19

Talking to Databases

In the next two chapters, we're going to look at how Java programs can interface with relational databases, or any database that can be accessed using Structured Query Language (SQL), using classes that come with the JDK.

First of all we will look at the basic ideas behind databases and how they store data. This leads naturally onto a discussion of SQL, the language that is used with many relational databases to both define and query data, and which is a prerequisite for database access in Java. Then we will look into the Java Database Connectivity (JDBC) class library, which provides a standard way for establishing and maintaining a Java program's connection to a database. Once you have a connection to a database, you can use SQL to access and process the contents.

In this first chapter, we'll take a brief tour of database concepts, SQL and JDBC. In the next chapter we will go into more depth on the capabilities provided by JDBC, and develop a database browsing application. In this chapter you will learn:

- What databases are
- What the basic SQL statements are and how you apply them
- What the rationale behind JDBC is
- How to write a simple JDBC program
- What the key elements of the JDBC API are

JDBC Concepts and Terminology

To make sure we have a common understanding of the jargon, we will first take a look at database terminology. Firstly, in general, **data access** is the process of retrieving or manipulating data that is taken from a remote or local **data source**. Data sources don't have to be relational – they can come in a variety of different forms. Some common examples of data sources that you might access are:

- A remote relational database on a server, for example, SQL Server
- A local relational database on your computer, for example, Personal Oracle or Microsoft Access
- A text file on your computer
- A spreadsheet
- A remote mainframe/midrange host providing data access
- An on-line information service (Dow Jones, etc.)

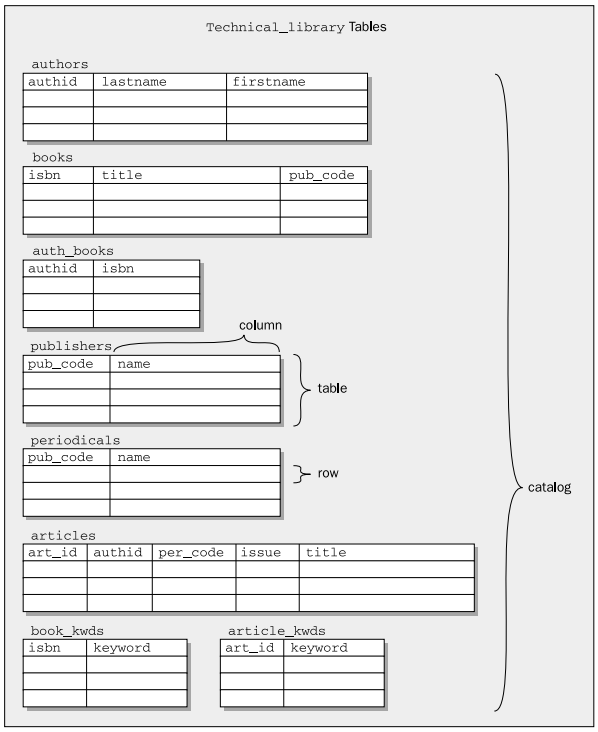
JDBC is, by definition, an interface to **relational** data sources. While it is conceivable that non-relational sources may be accessible through JDBC, we will be concentrating on relational databases throughout this chapter and the next. If you haven't met relational databases before, you should still be able to follow the discussion. The structure of relational databases is logical and fairly easy to learn, and while we can't provide a comprehensive tutorial on it here, we will cover enough of the basics to make what we are doing understandable.

The Java Database Connectivity (JDBC) library provides the means for executing SQL statements to access and operate on a relational database. JDBC was designed as an object-oriented, Java-based application programming interface (API) for database access, and is intended to be a standard to which Java developers and database vendors could adhere.

> *JDBC is based on other standard program-database interfaces – largely the X/Open SQL CLI (Call Level Interface) specification, but knowledge of these standards isn't necessary to use JDBC. However, if you've programmed database access before, you should be able to draw on that experience when using JDBC.*

The library is implemented in the `java.sql` package. It is a set of classes and interfaces that provide a uniform API for access to a broad range of databases.

The following figure shows the basic contents of the `technical_library` database (available from the Wrox Web site) that we will be using both to illustrate some key concepts, and as a base for the examples.



This shows the tables that make up the sample database. In case you are unfamiliar with relational databases, we will be going into what tables are in a moment.

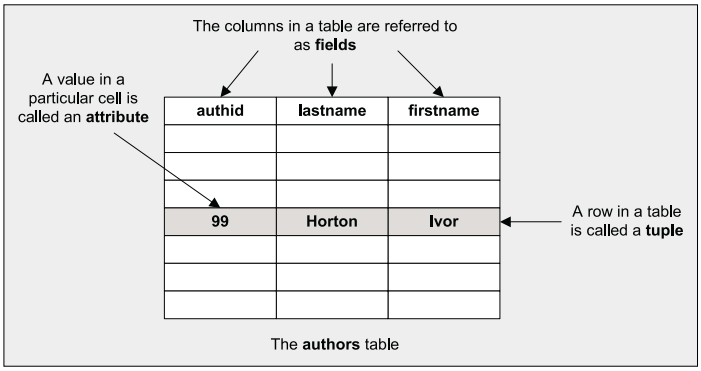
The operations that you want to carry out on a relational database are expressed in a language that was designed specifically for this purpose, the **Structured Query Language** – more commonly referred to as SQL. SQL is not like a conventional programming language, such as Java. SQL is a **declarative** language, which means that SQL statements tell the database server *what* you want to do, but not *how* it should be done – the *how* is up to the server. Each SQL command is analyzed by the database server and the operation it describes is carried out by a separate piece of software that is usually referred to as the **database engine**. A database engine will be associated with a particular implementation of a relational database, although not necessarily uniquely. Different commercial database implementations may use a common database engine.

Tables

A relational database is made of tables. A **table**, such as the `authors` table in our example above, is the primary database construct that you'll be dealing with. Any time that you define, create, update or delete data, you will do so within the context of a table.

When you create a table within a database, you are creating a 'template' for a rectangular grid that will contain the data. In relational parlance, a table is a collection of rows conforming to the specifications of the corresponding columns, and the table is called a **relation**. Each row implies that the set of data items that it contains are related in some way – it expresses a relationship between the data items, and a table can contain as many rows as you want.

The technical term for a row in a table is a **tuple**. The columns define the constituent parts of a row and are referred to as **fields**, and these column-defined items of data in a row are called **attributes**. Thus the number of columns for a given table is fixed.



Although a table is logically a set of rows with a fixed number of columns, the physical organization doesn't have to be like that. The physical organization can be anything at all as long as the logical organization – the way it appears when you access it – is as we have described.

Table Columns

As we said, a table behaves as if it is a rectangular grid of cells. The grid has a given number of columns and an arbitrary number of rows. Each column of cells in the grid stores data of a particular kind. Not only is it of a particular data type, it is also a specific category of information specified by the field name. For example, in the previous figure, the field with the name `lastname` is one of three fields defined for the `authors` table. It stores the last name of the authors that appear in the table, and the type of the data will be a text string. SQL has data types that correspond to the basic Java data types – we'll list them later.

It's fundamental to a relational database that the data items in each column in a table have consistent data types *and* semantics. For example, in the `authors` table, the `lastname` column, which is defined as a text field, will only be used to store text – not numbers, so the data type is preserved. The column's semantics must also be preserved, and since the column represents the last name of an author, you would not use this column to store an author's hobbies or favorite movie, even though such data items may be of the same type. If you wanted to record that information, you would need to create new columns in the table.

Table Rows

Each **row** in a table is the collection of data elements that make up an entity referred to as a **tuple**. Some data sources refer to a row as a **record**, and you will often see the term record used in a general purpose programming context when accessing a relational database. The term **recordset** is used to describe a collection of rows that is produced by executing an SQL command.

A row from the `authors` table in the database example that we showed earlier would look like:

```
20              Celko           Joe
```

This row contains data items for the `authid`, `lastname` and `firstname` columns. Although in this case there is a data item corresponding to each column, this does not have to be the case. A data item can be empty but it still appears in the row. Note that empty is not the same as zero, and `NULL` is used in SQL to denote the absence of a value. A rough analogy might be a variable in Java of type `Integer` that could contain a reference to an object that represents zero (or some other integer value of course), or could contain `null`. We will come back to the notion of `NULL` when we look at SQL data types.

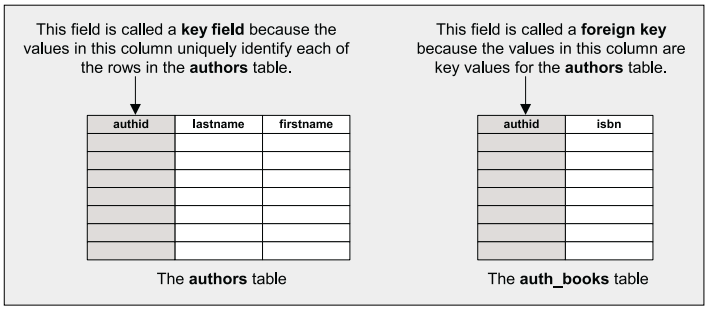
Database Catalog

In general, the **catalog** refers to the database **system tables**. System tables are similar to the tables used by your application, except that they are used to store information about the databases, the tables, and the composition of those tables, rather than storing application data. The catalog can be queried to find out what tables are available from a data source, what columns are defined for a given table, and so forth. The data describing the contents of a database is also referred to as **metadata** – data about data – or collectively as the **data dictionary**.

In the figure on page 999, 'catalog' refers to the entire `Technical_library` database.

Depending on the particular database to which you're connected, the catalog may contain other information related to accessing the database, such as security details, foreign keys and stored procedures.

The column values that together uniquely identify a particular row, make up what is referred to as the primary key**. Such columns are also called** primary key fields **or** primary key values**. Rows may also contain values that refer to key values in different tables. For example, the** `auth_books` **table contains a column** `author_id` **whose value is the primary key for a row in the** `authors` **table. When a table contains columns that are key columns for another table, values in those columns are referred to as** foreign keys.



Many database servers are capable of executing pre-built scripts, as well as SQL statements. These server-based scripts are called by the application, and are executed by the database server. They are frequently referred to as **stored procedures**, or just **procedures.** They are used to package up commonly used operations on the database, particularly those that enforce specific business rules.

Introducing SQL

Structured Query Language (SQL) is accepted internationally as the official standard for relational database access. A major reason for the acceptance of SQL as *the* relational query language was the move towards client/server architectures that began in the late 1980's.

> **Not all versions and dialects of SQL are created equal, however. As vendors have incorporated SQL into their products, extensions to the grammar have often been added. That was convenient for the database vendors but tough for anyone else trying to work with more than one database vendor. In order to ensure SQL's place as a standard for database access, organizations like ISO and ANSI have worked with the industry to develop standards for SQL. The current ISO operating standard is SQL-92, to which JDBC adheres. Conformance to the standard does not guarantee that your SQL will work in every case though. A database system that is in conformance with the standard for SQL is not obliged to implement all the capabilities that the standard defines. Indeed most database systems do not do so.**
>
> **SQL is different from other programming languages that you may be familiar with, in that it is declarative, not procedural. In other words, you don't use SQL to define complex processes, but rather use SQL to issue commands that define and manipulate data.**
>
> **If you need more information about using SQL, you may want to check out *Instant SQL* by Joe Celko (ISBN 1-874416-50-8), published by Wrox Press.**

The first thing that strikes you about SQL is that it is very readable. The way that each query is structured reads like a sentence in English. The syntax is easy to learn, and the constructs and concepts are very easy to grasp. Secondly, with SQL you always issue commands. You send the command to the database and the database either returns the required data, or performs the required action.

Let's look at the example that we saw earlier in the illustration – the `technical_library` database. How would we go about defining the tables we need to implement this database?

Try It Out — Designing Database Tables

The first step in designing the tables you need in your database is to decide what information you want to capture. Then you can design the tables around that information.

With our `technical_library` database, we want to keep track of the following kinds of things:

- Books
- Articles
- Authors
- Publishers

For each of these information categories, called **entities** in database jargon, we will want to record a specific set of data items, as follows:

Entity	Attribute
Books	ISBN
	Book title
	Author(s)
	Publisher
Articles	Author(s)
	Title
	Periodical it was published in
	Issue of publication
Authors	Last name
	First name
	Books published
	Articles published
Publishers	Publisher code
	Name

Let's start out with a table to keep track of the authors. We'll call this table `authors`, and describe the columns that we want for this table:

Column Heading	Description
`authid`	Unique identifier, since several authors could have the same name
`lastname`	Family name
`firstname`	First name
`address1`	Address line one
`address2`	Address line two
`city`	City
`state_prov`	State or province
`postcode`	Zip or postal code
`country`	Country
`phone`	Contact phone number
`fax`	Fax number
`email`	Email address

We need to assign a data type to each column heading that prescribes the form of the data in the column as it is stored in the table. Of course, these need to be data types meaningful to SQL, not necessarily Java data types. The data types for data in a relational database are those recognized by the SQL implementation supported by the database engine, and these types will have to be mapped to Java data types. Let's look at some examples of SQL data types:

SQL Data Type	Description
`CHAR`	Fixed length string of characters
`VARCHAR`	Variable length string of characters
`BOOLEAN`	Logical value – `true` or `false`
`SMALLINT`	Small integer value, from -127 to +127
`INTEGER`	Larger integer value, from -32767 to +32767
`NUMERIC`	A numeric value with a given **precision** – which is the number of decimal digits in the number, and a given **scale** – which is the number of digits after the decimal point. For instance, the value 234567.89 has a precision of 8 and a scale of 2.
`FLOAT`	Floating point value
`CURRENCY`	Stores monetary values
`DOUBLE`	Higher precision floating point value
`DATE`	Date
`TIME`	Time
`DATETIME`	Date and time
`RAW`	Raw binary data (can be used to store objects in a streamed binary format)

As we said earlier, `NULL` represents the absence of a value for any SQL type of data, but it is not the same as the `null` we have been using in Java. For one thing, you can't compare `NULL` with another `NULL` in SQL, you can only determine whether a particular attribute is or is not `NULL`. One effect of this is to introduce four potential values of type `BOOLEAN` – `TRUE` and `FALSE` which you would expect, `NULL` meaning the absence of a `BOOLEAN` value, and `UNKNOWN` which arises when the result cannot be determined – when you compare two `NULL` values for instance. Note that not all database systems allow `BOOLEAN` values to be `NULL` however.

From the data types in the authors table above, we can assign a data type for each column in the table that is appropriate for the kind of information in the column:

Column Name	Data Type
authid	INTEGER
lastname	VARCHAR
firstname	VARCHAR
address1	VARCHAR
address2	VARCHAR
city	VARCHAR
state_prov	VARCHAR
pzipcode	VARCHAR
country	VARCHAR
phone	VARCHAR
fax	VARCHAR
email	VARCHAR

How It Works

The column names tell us something about the information that will be stored in that field, but they don't tell the computer what type of information has been stored, or how to store it. While we are interested in the information stored in the columns, all the database engine wants to know is the *type* of information, and that's where the Data Type is invaluable.

You probably noticed that a column labeled authid has been placed at the top of the list of columns in the authors table. This is to give each record a unique identifier so that an individual record can easily be retrieved. Think of the nightmare you'd have if you were managing books and journals for a large library and you had several authors named John Smith. You wouldn't have any way of distinguishing one John Smith from the another. It's essential to give each row, or record, a unique identifier, and the author's ID serves this purpose here. Of course, it is not always necessary to introduce a column or columns specifically for this purpose. If your table already contains a column or combination of columns that will uniquely identify each row, then you can just use that.

Most of the data in the authors table is string information, so the VARCHAR data type was chosen as most convenient, since it allows as much or as little text information to be stored in that field as necessary. However, in practice it is likely to be more efficient to choose fixed length character fields.

Not all databases support VARCHAR**. In such cases, you will have to define these fields as** CHAR **type anyway, and estimate the maximum number of characters these fields will require. We will be doing this a little later when we get to a final definition of our database tables to allow the same table definitions to work with Access and other databases.**

The next requirement is to be able to store information about books. One possibility is to store the data in the `authors` table using extra columns. If we wanted to store books in the `authors` table, we might consider adding two columns – `title` and `publisher`. However, this would seriously restrict the amount of information about books written by a particular author – to one book in fact. Since each record in the `authors` table has a unique author ID, each author can have only one record in the table, and thus only one book could be recorded for each author. In practice, authors will frequently write more than one book or article so we must have a way to cope with this. A much more realistic approach is to store books in a separate table.

Try It Out — Defining the Books Table

We can easily store information about individual books by creating a table that will store the following information:

Column Heading	Description
`isbn`	ISBN is a globally unique identifier for a book
`title`	Title of the book
`pub_code`	Code identifying the publisher of the book

We also need an SQL data type assignment for each column in our `books` table.

Column Heading	Data Type
`isbn`	`VARCHAR`
`title`	`VARCHAR`
`pub_code`	`CHAR(8)`

Here the publisher's code is a fixed length character field. The other two fields vary in length so we have assigned `VARCHAR` as the most convenient type. Where this is not supported, we would have to use the `CHAR` type with a length sufficient to accommodate whatever data might turn up – something that is not always easy to decide.

How It Works

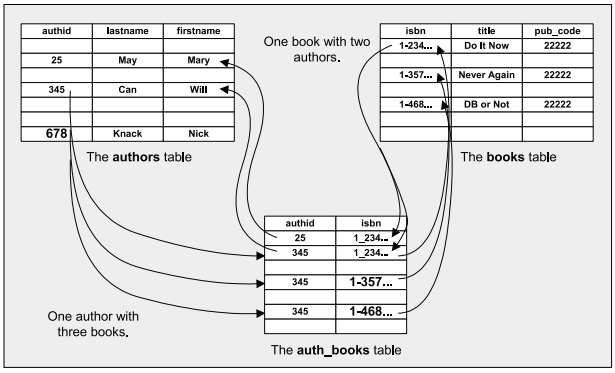
The `books` table allows us to record the ISBN for each book, which uniquely identifies the book, its title, and the publisher of the book. Notice, however, that we haven't included any information about the author. This isn't an oversight. Since more than one author can be involved in the writing of a book and an author can be involved in the writing of more than one book, we need to add some more information linking an author with a book, that will be independent of the `books` table. Let's see how we might do that.

Designing an Intersection Table

It is not difficult to see that we could create the link between an author and a book by using the `isbn` (the book identifier) and the `authid`. If we create a table with these two pieces of information, we can make a record of each combination of authors and the books they authored or co-authored. This table is simple enough – it merely contains a column for the author identifier and the ISBN. The data types must match the corresponding columns in the `authors` and `books` tables:

Column heading	Data Type
`authid`	`INTEGER`
`isbn`	`VARCHAR`

This table effectively provides links between the `authors` table and the `books` table. A table like this that links two or more tables is called an **intersection table**, and is illustrated below.



Now that we've decided on the design of the tables, let's see how we use SQL to create and add information to them. Rather than use the VARCHAR type for text fields, which is less widely supported, we will use fixed length CHAR types so that the database can be created in more environments. It may also be a bit more efficient.

SQL Statements

Most SQL statements, and certainly the ones we will be using, fall neatly into two groups:

- **Data Definition Language** (DDL) statements that are used to describe the tables and the data they contain
- **Data Manipulation Language** (DML) statements that are used to operate on data in the database. DML can be further divided into two groups:
  - **a.** **SELECT statements** — statements that return a set of results
  - **b.** Everything else — statements that don't return a set of results

In order to create the tables in our example, we would use DDL, which defines a syntax for commands such as `CREATE TABLE` and `ALTER TABLE`. We would use DDL statements to define the structure of the database. To carry out operations on the database, adding rows to a table or searching the data for instance, we would use DML statements.

Here's an example of a typical DDL statement:

```
CREATE TABLE authors (
  authid INT NOT NULL PRIMARY KEY,
  lastname CHAR(25) NOT NULL,
  firstname CHAR(15),
  address1 CHAR(25),
  address2 CHAR(25),
  city CHAR(25),
  state_prov CHAR(25),
  zipcode CHAR(10),
  country CHAR(15),
  phone CHAR(20),
  fax CHAR(20),
  email CHAR(25));
```

This is not so dissimilar to the data type assignments that we described earlier, but as you can see, in this SQL statement I've used fixed length `CHAR` fields rather than `VARCHAR` types. The values between parentheses are the number of characters in the field. Note that while it is not mandatory, by convention keywords in SQL are written in uppercase.

The clause `NOT NULL PRIMARY KEY` for the `authid` column tells the database two things. Firstly, no row of the table is allowed to contain a `NULL` value in this column. Every row in this column must always contain a valid value. Secondly, because this column is a primary key field, the database should create a unique index in the `authid` column. This ensures that there will be no more than one row with any given author ID. This greatly assists the database when searching through and ordering records. Think how difficult it would be to search for an entry in an encyclopedia without an index of unique values in one of the volumes.

This is the same principle on which database indexes work. Just to make sure we have some concrete information identifying an author, the `lastname` field is also not allowed to be `NULL`. Of course, all the tables in a database have to have unique names, otherwise it would not be possible to determine to which table you were referring. The names for the columns within a table must all be different for the same reason, and it is also helpful if you give all the non-key columns in all the tables in the database unique names, but this is not mandatory.

Now that we have a table created, we need to put data into the table. The SQL `INSERT` statement does exactly that.

INSERT Statements

There are three basic parts to an **insert statement**:

- Define the target table for inserting data
- Define the columns that will have values assigned
- Define the values for those columns

An insert statement begins with the keywords `INSERT INTO`, followed by the name of the target table:

```
INSERT INTO authors
```

You then supply a list of the names of the columns that will receive values. The columns are enclosed between parentheses:

```
(authid, lastname, firstname, email)
```

Lastly, you put the keyword `VALUES` followed by the values between parentheses for the columns you have identified:

```
VALUES (99, 'Phillips', 'Ron', 'ronp@happykitty.com')
```

Thus the complete `INSERT` statement is:

```
INSERT INTO authors (authid, lastname, firstname, email)
          VALUES (99, 'Phillips', 'Ron', 'ronp@happykitty.com')
```

The result of executing this statement is a new row inserted into the `authors` table. This statement does not fill in values for every column in the table, however. The SQL database will supply a `NULL` value where no values were supplied by the `INSERT` statement. If we had attempted to insert a row without a value for `authid`, the database would have reported an error, since the table was created with the `authid` column specified as `NOT NULL`.

A variation on the `INSERT` statement can be used when all column values are being filled by the statement: when no columns are specified, SQL assumes that the values following the `VALUES` keyword correspond to each column in the order that they were specified when the table was created. For example, you could add a row to the `books` table with the following statement:

```
INSERT INTO books (isbn, title, pub_code)
  VALUES ('1874416680', 'Beginning Linux Programming', 'WROX')
```

Since the `books` table contains only the three columns, the following statement has exactly the same results:

```
INSERT INTO books
  VALUES ('1874416680', 'Beginning Linux Programming', 'WROX')
```

Note how we have been spreading our SQL statements over two lines, just for readability. Whitespace is ignored generally in SQL, except in the middle of a string of course, so you can add whitespace wherever it helps to make your SQL code more readable.

Now, let us look at a basic SELECT statement, as this will give a starting point for getting some data back from the database we prepared earlier.

Select Statements

You use the SELECT statement to retrieve information from a database. There are four parts to an SQL SELECT statement:

- Defining what you want to retrieve
- Defining where you want to get it from
- Defining the conditions for retrieval – joining tables, and record filtering
- Defining the order in which you want to see the data.

So, how do you define what you want to retrieve? The first keyword in the SELECT statement, unsurprisingly, is SELECT. This tells the database that we intend to get some data back in the form of a **resultset**, sometimes referred to as a **recordset**. A **resultset** is just a table of data – with fixed numbers of columns and rows – that is some subset of data from a database table generated as a result of the SELECT statement.

The next identifier allows us to define what we want to see – it allows us to specify which columns we want to retrieve and to have as our resultset table headers. We specify each column name as part of a comma-separated list.

So our sample statement so far looks like:

```
SELECT firstname, lastname, authid
```

We now have to specify which table we want to retrieve the data from. When creating a table, there is nothing to stop the developer giving similar column names to each table, so we must ensure that there are no ambiguities when selecting similar column names from two or more tables.

We specify the table or tables that we wish to retrieve data from, in a **FROM clause**. This clause immediately follows the SELECT clause. A FROM clause consists of the keyword FROM, followed by a comma-separated list of tables that you wish to access:

```
FROM authors
```

Giving:

```
SELECT firstname, lastname, authid FROM authors
```

This is a complete statement that will retrieve the name, surname and author ID from each row in the `authors` table.

When a resultset is retrieved from the database, each resultset column has a label that, by default, is derived from the column names in the `SELECT` statement. It is also possible to provide **aliases** for the table column names that are to be used for the resultset column names. Aliases are also referred to as **correlation names**, and are often used so that column names can be abbreviated. Column aliases appear after the column names in the `SELECT` statement following the keyword `AS`. For example:

```
SELECT firstname, lastname, authid AS author_identifier
       FROM authors
```

would alias the `authid` as `author_identifier`. If you require an alias for a column name that includes whitespace, just put the alias between double quotes in the `SELECT` statement.

If you want to select all columns in a `SELECT` statement, there is a wildcard notation you can use. You just specify the columns as `*` to indicate that you want to select all the columns in a table. For example, to select all the columns from the `authors` table you would write:

```
SELECT * FROM authors
```

Suppose I wanted to limit the rows returned by a `SELECT` operation to only include authors that reside within the UK. In order to accomplish that, I would add a `WHERE` clause. `WHERE` clauses are used to filter the set of rows produced as the result of a `SELECT` operation. For example, to get a list of authors in the UK:

```
SELECT lastname, firstname FROM authors
  WHERE country = 'UK'
```

You can also specify multiple criteria for row selection in a `WHERE` clause. For example, I might want to get a list of authors in the UK for whom an email address is also on record:

```
SELECT lastname, firstname, phone FROM authors
  WHERE country = 'UK'
    AND email IS NOT NULL
```

Note the construction of the `WHERE` clause – there are two conditions that a row is required to satisfy before it will be returned. Firstly, the `country` field must contain a value that is equal to the string value `UK`, and the `email` field must not be `NULL`. If we wanted to find rows where the field is `NULL`, we would omit the `NOT`.

Let's look at one final example of a SELECT statement – a table join. Suppose we want to see a list of all authors and the books they have written. We can write a statement that will return this information like this:

```
SELECT a.lastname, a.firstname, b.title
  FROM authors a, books b, auth_books ab
  WHERE a.authid = ab.authid
    AND b.isbn = ab.isbn
```

The table join appears in the first line of the WHERE clause; we specify the condition for each row that the authid columns of the authors table and the auth_books table must be equal. We also specify that the isbn column of books and auth_books must be equal.

Notice also one small addition to the statement. As you saw earlier, we can alias column names by specifying an alternative name after each table identifier, or expression using the AS keyword. In this statement we are aliasing the table names in the FROM clause by simply putting the alias following the table name. The authors table is aliased as a, the books table is aliased as b and the auth_books table is aliased as ab. Back in the first part of the SELECT statement, the column names are 'qualified' by the aliases for each table. This is the way that column name ambiguities are removed. Since the authid column name appears in more than one table, if we did not have a qualifier in front of each usage of the authid column name, the database engine would have no way of knowing from which table the column was required. Note that if you specify an alias in the FROM clause, column names in the WHERE clause must be qualified with the appropriate table alias.

Update Statements

Update statements provide a way of modifying existing data in a table. Update statements are constructed in a similar way to SELECT statements. You first start with the UPDATE keyword, followed by the name of the table you wish to modify:

```
UPDATE authors
```

You then specify the SET keyword, and the data members you wish to modify, with their new values:

```
SET lastname = 'Burk'
```

Finally, the WHERE clause is used to filter the records that we wish to update. An update statement cannot be performed across a table join, so the WHERE clause is not used to specify a join of this type.

```
WHERE authid = 27
```

The full statement:

```
UPDATE authors SET lastname = 'Burk' WHERE authid = 27
```

will update the author record to reflect a change in last name for the author with the ID of 27.

Update statements do not return a resultset, they merely modify data in the database.

Delete Statements

Delete statements provide a way of deleting particular rows from tables in a database. Delete statements consist of the `DELETE` keyword, a `FROM` clause and a `WHERE` clause. For example:

```
DELETE FROM books WHERE isbn = '0131259075'
```

deletes the record in the `books` table with the ISBN value '0131259075'. In the case of the `books` table, there can only be one row with this value, since its primary key is the ISBN. If a similar `DELETE` statement were executed against the `auth_books` table, however, it would delete all rows with the matching ISBN value.

By now you should have a reasonably clear idea of:

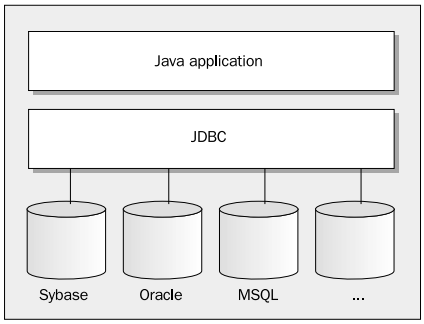
- The way SQL is constructed
- How to read SQL statements
- How to construct basic SQL statements

You can expect SQL statements to work with relational databases that adhere to ANSI standards, although each database typically implements a subset of the full standard. For this reason you need to understand the functionality of the SQL that is used by the underlying database you are using, as this will affect the way you use JDBC to write your Java applications.

The JDBC Package

The JDBC library was designed as an interface for executing SQL statements, and not as a high-level abstraction layer for data access. So, although it wasn't designed to automatically map Java classes to rows in a database, it allows large scale applications to be written to the JDBC interface without worrying too much about which database will be deployed with the application. A JDBC application is well insulated from the particular characteristics of the database system being used, and therefore doesn't have to be re-engineered for specific databases.

From the user's point of view, the Java application looks something like this:



JDBC manages this by having an implementation of the JDBC interface for each specific database – a **driver**. This handles the mapping of Java method calls in the JDBC classes to the database API. We'll learn more about this later on.

Relating JDBC to ODBC

One of the fundamental principles of JDBC's design was to make it practical to build JDBC drivers based on other database APIs. There is a very close mapping between the JDBC architecture and API, and their ODBC counterparts, fundamentally because they are all based on the same standard, the SQL X/Open CLI; but JDBC is a lot easier to use. Because of their common ancestry, they share some important conceptual components:

Driver Manager	Loads database drivers, and manages the connections between the application and the driver.
Driver	Translates API calls into operations for a specific data source.
Connection	A session between an application and a database.
Statement	A SQL statement to perform a query or update operation.
Metadata	Information about returned data, the database and the driver.
Result Set	Logical set of columns and rows of data returned by executing a statement.

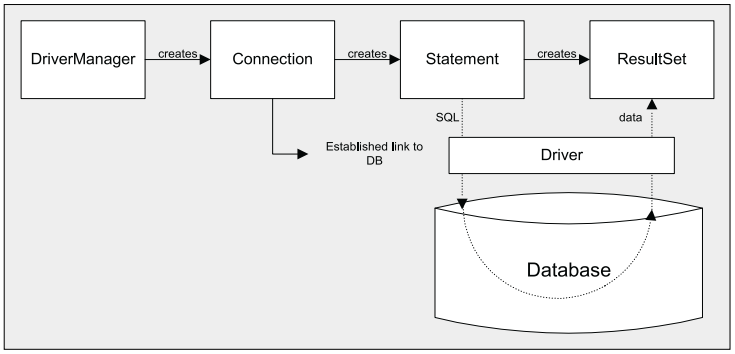
JDBC Basics

Assuming you have followed the instructions given at the start of the chapter, and have the requisite sample database and database driver installed on your machine, we are ready to look at a basic JDBC program that involves the following steps:

- Import the necessary classes
- Load the JDBC driver
- Identify the data source
- Allocate a `Connection` object
- Allocate a `Statement` object
- Execute a query using the `Statement` object
- Retrieve data from the returned `ResultSet` object
- Close the `ResultSet`
- Close the `Statement` object
- Close the `Connection` object

Throughout this chapter we will work towards accumulating a sufficient understanding of JDBC to implement the essential elements of such a program.

The JDBC architecture is based on a collection of Java interfaces and classes that together enable you to connect to data sources, to create and execute SQL statements, and to retrieve and modify data in a database. These operations are illustrated in the figure below:



Each of the boxes in the illustration represents a JDBC class or interface that has a fundamental role in accessing a relational database. All your work with JDBC will begin with the `DriverManager` class, but before we look at that let's set up our `technical_library` database so that we will be ready to use it as we progress.

Setting up a Database

Before we get started on learning about the `DriverManager` class, it's worth while setting up a sample database and a suitable JDBC driver for our code to use. The database is called `technical_library`, and it stores information on technical books. This is implemented as an MS Access database and you can download it together with the sample code from the Wrox Web site.

In MS Windows, to use this database with the application that we are going to construct in this chapter, you can set up an Access database driver for the database in the ODBC Data Source Administrator dialog box which can be found via Start -> Settings -> Control Panel, and then double-clicking on the ODBC Data Sources icon. Select the System DSN tab at the top of the dialog box, and click on the Add... button at the right. In the list box that comes up, select Microsoft Access Driver (*.mdb), and then click on Finish. A further dialog box will then come up with the title ODBC Microsoft Access Setup. In the Data Source Name text box at the top of the dialog, type in technical_library. Type in a suitable description in the Description text field if you wish. In the Database section of the dialog, click on the Select button, and in the file browsing dialog box that comes up, find and select your saved version of technical_library.mdb, then click OK. Now click on OK in the ODBC Microsoft Access Setup dialog. The System DSN section of the initial ODBC Data Source Administrator dialog should now have technical_library in the list of system data sources available. Click on the OK button at the bottom of the dialog to exit. Barring unforeseen problems you should now be able to use this database with the programs in this chapter.

If you're working with a database program other than Access you will need to obtain an appropriate driver for it if you do not already have one. An up-to-date list of suitable drivers for various databases can be found at http://www.javasoft.com/products/jdbc/jdbc.drivers.html.

If you have a database other than Access and the correct driver already set up, you can use a small Java class, `build_tables`, also included with the book's code, to create the sample database's tables. Simply run:

```
java build_tables buildlibrary_access.sql
```

Note that the program to set up the database takes a little time to run. If you are using something other than Access, you may need to edit the first few lines of the text file `buildlibrary_access.sql` to use the appropriate database driver, and possibly edit the rest of the instructions to accommodate the data types used by your database system. It is quite common for database systems not to support all of the SQL capabilities defined by the ANSI standard. If you have no luck getting the sample database up and running first time around, try reading on in this chapter and then re-reading your driver and database documentation before having another go. Having got a suitable database and JDBC driver installed, you can try running the InteractiveSQL program that we'll be using in this chapter to show how to send commands to a database. We'll build the application at the end of this chapter, by which time its workings should be plain to you.

In all the examples we will write in this chapter we'll be using the JDBC-ODBC Bridge driver — `sun.jdbc.odbc.JdbcOdbcDriver` — to access the MS Access database, `technical_library.mdb`, that we've assumed has been set up with a Microsoft Access ODBC driver as described in the section earlier.

DriverManager

JDBC database drivers are defined by classes that implement the `Driver` interface. The `DriverManager` class is responsible for establishing connections to the data sources, accessed through the JDBC drivers. If any JDBC driver has been identified in the `"jdbc.drivers"` system property (see below) on your computer, then the `DriverManager` class will attempt to load that when it is loaded.

The system properties are actually stored in a `Properties` object. The `Properties` class, defined in the `java.util` package, associates values with keys in a map, and the contents of the map defines a set of system properties. In general, each key is supplied as a `String` and the value corresponding to a key can be any valid object. Thus you can use a `Properties` object to supply as much information as is required by your driver – or anything else that interacts with the system properties for that matter. You just set the key/value pairs for the `Properties` object that are needed.

You can set the `"jdbc.drivers"` system property by calling the `setProperty()` method for the `System` class, for example:

```
System.setProperty("jdbc.drivers","sun.jdbc.odbc.JdbcOdbcDriver");
```

The first argument is the key for the property to be set and the second argument is the value. This statement identifies the JDBC-ODBC Bridge driver in the system property. This driver supports connections to any ODBC supported database. If you want to specify multiple drivers in the system property value, you should separate the driver names within the string by colons.

If the security manager permits it, you can obtain a reference to the `Properties` object for your system by calling the static `getProperties()` method for the `System` class. If there is no `Properties` object defined containing the system properties, one will be created with a default set of properties. The `Properties` class defines a `list()` method that you can use to list all your system properties as follows:

```
System.getProperties().list(System.out);   // List all properties
```

You could try this out in a simple program of your own if you want to see what your system properties are. The `Properties` class also defines a `setProperty()` method, so once you have a `Properties` object, you can set properties directly by calling this method for the object.

If a security manager is in effect and a security policy has been set up on your system, it may be that you will not be allowed to set the system property, in which case the `setProperty()` call with throw an exception of type `SecurityException`. In this situation, to include the driver that you want to use, you can load the driver explicitly by calling the static `forName()` method in the `Class` class, and passing a `String` object as an argument containing the driver class name. For example:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");   // Load the ODBC driver
```

The `forName()` method can throw an exception of type `ClassNotFoundException` if the driver class cannot be found, and this must be caught, so a call to the function has to appear in a `try` block with an appropriate `catch` block.

Each driver class will typically create an instance of itself when it is loaded, and register that instance by calling the `DriverManager` class method automatically. You don't need – indeed you can't – create `DriverManager` objects, and all the methods in the `DriverManager` class are `static`. There are `DriverManager` class methods that can be used to determine which drivers have been loaded, as well as methods that register or unregister drivers 'on the fly'. However, for the most part you will only need to call the method that establishes a connection to a data source.

When you need a connection to a JDBC driver, you don't create a new object encapsulating the connection yourself – you ask the `DriverManager` to do it for you. The `DriverManager` class provides several `static` methods for creating objects that implement the `Connection` interface, which we will get to in a moment, and that encapsulate a connection to a database. These are all overloaded versions of the `getConnection()` method.

Creating a Connection to a Data Source

A connection to a specific data source is represented by an object of a class that implements the `Connection` interface. Before you can execute any SQL statements, you must first have a `Connection` object. A `Connection` object represents an established connection to a particular data source, and you use it to create a `Statement` object that enables you to define and execute specific SQL statements. A `Connection` object can also be used to query the data source for information about the data in the database (the metadata), including the names of the available tables, information about the columns for a particular table, and so on.

There are three overloaded `getConnection()` methods in the `DriverManager` class that return a `Connection` object. In the simplest case you can obtain a `Connection` object that represents a session for your database with the following statement:

```
Connection databaseConnection = DriverManager.getConnection(source);
```

The argument, `source`, is a `String` object defining the URL that identifies where the database is located. Note that this is a `String` object specifying the URL, not an object of the `URL` class that we have seen earlier.

URLs and JDBC

As you saw when we discussed the `URL` class, a URL describes an electronic resource, such as a World Wide Web page, or a file on an FTP server, in a manner that uniquely identifies that resource. URLs play a central role in networked application development in Java. JDBC uses URLs to identify the locations of both drivers and data sources. JDBC URLs have the format:

jdbc:<subprotocol>://<data source identifier>

The scheme `jdbc` indicates that the URL refers to a JDBC data source. The sub-protocol identifies which JDBC driver to use. For example, the JDBC-ODBC Bridge uses the driver identifier `odbc`.

The JDBC driver dictates the format of the data source identifier. In our example above, the JDBC-ODBC Bridge simply uses the ODBC data source name. In order to use the ODBC driver with the `technical_library` ODBC data source, you would create a URL with the format:

```
jdbc:odbc:technical_library
```

The next step to getting data to or from a database is to create a `Connection` object. The `Connection` object essentially establishes a context in which you can create and execute SQL commands. Since the data source that we will use in this chapter's examples doesn't require a user name or password, the simplest form of the `getConnection()`method can be used.

We could exercise the `getConnection()` method in a working example.

Try It Out — Making a Connection

The following source code is a minimal JDBC program that creates a `Connection` object. In this instance the connection will be established using only the URL for the data source. In the next section we will look at how you can also supply a user ID and a password when this is necessary.

```
import java.sql.*;

public class MakingTheConnection
{
  public static void main(String[] args)
  {
    // Load the driver
    try
    {
      // Load the driver class
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

      // Define the data source for the driver
      String sourceURL = "jdbc:odbc:technical_library";

      // Create a connection through the DriverManager
      Connection databaseConnection =
                                DriverManager.getConnection(sourceURL);
    }
    catch(ClassNotFoundException cnfe)
    {
      System.err.println(cnfe);
    }
    catch(SQLException sqle)
    {
      System.err.println(sqle);
    }
  }
}
```

How It Works

Naturally we need to import the classes and interfaces for the JDBC library. These classes are defined in the `java.sql` package. The `forName()` method call at the beginning of `main()` ensures that the JDBC driver class required by our program is loaded. This will guarantee that any initialization that the JDBC driver must do will be completed before our code actually uses the driver. As we said earlier, the `forName()` method will throw a `ClassNotFoundException` if the driver class cannot be found, and this exception must be caught.

The `forName()` method call causes the Java interpreter's class loader to load the class for the driver specified by the argument. When the driver class is loaded, the class loader will determine if the driver class has any `static` initialization code. If it does, it will execute the `static` initialization code immediately after the class has been loaded. That is how the driver class is able to instantiate itself, and register the instance that is created with the `DriverManager` object. It can also execute other initialization code that may be required, such as loading a dynamic link library if the driver uses native methods for instance, and since this all happens when the class is loaded it is guaranteed to happen before any other driver methods get called.

Most JDBC methods handle errors by throwing an exception of the type `SQLException`, and the `getConnection()` method of the `DriverManager` class does exactly that, so we also have a `catch` block that handles the `SQLException` exception. In this example, a simple message will be displayed in the event of a problem loading the JDBC driver or creating a `Connection` to the data source. In the next chapter, you will learn more sophisticated error handling techniques.

More Complex Connections

If the database requires a user name and password in order to gain access to it, you can use the second form of the `getConnection()` method:

```
databaseConnection = DriverManager.getConnection(sourceURL,
                                                 myUserName,
                                                 myPassword);
```

All three arguments here are of type `String`. In some cases, however, the user name and password may not be enough to establish a connection. In order to accommodate those situations, the `DriverManager` class provides another `getConnection()` method that accepts a `Properties` object as an argument.

To supply the properties required by your JDBC driver, you can create a `Properties` object using the default class constructor, and then set the properties that you need by calling its `setProperty()` method. In general, at least the user name and password need to be set.

The code fragment below illustrates creation of a connection for the JBDC driver for ODBC.

```
import java.util.Properties;

// ...

  String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
  String sourceURL = "jdbc.odbc:technical_library";

  try
  {
    Class.forName (driverName);
    Properties prop = new Properties();
    prop.setProperty("user", "ItIsMe");
    prop.setProperty("password", "abracadabra");
    Connection databaseConnection = DriverManager.getConnection(sourceURL, prop);
  }
  catch(ClassNotFoundException cnfe)
  {
    System.err.println("Error loading " + driverName);
  }
  catch(SQLException sqle)
  {
    System.err.println(sqle);
  }
```

Note that the `Properties` class is imported from the `java.util` package.

While this pretty much covers everything that most developers will ever do with the `DriverManager` class, there are other methods that may be useful. We will take a look at these next.

Logging JDBC Driver Operations

The `DriverManager` class provides a pair of access methods for the `PrintWriter` object, that is used by the `DriverManager` class and all JDBC drivers, to record logging and trace information. These allow you to set, or reroute, the `PrintWriter` that the driver uses to log information. The two access methods are:

```
public static void setLogWriter(PrintWriter out)
public static PrintWriter getLogWriter()
```

You can disable logging by passing a `null` argument to the `setLogWriter()` method.

Examining the log can be pretty interesting. If you want to find out what's going on behind the scenes, take a look at the information generated by the JDBC-ODBC driver. You'll get a very good idea of how that driver works.

Your application can print to the `PrintWriter` stream using the static `println()` method defined in the `DriverManager` class. Just pass a `String` object as an argument containing the message you want to record in the log. This method is typically used by JDBC drivers, but it may prove useful for debugging or logging database-related errors or events.

Setting the Login Timeout

The `DriverManager` class provides a pair of access methods for the login timeout period. These allow you to specify a timeout period (in seconds) that limits the time that a driver is prepared to wait for logging in to the database. The two access methods are:

```
public static void setLoginTimeout(int seconds)
public static int getLoginTimeout()
```

Specifying a non-default timeout period can be useful for troubleshooting applications that are having difficulty connecting to a remote database server. For example, if your application is trying to connect to a very busy server, the applications might appear to have hung. You can tell the `DriverManager` to fail the connection attempt by specifying a timeout period. The code fragment below tells the `DriverManager` to fail the login attempt after 60 seconds:

```
String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
String sourceURL =  "jdbc:odbc:technical_library";

try
{
  Class.forName(driverName);
```

```
    // fail after 60 seconds
    DriverManager.setLoginTimeout(60);

    Connection databaseConnection = DriverManager.getConnection(sourceURL);
  }
  catch(ClassNotFoundException cnfe)
  {
    System.err.println("Error loading " + driverName);
  }
  catch(SQLException sqle)
  {
    System.err.println(sqle);
  }
```

More on Drivers

When the `DriverManager` class has been loaded, it is then possible to connect to a data source using a particular driver. A driver is represented by an object of type `Driver`. Driver implementations come in four flavors:

- JDBC-ODBC Bridge driver
- Native API partly-Java
- Net protocol all-Java client
- Native protocol all-Java

Understanding a little of how drivers are built, and their limitations, will help you to decide which driver is most appropriate for your application.

JDBC-ODBC Bridge Driver

The JDBC-ODBC Bridge – `"sun.jdbc.odbc.JdbcOdbcDriver"` – included with the JDK, enables Java applications to access data through drivers written to the ODBC standard. The driver bridge is very useful for accessing data in data sources for which no pure JDBC drivers exist.

The bridge works by translating the JDBC methods into ODBC function calls. It has the advantage of working with a huge number of ODBC drivers, but it only works under the Microsoft Windows and Sun Solaris operating systems.

Native API/Partly Java Driver

This class of driver is quite similar to the bridge driver. It consists of Java code that accesses data through native methods – typically calls to a particular vendor library. Like the bridge driver, this class of driver is convenient when a C data access library already exists, but it isn't usually very portable across platforms.

Net Protocol All Java Client

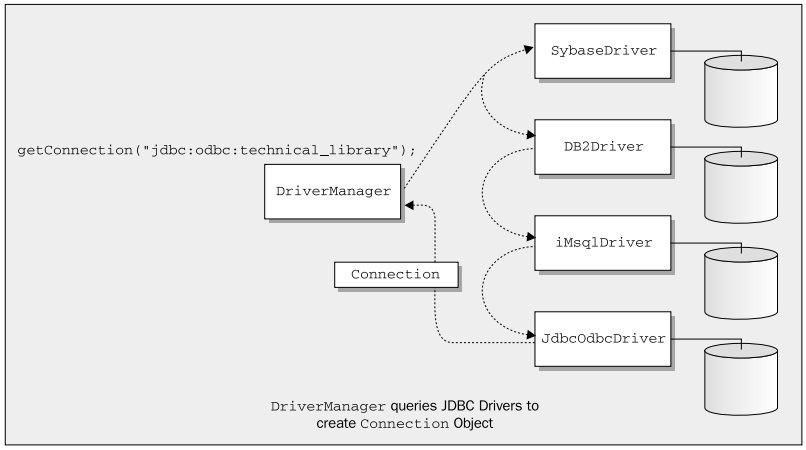
This class of driver is implemented as 'middleware', with the client driver completely implemented in Java. This client driver communicates with a separate middleware component (usually through TCP/IP) which translates JDBC requests into database access calls. This form of driver is an extension of the previous class, with the Java and native API separated into separate client and proxy processes.

Native Protocol All Java

This class of driver communicates directly to the database server using the server's native protocol. Unlike the previous driver type, there is no translation step that converts the Java-initiated request into some other form. The client talks directly to the server. If this class of driver is available for your database, then this is the one you should use.

> **There are a number of JDBC drivers available. As we mentioned earlier, the best source of up-to-date information about JDBC drivers is from the JavaSoft JDBC drivers page on their Web site:**
> http://www.javasoft.com/products/jdbc/jdbc.drivers.html**.**

The only time that you are likely to come into contact with the `Driver` object is when you install it. Your applications need not ever interact directly with the `Driver` object itself since the `DriverManager` class takes care of communicating with it. When you call the `getConnection()` method of the `DriverManager` class, it iterates through the drivers that are registered with the `DriverManager`, and asks each one in turn if it can handle the URL that you have passed to it. The first driver that can satisfy the connection defined by the URL creates a `Connection` object, which is passed back to the application by way of the `DriverManager`.



There are occasions, however, when you may want to query a specific driver for information, such as its version number. For example, you may know that a particular feature that your program makes use of wasn't incorporated into a driver until version 2.1. You can query the driver to get the version number so your program can handle an earlier version intelligently.

In order to get the `Driver` object, you call the static `getDriver()` method of the `DriverManager` class, passing the URL of the data source to it as the argument. If the `DriverManager` finds a driver that can accommodate the data source, a reference to a `Driver` object encapsulating it is returned. The code fragment below illustrates testing the version of a JDBC driver, looking for versions that are 1.1 or greater.

```
        // Load the driver class
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

        // Define the data source for the driver
        String sourceURL = "jdbc:odbc:technical_library";

        // Test for driver version
        int verMajor;
        float verComplete;
        float verPreferred;

        // Set the minimum preferred version
        verPreferred = 1.1f;

        // Set the driver
        Driver theDriver = DriverManager.getDriver(sourceURL);

        // Get the version number to the left of the decimal point, e.g. 1 out of 1.0
        verMajor = theDriver.getMajorVersion();

        /* Make a float of the complete version number by adding the minor number
           (to the right of the decimal point, e.g. 1108 out of 1.1108)
           on to verMajor */

           verComplete = Float.parseFloat(verMajor + "." +
                         theDriver.getMinorVersion());

        // Test to see if we have a suitable version of the driver
        if(verComplete >= verPreferred)
            System.out.println("Version " + verComplete + " found");
            //Make the connection...
        else
            System.out.println("Required version of driver (" +
                        verPreferred + ") not found");
            // Otherwise drop out...
```

In practice you could do a lot more that just output messages depending on the version of the driver that is available. Your program might choose to operate differently to account for the limitations of an earlier version for instance.

Statement Objects

A `Statement` object is an object of a class that implements the `Statement` interface. When a `Statement` object is created, it provides a workspace for you to create an SQL query, execute it, and retrieve any results that are returned. You can also assemble multiple SQL statements into a batch, and submit them for processing as a batch to the database.

`Statement` objects are created by calling the `createStatement()` method of a valid `Connection` object. Once you have created a `Statement` object, you can use it to execute an SQL query by calling the `executeQuery()` method for your `Statement` object. You pass a `String` object containing the text of your SQL query as the argument to the method.

The resultset from the query is returned as an object of type `ResultSet`. For instance, if you have a `Statement` object, `statement`, you could write:

```
ResultSet results = statement.executeQuery
                    ("SELECT lastname, firstname FROM authors");
```

This will execute the `SELECT` statement that appears in the argument.

When you want to batch up several SQL statements, you call the `addBatch()` method in the `Statement` object for each of them, passing the `String` object containing the SQL as the argument. When you finally want to execute the batch of SQL that you have created, you call `the executeBatch()` method for the statement object. To clear the batch of statements in readiness for a new set, you call the `clearBatch()` method for the `Statement` object. Because a batch of SQL statements can generate multiple resultsets, accessing them is a little complicated. It involves calling the `getResultSet()` method for the `Statement` object to retrieve the first resultset, and then using `getXXX()` methods for the object reference that is returned to access the contents. To move to the next resultset you call `getMoreResults()` for the `Statement` object. This returns true if the next result is another resultset, and false if the next result is not a resultset or there are no more results. You can then call `getResultSet()` again to obtain a reference to the next resultset if there is one. This is a relatively rare requirement so we won't go into further detail on this.

JDBC provides two other kinds of objects that you can use to execute SQL statements. These objects implement interfaces that are sub-interfaces of the `Statement` interface; the interface `PreparedStatement` that extends the `Statement` interface, and the interface `CallableStatement` that extends the `PreparedStatement` interface.

A `PreparedStatement` reference is returned by the `prepareStatement()` method in the `Connection` interface. In the simplest case, you just pass a `String` object specifying the text of an SQL statement to the method as the argument, but there is a more complex version of the method that provides you with more control over the resultset that is produced. `PreparedStatement` objects differ from `Statement` objects in that the SQL statement is pre-compiled, and can have placeholders for runtime parameter values. `PreparedStatement` objects are particularly useful when a statement will be executed many times (for example, adding new rows to a table), since substantial performance gains can be achieved in many cases.

This is due to the fact that a prepared statement is parsed once and reused, whereas the SQL for a `Statement` object has to be parsed by the server each time it is executed. `PreparedStatement` objects are also helpful when it is not convenient to create a single string containing the entire SQL statement. We'll see an example later in this chapter that will show the same SQL statement executed via both `Statement` and `PreparedStatement` objects.

A `CallableStatement` reference is returned by the `prepareCall()` method for a `Connection` object. There are two overloaded versions of this method, one requiring a single `String` JDBC-ODBC Bridge argument that defines the SQL for the stored procedure, and the other with additional parameters providing more control over the resultset that is produced. You use a `CallableStatement` object for calling procedures on the database. As we said earlier, many database engines have the ability to execute procedures. This allows business logic and rules to be defined at the server level, rather than relying on applications to replicate and enforce those rules.

Whichever type of `Statement` reference you are using, the results of an SQL query are always returned in the same way, so let's look at that.

ResultSet Objects

The results of executing an SQL query are returned in the form of an object that implements the `ResultSet` interface, and that contains the table produced by the SQL query. The `ResultSet` object contains something called a '**cursor**' that you can manipulate to refer to any particular row in the resultset. This initially points to a position immediately preceding the first row. Calling the `next()` method for the `ResultSet` object will move the cursor to the next position. You can reset the cursor to the first or last row at any time by calling the `first()` or `last()` method for the `ResultSet` object. You also have methods `beforeFirst()` and `afterLast()` to set the cursor position before the first row or after the last. The `previous()` method for the `ResultSet` object moves the cursor from its current position to the previous row. This ability to scroll backwards through a resultset is a recent innovation in JDBC (with JDBC 2) and is dependent on your database and your driver supporting this capability.

Usually you will want to process rows from a resultset in a loop, and you have a couple of ways to do this. Both the `next()` and `previous()` methods return `true` if the move is to a valid row, and `false` if you fall off the end, so you can use this to control a `while` loop. You could process all the rows in a resultset with the following loop:

```
while(resultset.next())
{
  // Process the row...
}
```

This assumes `resultset` is the object returned as a result of executing a query and the `resultset` object starts out in its default state with the cursor set to one before the first row. You can also use the `isLast()` or `isFirst()` methods to test whether you have reached the end or the beginning of the resultset.

Now we know how to get at the rows in a resultset, let's look into how we access the fields in a row.

Accessing Data in a Resultset

Using the `ResultSet` reference, you can retrieve the value of any column for the current row (as specified by the cursor) by name or by position. You can also determine information about the columns such as the number of columns returned, or the data types of columns. The `ResultSet` interface declares the following basic methods for retrieving column data for the current row as Java types:

`getAsciiStream()`	`getTimestamp()`	`getTime()`
`getBoolean()`	`getBinaryStream()`	`getString()`
`getDate()`	`getBytes()`	`getByte()`
`getInt()`	`getFloat()`	`getDouble()`
`getShort()`	`getObject()`	`getLong()`

Note that this is not a comprehensive list, but it is not likely you will need to know about the others. For a full list of the methods available take a look at the documentation for the `ResultSet` interface. There are overloaded versions of each of the methods shown above that provide two ways of identifying the column containing the data. The column can be selected by passing the SQL column name as a `String` argument, or by passing an index value for the column of type `int`, where the first column has the index value 1. Note that column names are not case sensitive so `"FirstName"` is the same as `"firstname"`.

The `getDate()`, `getTime()` and `getTimestamp()` methods return objects of type `Date`, `Time`, and `TimeStamp` respectively. The `getAsciiStream()` method returns an object of type `InputStream` that you can use to read the data as a stream of ASCII characters. This is primarily for use with values of the SQL type `LONGVARCHAR` which can be very long strings that you would want to read piecemeal. Most of the basic data access methods are very flexible in converting from SQL data types to Java data types. For instance if you use `getInt()` on a field of type `CHAR`, the method will attempt to parse the characters assuming they specify an integer. Equally, you can read numeric SQL types using the `getString()` method.

With all these methods, an absence of a value – an SQL `NULL` – is returned as the equivalent of zero, or `null` if an object reference is returned. Thus a `NULL` boolean field will return false and a `NULL` numeric field will return 0. If a database access error occurs when executing a `getXXX()` method for a resultset, an exception of type `SQLException` will be thrown.

The JDBC API provides access to metadata, not only for the `Connection` object, but also for the `ResultSet` object. The JDBC API provides a `ResultSetMetaData` object that lets you peek into the data behind the `ResultSet` object. If you plan on providing interactive browsing facilities in your JDBC applications, you'll find this particularly useful and we'll see how to do this later.

Together, these classes and interfaces make up the bulk of the JDBC components that you will be working with. Let's now put them into action with a simple code example.

Try It Out — Using a Connection

We'll do something useful with the `Connection` object created by changing our `MakingTheConnection` class into a new class for accessing the `technical_library` database. You can alter the code from the earlier example to that shown below:

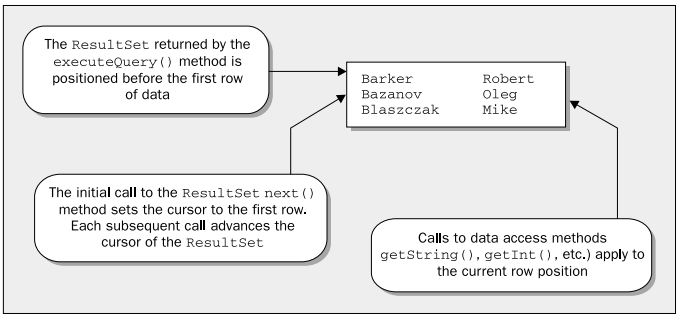
```
import java.sql.*;

public class MakingAStatement
{
  public static void main(String[] args)
  {
    // Load the driver
    try
    {
      // Load the driver class
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

      // This defines the data source for the driver
      String sourceURL = new String("jdbc:odbc:technical_library");

      // Create connection through the DriverManager
      Connection databaseConnection =
                           DriverManager.getConnection(sourceURL);

      Statement statement = databaseConnection.createStatement();

      ResultSet authorNames = statement.executeQuery
                             ("SELECT lastname, firstname FROM authors");

     // Output the resultset data
     while(authorNames.next())
      System.out.println(authorNames.getString("lastname")+" "+
      authorNames.getString("firstname"));
    }
    catch(ClassNotFoundException cnfe)
    {
      System.err.println(cnfe);
    }
    catch(SQLException sqle)
    {
      System.err.println(sqle);
    }
  }
}
```

You can save this as `MakingAStatement.java`. This program will list all the author names, one author to a line, with the last name first on each line.

How It Works

Once the connection has been established by the `getConnection()` method call, the next step is to create a `Statement` object that enables you to execute an SQL statement and retrieve the results. To create a `Statement` object we simply call the `createStatement()` method for the `Connection` object.

Once we have created the `statement` object, we execute an SQL query against the connected database by passing a `String` object as the argument to the `executeQuery()` method for the `statement` object. The `executeQuery()` method returns an object that implements the `ResultSet` interface. As the name implies, the `ResultSet` interface enables you to get at information that was retrieved by the query. You can think of the `ResultSet` interface as providing row-at-a-time access to a virtual table of results. The `ResultSet` object provides an internal **cursor** or logical pointer to keep track of its current row. When the `ResultSet` is first returned, the cursor is positioned just before the first row of data.



After executing the query and before any column data can be accessed, the row position needs to be advanced by calling the `next()` method, and we do this in the `while` loop condition. The `next()` method advances the row position and returns a `boolean` value that indicates if the `ResultSet` is positioned at a valid row (`true`), or that there are no more rows (`false`). Thus our `while` loop continues until we have output the data from all the rows that are in the resultset, `authorNames`.

Within the `while` loop, we access the data in the columns using the `getString()` method for the `ResultSet` object. In both cases we use the column names to reference the column. Accessing the columns by name has the advantage that you don't need to know the order of the columns. On the other hand you do need to know the column names. If you wanted to process the columns by their index position, you would just use the index values 1 and 2 to refer to data in the first and second columns respectively. Using the column position is slightly faster than using the column name since there is no additional overhead in matching a column name to determine a particular column position. It can also be more convenient to refer to columns using their position when you want to identify the column by means of an expression.

Note that in spite of the illustration above, the rows in the resultset are not ordered. If you want to output the rows in `lastname` order, you need to change the SQL statement to sort the rows, as follows:

```
ResultSet authorNames = statement.executeQuery(
                  "SELECT lastname, firstname FROM authors ORDER BY lastname");
```

The rows in the resultset will be sorted in `lastname` order – in ascending sequence by default. To sort in descending sequence you should add the keyword `DESC` to the end of the SQL statement. You can sort on multiple columns by separating the column names by commas. The sorting applies to the columns successively from left to right, so if you specify the sort columns as `lastname,firstname` in the `SELECT` statement then the rows in the resultset will be ordered by `lastname`, and where two last names are the same, by first name. For instance, if we want the rows in the resultset `authorNames` to be sorted in descending sequence, we could write:

```
ResultSet authorNames = statement.executeQuery(
         "SELECT lastname, firstname FROM authors
              ORDER BY lastname DESC, firstname DESC");
```

Note that we must supply the `DESC` keyword for each column name that we want it to apply to. If you omit it for a column the default ascending sequence will apply.

Getting Metadata for a Resultset

The `getMetaData()` method for a `ResultSet` object returns a reference to an object of type `ResultSetMetaData` that encapsulates the metadata for the resultset. The `ResultSetMetaData` interface declares methods that enable you to get items of metadata for the resultset.

The `getColumnCount()` method returns the number of columns in the resultset as a value of type `int`. For each column, you can get the column name and column type by calling the `getColumnName()` and `getColumnType()` methods respectively. In both cases you specify the column by its index value. The column name is returned as a `String` object and the column type is returned as an `int` value that identifies the SQL type. The `Types` class in the `java.sql` package defines public fields of type `int` that identify the SQL types, and the names of these class data members are the same as the SQL types they represent – such as `CHAR`, `VARCHAR`, `DOUBLE`, `INT`, `TIME`, and so on. Thus you could list the names of the columns in a resultset that were of type `CHAR` with the following code:

```
ResultSetMetaData metadata = results.getMetaData();
int columns = metadata.getColumnCount();                // Get number of columns

for(int i = 1 ; i<= columns ; i++)                      // For each column
  if(metadata.getColumnType(i) == Types.CHAR)           // if it is CHAR
     System.out.println(metadata.getColumnName(i));     // display the name
```

You could output the data value of each row of a `ResultSet` object, `results`, that were of SQL type `CHAR` with the following code:

```
ResultSetMetaData metadata = results.getMetaData();
int columns = metadata.getColumnCount();               // Get number of columns

int row = 0;                                           // Row number
while(results.next())                                  // For each row
```

```
{
  System.out.print("\nRow "+(++row)+":");              // increment row count
  for(int i = 1 ; i<= columns ; i++)                   // For each column
    if(metadata.getColumnType(i) == Types.CHAR)        // if it is CHAR display
it
      System.out.print(" "+results.getString(i));
}
```

You can also get the type name for a column as a `String` by calling the `getColumnTypeName()` method with the column number as the argument. Another very useful method is `getColumnDisplaySize()`, which returns the normal maximum number of characters required to display the data stored in the column. You pass the index number of the column that you are interested in as the argument. The return value is type `int`. You can use this to help format the output of column data.

There are a whole range of other methods that supply other metadata for a resultset that you will find in the documentation for the `ResultSetMetaData` interface. Here's a list of a few more that you may find useful – they all require an argument that is the column number as type `int`:

`getTableName()`	Returns the table name for the column as type `String`.
`getColumnLabel()`	Returns a `String` object that is the suggested label for a column for use in print-outs.
`getPrecision()`	Returns the number of decimal digits for a column as type `int`.
`getScale()`	Returns the number of decimal digits to the right of the decimal point for a column as type `int`.

`isSigned()`	Returns `true` if the column contains signed numbers.
`isCurrency()`	Returns `true` if the column contains currency values.
`isNullable()`	Returns an `int` value that can be: `columnNoNulls` indicating `NULL` is not allowed, `columnNullable` indicating `NULL` is allowed, `columnNullableUnknown` indicating it is not known if `NULL` is allowed.
`isWritable()`	Returns `true` if a write on the column is likely to succeed.

The Essential JDBC Program

We now have all the pieces to make up the essential JDBC program, which will initialize the environment, create `Connection` and `Statement` objects, and retrieve data by both position and column name.

Try It Out — Putting It All Together

Our application will execute two queries, one that selects specific columns by name, and another that selects all columns. First we will define the application class in outline, with the data members and the `main()` function and the other methods in the class:

```
import java.sql.*;

public class EssentialJDBC
{

  public static void main (String[] args)
  {
    EssentialJDBC SQLExample = new EssentialJDBC();   // Create application object

    SQLExample.getResultsByColumnName();
    SQLExample.getResultsByColumnPosition();
    SQLExample.getAllColumns();
    SQLExample.closeConnection();
  }

  public EssentialJDBC()
  {
    // Constructor to establish the connection and create a Statement object...
  }

  void getResultsByColumnName()
  {
    // Execute wildcard query and output selected columns...
  }

  void getResultsByColumnPosition()
  {
    // Execute ID and name query and output results...
  }

  void getAllColumns()
  {
    // Execute wildcard query and output all columns...
  }
```

```
    // Close the connection
    void closeConnection()
    {
      if(connection != null)
      try
      {
        connection.close();
        connection = null;
      }
      catch (SQLException ex)
      {
        System.out.println("\nSQLException------------------\n");
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("Message : " + ex.getMessage());
      }
    }

    Connection connection;
    Statement statement;
    String sourceURL = "jdbc:odbc:technical_library";
    String queryIDAndName = "SELECT authid, lastname, firstname FROM authors";
    String queryWildcard = "SELECT * FROM authors";          // Select all columns
  }
```

The data source is identified by a URL in the form, `jdbc:driver_name:datasource`. The data source identifier format is defined by the driver. In the case of the JDBC-ODBC Bridge, the data source is the ODBC source name. We have defined a `closeConnection()` method here that closes the connection when we are done. Notice that this method tests the value of the connection to ensure that we don't try to close a `null` connection.

Next we can fill in the details of the constructor for the class. This will establish a connection with the database and create a `Statement` object that will be used for executing queries.

```
    public EssentialJDBC()
    {
      try
      {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        connection = DriverManager.getConnection(sourceURL);
        statement = connection.createStatement();
      }
      catch(SQLException sqle)
      {
        System.err.println("Error creating connection");
      }
      catch(ClassNotFoundException cnfe)
      {
        System.err.println(cnfe.toString());
      }
    }
```

Next we can code the `getResultsByColumnName()` method. We will be using the `statement` object created from the `connection` object in the constructor to execute the SQL query to get a resultset back. A `while` loop with a call to `next()` as the condition will iterate through all the rows starting at the first:

```
void getResultsByColumnName()
{
  try
  {
    ResultSet authorResults = statement.executeQuery(queryWildcard);
    int row = 0;

    while(authorResults.next())
      System.out.println("Row " + (++row) + ") "+
                         authorResults.getString("authid")+ " " +
                         authorResults.getString("lastname")+ " , "+
                         authorResults.getString("firstname"));

    authorResults.close();
  }
  catch (SQLException sqle)
  {
    System.err.println ("\nSQLException------------------\n");
    System.err.println ("SQLState: " + sqle.getSQLState());
    System.err.println ("Message : " + sqle.getMessage());
  }
}
```

The `SQLException` handling code here doesn't provide very elegant error handling for this program, but we are obliged to catch this exception.

We can now define the `getResultsByColumnPosition()` method. This will use the query for the ID and names columns where the order of the columns is determined by the order of the column names in the query:

```
void getResultsByColumnPosition()
{
  try
  {
    ResultSet authorResults = statement.executeQuery(queryIDAndName);

    int row = 0;
    while (authorResults.next())
    {
      System.out.print("\nRow " + (++row) + ") ");
      for(int i = 1 ; i<=3 ; i++)
        System.out.print((i>1?", ":" ")+authorResults.getString(i));
    }
    authorResults.close();                          // Close the result set
  }
```

```
      catch (SQLException ex)
      {
        System.err.println("\nSQLException------------------\n");
        System.err.println("SQLState: " + ex.getSQLState());
        System.err.println("Message : " + ex.getMessage());
      }
    }
```

Next we can define the `getAllColumns()` method. This uses the wildcard form of `SELECT` statement where the * for the columns to be selected will retrieve all columns in the authors table. In general we won't necessarily know how many columns are returned in the resultset, but we can implement the method so that it will deal with any number of columns as well as any number of rows:

```
    void getAllColumns()
    {
      try
      {
        ResultSet authorResults = statement.executeQuery(queryWildcard);

        ResultSetMetaData metadata = authorResults.getMetaData();
        int columns = metadata.getColumnCount();          // Column count
        int row = 0;
        while (authorResults.next())
        {
          System.out.print("\nRow " + (++row) + ") ");
          for(int i = 1 ; i<=columns ; i++)
            System.out.print((i>1?", ":" ")+authorResults.getString(i));
        }

        authorResults.close();                            // Close the result set
      }
      catch (SQLException ex)
      {
        System.err.println("\nSQLException------------------\n");
        System.err.println("SQLState: " + ex.getSQLState());
        System.err.println("Message : " + ex.getMessage());
      }
    }
```

Running the `EssentialJDBC` program produces three sets of results. The first two sets are the same and consist of the ID and the name columns from the authors table. The third set lists all columns. Although the additional columns are null, you can see that we get them all in this case.

How It Works

The `EssentialJDBC` class provides a `main()` method to declare and allocate an `EssentialJDBC` object by calling the class constructor. It then calls the `getResultsByColumnName()`, the `getResultsByColumnPosition()`, and the `getAllColumns()` methods of the new object.

The constructor initializes member variables, and loads the `JdbcOdbc` driver class. It then creates a `Connection` object by calling the static `getConnection()` method of the `DriverManager` class. It then uses the `Connection` object to create a `Statement` object.

The bulk of the work is done in the three `getXXX()` methods. All three use the same `Statement` object to execute an SQL query. The difference between the three methods is how they retrieve the returned data.

The `getResultsByColumnName()` method executes the wildcard form of an SQL `SELECT` statement where the column names are specified by an `*` and the column ordering, of the returned results, is determined by the database engine. This query is executed by calling the `executeQuery()` method of the `Statement` object and this method returns the data in a `ResultSet` object. Since the column ordering is unknown ahead of time, we retrieve data by explicitly specifying the column names. The column data is retrieved as strings and written to the standard output stream. Finally, the `ResultSet` is closed. Note that the garbage collection of Java will handle this automatically anyway, but calling `close()` explicitly ensures that the resources, used by the `ResultSet` object, will be cleaned up sooner.

The `getResultsByColumnPosition()` method executes a `SELECT` that explicitly specifies the columns required by name so the column ordering in the resultset is the same as the sequence of column names in the `SELECT` statement. We can therefore use the column position index values to retrieve the data from the `ResultSet`. Like the previous method, the column data is retrieved as strings and printed to the console for each row returned. Finally, the `ResultSet` object is closed as before.

The `getAllColumns()` method uses the wildcard form of `SELECT` statement to retrieve a resultset containing all columns from the `authors` table – the entire table in other words. The method gets the count of the number of columns by means of the `ResultSetMetaData` object for the `ResultSet` object created as a result of the query. This is used to output however many columns there are in the resultset.

Using a PreparedStatement Object

Let's put a prepared statement into action now, to go through the mechanics in a practical context. This won't really show the capabilities of this – we will get to that in the next chapter. We will code an example that will execute the same SQL `SELECT` statement using both `Statement` and `PreparedStatement` objects. For each of these, the results will be displayed along with the metadata.

Try It Out — Statements and Metadata

First `import` the necessary classes from the `java.sql` package. Then define the `StatementTest` class and its member data. Define the class entry point `main()`, which will instantiate a `StatementTest` object, and then call the methods `doStatement()` and `doPreparedStatement()` for that object.

```
import java.sql.*;

public class StatementTest
{
  public static void main(String[] args)
  {
    StatementTest SQLExample;
    try
    {
      SQLExample = new StatementTest();
      SQLExample.doStatement();
      SQLExample.doPreparedStatement();
    }
    catch(SQLException sqle)
    {
      System.err.println("SQL Exception: " + sqle);
    }
    catch(ClassNotFoundException cnfe)
    {
      System.err.println(cnfe.toString());
    }
  }

  Connection databaseConnection;                // Connection to the database
  String driverName;                            // Database driver name
  String sourceURL;                             // Database location
}
```

Next we will define the `StatementTest` class constructor. This constructor assigns the driver name and the source URL that defines where the data will come from. It then loads the driver and calls the static `getConnection()` method of the `DriverManager` class to establish the database connection.

```
  public StatementTest() throws SQLException, ClassNotFoundException
  {
    driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
    sourceURL = "jdbc:odbc:technical_library";

    Class.forName (driverName);
    databaseConnection = DriverManager.getConnection(sourceURL);
  }
```

Next we will define the `doStatement()` method that is called in `main()`. This method shows once again how we create a `Statement` object, and use it to execute a query. The `ResultSet` object that is returned by the `executeQuery()` method of the `Statement` object is passed to the `showResults()` method of the `StatementTest` class to display the results.

```
public void doStatement() throws SQLException
{
  Statement myStatement = databaseConnection.createStatement();
  ResultSet myResults = myStatement.executeQuery
                        ("SELECT authid, lastname, firstname FROM authors");

  showResults(myResults);
}
```

Now we will define the `doPreparedStatement()` method. This method demonstrates how a `PreparedStatement` is created and executed. For the time being we will define it so that it operates in the same fashion as the `doStatement()` method.

```
public void doPreparedStatement() throws SQLException
{
  PreparedStatement myStatement = databaseConnection.prepareStatement
                ("SELECT authid, lastname, firstname FROM authors");
  ResultSet myResults = myStatement.executeQuery();
  showResults(myResults);
}
```

Firstly, define the `showResults()` method. This method is passed a `ResultSet` object, from which it extracts both data and metadata. Notice that the first thing this method does is retrieve the `ResultSetMetaData` object, from which it determines the number of columns returned. It then loops through and retrieves each column value as a string and prints it out.

After the data is displayed, the method extracts information about each column and displays that too.

```
public void showResults(ResultSet myResults) throws SQLException
{
  // Retrieve ResultSetMetaData object from ResultSet
  ResultSetMetaData myResultMetadata = myResults.getMetaData();

  // How many columns were returned?
  int numColumns = myResultMetadata.getColumnCount();

  System.out.println("-------------Query Results----------------");
  // Loop through the ResultSet and get data
  while(myResults.next())
  {
    for(int column = 1; column <= numColumns; column++)
      System.out.print(myResults.getString(column) + "\t");
    System.out.print("\n");
  }
```

```
    System.out.println("\n\n-----------Query Metadata----------------");
    System.out.println("ResultSet contains " + numColumns + " columns");
    for (int column = 1; column <= numColumns; column++)
    {
      System.out.println("Column " + column);
      // Print the column name
      System.out.println("\tcolumn\t\t:" +
                                    myResultMetadata.getColumnName(column));
      // Print the label name
      System.out.println("\tlabel\t\t:" +
                                   myResultMetadata.getColumnLabel(column));
      // Print the column's display size
      System.out.println("\tdisplay width\t:" +
                              myResultMetadata.getColumnDisplaySize(column) +
                               " characters");
      // Print the column's type
      System.out.println("\tdata type:\t:" +
                                 myResultMetadata.getColumnTypeName(column));
    }
  }
```

When you run the `StatementTest` program, you should get the following results twice:

```
-------------Query Results----------------
1       Gross                           Christian
2       Roche                           Kevin
3       Tracy                           Michael
4       Horton                          Ivor
...
52      Pompeii                         John
53      Brown                           Marc
54      Woelk                           Darrel


-----------Query Metadata----------------
ResultSet contains 3 columns
Column 1
        name            :authid
        label           :authid
        display width   :11 characters
        data type:      :LONG
Column 2
        name            :lastname
        label           :lastname
        display width   :25 characters
        data type:      :CHAR
Column 3
        name            :firstname
        label           :firstname
        display width   :15 characters
        data type:      :CHAR
```

How It Works

All we've done in this example is to take the concepts that you've seen in this chapter and put them all together into a working program.

The program creates and executes both a `Statement` and a `PreparedStatement` object, which should produce identical results. In this case, there were no parameters for the `PreparedStatement` (not to worry – you'll have more than enough `PreparedStatement` objects in the next chapter!). Since the results were identical, the `ResultSetMetaData` is identical for the two executed SQL statements as well.

Notice that all of the exception handling for this example is handled within `main()`. Each of the other methods that might generate exceptions declare those exceptions in their `throws` clause. If an exception occurs within any of those methods, the method will simply throw that exception back to the calling routine – `main()`.

The InteractiveSQL Tool

So far our examples have been console applications. In practice you will want to implement your database programs as interactive windowed applications, so let's apply what we know to creating an example. We will build an interactive SQL tool that will execute SQL statements to retrieve a resultset.

The InteractiveSQL tool will be a simple front end to the JDBC API. It will provide a means of entering and executing SQL statements, and have a display area for viewing results. This tool will be pretty basic in terms of functionality, but may come in handy for experimenting with SQL statements. You can always add extensions to this utility as you become more familiar with JDBC.

We will set our requirements for the InteractiveSQL tool to be fairly simple:

- Enable the user to enter and execute a SQL command
- Display the result set from a SQL query
- Display error information where appropriate

We will implement this as an application with a window based on the Swing class `JFrame`. We will also use a Swing component that is particularly useful for database applications – a table defined by the `JTable` class. The `JTable` class is defined in the `javax.swing.table` package along with some other classes and interfaces that support tables of data. The resultset that is generated when you execute an SQL `SELECT` statement is a rectangular table of data values, so a `JTable` component is ideal for displaying resultsets. Let's explore the basics of the `JTable` component so we can apply it to our InteractiveSQL program.

Using Tables

You use a `JTable` component to display a rectangular array of data on the screen. The items of data in the table do not have to be of all the same type, in fact each column of data in the table can be of a different type, either a basic type or class type. This is precisely the situation we have with a resultset. There are several ways to create a `JTable` component but we will just consider the most convenient in the database context, which is to use an object that encapsulates the data that is to be displayed in the table and implements the `TableModel` interface. You can create a `JTable` object directly from such an object by passing a reference of type `TableModel` to a `JTable` constructor. For example:

```
JTable table = new JTable(model);
```

Here, `model` is a variable of type `TableModel` that stores a reference to your object encapsulating the data to be displayed – the resultset in other words. All we need is a class to define this object, so next we need to know how to implement the `TableModel` interface.

Understanding the TableModel Interface

The `TableModel` interface declares methods that are used by a `JTable` object to access the data item to be displayed at each position in the table. This interface is defined in the `javax.swing.table` package, along with the `JTable` class. Our class encapsulating a resultset will need to implement this interface, and therefore define all the methods that the interface declares. The bad news is that there are nine of them. The good news is that there is an abstract class, `AbstractTableModel`, that implements six of them so that if we extend this class we have a minimum of three methods to define. The full set of methods declared in the `TableModel` interface is as follows:

`getColumnCount()`	Returns the number of columns in the table model as type `int`.
`getRowCount()`	Returns the number of rows in the table model as type `int`.
`getValueAt(int row, int column)`	Returns the value of the data item in the table model at the position specified by the argument as type `Object`.
`getColumnClass(int column)`	Returns the class type of the data in the columns specified by the argument as type `Class`.
`getColumnName(int column)`	Returns the name of the columns specified by the argument as type `String`.
`setValueAt(Object value, int row, int column)`	Sets the value, `value`, for the data item in the table model at the position specified by the last two arguments. There is no return value.
`isCellEditable(int row, int column)`	Returns `true` if the data item at the position specified by the arguments is editable, and `false` otherwise.
`addTableModelListener (TableModelListener tml)`	Adds a listener that is notified each time the table model is altered.
`removeTableModelListener (TableModelListener tml)`	Removes a listener that was listening for table model changes.

Remember, all these methods are called by the `JTable` object, so these provide the means whereby the `JTable` object accesses and manipulates data in the table model. The `AbstractTableModel` class provides default implementations for the last six methods in the list above, so the minimum you have to supply when you extend this class is the first three.

Defining a Table Model

We want to define a class that encapsulates a resultset, and it will be convenient to make the object of our class have the capability to accept a new resultset at any time. This will enable a single `JTable` object to display a series of different resultsets, just by setting a new resultset in the underlying `TableModel` object. We can do this by providing a method that will accept a `ResultSet` object as an argument, and making the contents available through the `TableModel` interface. With this in mind, the basic outline of the class will be:

```
import java.sql.*;
import javax.swing.table.*;

class ResultsModel extends AbstractTableModel
{
  public void setResultSet(ResultSet results)
  {
    // Make the data in the resultset available through the TableModel
interface...
  }
  public int getColumnCount()
  {
    // Return number of columns...
  }

  public int getRowCount()
  {
    // Return number of rows...
  }

  public Object getValueAt(int row, int column)
  {
    // Return the value at position row,column...
  }

  public String getColumnName(int column)
  {
    // Return the name for the column...
  }
}
```

We could access the data to be returned by the `TableModel` interface methods by going back to the original `ResultSet` object as necessary. This may involve going back to the database each time, and it will be generally more convenient and probably more efficient to cache the data from the resultset in the `ResultsModel` object. This means that the `setResultSet()` method will need to set this up. We will need to store two sets of information in the `ResultsModel` object – the column names, which are `String` objects, and the data in the table, which could be any of the types matching the SQL data types. To keep it simple we will access the data using the `getString()` method for the `ResultSet` object. Any of the SQL data types we will be using can be extracted as type `String`, and it means that we will only have to deal with strings at this point.

We can store the column names in a `String` array, so we can add a data member to the `ResultsModel` class to provide for this:

```
    String[] columnNames = new String[0];          // Empty array of names
```

Defining `columnNames` as an array with zero elements ensures that we start out with a non-null array, even though there is no resultset initially. We won't know in advance how many rows or columns of data there are, so we won't want to use an array. A `Vector` object will provide sufficient flexibility to accommodate whatever we need – indeed, although we won't need it here, it can store data items that differ in type, as long as they are objects. We can store the contents of a row as an array of `String` objects that we can then store as an element in the `Vector`. We can define the data member that stores the rows of data values as:

```
  Vector dataRows = new Vector();              // Empty vector of rows
```

Of course, we have set the type for all the data items in a row to `String` here, but if we wanted to accommodate different types within a row a `Vector` object could be used to store each row. These objects could then be stored in the `Vector` object, `dataRows`. We mustn't forget the `import` statement for the `Vector` class – it's defined in `java.util`.

We can now implement the `setResultSet()` method in our `ResultsModel` class as:

```
    public void setResultSet(ResultSet results)
    {
      try
      {
        ResultSetMetaData metadata = results.getMetaData();

        int columns =  metadata.getColumnCount();    // Get number of columns
        columnNames = new String[columns];           // Array to hold names

        // Get the column names
        for(int i = 0; i < columns; i++)
          columnNames[i] = metadata.getColumnLabel(i+1);


        // Get all rows.
        dataRows = new Vector();                     // New Vector to store the data
        String[] rowData;                            // Stores one row
        while(results.next())                        // For each row...
        {
          rowData = new String[columns];             // create array to hold the
  data
          for(int i = 0; i < columns; i++)           // For each column
             rowData[i] = results.getString(i+1);    // retrieve the data item

          dataRows.addElement(rowData);              // Store the row in the vector
        }
```

```
        fireTableChanged(null);           // Signal the table there is new model
  data
      }
      catch (SQLException sqle)
      {
        System.err.println(sqle);
      }
    }
```

To get the column names and the number of columns, we need access to the `ResultSetMetaData` object corresponding to the `ResultSet` object. The `getColumnLabel()` method for the `metadata` object returns the label to be used to name the column. This will either be the name of the column as known to the database, or the alias if you specify one in the `SELECT` statement used to create the resultset. The column names are stored in the array, `columnNames`.

We create a new `Vector` object to hold the rows from the resultset and store the reference in `dataRows`. This will replace any existing `Vector` object, which will be discarded. Each element in the vector will be an array of `String` objects, `rowData`, that we create in the `while` loop and set the values of its elements in the nested `for` loop, and once it's done we store it in `dataRows`. After all the rows from the resultset have been stored, we call the `fireTableChanged()` method that our class inherits from the base class. This method notifies all listeners for the `JTable` object for this model that the model has changed, so the `JTable` object should redraw itself from scratch. The argument is a reference to an object of type `TableModelEvent` that can be used to specify the parts of the model that have changed. This is passed to the listeners. We can pass a `null` here as we want to invalidate the whole thing.

The method to return the number of columns is now very easy to implement:

```
    public int getColumnCount()
    { return columnNames.length; }
```

The column count is the number of elements in the `columnNames` array.

Supplying the row count is just as easy:

```
    public int getRowCount()
    {
      if(dataRows == null)
        return 0;
      else
        return dataRows.size();
    }
```

The number of rows corresponds to the size of the `Vector` object, `dataRows`. We check to make sure that the value of the `dataRows` vector is not `null` to ensure that the initialization of the InteractiveSQL GUI can take place even when the vector has not been initialized.

The last method that completes the class definition as we have it provides access to the data values. We can implement this as follows:

```
public Object getValueAt(int row, int column)
{ return ((String[])(dataRows.elementAt(row)))[column]; }
```

The `elementAt()` method returns the element in the `Vector` at the position specified by the argument. This is returned as type `Object`, so we must cast it to type `String[]` before we can index it to access the value of the data item.

There is one other method we should add to the class – an implementation of `getColumnName()` that will return the column name given a column index. We can implement this as:

```
public String getColumnName(int column)
{
  return columnNames[column] == null ? "No Name" : columnNames[column];
}
```

We take the precaution here of dealing with a null column name by supplying a default column name in this case.

The Application GUI

The figure seen here shows the user interface for the InteractiveSQL tool. The text field at the top provides an entry area for typing in the SQL statement, and will be implemented using a `JTextField` component. The results display provides a scrollable area for the results of the executed SQL command. This will be implemented using a `JScrollPane` component. A status line, implemented as a `JTextArea` component provides the user with the number of rows returned from the query, or the text of any `SQLException` object generated by the query.

The illustration above also shows the menu items in the File menu, and the tooltip prompt for the SQL input area. The Clear query menu item will just clear the input area where you enter an SQL query.

Try It Out — Defining the GUI

We will derive the `InteractiveSQL` class from the `JFrame` class, and make this the foundation for the application. Its constructor will be responsible for loading the JDBC driver class, creating a connection to the database, and creating the user interface. The code is as follows:

```
import java.awt.*;
import java.awt.event.*;                // For event classes
import javax.swing.*;                   // For Swing components
import javax.swing.table.*;             // For the table classes
import java.sql.*;                      // For JDBC classes

public class InteractiveSQL extends JFrame
{
  public static void main(String[] args)
  { // Create the application object
    InteractiveSQL theApp = new InteractiveSQL("sun.jdbc.odbc.JdbcOdbcDriver",
                                               "jdbc:odbc:technical_library",
                                               "guest",
                                               "guest");
  }

  public InteractiveSQL(String driver, String url,
                        String user , String password)
  {
    super("InteractiveSQL");                          // Call base constructor
    setBounds(0, 0, 400, 300);                        // Set window bounds
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);       // Close window operation
    addWindowListener(new WindowHandler());           // Listener for window close

    // Add the input for SQL statements at the top
    command.setToolTipText("Key SQL commmand and press Enter");
    getContentPane().add(command, BorderLayout.NORTH);

    // Add the status reporting area at the bottom
    status.setLineWrap(true);
    status.setWrapStyleWord(true);
    getContentPane().add(status, BorderLayout.SOUTH);

    // Create the menubar from the menu items
    JMenu fileMenu = new JMenu("File");               // Create File menu
    fileMenu.setMnemonic('F');                        // Create shortcut
    fileMenu.add(clearQueryItem);                     // Add clear query item
    fileMenu.add(exitItem);                           // Add exit item
    menuBar.add(fileMenu);                            // Add menu to the menubar
    setJMenuBar(menuBar);                             // Add menubar to the window
```

```
      // Establish a database connection and set up the table
      try
      {
        Class.forName(driver);                          // Load the driver
        connection = DriverManager.getConnection(url, user, password);
        statement = connection.createStatement();

        model = new ResultsModel();                // Create a table model
        JTable table = new JTable(model);          // Create a table from the model
        table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);   // Use scrollbars
        resultsPane = new JScrollPane(table);        // Create scrollpane for table
        getContentPane().add(resultsPane, BorderLayout.CENTER);

      }
      catch(ClassNotFoundException cnfe)
      {
        System.err.println(cnfe);                   // Driver not found
      }
      catch(SQLException sqle)
      {
        System.err.println(sqle);                   // error connection to database
      }
      pack();
      setVisible(true);
    }
    class WindowHandler extends WindowAdapter

    {
      // Handler for window closing event
      public void windowClosing(WindowEvent e)
      {
        dispose();                                  // Release the window resources
        System.exit(0);                             // End the application
      }
    }

    JTextField command = new JTextField();      // Input area for SQL
    JTextArea status = new JTextArea(3,1);      // Output area for status and errors
    JScrollPane resultsPane;

    JMenuBar menuBar = new JMenuBar();                        // The menu bar
    JMenuItem clearQueryItem = new JMenuItem("Clear query");  // Clear SQL item
    JMenuItem exitItem = new JMenuItem("Exit");               // Exit item

    Connection connection;                       // Connection to the database
    Statement statement;                         // Statement object for queries
    ResultsModel model;                          // Table model for resultset
  }
```

You can try running the application as it is and you should see the basic application interface displayed in the window with a working close operation.

How It Works

The constructor is passed the arguments required to load the appropriate driver and create a `Connection` to a database. The first executable statement in this constructor calls the constructor for the `JFrame` class, passing a default window title to it. The constructor then creates and arranges the user interface components. Most of this should be familiar to you but let's pick out a few things that are new, or worthy of a second look.

You can see how we add a tooltip for the `JTextField` component, `command`, – the input area for an SQL statement. Don't forget that you can add a tooltip for any Swing component in the same way.

We define the `JTextArea` object `status` so that it can display three lines of text. The first argument to the constructor is the number of lines of text, and the second argument is the number of columns. Some of the error messages can be quite long, so we call both the `setLineWrap()` method to make lines wrap automatically, and the `setWrapStyleWord()` method to wrap a line at the end of a word – that is on whitespace – rather than in the middle of a word. In both cases the `true` argument switches the facility on.

We create the `JTable` object using the default `ResultsModel` object which will contain no data initially. Since the number of columns in a resultset will vary depending on the SQL query that is executed, we wrap our `JTable` object in a `JScrollPane` object to provide automatic scrolling as necessary. The scrollbars will appear whenever the size of the `JTable` object is larger than the size of the scroll pane. By default, a `JTable` object will resize the width of its columns to fit within the width of the `JTable` component. To inhibit this and allow the scroll pane scrollbars to be used, we call the `setAutoResizeMode()` method with the argument as `JTable.AUTO_RESIZE_OFF`.

This not only inhibits the default resizing action when the table is displayed, but also allows you to change the size of a column when the table is displayed without affecting the size of the other columns. You change the size of a column by dragging the side of the column name using the mouse. There are other values you can pass to this method that affect how resizing is handled:

`AUTO_RESIZE_ALL_COLUMNS`	Adjusts the sizes of all columns to take up the change in width of the column being resized. This maintains the overall width of the table.
`AUTO_RESIZE_NEXT_COLUMN`	Adjusts the size of the next column to provide for the change in the column being altered in order to maintain the total width of the table.
`AUTO_RESIZE_LAST_COLUMN`	Adjusts the size of the last column to provide for the change in the column being altered in order to maintain the total width of the table.
`AUTO_RESIZE_SUBSEQUENT_COLUMNS`	Adjusts the size of the columns to the right to provide for the change in the column being altered in order to maintain the total width of the table.

The program only works with the database and driver hard-coded in the program. We can make it a lot more flexible by allowing command line argument to be supplied that specify the database and driver, as well as the user ID and password.

Handling Command Line Arguments

All we need to do is to alter `main()` to accept up to four command line arguments: these will be the values for the user name, password, database URL and JDBC driver

Try It Out — Using Command Line Parameters

We need to modify the code in `main()` to:

```
public static void main(String[] args)
{
   // Set default values for the command line args
   String user     = "guest";
   String password = "guest";
   String url      = "jdbc:odbc:technical_library";
   String driver   = "sun.jdbc.odbc.JdbcOdbcDriver";

   // Up to 4 arguments in the sequence database url,driver url, user ID, password
   switch(args.length)
   {
     case 4:                                  // Start here for four arguments
       password = args[3];
       // Fall through to the next case
     case 3:                                  // Start here for three arguments
       user = args[2];
       // Fall through to the next case
     case 2:                                  // Start here for two arguments
       driver = args[1];
       // Fall through to the next case
     case 1:                                  // Start here for one argument
       url = args[0];
   }
    InteractiveSQL theApp = new InteractiveSQL(driver, url, user, password);
}
```

How It Works

This enables you to optionally specify the JDBC URL, the JDBC driver, the user name and the password, on the command line. The mechanism that handles the optional parameters is pretty simple. The `switch` statement tests the number of parameters that were specified on the command line. If one parameter was passed, it is interpreted as the JDBC URL. If two parameters were passed, the second parameter is assumed to be the driver URL, and so on. There are no `break` statements, so control always drops through from the starting case to include each of the following cases.

Handling Events

To make the program operational, we need an event to add the handling logic for the menu items and the input field for the SQL statement. We can handle the menu items in the usual way by making the `InteractiveSQL` class implement the `ActionListener` interface. For the `JTextField` object we can add an action listener to respond to the *Enter* key being pressed at the end of entering an SQL statement.

Try It Out — Events in `InteractiveSQL`

First, we need to add the interface declaration to the class definition. The `InteractiveSQL` class will implement the `ActionListener` interface to handle menu events so you should change the first line of the definition to:

```
public class InteractiveSQL extends JFrame
                             implements ActionListener
```

We can now add code to the class constructor to add listeners for the menu item action events, and the `JTextField` action event.

```
public InteractiveSQL(String driver, String url,
                      String user , String password)
  {
    super("InteractiveSQL");                          // Call base constructor
    setBounds(0, 0, 400, 300);                        // Set window bounds
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);       // Close window operation
    addWindowListener(new WindowHandler());           // Listener for window close

    // Add the input for for SQL statements at the top
    command.setToolTipText("Key SQL commmand and press Enter");
    command.addActionListener(this);
    getContentPane().add(command, BorderLayout.NORTH);
    // Add the status reporting area at the bottom
    status.setLineWrap(true);
    status.setWrapStyleWord(true);
    getContentPane().add(status, BorderLayout.SOUTH);

    // Create the menubar from the menu items
    JMenu fileMenu = new JMenu("File");               // Create File menu
    fileMenu.setMnemonic('F');                        // Create shortcut
    clearQueryItem.addActionListener(this);
    exitItem.addActionListener(this);
    fileMenu.add(clearQueryItem);                     // Add clear query item
    fileMenu.add(exitItem);                           // Add exit item
    menuBar.add(fileMenu);                            // Add menu to the menubar
    setJMenuBar(menuBar);                             // Add menubar to the window
```

```
    // Establish a database connection and set up the table
    try
    {
      Class.forName(driver);                          // Load the driver
      connection = DriverManager.getConnection(url, user, password);
      statement = connection.createStatement();

      model = new ResultsModel();                 // Create a table model
      JTable table = new JTable(model);           // Create a table from the model
      table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);   // Use scrollbars
      resultsPane = new JScrollPane(table);        // Create scrollpane for table
      getContentPane().add(resultsPane, BorderLayout.CENTER);

    }
    catch(ClassNotFoundException cnfe)
    {
      System.err.println(cnfe);                    // Driver not found
    }
    catch(SQLException sqle)
    {
      System.err.println(sqle);                    // error connection to database
    }
    pack();
    setVisible(true);
  }
```

That's all we need to do to add the listeners. We can now add the `actionPerformed()` method. This method will get the reference to the component that originated the event from the event object, and use that to determine what action is necessary:

```
  public void actionPerformed(ActionEvent e)
  {
    Object source = e.getSource();
    if(source == command)                      // Enter key for text field input
      executeSQL();

    else if(source == clearQueryItem)          // Clear query menu item
      command.setText("");                     // Clear SQL entry
    else if(source == exitItem)                // Exit menu item
    {
      dispose();                               // Release the window resources
      System.exit(0);                          // End the application
    }
  }
```

The final piece is the `executeSQL()` method that will execute the statements that was entered in the text field command, and pass the resultset generated to the table model object, `model`, that supplies the data for the `JTable` object:

```
public void executeSQL()
{
  String query = command.getText();      // Get the SQL statement
  if(query == null)                      // If there's nothing we are done
    return;
  try
  {
    model.setResultSet(statement.executeQuery(query));
    status.setText("Resultset has " + model.getRowCount() + " rows.");
  }
  catch (SQLException sqle)
  {
    status.setText(sqle.getMessage());        // Display error message
  }
}
```

Compile and run the application, and remember the command line parameters: `<URL> <driver> <user id> <password>`. For example, try:

```
java InteractiveSQL jdbc:odbc:technical_library
```

and then execute the SQL query:

```
SELECT firstname AS "First Name", lastname AS Surname FROM authors
                    ORDER BY lastname,firstname
```

You should get the resultset displayed here:

How It Works

Pressing the *Enter* key after typing in the SQL query to be executed causes an action event to be generated for the `JTextField` object, `command`, that receives the input. The `actionPerformed()` method identifies the event by comparing the reference to the originating object with each of the objects in the GUI that have the application object as the action listener. When the event originates with the `command` object, the `executeSQL()` method is called. This retrieves the query from the text field and executes it using the `Statement` object. The `ResultSet` object that is returned is passed to the `setResultSet()` method for the `model` object. The model object extracts the data from the resultset and alerts the `JTable` object. This causes the data in the resultset to be displayed.

If an error occurs when the query is executed, a `SQLException` will be thrown. The catch block handling this exception passes the message text from the exception object to the `JTextArea` object, `status`.

You now have a simple but useful tool for executing SQL statements through JDBC.

Summary

In this chapter you've been introduced to JDBC programming, and seen it in action. The important points covered in this chapter are:

- The fundamental classes in JDBC are:
- `DriverManager` – manages loading of JDBC drivers and connections to client applications
- `Connection` – provides a connection to a specific data source
- `Statement` – provides a context for executing SQL statements
- `ResultSet` – provides a means for accessing data returned from an executed `Statement`
- The essential JDBC program has the following basic structure when writing:
  - Import the necessary classes
  - Load the JDBC driver
  - Identify the data source
  - Allocate a `Connection` object
  - Allocate a `Statement` object
  - Execute a query using the `Statement` object
  - Retrieve data from the returned `ResultSet` object
  - Close the `ResultSet`
  - Close the `Statement` object
  - Close the `Connection` object
- The `JTable` component provides an easy and convenient way to display the results of database queries.
- A table model can provide the data to be displayed by a `JTable` component. A table model is an object of a class that implements the `TableModel` interface.

Exercises

1. Write a program that outputs all authors in the `technical_library` database with last names starting with the letters A through H.

2. Write a program that lists all books and the authors for that book. (Hint: you will need an SQL join and the `author_books` table.)

3. Modify the `InteractiveSQL` program to allow the user to specify which database to use.

4. Modify the `InteractiveSQL` program to provide separate input areas for each part of a `SELECT` statement – one for the table columns, one for the table, one for a possible `WHERE` clause, and so on.