Objectives

- To understand the concepts of databases and database management systems (§34.2).
- To understand the relational data model: relational data structures, constraints, and languages (§34.2).
- To use SQL to create and drop tables and to retrieve and modify data (§34.3).
- To learn how to load a driver, connect to a database, execute statements, and process result sets using JDBC (§34.4).
- To use prepared statements to execute precompiled SQL statements (§34.5).
- To use callable statements to execute stored SQL procedures and functions (§34.6).
- To explore database metadata using the DatabaseMetaData and ResultSetMetaData interfaces (§34.7).
34.1 Introduction

Java provides the API for developing database applications that works with any relational database systems.

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 for courses online. Database systems play an important role in society and in commerce.

This chapter introduces database systems, the SQL language, and how to develop database applications using Java. If you already know SQL, you can skip Sections 34.2 and 34.3.

34.2 Relational Database Systems

SQL is the standard database language for defining and accessing databases.

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 34.1.

A database is a repository of data that form information. When you purchase a database system—such as MySQL, Oracle, IBM’s DB2 and Informix, Microsoft SQL Server, or Sybase—from a software vendor, you actually purchase the software comprising a database management system (DBMS). 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. Application programs may be standalone GUI applications or Web applications, and may access several different database systems in the network, as shown in Figure 34.2.

Most of today’s database systems are relational database systems. They are based on the relational data model, which has three key components: structure, integrity, and language.
Relational Database Systems

**Structure** defines the representation of the data. **Integrity** imposes constraints on the data. **Language** provides the means for accessing and manipulating data.

### 34.2.1 Relational Structures

The relational model is built around a simple and natural structure. A *relation* is actually a table that consists of nonduplicate 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 relational database theory, a row is called a *tuple* and a column is called an *attribute*. Figure 34.3 shows a sample table that stores information about the courses offered by a university. The table has eight tuples, and each tuple has five attributes.

![Figure 34.3](image)

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 34.3) 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, *Student* and *Enrollment*, as shown in
Figures 34.4 and 34.5. 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.

<table>
<thead>
<tr>
<th>Student Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn</td>
</tr>
<tr>
<td>444111110</td>
</tr>
<tr>
<td>444111111</td>
</tr>
<tr>
<td>444111112</td>
</tr>
<tr>
<td>444111113</td>
</tr>
<tr>
<td>444111114</td>
</tr>
<tr>
<td>444111115</td>
</tr>
<tr>
<td>444111116</td>
</tr>
<tr>
<td>444111117</td>
</tr>
<tr>
<td>444111118</td>
</tr>
<tr>
<td>444111119</td>
</tr>
<tr>
<td>444111120</td>
</tr>
</tbody>
</table>

**Figure 34.4** A Student table stores student information.

<table>
<thead>
<tr>
<th>Enrollment Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn</td>
</tr>
<tr>
<td>444111110</td>
</tr>
<tr>
<td>444111110</td>
</tr>
<tr>
<td>444111110</td>
</tr>
<tr>
<td>444111111</td>
</tr>
<tr>
<td>444111111</td>
</tr>
<tr>
<td>444111111</td>
</tr>
<tr>
<td>444111112</td>
</tr>
<tr>
<td>444111112</td>
</tr>
<tr>
<td>444111112</td>
</tr>
<tr>
<td>444111113</td>
</tr>
<tr>
<td>444111113</td>
</tr>
<tr>
<td>444111114</td>
</tr>
<tr>
<td>444111115</td>
</tr>
<tr>
<td>444111115</td>
</tr>
<tr>
<td>444111116</td>
</tr>
<tr>
<td>444111116</td>
</tr>
<tr>
<td>444111118</td>
</tr>
<tr>
<td>444111118</td>
</tr>
</tbody>
</table>

**Figure 34.5** An Enrollment table stores student enrollment information.

### 34.2.2 Integrity Constraints

An integrity constraint imposes a condition that all the legal values in a table must satisfy. Figure 34.6 shows an example of some integrity constraints in the Subject and Course tables.

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 intrarelational constraints, meaning that a constraint involves only one relation. The foreign key constraint is interrelational, meaning that a constraint involves more than one relation.
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 (in the `Course` table) must be greater than 0 and less than 5. You can also specify whether an attribute can be `null`, which is a special value in a database meaning unknown or not applicable. As shown in the `Student` table, `birthDate` may be `null`.

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 a 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. As shown in Figure 34.6, `courseId` is the primary key in the `Course` table.

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 nonnull value on $FK$ in $R$ must match a primary key value in $T$.
As shown in Figure 34.6, 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.

Enforcing Integrity Constraints

The database management system enforces integrity constraints and rejects operations that would violate them. For example, if you attempted to insert the new record (‘11115’, ‘CSCI’, ‘2490’, ‘C++ Programming’, 0) into the Course table, it would fail because the credit hours must be greater than 0; if you attempted 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 attempted to delete a record from 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 primary key constraints and foreign key constraints, but not all database systems support domain constraints. In the Microsoft Access database, for example, you cannot specify the constraint that numOfCredits is greater than 0 and less than 5.

34.1 What are superkeys, candidate keys, and primary keys?
34.2 What is a foreign key?
34.3 Can a relation have more than one primary key or foreign key?
34.4 Does a foreign key need to be a primary key in the same relation?
34.5 Does a foreign key need to have the same name as its referenced primary key?
34.6 Can a foreign key value be null?

34.3 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 many 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.

SQL can be used on MySQL, Oracle, Sybase, IBM DB2, IBM Informix, Borland InterBase, MS Access, or any other relational database system. This chapter uses MySQL to demonstrate SQL and uses MySQL, Access, and Oracle to demonstrate Java database programming. The Companion Web site contains the following supplements on how to install and use three popular databases: MySQL, Oracle, and Access:

MySQL Tutorial
- Supplement IV.B: Tutorial for MySQL
Oracle Tutorial
- Supplement IV.C: Tutorial for Oracle
Access Tutorial
- Supplement IV.D: Tutorial for Microsoft Access
34.3.1 Creating a User Account on MySQL

Assume that you have installed MySQL 5 with the default configuration. To match all the examples in this book, you should create a user named `scott` with the password `tiger`. You can perform the administrative tasks using the MySQL Workbench or using the command line. MySQL Workbench is a GUI tool for managing MySQL databases. Here are the steps to create a user from the command line:

1. From the DOS command prompt, type
   ```
   mysql -uroot -p
   ```
   You will be prompted to enter the root password, as shown in Figure 34.7.

2. At the mysql prompt, enter
   ```
   use mysql;
   ```

3. To create user `scott` with password `tiger`, enter
   ```
   create user 'scott'@'localhost' identified by 'tiger';
   ```

4. To grant privileges to `scott`, enter
   ```
   grant select, insert, update, delete, create, create view, drop,
       execute, references on *.* to 'scott'@'localhost';
   ```
   - If you want to enable remote access of the account from any IP address, enter
     ```
     grant all privileges on *.* to 'scott'@'%' identified by 'tiger';
     ```
   - If you want to restrict the account’s remote access to just one particular IP address, enter
     ```
     grant all privileges on *.* to 'scott'@'ipAddress' identified by 'tiger';
     ```

5. Enter
   ```
   exit;
   ```
   to exit the MySQL console.

![Image](image.png)  
**Figure 34.7** You can access a MySQL database server from the command window.
By default, the server contains two databases named `mysql` and `test`. The `mysql` database contains the tables that store 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 delete an existing database using the command `DROP DATABASE databasename`.

### 34.3.2 Creating a Database

To match the examples in the book, you should create a database named `javabook`. Here are the steps to create it:

1. From the DOS command prompt, type

   ```
   mysql -u scott -p tiger
   ```

   to login to `mysql`, as shown in Figure 34.8.

2. At the `mysql` prompt, enter

   ```
   CREATE DATABASE javabook;
   ```

   and then type

   ```
   USE javabook;
   ```

   as shown in Figure 34.9.

   For your convenience, the SQL statements for creating and initializing tables used in the book are provided in Supplement IV.A. You can download the script for MySQL and save it to `script.sql`. To execute the script, first switch to the `javabook` database using the following command:

   ```
   USE javabook;
   ```

   and then type

   ```
   SOURCE script.sql;
   ```
34.3.3 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:

```sql
create table Course (  
courseId char(5),  
subjectId char(4) not null,  
courseNumber integer,  
title varchar(50) not null,  
umOfCredits 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 50 characters. `integer` specifies that `courseNumber` is an integer. The primary key is `courseId`.

The tables `Student` and `Enrollment` can be created as follows:

```sql
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),  
depId 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(ssn),  
foreign key (courseId) references Course(courseId)
);
```

**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`.

Figure 34.10 shows how to enter the `create table` statement from the MySQL console.

![Figure 34.10](image)

**Figure 34.10** A table is created using the `create table` statement.

If you make typing errors, you have to retype the whole command. To avoid retyping, you can save the command in a file, and then run the command from the file. To do so, create a text file to contain commands, named, for example, `test.sql`. You can create the text file using any text editor, such as Notepad, as shown in Figure 34.11a. To comment a line, precede it with two dashes. You can now run the script file by typing `source test.sql` from the SQL command prompt, as shown in Figure 34.11b.

![Figure 34.11](image)

**Figure 34.11** (a) You can use Notepad to create a text file for SQL commands. (b) You can run the SQL commands in a script file from MySQL.

### 34.3.4 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 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.

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

The column names are optional. If they are omitted, all the 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 in SQL.

The syntax to update a table is:

```sql
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.

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

The syntax to delete records from a table is:

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

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

```sql
delete Course
where title = 'Database Systems';
```

The following statement deletes all the records from the **Course** table:

```sql
delete Course;
```

### 34.3.5 Simple Queries

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

```sql
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 optional **where** clause specifies the conditions for the selected rows.

**Query 1**: Select all the students in the CS department, as shown in Figure 34.12.

```sql
select firstName, mi, lastName
from Student
where deptId = 'CS';
```
34.3.6 Comparison and Boolean Operators

SQL has six comparison operators, as shown in Table 34.1, and three Boolean operators, as shown in Table 34.2.

<table>
<thead>
<tr>
<th>Table 34.1</th>
<th>Comparison Operators</th>
<th>Table 34.2</th>
<th>Boolean Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Operator</td>
<td>Description</td>
</tr>
<tr>
<td>=</td>
<td>Equal to</td>
<td>not</td>
<td>Logical negation</td>
</tr>
<tr>
<td>&lt;&gt; or !=</td>
<td>Not equal to</td>
<td>and</td>
<td>Logical conjunction</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>or</td>
<td>Logical disjunction</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note**

The comparison and Boolean operators in SQL have the same meanings as in Java. In SQL the equal to operator is =, but in Java it is ==. In SQL the not equal to operator is <> or !=, but in Java it is !=. 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 ZIP code 31411.

```sql
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 use an asterisk (*), which stands for all the attributes. For example, the following query displays all the attributes of the students who are in the CS dept and live in ZIP code 31411:

```sql
select *
from Student
where deptId = 'CS' and zipCode = '31411';
```
34.3.7 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

$$ s \text{ like } p \text{ or } s \text{ not like } p $$

You can use the wildcard 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 '_ni%'` matches any string whose second and third letters are $m$ and $i$. `lastName not like '_ni%'` excludes any string whose second and third letters are $m$ and $i$.

**Note**

In earlier versions of MS Access, the wildcard character is *, and the character ? matches any single character.

The **between-and** operator checks whether a value $v$ is between two other values, $v_1$ and $v_2$, using the following syntax:

$$ v \text{ between } v_1 \text{ and } v_2 \text{ or } v \text{ not between } v_1 \text{ and } v_2 $$

$v \text{ between } v_1 \text{ and } v_2$ is equivalent to $v \geq v_1 \text{ and } v \leq v_2$, and $v \text{ not between } v_1 \text{ and } v_2$ is equivalent to $v < v_1 \text{ or } v > v_2$.

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

$$ v \text{ is null } \text{ or } v \text{ is not null } $$

**Query 3**: Get the Social Security numbers of the students whose grades are between ‘C’ and ‘A’.

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

34.3.8 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. Sometimes it is desirable to give more descriptive names in the result heading. You can use the column aliases with the following syntax:

```sql
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 34.13.

![Figure 34.13](image-url)

**Figure 34.13** You can use a column alias in the display.
select lastName as "Last Name", zipCode as "Zip Code"
from Student
where deptId = 'CS';

Note
The as keyword is optional in MySQL and Oracle, but it is required in MS Access.

34.3.9 The Arithmetic Operators
You can use the arithmetic operators * (multiplication), / (division), + (addition), and – (subtraction) in SQL.

Query 5: Assume that a credit hour is 50 minutes of lectures, and get the total minutes for each course with the subject CSCI. The query result is shown in Figure 34.14.

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

34.3.10 Displaying Distinct Tuples
SQL provides the distinct keyword, which can be used to eliminate duplicate tuples in the result. Figure 34.15a displays all the subject IDs used by the courses and Figure 34.15b displays all the distinct subject IDs used by the courses using the following statement.

select distinct subjectId as "Subject ID"
from Course;

Figure 34.14 You can use arithmetic operators in SQL.

Figure 34.15 (a) The duplicate tuples are displayed. (b) The distinct tuples are displayed.
When there is more than one column in the `select` clause, the `distinct` keyword applies to the whole tuple in the result. For example, the following statement displays all tuples with distinct `subjectId` and `title`, as shown in Figure 34.16. Note that some tuples may have the same `subjectId`, but different `title`. These tuples are distinct.

```sql
select distinct subjectId, title
from Course;
```

![Figure 34.16](image)

The keyword `distinct` applies to the entire tuple.

### 34.3.11 Displaying Sorted Tuples

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

```sql
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 after `columns-to-be-sorted`, 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 first 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 descending order and secondarily on their first names in ascending order. The query result is shown in Figure 34.17.

![Figure 34.17](image)

You can sort results using the `order by` clause.
select lastName, firstName, deptId
from Student
where deptId = 'CS'
order by lastName desc, firstName asc;

34.3.12 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 34.18.

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 one item 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 34.19. **Student** and **Enrollment** have the same attribute **ssn**. To distinguish them in a query, use **Student.ssn** and **Enrollment.ssn**.

**Figure 34.18** **Student** and **Enrollment** are joined on **ssn**.

**Figure 34.19** Query 7 demonstrates queries involving multiple tables.
For more features of SQL, see Supplement IV.H and Supplement IV.I.

34.7 Create the tables **Course**, **Student**, and **Enrollment** using the **create table** statements in Section 34.3.3, Creating and Dropping Tables. Insert rows into the **Course**, **Student**, and **Enrollment** tables using the data in Figures 34.3, 34.4, and 34.5.

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

34.9 List all students whose last names contain the letter e two times.

34.10 List all students whose birthdays are null.

34.11 List all students who take Math courses.

34.12 List the number of courses in each subject.

34.13 Assume that each credit hour is 50 minutes of lectures. Get the total minutes for the courses that each student takes.

### 34.4 JDBC

**JDBC** is the Java API for accessing relational database.

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 34.20. 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 and are normally provided by the database vendors. You need MySQL JDBC drivers to access the MySQL database, and Oracle JDBC drivers to

![Figure 34.20](image-url)  
Java programs access and manipulate databases through JDBC drivers.
access the Oracle database. For the Access database, use the JDBC-ODBC bridge driver included in the JDK. ODBC is a technology developed by Microsoft for accessing databases on the Windows platform. An ODBC driver is preinstalled on Windows. The JDBC-ODBC bridge driver allows a Java program to access any ODBC data source.

### 34.4.1 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 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 API defines these interfaces, and the JDBC driver vendors provide the implementation for the interfaces. Programmers use these interfaces.

The relationship of these interfaces is shown in Figure 34.21. 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 34.21: JDBC classes enable Java programs to connect to the database, send SQL statements, and process results.](image)

1. **Loading drivers.**
   
   An appropriate driver must be loaded using the statement shown below before connecting to a database.

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

   A driver is a concrete class that implements the `java.sql.Driver` interface. The drivers for Access, MySQL, and Oracle are listed in Table 34.3. If your program accesses several different databases, all their respective drivers must be loaded.

you have to add `mysqljdbc.jar` and `ojdbc6.jar` in the classpath using the following DOS command on Windows:

```
set classpath=%classpath%;c:\book\mysqljdbc.jar;c:\book\ojdbc6.jar
```

If you use an IDE such as Eclipse or NetBeans, you need to add these jar files into the library in the IDE.

**Note**

`com.mysql.jdbc.Driver` is a class in `mysqljdbc.jar`, and `oracle.jdbc.driver.OracleDriver` is a class in `ojdbc6.jar`. `mysqljdbc.jar` and `ojdbc6.jar` contain many classes to support the driver. These classes are used by JDBC, but not directly by JDBC programmers. When you use a class explicitly in the program, it is automatically loaded by the JVM. The driver classes, however, are not used explicitly in the program, so you have to write the code to tell the JVM to load them.

**Note**

Java 6 supports automatic driver discovery, so you don’t have to load the driver explicitly. At the time of this writing, however, this feature is not supported for all database drivers. To be safe, load the driver explicitly.

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 34.4 lists the URL patterns for the Access, MySQL, and Oracle databases.

### Table 34.3  JDBC Drivers

<table>
<thead>
<tr>
<th>Database</th>
<th>Driver Class</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td><code>sun.jdbc.odbc.JdbcOdbcDriver</code></td>
<td>Already in JDK</td>
</tr>
<tr>
<td>MySQL</td>
<td><code>com.mysql.jdbc.Driver</code></td>
<td>Companion Web site</td>
</tr>
<tr>
<td>Oracle</td>
<td><code>oracle.jdbc.driver.OracleDriver</code></td>
<td>Companion Web site</td>
</tr>
</tbody>
</table>

### Table 34.4  JDBC URLs

<table>
<thead>
<tr>
<th>Database</th>
<th>URL Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td><code>jdbc:odbc:datasource</code></td>
</tr>
<tr>
<td>MySQL</td>
<td><code>jdbc:mysql://hostname/dbname</code></td>
</tr>
<tr>
<td>Oracle</td>
<td><code>jdbc:oracle:thin:@hostname:port#;oracleDBSID</code></td>
</tr>
</tbody>
</table>

For an ODBC data source, the `databaseURL` is `jdbc:odbc:datasource`. An ODBC data source can be created using the ODBC Data Source Administrator on Windows. See Supplement IV.D, 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:

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

The `databaseURL` for a MySQL database specifies the host name and database name to locate a database. For example, the following statement creates a `Connection` object for the local MySQL database `javabook` with username `scott` and password `tiger`:

```java
Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/javabook", "scott", "tiger");
```

Recall that by default MySQL contains two databases named `mysql` and `test`. Section 34.3.2, Creating a Database, created a custom database named `javabook`. We will use `javabook` in the examples.

The `databaseURL` for an Oracle database specifies the `hostname`, the `port#` where the database listens for incoming connection requests, and the `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 the username `scott` and password `tiger`:

```java
Connection connection = DriverManager.getConnection("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl", "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` 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:

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

4. Executing statements.

SQL data definition language (DDL) and update statements can be executed using `executeUpdate(String sql)` , and an 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))`:

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

This next code executes the SQL query `select firstName, mi, lastName from Student where lastName = 'Smith'`:

```java
// Select the columns from the Student table
ResultSet resultSet = statement.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 the various get methods to retrieve values from a current row. For example, the following code displays all the results from the preceding SQL query.
// Iterate through the result and print the student names
while (resultSet.next())
    System.out.println(resultSet.getString(1) + " " +
                        resultSet.getString(2) + " " + resultSet.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 row, third row, and so on, to the last row.

Listing 34.1 is a 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**.

**Listing 34.1 SimpleJDBC.java**

```java
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");

        // Connect to a database
        Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/javabook", "scott", "tiger");
        System.out.println("Database connected");

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

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

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

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

The statement in line 7 loads a JDBC driver for MySQL, and the statement in lines 11–13 connects to a local MySQL database. You can change them to connect to an Access or Oracle database. The program creates a `Statement` object (line 16), executes an SQL statement and returns a `ResultSet` object (lines 19–21), and retrieves the query result from the `ResultSet` object (lines 24–26). The last statement (line 29) closes the connection and releases resources related to the connection.

**Note**

If you run this program from the DOS prompt, specify the appropriate driver in the classpath, as shown in Figure 34.22.
The classpath directory and jar files are separated by commas. The period (.) represents the current directory. For convenience, the driver files are placed under the \texttt{lib} directory.

\textbf{Caution}\hspace{0.5cm} Do not use a semicolon (;) to end the Oracle SQL command in a Java program. The semicolon may not work with the Oracle JDBC drivers. It does work, however, with the other drivers used in the book.

\textbf{Note}\hspace{0.5cm} The \texttt{Connection} interface handles transactions and specifies how they are processed. By default, a new connection is in autocommit 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 the results have been retrieved. You can use the \texttt{setAutoCommit(false)} method to disable autocommit, so that all SQL statements are grouped into one transaction that is terminated by a call to either the \texttt{commit()} or the \texttt{rollback()} method. The \texttt{rollback()} method undoes all the changes made by the transaction.

\subsection*{34.4.2 Accessing a Database from a Java Applet}

If you are 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. This section gives an example that 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 34.23. The code in Listing 34.2 uses the MySQL database on the localhost.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{applet.png}
\caption{A Java applet can access the database on the server.}
\end{figure}
Listing 34.2 FindGrade.java

```java
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 Statement object
        initializeDB();

        jbtShowGrade.addActionListener(new ActionListener() {
            @Override
            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);

        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://localhost/javabook", "scott", "tiger");
            // ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl", "scott", "tiger");
            System.out.println("Database connected");

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

    private void jbtShowGrade_actionPerformed(ActionEvent e) {
```
Chapter 34 Java Database Programming

```java
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 firstName = rset.getString(1);
        String mi = rset.getString(2);
        String lastName = 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();
}
```

The `initializeDB()` method (lines 36–56) loads the MySQL driver (line 39), connects to the MySQL database on host `liang.armstrong.edu` (lines 44–45) and creates a statement (line 51).

You can run the applet as a standalone from the `main` method (note that the listing for the `main` method is omitted for all the applets in the book for brevity) or test the applet using the `appletviewer` utility, as shown in Figure 34.23. 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 a MySQL driver, you should specify the driver in the archive attribute in the `applet` tag, as follows:

```html
<applet
    code = "FindGrade"
    archive = "FindGrade.jar, lib/mysqljdbc.jar"
    width = 380
    height = 80
>
</applet>
```

**Note**
For information on how to create an archive file, see Supplement III.Q, Packaging and Deploying Java Projects. The `FindGrade.jar` file can be created using the following command:

```bash
c:\book>jar -cf FindGrade.jar FindGrade.class
    FindGrade$1.class
```
Note
To access the database from an applet, security restrictions make it necessary for the applet to be downloaded from the server where the database is located. Therefore, you have to deploy the applet on the server.

Note
There is a security hole in this program. If you enter 1' or true or '1 in the SSN field, you will get the first student's score, because the query string now becomes

```sql
SELECT firstName, mi, lastName, title, grade
FROM Student, Enrollment, Course
WHERE Student.ssn = '1' or true or '1' AND
    Enrollment.courseId = '' AND
    Enrollment.courseId = Course.courseId AND
    Enrollment.ssn = Student.ssn;
```

You can avoid this problem by using the `PreparedStatement` interface, which is discussed in the next section.

34.14 What are the advantages of developing database applications using Java?
34.15 Describe the following JDBC interfaces: `Driver`, `Connection`, `Statement`, and `ResultSet`.
34.16 How do you load a JDBC driver? What are the driver classes for MySQL, Access, and Oracle?
34.17 How do you create a database connection? What are the URLs for MySQL, Access, and Oracle?
34.18 How do you create a `Statement` and execute an SQL statement?
34.19 How do you retrieve values in a `ResultSet`?
34.20 Does JDBC automatically commit a transaction? How do you set autocommit to false?

34.5 `PreparedStatement`

`PreparedStatement` enables you to create parameterized SQL statements.

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 don’t contain any parameters. The `PreparedStatement` interface, extending `Statement`, is used to execute a precompiled SQL statement with or without parameters. Since the SQL statements are precompiled, they are efficient for repeated executions.

A `PreparedStatement` object is created using the `prepareStatement` method in the `Connection` interface. For example, the following code creates a `PreparedStatement` for an SQL `INSERT` statement:

```java
Statement preparedStatement = 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`. It also 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:

```java
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", and "Ryan" to the placeholders for `firstName`, `mi`, and `lastName` in `preparedStatement`:

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

After setting the parameters, you can execute the prepared statement by invoking `executeQuery()` for a SELECT statement and `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 don’t have any parameters, because the SQL statements are already specified in the `preparedStatement` method when the object of `PreparedStatement` is created.

Using a prepared SQL statement, Listing 34.2 can be improved as in Listing 34.3.

**Listing 34.3 FindGradeUsingPreparedStatement.java**

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

public class FindGradeUsingPreparedStatement extends JApplet {

    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 preparedStatement;

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

        jbtShowGrade.addActionListener(new ActionListener() {
            @Override
            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);
```
34.5 PreparedStatement

```java
add(jPanel1, BorderLayout.NORTH);
}

private void initializeDB() {
    try {
        // Load the JDBC drivers
        Class.forName("com.mysql.jdbc.Driver");
        System.out.println("Driver loaded");
        
        // Establish a connection
        Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/javabook", "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";
        
        PreparedStatement preparedStatement = connection.prepareStatement(queryString);
        
        try {
            preparedStatement.setString(1, ssn);
            preparedStatement.setString(2, courseId);
            ResultSet rset = preparedStatement.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();
        }
    }
}
```

```java
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 = ? and Enrollment.courseId = ? " + "and Enrollment.courseId = Course.courseId";
        
        PreparedStatement preparedStatement = connection.prepareStatement(queryString);
        
        try {
            preparedStatement.setString(1, ssn);
            preparedStatement.setString(2, courseId);
            ResultSet rset = preparedStatement.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();
        }
    }
}
```

```java
main method omitted
```
This example does exactly the same thing as Listing 34.2 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.

A prepared query string is defined in lines 50–53 with `ssn` and `courseId` as parameters. An SQL prepared statement is obtained in line 56. Before executing the query, the actual values of `ssn` and `courseId` are set to the parameters in lines 67–68. Line 69 executes the prepared statement.


34.22 What are the benefits of using prepared statements?

### 34.6 CallableStatement

CallableStatement enables you to execute SQL stored procedures.

The CallableStatement interface is designed to execute SQL-stored procedures. The procedures may have IN, OUT or IN OUT parameters. An IN parameter receives a value passed to the procedure when it is called. An OUT parameter returns a value after the procedure is completed, but it doesn’t contain any value when the procedure is called. An IN OUT parameter contains a value passed to the procedure when it is called, and returns a value after it is completed. For example, the following procedure in Oracle PL/SQL has IN parameter `p1`, OUT parameter `p2`, and IN OUT parameter `p3`.

```sql
create or replace procedure sampleProcedure
(p1 in varchar, p2 out number, p3 in out integer)
is
begin
    /* do something */
    end sampleProcedure;
/
```

**Note**
The syntax of stored procedures is vendor specific. We use both Oracle and MySQL for demonstrations of stored procedures in this book.

A CallableStatement object can be created using the `prepareCall(String call)` method in the Connection interface. For example, the following code creates a CallableStatement `cstmt` on Connection `connection` for the procedure `sampleProcedure`.

```java
CallableStatement callableStatement = connection.prepareCall("{call sampleProcedure(? , ?, ?)}");
```

 `{call sampleProcedure(? , ?, ...)}` is referred to as the SQL escape syntax, which signals the driver that the code within it should be handled differently. The driver parses the escape syntax and translates it into code that the database understands. In this example, `sampleProcedure` is an Oracle procedure. The call is translated to the string `begin sampleProcedure(? , ?, ?); end` and passed to an Oracle database for execution.

You can call procedures as well as functions. The syntax to create an SQL callable statement for a function is:

```java
{? = call functionName(? , ?, ...)}
```
CallableStatement inherits PreparedStatement. Additionally, the CallableStatement interface provides methods for registering the OUT parameters and for getting values from the OUT parameters.

Before calling an SQL procedure, you need to use appropriate set methods to pass values to IN and IN OUT parameters, and use registerOutParameter to register OUT and IN OUT parameters. For example, before calling procedure sampleProcedure, the following statements pass values to parameters \texttt{p1 (IN)} and \texttt{p3 (IN OUT)} and register parameters \texttt{p2 (OUT)} and \texttt{p3 (IN OUT)}:

```java
 callableStatement.setString(1, "Dallas"); // Set Dallas to p1
 callableStatement.setLong(3, 1); // Set 1 to p3
 // Register OUT parameters
 callableStatement.registerOutParameter(2, java.sql.Types.DOUBLE);
 callableStatement.registerOutParameter(3, java.sql.Types.INTEGER);
```

You can use \texttt{execute()} or \texttt{executeUpdate()} to execute the procedure depending on the type of SQL statement, then use get methods to retrieve values from the OUT parameters. For example, the next statements retrieve the values from parameters \texttt{p2} and \texttt{p3}.

```java
 double d = callableStatement.getDouble(2);
 int i = callableStatement.getInt(3);
```

Let us define a MySQL function that returns the number of the records in the table that match the specified \texttt{firstName} and \texttt{lastName} in the \texttt{Student} table.

```java
/* For the callable statement example. Use MySQL version 5 */
 drop function if exists studentFound;

delimiter //

create function studentFound(first varchar(20), last varchar(20))
 returns int
begin
 declare result int;

 select count(*) into result
 from Student
 where Student.firstName = first and
 Student.lastName = last;

 return result;
end;

//
delimiter ;
/* Please note that there is a space between delimiter and ; */
```

If you use an Oracle database, the function can be defined as follows:

```sql
create or replace function studentFound
 (first varchar2, last varchar2)
 returns number is
numberOfSelectedRows number := 0;
begin
 select count(*) into numberOfSelectedRows
 from Student
```
Suppose the function `studentFound` is already created in the database. Listing 34.4 gives an example that tests this function using callable statements.

**Listing 34.4 TestCallableStatement.java**

```
import java.sql.*;

public class TestCallableStatement {
    public static void main(String[] args) throws Exception {
        Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/javabook", "scott", "tiger");
        CallableStatement callableStatement = connection.prepareCall("{? = call studentFound(?, ?)}");
        java.util.Scanner input = new java.util.Scanner(System.in);
        System.out.print("Enter student's first name: ");
        String firstName = input.nextLine();
        System.out.print("Enter student's last name: ");
        String lastName = input.nextLine();
        callableStatement.setString(2, firstName);
        callableStatement.setString(3, lastName);
        callableStatement.registerOutParameter(1, Types.INTEGER);
        callableStatement.execute();
        if (callableStatement.getInt(1) >= 1)
            System.out.println(firstName + " " + lastName + " is in the database");
        else
            System.out.println(firstName + " " + lastName + " is not in the database");
    }
}
```

Enter student's first name: Jacob
Enter student's last name: Smith
Jacob Smith is in the database

Enter student's first name: John
Enter student's last name: Smith
John Smith is not in the database
The program loads a MySQL driver (line 6), connects to a MySQL database (lines 7–9), and creates a callable statement for executing the function `studentFound` (lines 15–16).

The function’s first parameter is the return value; its second and third parameters correspond to the first and last names. Before executing the callable statement, the program sets the first name and last name (lines 24–25) and registers the `OUT` parameter (line 26). The statement is executed in line 27.

The function’s return value is obtained in line 29. If the value is greater than or equal to 1, the student with the specified first and last name is found in the table.

### 34.23 Describe callable statements. How do you create instances of `CallableStatement`? How do you execute a `CallableStatement`? How do you register `OUT` parameters in a `CallableStatement`?

### 34.7 Retrieving Metadata

The database metadata such as database URL, username, JDBC driver name can be obtained using the `DatabaseMetaData` interface and result set metadata such as table column count and column names can be obtained using the `ResultSetMetaData` interface.

JDBC provides the `DatabaseMetaData` interface for obtaining database-wide information, and the `ResultSetMetaData` interface for obtaining information on the specific `ResultSet`.

#### 34.7.1 Database Metadata

The `Connection` interface establishes a connection to a database. It is within the context of a connection that SQL statements are executed and results are returned. A connection also provides access to database metadata information that describes the capabilities of the database, supported SQL grammar, stored procedures, and so on. To obtain an instance of `DatabaseMetaData` for a database, use the `getMetaData` method on a `Connection` object like this:

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

If your program connects to a local MySQL database, the program in Listing 34.5 displays the database information statements shown in Figure 34.24.

**Listing 34.5 TestDatabaseMetaData.java**

```java
import java.sql.*;

public class TestDatabaseMetaData {

    public static void main(String[] args) throws SQLException, ClassNotFoundException {

        // Load the JDBC driver
        Class.forName("com.mysql.jdbc.Driver");
        System.out.println("Driver loaded");

        // Connect to a database
        Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/javabook", "scott", "tiger");
        System.out.println("Database connected");

        DatabaseMetaData dbMetaData = connection.getMetaData();
        System.out.println("database URL: " + dbMetaData.getURL());
        System.out.println("database username: " +
```
Chapter 34  Java Database Programming

```java
18       dbMetaData.getUserName());
19      System.out.println("database product name: " +
20           dbMetaData.getDatabaseProductName());
21      System.out.println("database product version: " +
22           dbMetaData.getDatabaseProductVersion());
23      System.out.println("JDBC driver name: " +
24           dbMetaData.getDriverName());
25      System.out.println("JDBC driver version: " +
26           dbMetaData.getDriverVersion());
27      System.out.println("JDBC driver major version: " +
28           dbMetaData.getDriverMajorVersion());
29      System.out.println("JDBC driver minor version: " +
30           dbMetaData.getDriverMinorVersion());
31      System.out.println("Max number of connections: " +
32           dbMetaData.getMaxConnections());
33      System.out.println("MaxTableNameLength: " +
34           dbMetaData.getMaxTableNameLength());
35      System.out.println("MaxColumnsInTable: " +
36           dbMetaData.getMaxColumnsInTable());
37
38      // Close the connection
39      connection.close();
40  }
```

**Figure 34.24** The `DatabaseMetaData` interface enables you to obtain database information.

### 34.7.2 Obtaining Database Tables

You can identify the tables in the database through database metadata using the `getTables` method. Listing 34.6 displays all the user tables in the test database on a local MySQL database. Figure 34.25 shows a sample output of the program.

**Listing 34.6  FindUserTables.java**

```java
import java.sql.*;

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

// Connect to a database
Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/javabook", "scott", "tiger");
System.out.println("Database connected");

DatabaseMetaData dbMetaData = connection.getMetaData();

ResultSet rsTables = dbMetaData.getTables(null, null, null, null);
new String[] {
  "TABLE"});
System.out.print("User tables: ");
while (rsTables.next())
  System.out.print(rsTables.getString("TABLE_NAME") + " ");
// Close the connection
connection.close();

Figure 34.25 You can find all the tables in the database.

Line 17 obtains table information in a result set using the getTables method. One of the columns in the result set is TABLE_NAME. Line 21 retrieves the table name from this result set column.

34.7.3 Result Set Metadata

The ResultSetMetaData interface describes information pertaining to the result set. A ResultSetMetaData object can be used to find 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, Listing 34.7 displays all the column names and contents resulting from the SQL SELECT statement select * from Enrollment. The output is shown in Figure 34.26.

Listing 34.7 TestResultSetMetaData.java

import java.sql.*;
public class TestResultSetMetaData {
  public static void main(String[] args)
    throws SQLException, ClassNotFoundException {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    System.out.println("Driver loaded");
  
  Figure 34.26 You can find all the tables in the database.
Chapter 34 Java Database Programming

9 // Connect to a database
10 Connection connection = DriverManager.getConnection
11 ("jdbc:mysql://localhost/javabook", "scott", "tiger");
12 System.out.println("Database connected");
13
14 // Create a statement
15 Statement statement = connection.createStatement();
16
17 // Execute a statement
18 ResultSet resultSet = statement.executeQuery
19 ("select * from Enrollment");
20
21 ResultSetMetaData rsMetaData = resultSet.getMetaData();
22 for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
23 System.out.printf("%-12s\t", rsMetaData.getColumnName(i));
24 System.out.println();
25
26 // Iterate through the result and print the students' names
27 while (resultSet.next()) {
28 for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
29 System.out.printf("%-12s\t", resultSet.getObject(i));
30 System.out.println();
31 }
32
33 // Close the connection
34 connection.close();
35
36

FIGURE 34.26 The ResultSetMetaData interface enables you to obtain result set information.

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

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

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

KEY TERMS

database system 1212
primary key constraint 1215
domain constraint 1215
relational database 1215
foreign key constraint 1215
Structured Query Language (SQL) 1216
integrity constraint 1214
CHAPTER SUMMARY

1. This chapter introduced the concepts of database systems, relational databases, relational data models, data integrity, and SQL. You learned how to develop database applications using Java.

2. 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.

3. 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.

4. 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 in the classpath before running the program.

5. 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.

6. 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.

7. The PreparedStatement interface is designed to execute dynamic SQL statements with parameters. These SQL statements are precompiled for efficient use when repeatedly executed.

8. 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.

TEST QUESTIONS

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

PROGRAMMING EXERCISES

*34.1 (Access and update a Staff table) Write a Java applet that views, inserts, and updates staff information stored in a database, as shown in Figure 34.27a. The View button displays a record with a specified ID. The Staff table is created as follows:

```java
create table Staff (
    id char(9) not null,
```
**34.2 (Visualize data) Write a program that displays the number of students in each department in a pie chart and a bar chart, as shown in Figure 34.27b. The number of students for each department can be obtained from the Student table (see Figure 34.4) using the following SQL statement:

```sql
select deptId, count(*)
from Student
where deptId is not null
group by deptId;
```

**34.3 (Connection dialog) Develop a class 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 34.28. When the OK button is clicked, a Connection object for the database is stored in the connection property. You can then use the getConnection() method to return the connection.
*34.4  *(Find grades)* Listing 34.2, FindGrade.java, presented an applet that finds a student’s grade for a specified course. Rewrite the program to find all the grades for a specified student, as shown in Figure 34.29.

![Figure 34.29](image)

The program displays the grades for the courses for a specified student.

*34.5  *(Display table contents)* Write a program that displays the content for a given table. As shown in Figure 34.30a, you enter a table and click the *Show Contents* button to display the table contents in the text area.

![Figure 34.30](image)

(a) Enter a table name to display the table contents. (b) Select a table name from the combo box to display its contents.

*34.6  *(Find tables and showing their contents)* Write a program that fills in table names in a combo box, as shown in Figure 34.30b. You can select a table from the combo box to display its contents in the text area.

**34.7  *(Populate Quiz table)* Create a table named *Quiz* as follows:

```java
create table Quiz(
    questionId int,
    question varchar(4000),
    choicea varchar(1000),
    choiceb varchar(1000),
    choiccc varchar(1000),
    choiced varchar(1000),
    answer varchar(5));
```

The *Quiz* table stores multiple-choice questions. Suppose the multiple-choice questions are stored in a text file accessible from *www.cs.armstrong.edu/liang/data/Quiz.txt* in the following format:

```
1. question1
   a. choice a
   b. choice b
```
Write a program that reads the data from the file and populate it into the Quiz table.

**34.8** *(Populate Salary table)* Create a table named Salary as follows:

```sql
create table Salary(
    firstName varchar(100),
    lastName varchar(100),
    rank varchar(15),
    salary float);
```

Obtain the data for salary from [http://cs.armstrong.edu/liang/data/Salary.txt](http://cs.armstrong.edu/liang/data/Salary.txt) and populate it into the Salary table in the database.

**34.9** *(Copy table)* Suppose the database contains a student table defined as follows:

```sql
create table Student1 (
    username varchar(50) not null,
    password varchar(50) not null,
    fullname varchar(200) not null,
    constraint pkStudent primary key (username)
);
```

Create a new table named Student2 as follows:

```sql
create table Student2 (
    username varchar(50) not null,
    password varchar(50) not null,
    firstname varchar(100),
    lastname varchar(100),
    constraint pkStudent primary key (username)
)
```

A full name is in the form of `firstname mi lastname` or `firstname lastname`. For example, *John K Smith* is a full name. Write a program that copies table Student1 into Student2. Your task is to split a full name into `firstname`, `mi`, and `lastname` for each record in Student1 and store a new record into Student2.

**34.10** *(Record unsubmitted exercises)* The following three tables store information on students, assigned exercises, and exercise submission in LiveLab. LiveLab is an automatic grading system for grading programming exercises.

```sql
create table AGSStudent (
    username varchar(50) not null,
    password varchar(50) not null,
    fullname varchar(200) not null,
    instructorEmail varchar(100) not null,
```
The `AGSStudent` table stores the student information. The `ExerciseAssigned` table assigns the exercises by an instructor. The `AGSLog` table stores the grading results. When a student submits an exercise, a record is stored in the `AGSLog` table. However, there is no record in `AGSLog` if a student did not submit the exercise.

Write a program that adds a new record for each student and an assigned exercise to the student in the `AGSLog` table if a student has not submitted the exercise. The record should have the default values on `score` and `submitted`. For example, if the tables contain the following data in `AGSLog` before you run this program, the `AGSLog` table now contains the new records after the program runs.

<table>
<thead>
<tr>
<th>AGSStudent</th>
<th>ExerciseAssigned</th>
<th>AGSLog after the program runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>password</td>
<td>fullname</td>
</tr>
<tr>
<td>abc</td>
<td>p1</td>
<td>John Roo</td>
</tr>
<tr>
<td>cde</td>
<td>p2</td>
<td>Yao Mi</td>
</tr>
<tr>
<td>wbc</td>
<td>p3</td>
<td>F3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>username</td>
<td>exerciseName</td>
<td>score</td>
</tr>
<tr>
<td>abc</td>
<td>e1</td>
<td>9</td>
</tr>
<tr>
<td>wbc</td>
<td>e2</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>