.
- **java.sql.ResultSet** An interface that represents a query result. Use next() to move cursor to the next row. Use getString(i) to get the ith element as a string from the current row in the result set. For a scrollable result set, you can use the methods first(), last(), next(), previous(), absolute(), or relative() to move the cursor. For an updateable result set, you can use the methods insertRow(), deleteRow(), and updateRow() to insert, delete, and update rows.
- **java.sql.DatabaseMetadata** An interface for retrieving database metadata. An instance of DatabaseMetadata can be obtained from a Connection instance using its getMetaData() method.
- **java.sql.ResultSetMetadata** An interface for retrieving metadata of a result set. An instance of ResultSetMetadata can be obtained from a ResultSet instance using its getMetaData() method. You can use the getColumnCount() method to find the number of columns in the result and the getColumnName(int) method to get the column names.

Key Terms

- **database system** consists of a database, the software that stores and manages data in the database, and the application programs that present data and enable the user to interact with the database system.
- **relational database** based on the relational data model. A relational database stores data in tables (also known as relations). A relational data model has three key components: structure, integrity, and languages. *Structure* defines the representation of the data. *Integrity* imposes constraints on the data. *Language* provides the means for accessing and manipulating data.
- **integrity constraint** imposes a condition that all legal values of the tables must satisfy. In general, there are three types of constraints: *domain constraints*, *primary key constraints*, and *foreign key constraints*. DBMS enforces integrity constraints and rejects any operation that would violate them.

- **domain constraint** specifies the permissible values for an attribute. Domains can be specified using standard data types, such as integers, floating-point numbers, fixed-length strings, and variant-length strings. The standard data type specifies a broad range of values.
- **primary key constraint** specifies that the values of the primary key are unique in a relation.
- **foreign key constraint** defines the relationships among relations. A foreign key is an attribute or a set of attributes in one relation that refers to the primary key in another relation.
- **Structured Query Language (SQL)** the language for defining tables and integrity constraints and for accessing and manipulating data.

Chapter Summary

- This chapter introduced the concept of database systems, relational database, relational data model, data integrity, and SQL. You learned how to develop database applications using Java.
- The Java API for developing Java database applications is called *JDBC*. JDBC provides Java programmers with a uniform interface for accessing and manipulating a wide range of relational databases.
- The JDBC API consists of classes and interfaces for establishing connections with databases, sending SQL statements to databases, processing the results of SQL statements, and obtaining database metadata.
- Since a JDBC driver serves as the interface to facilitate communications between JDBC and a proprietary database, JDBC drivers are database-specific. A JDBC-ODBC bridge driver is included in JDK to support Java programs that access databases through ODBC drivers. If you use a driver other than the JDBC-ODBC bridge driver, make sure it is on the classpath before running the program.
- Four key interfaces are needed to develop any database application using Java: Driver, Connection, Statement, and ResultSet. These interfaces define a framework for generic SQL database access. The JDBC driver vendors provide implementation for them.
- A JDBC application loads an appropriate driver using the Driver interface, connects to the database using the Connection interface, creates and executes SQL statements using the Statement interface, and processes the result using the ResultSet interface if the statements return results.
- The PreparedStatement interface is designed to execute dynamic SQL statements and SQL-stored procedures with IN parameters. These SQL statements and stored procedures are precompiled for efficient use when repeatedly executed.
- Database *metadata* is information that describes the database itself. JDBC provides the DatabaseMetaData interface for obtaining database-wide information and the ResultSetMetaData interface for obtaining information on the specific ResultSet.

Review Questions

Section 26.2 Relational Database Systems

26.1 What are superkeys, candidate keys, and primary keys? How do you create a table with a primary key?
26.2 What is a foreign key? How do you create a table with a foreign key?
26.3 Can a relation have more than one primary key or foreign key?
26.4 Does a foreign key need to be a primary key in the same relation?
26.5 Does a foreign key need to have the same name as its referenced primary key?
26.6 Can a foreign key value be null?

Section 26.3 SQL
26.7 Create tables Course, Student, and Enrollment using the create table statements in Section 26.3.1, "Creating and Dropping Tables." Insert rows into Course, Student, and Enrollment using the data in Figures 26.3, 26.4, and 26.5.

26.8 List all CSCI courses with at least four credit hours.

26.9 List all students whose last name contains the letter e two times.

26.10 List all students whose birthday is null.

26.11 List all student whose take Math courses.

26.12 List the number of the course in each subject.

26.13 Assume each credit hour is fifty minutes of lectures, get the total minutes for the courses that each student take.

Section 26.4 JDBC

26.14 What are the advantages of developing database applications using Java?

26.15 Describe the following JDBC interfaces: Driver, Connection, Statement, and ResultSet.

26.16 How do you load a JDBC driver? What are driver class names for MySQL, Access, and Oracle?

26.17 How do you create a database connection? What are the URLs for MySQL, Access, and Oracle?

26.18 How do you create a Statement and execute an SQL statement?

26.19 How do you retrieve values in a ResultSet?

26.20 Does JDBC automatically commit a transaction? How do you set auto-commit to false?

Section 26.5 PreparedStatement
26.21 Describe prepared statements. How do you create instances of PreparedStatement? How do you execute a PreparedStatement? How do you set parameter values in a PreparedStatement?

26.22 What are benefits of using prepared statements?

Section 26.6 Retrieving Metadata

26.23 What is DatabaseMetaData for? Describe the methods in DatabaseMetaData. How do you create an instance of DatabaseMetaData?

26.24 What is ResultSetMetaData for? Describe the methods in ResultSetMetaData. How do you create an instance of ResultSetMetaData?

26.25 How do you find the number of columns in a result set? How do you find the column names in a result set?

Section 26.8 Batch Processing

26.26 What is batch processing in JDBC? What are the benefits of using batch processing?

26.27 How do you add an SQL statement to a batch? How do you execute a batch?

26.28 Can you execute a SELECT statement in a batch?

26.29 How do you know if a JDBC driver supports batch updates?

Section 26.9 Scrollable and Updateable Result Set

26.30 What is a scrollable result set? What is an updateable result set?

26.31 How do you create a scrollable and updateable ResultSet?

26.32 How do you know if a JDBC driver supports a scrollable and updateable ResultSet?

Section 26.10 Storing and Retrieving Images in JDBC

26.33 How do you store images into a database?

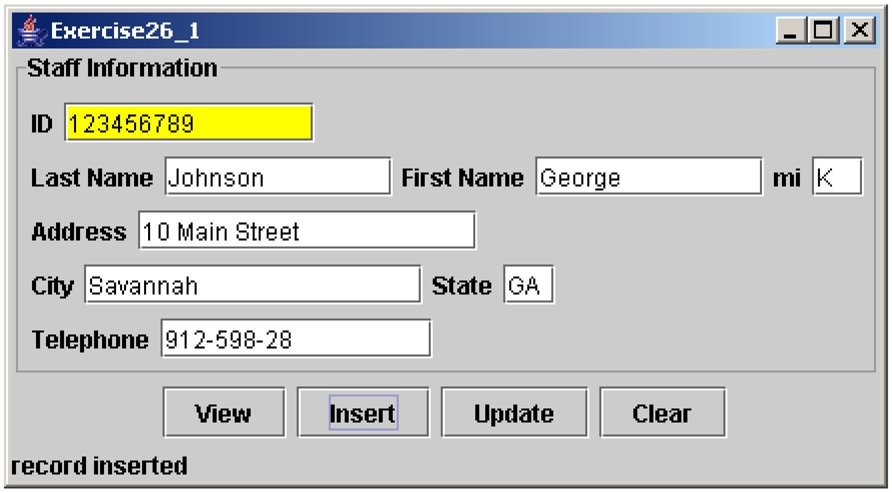
26.34 How do you retrieve images from a database?

26.35 Does Oracle support SQL3 BLOB type and CLOB type? How about MySQL and Access?

Exercises

26.1 Write a Java applet that views, inserts, and updates staff information stored in a database, as shown in Figure 26.27. The Staff table is created as follows:

```
create table Staff (
  id char(9) not null,
  lastName varchar(15),
  firstName varchar(15),
  mi char(1),
  address varchar(20),
  city varchar(20),
  state char(2),
  telephone char(10),
  email varchar(40),
  primary key (id)
);
```



ID: 123456789
Last Name: Johnson
First Name: George
mi: K
Address: 10 Main Street
City: Savannah
State: GA
Telephone: 912-598-28

record inserted

Figure 26.27

The applet lets you view, insert, and update staff information.

26.2 Write a program to display the number of students in each department in a pie chart and a bar chart, as shown in Figure 26.28. The number of students for each department can be obtained from the Student table (See Figure 26.4) using the following SQL statement:

```
select deptId, count(*)
from Student
group by deptId;
```

Use the PieChart component and the BarChart component created in Exercise 22.6 to display the data.

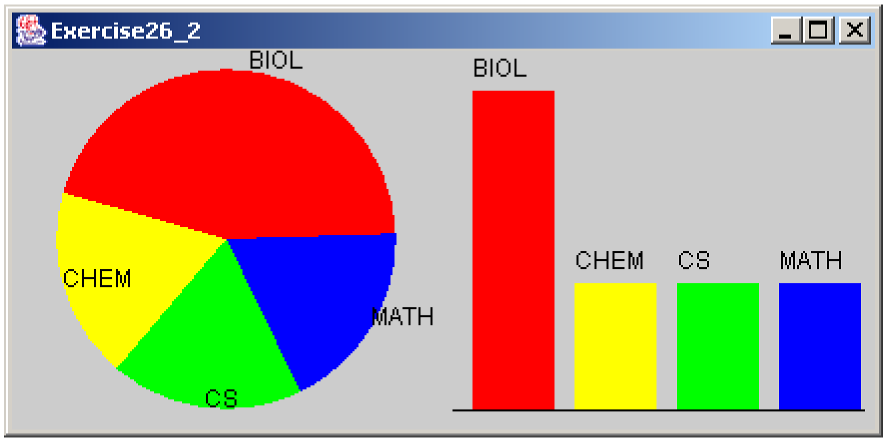
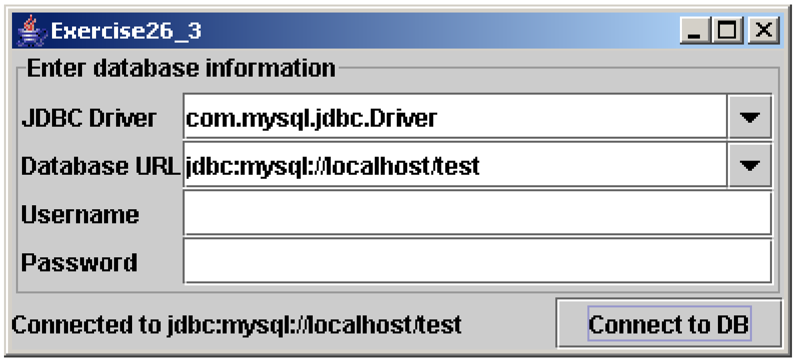


Figure 26.28

The PieChart and BarChart components display the query data obtained from the data module.

26.3 Develop a JavaBeans component named DBConnectionPanel that enables the user to select or enter a JDBC driver and a URL and to enter a username and password, as shown in Figure 26.29. Upon clicking the OK button, a Connection object for the database is stored in the connection property. You can then use the getConnection() method to return the connection.



JDBC Driver: com.mysql.jdbc.Driver
Database URL: jdbc:mysql://localhost/test
Connected to jdbc:mysql://localhost/test

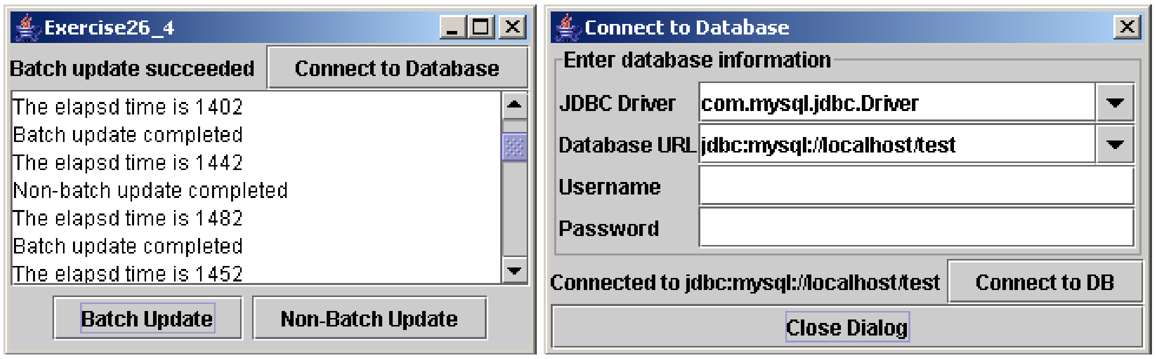
Figure 26.29

The DBConnectionPanel component enables the user to enter database information.

26.4 Write a program to insert 1000 records to a database and compare the performance with and without batch updates, as shown in Figure 26.30. Suppose the table is defined as follows:

```
create table Temp(num1 double, num2 double, num3 double)
```

Use the Math.random() method to generate random numbers for each record. Create a dialog box that contains DBConnectionPanel discussed in the preceding exercise. Use this dialog box to connect to the database.



Batch update succeeded

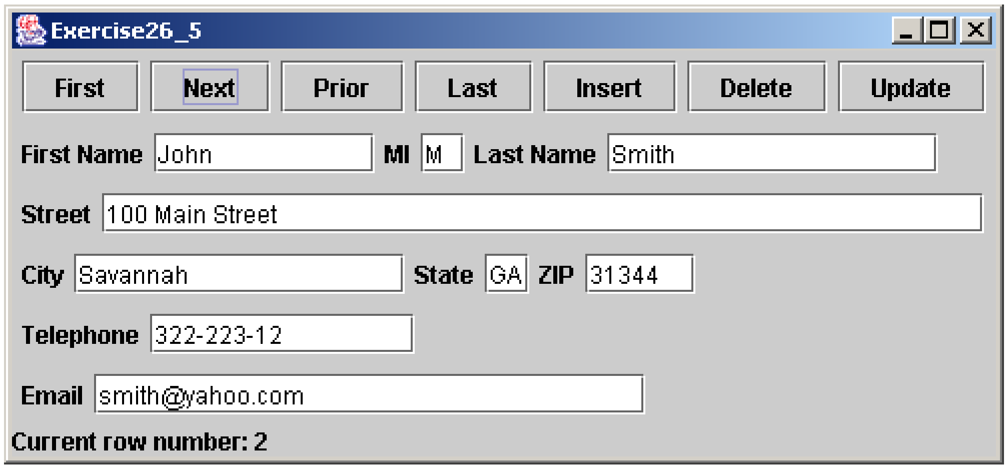
```
The elapsd time is 1402
Batch update completed
The elapsd time is 1442
Non-batch update completed
The elapsd time is 1482
Batch update completed
The elapsd time is 1452
```

JDBC Driver: com.mysql.jdbc.Driver
Database URL: jdbc:mysql://localhost/test
Connected to jdbc:mysql://localhost/test

Figure 26.30

The program demonstrates the performance improvements that result from using the batch updates.

26.5 Write a program to use the buttons First, Next, Prior, Last, Insert, Delete, and Update display and modify a single record in the Address table, as shown in Figure 26.31.



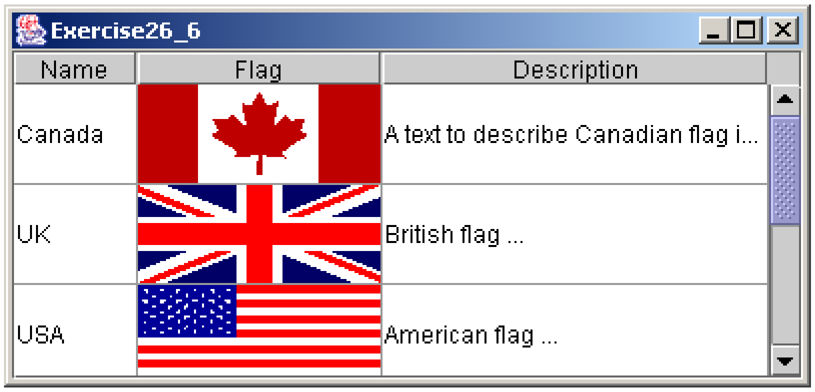
First Name	John	MI	M	Last Name	Smith
Street	100 Main Street				
City	Savannah	State	GA	ZIP	31344
Telephone	322-223-12				
Email	smith@yahoo.com				

Current row number: 2

Figure 26.31

You can use the buttons to display and modify a single record in the Address table.

26.6 Write a program that uses <u>JTable</u> to display the Country table created in Example 26.6, as shown in Figure 26.32.



Name	Flag	Description
Canada		A text to describe Canadian flag i...
UK		British flag ...
USA		American flag ...

Figure 26.32

The Country table is displayed in a <u>JTable</u> instance.

26.7 Revise Example 26.5, "Scrolling and Updating Tables," to enable it to insert all types of columns (not just strings).