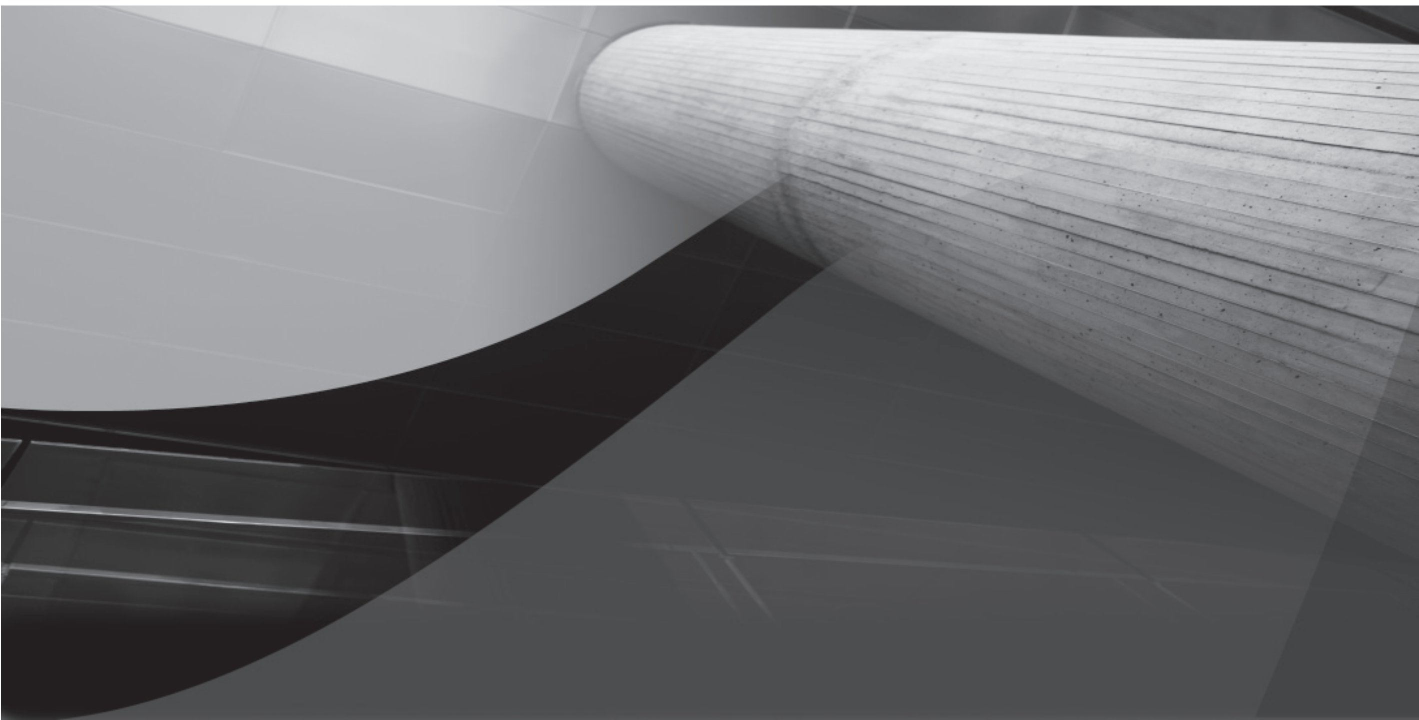


CHAPTER 1

Introduction

In this chapter, you will learn about the following:

- Relational databases.
- The Structured Query Language (SQL), which is used to access a database.
- SQL*Plus, Oracle's interactive text-based tool for running SQL statements.
- SQL Developer, which is a graphical tool for database development.
- PL/SQL, Oracle's procedural programming language. PL/SQL allows you to develop programs that are stored in the database.

Let's plunge in and consider what a relational database is.

What Is a Relational Database?

The concept of a relational database was originally developed back in 1970 by Dr. E.F. Codd. He laid down the theory of relational databases in his seminal paper entitled "A Relational Model of Data for Large Shared Data Banks," published in *Communications of the ACM* (Association for Computing Machinery), Vol. 13, No. 6, June 1970.

The basic concepts of a relational database are fairly easy to understand. A *relational database* is a collection of related information that has been organized into *tables*. Each table stores data in *rows*; the data is arranged into *columns*. The tables are stored in database *schemas*, which are areas where users may store their own tables. A user may grant *permissions* to other users so they can access their tables.

Most of us are familiar with data being stored in tables—stock prices and train timetables are sometimes organized into tables. One example table used in this book records customer information for an imaginary store; the table stores the customer first names, last names, dates of birth (dobs), and phone numbers:

```
first_name last_name  dob         phone
---------- ---------- ----------- ------------
John       Brown      01-JAN-1965 800-555-1211
Cynthia    Green      05-FEB-1968 800-555-1212
Steve      White      16-MAR-1971 800-555-1213
Gail       Black                  800-555-1214
Doreen     Blue       20-MAY-1970
```

This table could be stored in a variety of forms:

- A card in a box
- An HTML file on a web page
- A table in a database

An important point to remember is that the information that makes up a database is different from the system used to access that information. The software used to access a database is known as a *database management system*. The Oracle database is one such piece of software; other examples include SQL Server, DB2, and MySQL.

Of course, every database must have some way to get data in and out of it, preferably using a common language understood by all databases. Database management systems implement a standard language known as *Structured Query Language*, or SQL. Among other things, SQL allows you to retrieve, add, modify, and delete information in a database.

Introducing the Structured Query Language (SQL)

Structured Query Language (SQL) is the standard language designed to access relational databases. SQL should be pronounced as the letters "S-Q-L."

NOTE
"S-Q-L" is the correct way to pronounce SQL according to the American National Standards Institute. However, the single word "sequel" is frequently used instead.

SQL is based on the groundbreaking work of Dr. E.F. Codd, with the first implementation of SQL being developed by IBM in the mid-1970s. IBM was conducting a research project known as System R, and SQL was born from that project. Later, in 1979, a company then known as Relational Software Inc. (known today as Oracle Corporation) released the first commercial version of SQL. SQL is now fully standardized and recognized by the American National Standards Institute.

SQL uses a simple syntax that is easy to learn and use. You'll see some simple examples of its use in this chapter. There are five types of SQL statements, outlined in the following list:

- **Query statements** retrieve rows stored in database tables. You write a query using the SQL `SELECT` statement.
- **Data Manipulation Language (DML) statements** modify the contents of tables. There are three DML statements:
  - **`INSERT`** adds rows to a table.
  - **`UPDATE`** changes rows.
  - **`DELETE`** removes rows.
- **Data Definition Language (DDL) statements** define the data structures, such as tables, that make up a database. There are five basic types of DDL statements:
  - **`CREATE`** creates a database structure. For example, `CREATE TABLE` is used to create a table; another example is `CREATE USER`, which is used to create a database user.
  - **`ALTER`** modifies a database structure. For example, `ALTER TABLE` is used to modify a table.
  - **`DROP`** removes a database structure. For example, `DROP TABLE` is used to remove a table.
  - **`RENAME`** changes the name of a table.
  - **`TRUNCATE`** deletes all the rows from a table.

- **Transaction Control (TC) statements** either permanently record any changes made to rows, or undo those changes. There are three TC statements:
  - **`COMMIT`** permanently records changes made to rows.
  - **`ROLLBACK`** undoes changes made to rows.
  - **`SAVEPOINT`** sets a "save point" to which you can roll back changes.
- **Data Control Language (DCL) statements** change the permissions on database structures. There are two DCL statements:
  - **`GRANT`** gives another user access to your database structures.
  - **`REVOKE`** prevents another user from accessing your database structures.

There are many ways to run SQL statements and get results back from the database, some of which include programs written using Oracle Forms and Reports. SQL statements may also be embedded within programs written in other languages, such as Oracle's Pro*C++, which allows you to add SQL statements to a C++ program. You can also add SQL statements to a Java program using JDBC; for more details, see my book *Oracle9*i *JDBC Programming* (Oracle Press, 2002).

Oracle also has a tool called SQL*Plus that allows you to enter SQL statements using the keyboard or to run a script containing SQL statements. SQL*Plus enables you to conduct a "conversation" with the database; you enter SQL statements and view the results returned by the database. You'll be introduced to SQL*Plus next.

Using SQL*Plus

If you're at all familiar with the Oracle database, chances are that you're already familiar with SQL*Plus. If you're not, don't worry: you'll learn how to use SQL*Plus in this book.

In the following sections, you'll learn how to start SQL*Plus and run a query.

Starting SQL*Plus

If you're using Windows XP Professional Edition and Oracle Database 11*g*, you can start SQL*Plus by clicking START and selecting All Programs | Oracle | Application Development | SQL Plus.

Figure 1-1 shows SQL*Plus running on Windows XP. SQL*Plus asks you for a username. Figure 1-1 shows the `scott` user connecting to the database (`scott` is an example user that is contained in many Oracle databases; `scott` has a default password of `tiger`). The host string after the @ character tells SQL*Plus where the database is running. If you are running the database on your own computer, you'll typically omit the host string (that is, you enter `scott/tiger`)—doing this causes SQL*Plus to attempt to connect to a database on the same machine on which SQL*Plus is running. If the database isn't running on your machine, you should speak with your database administrator (DBA) to get the host string. If the `scott` user doesn't exist or is locked, ask your DBA for an alternative user and password (for the examples in the first part of this chapter, you can use any user; you don't absolutely have to use the `scott` user).

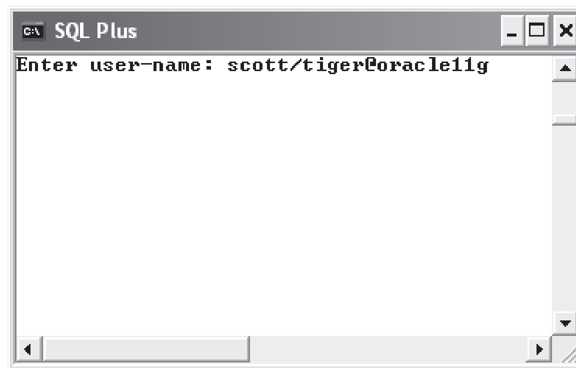


FIGURE 1-1 *Oracle Database 11*g *SQL*Plus Running on Windows XP*

If you're using Windows XP and Oracle Database 10*g* or below, you can run a special Windows-only version of SQL*Plus. You start this version of SQL*Plus by clicking Start and selecting All Programs | Oracle | Application Development | SQL Plus. The Windows-only version of SQL*Plus is deprecated in Oracle Database 11*g* (that is, it doesn't ship with 11*g*), but it will still connect to an 11*g* database. Figure 1-2 shows the Windows-only version of Oracle Database 10*g* SQL*Plus running on Windows XP.

NOTE
*The Oracle Database 11*g *version of SQL*Plus is slightly nicer than the Windows-only version. In the 11*g *version, you can scroll through previous commands you've run by pressing the* UP *and* DOWN ARROW *keys on the keyboard.*

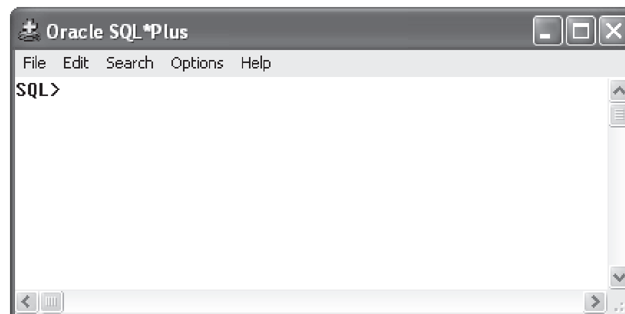


FIGURE 1-2 *Oracle Database 10*g *SQL*Plus Running on Windows XP*

Starting SQL*Plus from the Command Line

You can also start SQL*Plus from the command line. To do this, you use the `sqlplus` command. The full syntax for the `sqlplus` command is

```
sqlplus [user_name[/password[@host_string]]]
```

where

- *user_name* is the name of the database user.
- *password* is the password for the database user.
- *host_string* is the database you want to connect to.

The following examples show `sqlplus` commands:

```
sqlplus scott/tiger
sqlplus scott/tiger@orcl
```

If you're using SQL*Plus with a Windows operating system, the Oracle installer automatically adds the directory for SQL*Plus to your path. If you're using a non-Windows operating system (for example, Unix or Linux), either you must be in the same directory as the SQL*Plus program to run it or, better still, you should add the directory to your path. If you need help with that, talk to your system administrator.

For security, you can hide the password when connecting to the database. For example, you can enter

```
sqlplus scott@orcl
```

SQL*Plus then prompts you to enter the password. As you type in the password, it is hidden from prying eyes. This also works when starting SQL*Plus in Windows.

You can also just enter

```
sqlplus
```

SQL*Plus then prompts you for the user name and password. You can specify the host string by adding it to the user name (for example, `scott@orcl`).

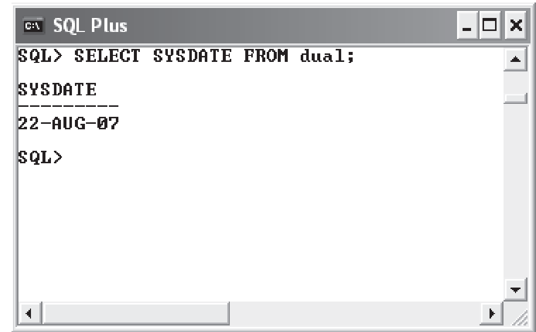
Performing a SELECT Statement Using SQL*Plus

Once you're logged onto the database using SQL*Plus, go ahead and run the following `SELECT` statement (it returns the current date):

```
SELECT SYSDATE FROM dual;
```

`SYSDATE` is a built-in database function that returns the current date, and the `dual` table is a table that contains a single row. The `dual` table is useful when you need the database to evaluate an expression (e.g., 2 * 15 / 5), or when you want to get the current date.

NOTE
*SQL statements directly entered into SQL*Plus are terminated using a semicolon character (;).*



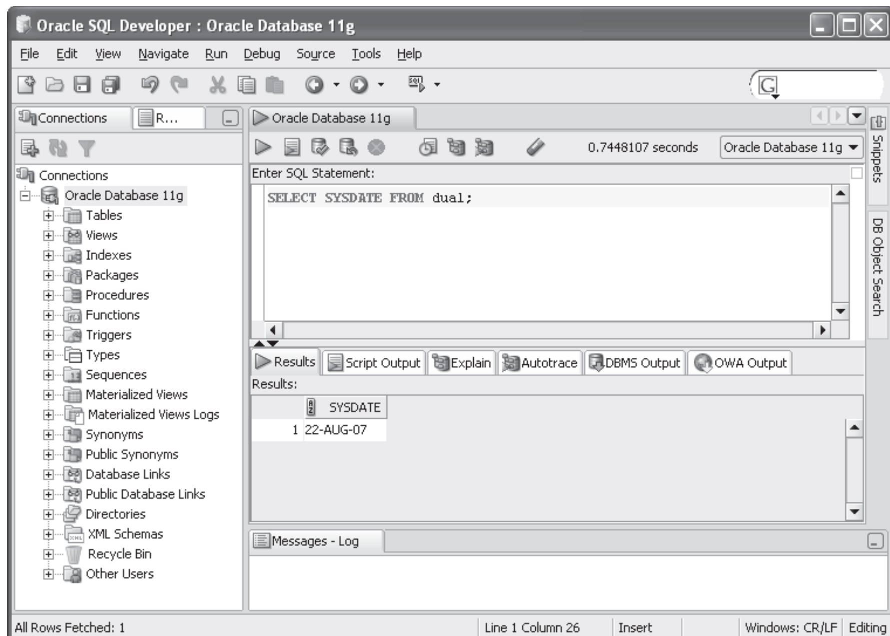
This illustration shows the results of this SELECT statement in SQL*Plus running on Windows. As you can see, the query displays the current date from the database.

You can edit your last SQL statement in SQL*Plus by entering EDIT. Doing this is useful when you make a mistake or you want to make a change to your SQL statement. On Windows, when you enter EDIT you are taken to the Notepad application; you then use Notepad to edit your SQL statement. When you exit Notepad and save your statement, the new statement is passed back to SQL*Plus, where you can re-execute it by entering a forward slash (/). On Linux or Unix, the default editor is typically set to vi or emacs.

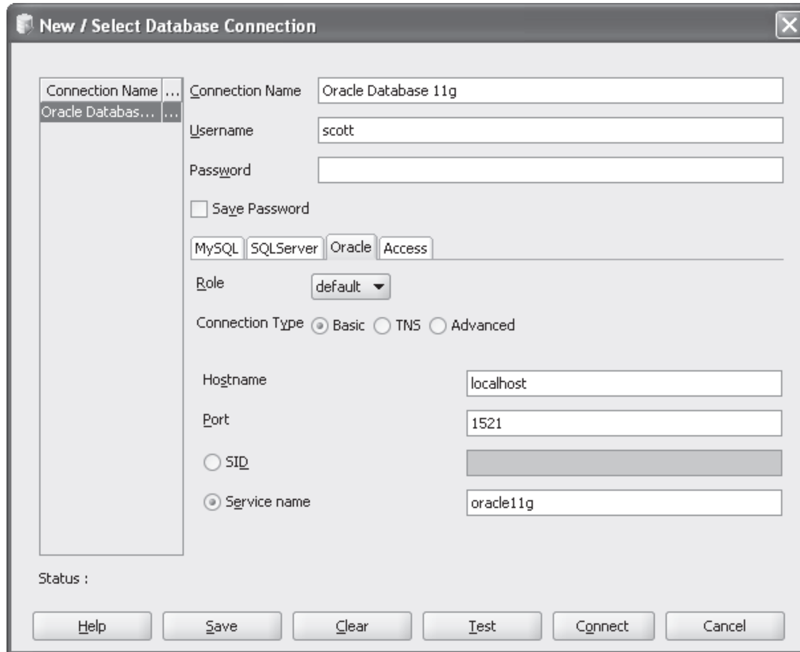
NOTE
*You'll learn more about editing SQL statements using SQL*Plus in Chapter 3.*

SQL Developer

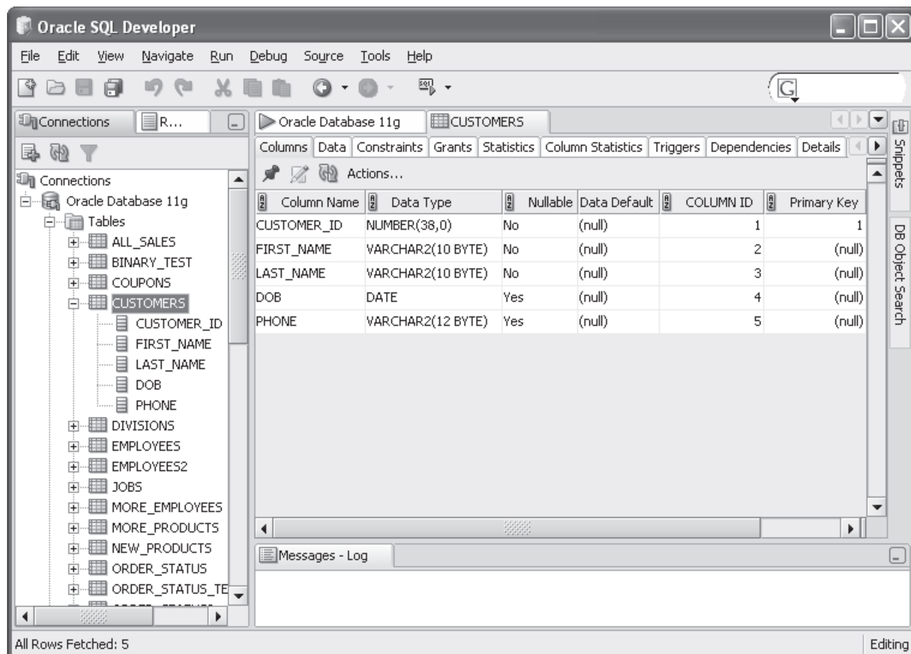
You can also enter SQL statements using SQL Developer. SQL Developer uses a very nice graphical user interface through which you can enter SQL statements, examine database tables, run scripts, edit and debug PL/SQL code, and much more. SQL Developer can connect to any Oracle Database, version 9.2.0.1 and higher, and runs on Windows, Linux, and Mac OSX. The following illustration shows SQL Developer running.



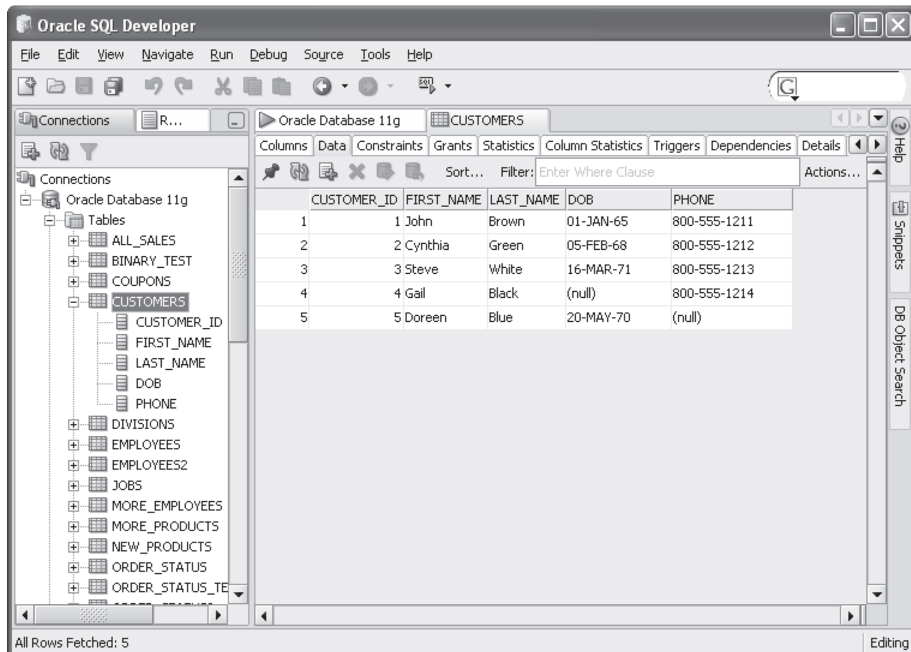
You need to have Java installed on your computer before you can run SQL Developer. If you're using Windows XP Professional Edition and Oracle Database 11*g*, you start SQL Developer by clicking Start and selecting All Programs | Oracle | Application Development | SQL Developer. SQL Developer will prompt you to select the Java executable. You then browse to the location where you have installed it and select the executable. Next, you need to create a connection by right-clicking Connections and selecting New Connection, as shown in the following illustration.



Once you've created a connection and tested it, you can use it to connect to the database and run queries, examine database tables, and so on. The following illustration shows the details for a database table named `customers`.



You can also view the data stored in a table, as shown in the following illustration.



You can see full details on using SQL Developer by selecting Help | Table of Contents from the menu bar in SQL Developer.

In the next section, you'll learn how to create the imaginary store schema used throughout this book.

Creating the Store Schema

The imaginary store sells items such as books, videos, DVDs, and CDs. The database for the store will hold information about the customers, employees, products, and sales. The SQL*Plus script to create the database is named `store_schema.sql`, which is located in the `SQL` directory where you extracted the Zip file for this book. The `store_schema.sql` script contains the DDL and DML statements used to create the `store` schema. You'll now learn how to run the `store_schema.sql` script.

Running the SQL*Plus Script to Create the Store Schema

You perform the following steps to create the `store` schema:

1. Start SQL*Plus.
2. Log into the database as a user with privileges to create new users, tables, and PL/SQL packages. I run scripts in my database using the `system` user; this user has all the required privileges. You may need to speak with your database administrator about setting up a user for you with the required privileges (they might also run the `store_schema.sql` script for you).
3. Run the `store_schema.sql` script from within SQL*Plus using the `@` command.

The `@` command has the following syntax:

```
@ directory\store_schema.sql
```

where *directory* is the directory where your `store_schema.sql` script is located.

For example, if the script is stored in `E:\sql_book\SQL`, then you enter

```
@ E:\sql_book\SQL\store_schema.sql
```

If you have placed the `store_schema.sql` script in a directory that contains spaces, then you must place the directory and script in quotes after the `@` command. For example:

```
@ "E:\Oracle SQL book\sql_book\SQL\store_schema.sql"
```

If you're using Unix or Linux and you saved the script in a directory named `SQL` in the `tmp` file system, then you enter

```
@ /tmp/SQL/store_schema.sql
```

NOTE
Windows uses backslash characters (\) in directory paths, whereas Unix and Linux use forward slash characters (/).

The first executable line in the `store_schema.sql` script attempts to drop the `store` user, generating an error because the user doesn't exist yet. Don't worry about the error: the line is there so you don't have to manually drop the `store` user when recreating the schema later in the book.

When the `store_schema.sql` script has finished running, you'll be connected as the `store` user. If you want to, open the `store_schema.sql` script using a text editor like Windows Notepad and examine the statements contained in it. Don't worry about the details of the statements contained in the script—you'll learn the details as you progress through this book.

NOTE

*To end SQL*Plus, you enter* `EXIT`*. To reconnect to the* `store` *schema in SQL*Plus, you enter* `store` *as the user name with a password of* `store_password`*. While you're connected to the database, SQL*Plus maintains a database session for you. When you disconnect from the database, your session is ended. You can disconnect from the database and keep SQL*Plus running by entering* `DISCONNECT`*. You can then reconnect to a database by entering* `CONNECT`*.*

Data Definition Language (DDL) Statements Used to Create the Store Schema

As mentioned earlier, Data Definition Language (DDL) statements are used to create users and tables, plus many other types of structures in the database. In this section, you'll see the DDL statements used to create the `store` user and some of the tables.

NOTE

The SQL statements you'll see in the rest of this chapter are the same as those contained in the `store_schema.sql` *script. You don't have to type the statements in yourself: you just run the* `store_schema.sql` *script.*

The next sections describe the following:

- How to create a database user
- The commonly used data types used in an Oracle database
- Some of the tables in the imaginary store

Creating a Database User

To create a user in the database, you use the `CREATE USER` statement. The simplified syntax for the `CREATE USER` statement is as follows:

```
CREATE USER user_name IDENTIFIED BY password;
```

where

- *`user_name`* is the user name
- *`password`* is the password for the user

For example, the following CREATE USER statement creates the store user with a password of store_password:

```
CREATE USER store IDENTIFIED BY store_password;
```

If you want the user to be able to work in the database, the user must be granted the necessary *permissions* to do that work. In the case of store, this user must be able to log onto the database (which requires the connect permission) and create items like database tables (which requires the resource permission). Permissions are granted by a privileged user (for example, the system user) using the GRANT statement.

The following example grants the connect and resource permissions to store:

```
GRANT connect, resource TO store;
```

Once a user has been created, the database tables and other database objects can be created in the associated schema for that user. Many of the examples in this book use the store schema. Before I get into the details of the store tables, you need to know about the commonly used Oracle database types.

The Common Oracle Database Types

There are many types that may be used to handle data in an Oracle database. Some of the commonly used types are shown in Table 1-1.

You can see all the data types in the appendix. The following table illustrates a few examples of how numbers of type NUMBER are stored in the database.

Format	Number Supplied	Number Stored
NUMBER	1234.567	1234.567
NUMBER(6, 2)	123.4567	123.46
NUMBER(6, 2)	12345.67	Number exceeds the specified precision and is therefore rejected by the database.

Examining the Store Tables

In this section, you'll learn how the tables for the store schema are created. Some of the information held in the store schema includes

- Customer details
- Types of products sold
- Product details
- A history of the products purchased by the customers
- Employees of the store
- Salary grades

The following tables are used to hold the information:

- **customers** holds the customer details.
- **product_types** holds the types of products sold by the store.

- **`products`** holds the product details.
- **`purchases`** holds which products were purchased by which customers.
- **`employees`** holds the employee details.
- **`salary_grades`** holds the salary grade details.

Oracle Type	Meaning
`CHAR(`*`length`*`)`	Stores strings of a fixed length. The *`length`* parameter specifies the length of the string. If a string of a smaller length is stored, it is padded with spaces at the end. For example, `CHAR(2)` may be used to store a fixed-length string of two characters; if `'C'` is stored in a `CHAR(2)`, then a single space is added at the end; `'CA'` is stored as is, with no padding.
`VARCHAR2(`*`length`*`)`	Stores strings of a variable length. The *`length`* parameter specifies the maximum length of the string. For example, `VARCHAR2(20)` may be used to store a string of up to 20 characters in length. No padding is used at the end of a smaller string.
`DATE`	Stores dates and times. The `DATE` type stores the century, all four digits of a year, the month, the day, the hour (in 24-hour format), the minute, and the second. The `DATE` type may be used to store dates and times between January 1, 4712 B.C. and December 31, 4712 A.D.
`INTEGER`	Stores integers. An integer doesn't contain a floating point: it is a whole number, such as 1, 10, and 115.
`NUMBER(` *`precision`*`,` *`scale`* `)`	Stores floating point numbers, but may also be used to store integers. The *`precision`* is the maximum number of digits (left and right of a decimal point, if used) that may be used for the number. The maximum precision supported by the Oracle database is 38. The *`scale`* is the maximum number of digits to the right of a decimal point (if used). If neither *`precision`* nor *`scale`* is specified, any number may be stored up to a precision of 38 digits. Any attempt to store a number that exceeds the *`precision`* is rejected by the database.
`BINARY_FLOAT`	Introduced in Oracle Database 10*g*, stores a single precision 32-bit floating point number. You'll learn more about `BINARY_FLOAT` later in the section "The `BINARY_FLOAT` and `BINARY_DOUBLE` Types."
`BINARY_DOUBLE`	Introduced in Oracle Database 10*g*, stores a double precision 64-bit floating point number. You'll learn more about `BINARY_DOUBLE` later in the section "The `BINARY_FLOAT` and `BINARY_DOUBLE` Types."

TABLE 1-1 *Commonly Used Oracle Data Types*

NOTE
The `store_schema.sql` *script creates other tables and database items not mentioned in the previous list. You'll learn about these items in later chapters.*

In the following sections, you'll see the details of some of the tables, and you'll see the `CREATE TABLE` statements included in the `store_schema.sql` script that create the tables.

The customers Table The `customers` table holds the details of the customers. The following items are held in this table:

- First name
- Last name
- Date of birth (dob)
- Phone number

Each of these items requires a column in the `customers` table. The `customers` table is created by the `store_schema.sql` script using the following `CREATE TABLE` statement:

```
CREATE TABLE customers (
  customer_id INTEGER CONSTRAINT customers_pk PRIMARY KEY,
  first_name VARCHAR2(10) NOT NULL,
  last_name VARCHAR2(10) NOT NULL,
  dob DATE,
  phone VARCHAR2(12)
);
```

As you can see, the `customers` table contains five columns, one for each item in the previous list, and an extra column named `customer_id`. The columns are

- **`customer_id`** Contains a unique integer for each row in the table. Each table should have one or more columns that uniquely identifies each row; the column(s) are known as the *primary key*. The `CONSTRAINT` clause indicates that the `customer_id` column is the primary key. A `CONSTRAINT` clause restricts the values stored in a column, and, for the `customer_id` column, the `PRIMARY KEY` keywords indicate that the `customer_id` column must contain a unique value for each row. You can also attach an optional name to a constraint, which must immediately follow the `CONSTRAINT` keyword—for example, `customers_pk`. You should always name your primary key constraints, so that when a constraint error occurs it is easy to spot where it happened.
- **`first_name`** Contains the first name of the customer. You'll notice the use of the `NOT NULL` constraint for this column—this means that a value must be supplied for `first_name` when adding or modifying a row. If a `NOT NULL` constraint is omitted, a user doesn't need to supply a value and the column can remain empty.

- **last_name** Contains the last name of the customer. This column is NOT NULL, and therefore a value must be supplied when adding or modifying a row.
- **dob** Contains the date of birth for the customer. Notice that no NOT NULL constraint is specified for this column; therefore, the default NULL is assumed, and a value is optional when adding or modifying a row.
- **phone** Contains the phone number of the customer. This is an optional value.

The store_schema.sql script populates the customers table with the following rows:

```
customer_id first_name last_name  dob       phone
----------- ---------- ---------- --------- ------------
          1 John       Brown      01-JAN-65 800-555-1211
          2 Cynthia    Green      05-FEB-68 800-555-1212
          3 Steve      White      16-MAR-71 800-555-1213
          4 Gail       Black                800-555-1214
          5 Doreen     Blue       20-MAY-70
```

Notice that customer #4's date of birth is null, as is customer #5's phone number.

You can see the rows in the customers table for yourself by executing the following SELECT statement using SQL*Plus:

```
SELECT * FROM customers;
```

The asterisk (*) indicates that you want to retrieve all the columns from the customers table.

NOTE
In this book, SQL statements shown in **bold** *are statements you should type in and run if you want to follow along with the examples. Non-bold statements are statements you don't need to type in.*

The product_types Table The product_types table holds the names of the product types sold by the store. This table is created by the store_schema.sql script using the following CREATE TABLE statement:

```
CREATE TABLE product_types (
  product_type_id INTEGER CONSTRAINT product_types_pk PRIMARY KEY,
  name VARCHAR2(10) NOT NULL
);
```

The product_types table contains the following two columns:

- **product_type_id** uniquely identifies each row in the table; the product_type_id column is the primary key for this table. Each row in the product_types table must have a unique integer value for the product_type_id column.
- **name** contains the product type name. It is a NOT NULL column, and therefore a value must be supplied when adding or modifying a row.

The `store_schema.sql` script populates the `product_types` table with the following rows:

```
product_type_id name
--------------- ----------
              1 Book
              2 Video
              3 DVD
              4 CD
              5 Magazine
```

The `product_types` table contains the product types for the store. Each product sold by the store must be one of these types.

You can see the rows in the `product_types` table for yourself by executing the following `SELECT` statement using SQL*Plus:

```
SELECT * FROM product_types;
```

The products Table The `products` table holds the products sold by the store. The following pieces of information are held for each product:

- Product type
- Name
- Description
- Price

The `store_schema.sql` script creates the `products` table using the following `CREATE TABLE` statement:

```
CREATE TABLE products (
  product_id INTEGER CONSTRAINT products_pk PRIMARY KEY,
  product_type_id INTEGER
    CONSTRAINT products_fk_product_types
    REFERENCES product_types(product_type_id),
  name VARCHAR2(30) NOT NULL,
  description VARCHAR2(50),
  price NUMBER(5, 2)
);
```

The columns in this table are as follows:

- **`product_id`** uniquely identifies each row in the table. This column is the primary key of the table.
- **`product_type_id`** associates each product with a product type. This column is a reference to the `product_type_id` column in the `product_types` table; it is known as a *foreign key* because it references a column in another table. The table containing the foreign key (the `products` table) is known as the *detail* or *child* table, and the table that is referenced (the `product_types` table) is known as the *master* or *parent* table.

This type of relationship is known as a *master-detail* or *parent-child* relationship. When you add a new product, you associate that product with a type by supplying a matching `product_types.product_type_id` value in the `products.product_type_id` column (you'll see an example later).

- **`name`** contains the product name, which must be specified, as the name column is `NOT NULL`.
- **`description`** contains an optional description of the product.
- **`price`** contains an optional price for a product. This column is defined as `NUMBER(5, 2)`—the precision is 5, and therefore a maximum of 5 digits may be supplied for this number. The scale is 2; therefore 2 of those maximum 5 digits may be to the right of the decimal point.

The following is a subset of the rows stored in the `products` table:

```
product_id    product_type_id name         description       price
---------- ------------------ ------------ ------------ ----------
         1                  1 Modern       A                 19.95
                              Science      description
                                           of modern
                                           science

         2                  1 Chemistry    Introduction         30
                                           to Chemistry

         3                  2 Supernova    A star            25.99
                                           explodes

         4                  2 Tank War     Action movie      13.95
                                           about a
                                           future war
```

The first row in the `products` table has a `product_type_id` of 1, which means the product is a book (this `product_type_id` matches the "book" product type in the `product_types` table). The second product is also a book, but the third and fourth products are videos (their `product_type_id` is 2, which matches the "video" product type in the `product_types` table).

You can see all the rows in the `products` table for yourself by executing the following `SELECT` statement using SQL*Plus:

```
SELECT * FROM products;
```

The purchases Table The `purchases` table holds the purchases made by a customer. For each purchase made by a customer, the following information is held:

- Product ID
- Customer ID
- Number of units of the product that were purchased by the customer

The `store_schema.sql` script uses the following `CREATE TABLE` statement to create the `purchases` table:

```
CREATE TABLE purchases (
  product_id INTEGER
    CONSTRAINT purchases_fk_products
    REFERENCES products(product_id),
  customer_id INTEGER
    CONSTRAINT purchases_fk_customers
    REFERENCES customers(customer_id),
  quantity INTEGER NOT NULL,
  CONSTRAINT purchases_pk PRIMARY KEY (product_id, customer_id)
);
```

The columns in this table are as follows:

- **`product_id`** contains the ID of the product that was purchased. This must match a `product_id` column value in the `products` table.
- **`customer_id`** contains the ID of a customer who made the purchase. This must match a `customer_id` column value in the `customers` table.
- **`quantity`** contains the number of units of the product that were purchased by the customer.

The `purchases` table has a primary key constraint named `purchases_pk` that spans two columns: `product_id` and `customer_id`. The combination of the two column values must be unique for each row. When a primary key consists of multiple columns, it is known as a *composite* primary key.

The following is a subset of the rows that are stored in the `purchases` table:

```
product_id customer_id   quantity
---------- ----------- ----------
         1           1          1
         2           1          3
         1           4          1
         2           2          1
         1           3          1
```

As you can see, the combination of the values in the `product_id` and `customer_id` columns is unique for each row.

You can see all the rows in the `purchases` table for yourself by executing the following `SELECT` statement using SQL*Plus:

```
SELECT * FROM purchases;
```

The employees Table The `employees` table holds the details of the employees. The following information is held in the table:

- Employee ID

- The ID of the employee's manager (if applicable)
- First name
- Last name
- Title
- Salary

The `store_schema.sql` script uses the following `CREATE TABLE` statement to create the `employees` table:

```
CREATE TABLE employees (
  employee_id INTEGER CONSTRAINT employees_pk PRIMARY KEY,
  manager_id INTEGER,
  first_name VARCHAR2(10) NOT NULL,
  last_name VARCHAR2(10) NOT NULL,
  title VARCHAR2(20),
  salary NUMBER(6, 0)
);
```

The `store_schema.sql` script populates the `employees` table with the following rows:

```
employee_id manager_id first_name last_name  title             salary
----------- ---------- ---------- ---------- ------------- ----------
          1            James      Smith      CEO               800000
          2          1 Ron        Johnson    Sales Manager     600000
          3          2 Fred       Hobbs      Salesperson       150000
          4          2 Susan      Jones      Salesperson       500000
```

As you can see, James Smith doesn't have a manager. That's because he is the CEO of the store.

The salary_grades Table The `salary_grades` table holds the different salary grades available to employees. The following information is held:

- Salary grade ID
- Low salary boundary for the grade
- High salary boundary for the grade

The `store_schema.sql` script uses the following `CREATE TABLE` statement to create the `salary_grades` table:

```
CREATE TABLE salary_grades (
  salary_grade_id INTEGER CONSTRAINT salary_grade_pk PRIMARY KEY,
  low_salary  NUMBER(6, 0),
  high_salary NUMBER(6, 0)
);
```

The `store_schema.sql` script populates the `salary_grades` table with the following rows:

```
salary_grade_id low_salary high_salary
--------------- ---------- -----------
              1          1      250000
              2     250001      500000
              3     500001      750000
              4     750001      999999
```

Adding, Modifying, and Removing Rows

In this section, you'll learn how to add, modify, and remove rows in database tables by using the SQL INSERT, UPDATE, and DELETE statements. You can make your row changes permanent in the database using the COMMIT statement, or you can undo them using the ROLLBACK statement. This section doesn't exhaustively cover all the details of using these statements; you'll learn more about them in Chapter 8.

Adding a Row to a Table

You use the INSERT statement to add new rows to a table. You can specify the following information in an INSERT statement:

- The table into which the row is to be inserted
- A list of columns for which you want to specify column values
- A list of values to store in the specified columns

When inserting a row, you need to supply a value for the primary key and all other columns that are defined as NOT NULL. You don't have to specify values for the other columns if you don't want to; those columns will be automatically set to null if you omit values for them.

You can tell which columns are defined as NOT NULL using the SQL*Plus DESCRIBE command. The following example DESCRIBEs the customers table:

```
SQL> DESCRIBE customers
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------
 CUSTOMER_ID                               NOT NULL NUMBER(38)
 FIRST_NAME                                NOT NULL VARCHAR2(10)
 LAST_NAME                                 NOT NULL VARCHAR2(10)
 DOB                                                DATE
 PHONE                                              VARCHAR2(12)
```

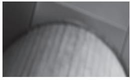
As you can see, the customer_id, first_name, and last_name columns are NOT NULL, meaning that you must supply a value for these columns. The dob and phone columns don't require a value; you could omit the values if you wanted, and they would be automatically set to null.

Go ahead and run the following INSERT statement, which adds a row to the customers table; notice that the order of values in the VALUES list matches the order in which the columns are specified in the column list:

```
SQL> INSERT INTO customers (
  2    customer_id, first_name, last_name, dob, phone
```

```
  3  ) VALUES (
  4    6, 'Fred', 'Brown', '01-JAN-1970', '800-555-1215'
  5  );

1 row created.
```



NOTE
*SQL*Plus automatically numbers lines after you hit* ENTER *at the end of each line.*

In the previous example, SQL*Plus responds that one row has been created after the `INSERT` statement is executed. You can verify this by running the following `SELECT` statement:

```
SELECT *
FROM customers;

CUSTOMER_ID FIRST_NAME LAST_NAME  DOB       PHONE
----------- ---------- ---------- --------- ------------
          1 John       Brown      01-JAN-65 800-555-1211
          2 Cynthia    Green      05-FEB-68 800-555-1212
          3 Steve      White      16-MAR-71 800-555-1213
          4 Gail       Black                800-555-1214
          5 Doreen     Blue       20-MAY-70
          6 Fred       Brown      01-JAN-70 800-555-1215
```

Notice the new row that has been added to the end of the table.

By default, the Oracle database displays dates in the format `DD-MON-YY`, where `DD` is the day number, `MON` is the first three characters of the month (in uppercase), and `YY` is the last two digits of the year. The database actually stores all four digits for the year, but by default it only displays the last two digits.

When a row is added to the `customers` table, a unique value for the `customer_id` column must be given. The Oracle database will prevent you from adding a row with a primary key value that already exists in the table; for example, the following `INSERT` statement causes an error because a row with a `customer_id` of 1 already exists:

```
SQL> INSERT INTO customers (
  2    customer_id, first_name, last_name, dob, phone
  3  ) VALUES (
  4    1, 'Lisa', 'Jones', '02-JAN-1971', '800-555-1225'
  5  );

INSERT INTO customers (
*
ERROR at line 1:
ORA-00001: unique constraint (STORE.CUSTOMERS_PK) violated
```

Notice that the name of the constraint is shown in the error (`CUSTOMERS_PK`). That's why you should always name your primary key constraints; otherwise, the Oracle database assigns an unfriendly system-generated name to a constraint (for example, `SYS_C0011277`).

Modifying an Existing Row in a Table

You use the UPDATE statement to change rows in a table. Normally, when you use the UPDATE statement, you specify the following information:

- The table containing the rows that are to be changed
- A WHERE clause that specifies the rows that are to be changed
- A list of column names, along with their new values, specified using the SET clause

You can change one or more rows using the same UPDATE statement. If more than one row is specified, the same change will be made for all the rows. The following example updates customer #2's last_name to Orange:

```
UPDATE customers
SET last_name = 'Orange'
WHERE customer_id = 2;

1 row updated.
```

SQL*Plus confirms that one row was updated.

CAUTION
If you forget to add a WHERE clause, then all the rows will be updated.

The following query confirms the update worked:

```
SELECT *
FROM customers
WHERE customer_id = 2;

CUSTOMER_ID FIRST_NAME LAST_NAME  DOB       PHONE
----------- ---------- ---------- --------- ------------
          2 Cynthia       Orange      05-FEB-68 800-555-1212
```

Removing a Row from a Table

You use the DELETE statement to remove rows from a table. You typically use a WHERE clause to limit the rows you wish to delete; if you don't, *all* the rows will be deleted from the table.

The following DELETE statement removes customer #2:

```
DELETE FROM customers
WHERE customer_id = 2;

1 row deleted.
```

To undo the changes you've made to the rows, you use ROLLBACK:

```
ROLLBACK;

Rollback complete.
```

Go ahead and run the ROLLBACK to undo any changes you've made so far. That way, your results will match those shown in subsequent chapters.

NOTE
You can make changes to rows permanent using COMMIT*. You'll see how to do that in Chapter 8.*

The BINARY_FLOAT and BINARY_DOUBLE Types

Oracle Database 10*g* introduced two new data types: BINARY_FLOAT and BINARY_DOUBLE. BINARY_FLOAT stores a single precision 32-bit floating point number; BINARY_DOUBLE stores a double precision 64-bit floating point number. These new data types are based on the IEEE (Institute of Electrical and Electronics Engineers) standard for binary floating-point arithmetic.

Benefits of BINARY_FLOAT and BINARY_DOUBLE

BINARY_FLOAT and BINARY_DOUBLE are intended to complement the existing NUMBER type. BINARY_FLOAT and BINARY_DOUBLE offer the following benefits over NUMBER:

- **Smaller storage required** BINARY_FLOAT and BINARY_DOUBLE require 5 and 9 bytes of storage space, whereas NUMBER might use up to 22 bytes.
- **Greater range of numbers represented** BINARY_FLOAT and BINARY_DOUBLE support numbers much larger and smaller than can be stored in a NUMBER.
- **Faster performance of operations** Operations involving BINARY_FLOAT and BINARY_DOUBLE are typically performed faster than NUMBER operations. This is because BINARY_FLOAT and BINARY_DOUBLE operations are typically performed in the hardware, whereas NUMBERs must first be converted using software before operations can be performed.
- **Closed operations** Arithmetic operations involving BINARY_FLOAT and BINARY_DOUBLE are closed, which means that either a number or a special value is returned. For example, if you divide a BINARY_FLOAT by another BINARY_FLOAT, a BINARY_FLOAT is returned.
- **Transparent rounding** BINARY_FLOAT and BINARY_DOUBLE use binary (base 2) to represent a number, whereas NUMBER uses decimal (base 10). The base used to represent a number affects how rounding occurs for that number. For example, a decimal floating-point number is rounded to the nearest decimal place, but a binary floating-point number is rounded to the nearest binary place.

TIP
If you are developing a system that involves a lot of numerical computations, you should use BINARY_FLOAT *and* BINARY_DOUBLE *to represent numbers. Of course, you must be using Oracle Database 10*g *or higher.*

Using BINARY_FLOAT and BINARY_DOUBLE in a Table

The following statement creates a table named binary_test that contains a BINARY_FLOAT and a BINARY_DOUBLE column:

```
CREATE TABLE binary_test (
  bin_float BINARY_FLOAT,
  bin_double BINARY_DOUBLE
);
```

NOTE
You'll find a script named oracle_10g_examples.sql *in the* SQL *directory that creates the* binary_test *table in the* store *schema. The script also performs the* INSERT *statements you'll see in this section. You can run this script if you are using Oracle Database 10*g *or higher.*

The following example adds a row to the binary_test table:

```
INSERT INTO binary_test (
  bin_float, bin_double
) VALUES (
  39.5f, 15.7d
);
```

Notice that f indicates a number is a BINARY_FLOAT, and d indicates a number is a BINARY_DOUBLE.

Special Values

You can also use the special values shown in Table 1-2 with a BINARY_FLOAT or BINARY_DOUBLE.

The following example inserts BINARY_FLOAT_INFINITY and BINARY_DOUBLE_INFINITY into the binary_test table:

```
INSERT INTO binary_test (
  bin_float, bin_double
) VALUES (
  BINARY_FLOAT_INFINITY, BINARY_DOUBLE_INFINITY
);
```

Special Value	Description
BINARY_FLOAT_NAN	Not a number (NaN) for the BINARY_FLOAT type
BINARY_FLOAT_INFINITY	Infinity (INF) for the BINARY_FLOAT type
BINARY_DOUBLE_NAN	Not a number (NaN) for the BINARY_DOUBLE type
BINARY_DOUBLE_INFINITY	Infinity (INF) for the BINARY_DOUBLE type

TABLE 1-2 *Special Values*

The following query retrieves the rows from `binary_test`:

```
SELECT *
FROM binary_test;

 BIN_FLOAT BIN_DOUBLE
---------- ----------
 3.95E+001  1.57E+001
       Inf        Inf
```

Quitting SQL*Plus

You use the `EXIT` command to quit from SQL*Plus. The following example quits SQL*Plus using the `EXIT` command:

```
EXIT
```

NOTE
*When you exit SQL*Plus in this way, it automatically performs a `COMMIT` for you. If SQL*Plus terminates abnormally—for example, if the computer on which SQL*Plus is running crashes—a `ROLLBACK` is automatically performed. You'll learn more about this in Chapter 8.*

Introducing Oracle PL/SQL

PL/SQL is Oracle's procedural language that allows you to add programming constructs around SQL statements. PL/SQL is primarily used for creating procedures and functions in a database that contain business logic. PL/SQL contains standard programming constructs such as

- Variable declarations
- Conditional logic (if-then-else, and so on)
- Loops
- Procedures and functions

The following `CREATE PROCEDURE` statement creates a procedure named `update_product_price()`. The procedure multiplies the price of a product by a factor—the product ID and the factor are passed as parameters to the procedure. If the specified product doesn't exist, the procedure takes no action; otherwise, it updates the product price.

NOTE
Don't worry about the details of the PL/SQL shown in the following listing—you'll learn all about PL/SQL in Chapter 11. I just want you to get a feel for PL/SQL at this stage.

```
CREATE PROCEDURE update_product_price (
  p_product_id IN products.product_id%TYPE,
  p_factor     IN NUMBER
) AS
  product_count INTEGER;
```

```
BEGIN
  -- count the number of products with the
  -- supplied product_id (will be 1 if the product exists)
  SELECT COUNT(*)
  INTO product_count
  FROM products
  WHERE product_id = p_product_id;

  -- if the product exists (i.e. product_count = 1) then
  -- update that product's price
  IF product_count = 1 THEN
    UPDATE products
    SET price = price * p_factor
    WHERE product_id = p_product_id;
    COMMIT;
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END update_product_price;
/
```

Exceptions are used to handle errors that occur in PL/SQL code. The `EXCEPTION` block in the previous example performs a `ROLLBACK` if an exception is thrown in the code.

Summary

In this chapter, you have learned the following:

- A relational database is a collection of related information that has been organized into structures known as tables. Each table contains rows that are further organized into columns. These tables are stored in the database in structures known as schemas, which are areas where database users may store their objects (such as tables and PL/SQL procedures).
- Structured Query Language (SQL) is the standard language designed to access relational databases.
- SQL*Plus allows you to run SQL statements and SQL*Plus commands.
- SQL Developer is a graphical tool for database development.
- How to run `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.
- PL/SQL is Oracle's procedural language that contains programming statements.

In the next chapter, you'll learn more about retrieving information from database tables.