Introduction to the Oracle WebServer™

Release 2.1

Part No. [[Part Number]]

ORACLE®

Enabling the Information Age

Introduction to the Oracle WebServer, 2.1

Part No. [[Part Number]]

Primary Authors: Martin Gruber, Kennan Rossi

Contributing Authors: Charles Prael

Contributors: Seshu Adunuthula, Mala Anand, Thomas Kyte, Shehzaad Nakhoda

Preface

Introduction to the Oracle WebServer is part of the Oracle WebServer documentation set, which contains:

- This book, *Introduction to the Oracle WebServer*
- I*ntroduction to the Web Request Broker*
- The *Oracle WebServer Installation Guide* for your platform.
- The Oracle WebServer online documentation

This book is a high-level overview of the Oracle WebServer, providing conceptual background information for the online documentation, which provides in-depth information in `html` format. To view the `html` files, you can use any standard web browser program that supports tables and client-side image maps.

Once you have installed the Oracle WebServer, you can follow the link from the product home page to the online documentation.

This Preface discusses this Guide's:

- Organization
- Typographic conventions
- Related documents

How this Guide is Organized

- Chapter 1, "Oracle WebServer Overview" briefly introduces the Oracle WebServer.

- Chapter 2, “The Web Listener” describes the capabilities of the Web Listener (HTTP daemon) component of the Oracle WebServer and explains related concepts.
- Chapter 3, “The Secure Sockets Layer” describes the Oracle WebServer implementation of the Secure Sockets Layer (SSL) and how clients can perform secure transactions with an Oracle WebServer site.
- Chapter 4, “Oracle WebServer Administration” provides a brief introduction to Oracle WebServer administration using the WebServer Manager, a collection of HTML forms and related utilities.
- Chapter 5, “Overview of the Oracle7 Server, SQL, and PL/SQL” discusses Oracle7 database concepts, and introduces SQL and PL/SQL programming.

Conventions Used in This Manual

Feature	Example	Explanation
monospace	`enum`	Identifies code elements.
boldface	**mna.h** **timeout**	Identifies file names and function arguments when used in text.
italics	*file1*	Identifies a place holder in command or function call syntax; replace this place holder with a specific value or string.
ellipses	n,...	Indicates that the preceding item can be repeated any number of times.

Table 1: Conventions

Example Conventions

This Guide shows code in this font:

```
applet.addParam("text", "This is an applet test.");
```

Related Documents

Part No.	Document Title
	Introduction to the Web Request Broker

Table 2: Related Documents

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. At the back of our printed manuals is a Reader's Comment Form, which we encourage you to use to tell us what you like and dislike about this manual or other Oracle manuals. If the form is not available, please use the following address or FAX number.

Oracle WebServer Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065
U.S.A.
FAX: 415-506-7200
email: owsdoc@us.oracle.com

Contents

1 Oracle WebServer Overview

The Oracle WebServer is a HyperText Transfer Protocol (HTTP) Internet Server with unprecedented database integration and a powerful development environment. When the WebServer receives a Uniform Resource Locator (URL) from a browser located either on the World Wide Web or on a local network using the Web's protocol (HTTP), it draws on information from the database and the operating system's (OS) file system as necessary to respond to the request. The file system can be used for static (hardcoded) Web pages, or for CGI scripts that do not access the database, and the database is used for Web pages that are generated at runtime using "live" data. Although you can run the Oracle WebServer without Oracle7, one of the great advantages of the product is its tight integration with the Oracle7 Universal Server, the leading information management product in the world.

The Web Listener is the component that receives a URL from a Web browser and sends back the appropriate output. When the Web Listener receives a URL, it determines whether the request requires the use of a Service to be accessed through the Web Request Broker (WRB), a program to be accessed through the CGI interface, or whether access to the file system of the machine on which the Listener resides is sufficient. If WRB access is required, the Listener passes the request to *WRB Dispatcher* for processing; then it returns to the task of listening for more incoming HTTP requests.

The WRB Dispatcher handles requests with the aid of a pool of processes called *WRB Executable Engines (WRBXs)*. Each WRBX interfaces to a back-end application using the WRB API. These applications are called *WRB cartridges*. The WRB API is designed so third parties can add their own cartridges. For more information on the WRB, see *Introduction to the Web Request Broker*.

CHAPTER

2 The Web Listener

The Oracle Web Listener is a HyperText Transfer Protocol (HTTP) engine that responds to requests for hypermedia documents from web browsers (clients). This section summarizes the Web Listener’s capabilities.

Network Communication

Every Oracle Web Listener process accepts connections from web browsers (clients) on one or more IP address / port combinations using HTTP to encode requests for hypertext documents, and the Transmission Control Protocol / Internet Protocol (TCP / IP) as the underlying connection protocol. Several Web Listener processes may run on single host computer at the same time (see the *Oracle Web Listener Administration Form* for more information).

Ports

A *port* is a number that TCP uses to identify a communication channel associated with a specific program. For example, the login program usually accepts login requests on port 49, and the Domain Name Service (DNS) usually accepts name lookup requests on port 53. HTTP engines, such as the Oracle Web Listener, usually accept ordinary connections on port 80 and secure connections on port 443.

Programs must execute with root permissions to access port numbers 1 through 1023, whereas any program can access port numbers 1024 through 65535.

IP Addresses

An Internet Protocol (IP) address is a unique number that identifies exactly one computer on the Internet, although each computer can be represented on the Internet by several addresses. A computer can use different addresses for different Internet functions. For example, a single computer might have one IP address on which it acts as an email routing server, and another address on which it acts as a DNS server.

DNS Host and Domain Names

A *fully qualified host name* is a unique character string that identifies exactly one computer on the Internet, although each computer can be represented on the Internet by several host names. The Domain Name Service (DNS) maintains a distributed database that maps host names to IP addresses. *DNS servers* provide Internet computers with lookup services for this database. `hal.us.oracle.com` is an example of a fully qualified host name.

A *domain* is a named DNS host name space, such as `us.oracle.com`. All host names within a domain must be unique. For example, there must be only one host named `hal` in the `us.oracle.com` domain.

Secure Connections

The Oracle Web Listener can accept secure connections using the Secure Sockets Layer (SSL), an emerging standard for secure data transmission. (The combination of IP address and port number is called a *socket*.) See *The Secure Sockets Layer* for more information on the Oracle WebServer's implementation of SSL.

Clients use a secure version of HTTP called HTTPS to establish secure connections with SSL. To establish a secure connection with a Web Listener process, a request URL must use "https" instead of "http" must specify a port on which the Web Listener has enabled SSL, for example:

```
https://www.blob.com:443/
```

Proxies

You can configure an Oracle Web Listener process to act as a *proxy server*, an HTTP engine running on a firewall machine that allows clients inside the firewall to access web sites outside the firewall. The Oracle Web Listener implements the proxy behavior defined by Luotonen and Altis in *World-Wide*

Web Proxies (http://www.w3.org:80/hypertext/WWW/Proxies/) under the auspices of the World-Wide Web Consortium (W3C).

File Handling

The Oracle Web Listener uses a *virtual file system* to keep track of the files it makes available to clients. The virtual file system maps the pathnames used in request URLs (Uniform Resource Locators) to the file system maintained by the host machine's operating system.

File Memory Mapping

The Web Listener uses the host operating system's memory mapping capability when opening a file requested by a client so that the Web Listener's virtual address space refers directly to the file's contents. This speeds access, and makes it possible for several clients to share the same copy of the file, which conserves the Web Listener's memory resources.

File Caching

Ordinarily, when the Web Listener opens a requested file, the file remains open and mapped into memory until all clients using the file are finished with it, at which point the Web Listener closes the file and frees the memory mapping associated with it. The Web Listener allows you to specify files to be *cached.* Cached files are opened when a client requests them, and remain open for the life of the Web Listener process. This optimizes access times for files, such as your home page, that are requested often.

File Protection

The Oracle Web Listener allows you to make secure specific virtual files or directories by assigning authentication and/or restriction schemes to protect them.

Authentication Schemes

When a file or directory is protected by an *authentication scheme*, a client requesting access to it must provide a user name and password. Authentication schemes allow you to define named groups of user name/password combinations, and named *realms* that are groups of these groups. You can then assign user, group, and realm names to virtual files and directories, requiring any client requesting access to input one of the specified user name/password combinations.

The Web Listener supports two authentication schemes: *basic authentication* and *digest authentication*. Both schemes are identical, except that digest authentication transmits passwords from client to server in an encrypted form called a *digest*, whereas basic authentication sends unencrypted passwords, making it considerably less secure.

Some older web browsers don't support digest authentication, but for files or directories that require authentication, you should use digest authentication whenever possible.

Restriction Schemes

When a file or directory is protected by a *restriction scheme*, only a client accessing the Web Listener from a trusted group of host machines may access it. The two restriction schemes that the Web Listener supports are *IP-based restriction* and *Domain-based restriction*. IP-based restriction allows you to define groups of trusted hosts identified by IP address, whereas Domain-based restriction allows you to define groups of trusted hosts identified by DNS host or domain name.

File Format Negotiation

The Oracle Web Listener can maintain several versions of a document in different formats, and provide it to clients in the formats they prefer.

Language Formats

For example, if a client requests a document in French, the Web Listener checks to see if it has a French version of the document, and if so, returns that version to the client. Otherwise, the Web Listener returns the version of the document in its own default language (usually English).

MIME Formats

Similarly, a client can request a document of a particular Multipurpose Internet Mail Extensions (MIME) type. If the Web Listener can find a version of the requested file in the requested MIME format, it returns that version to the client. Otherwise, it returns the version of the file that has the smallest size.

Encodings

The Web Listener can also maintain versions of a document that have been encoded by programs such as a compression utilities. For example, if a client can uncompress a document that has been compressed by the gzip program, the Web Listener can return to the client the compressed version of document instead of the uncompressed document, saving transfer time.

Filename Extensions

The Web Listener uses filename extensions to identify a file's format, which can represent language, MIME type, or encoding. For ease of maintenance, it's best to advertise a file to clients only by its base name, allowing clients and server to negotiate formats transparently.

Dynamic Document Generation

Using the Oracle Web Listener, your web site can respond to client requests by generating HTML documents dynamically. This allows you to customize your WebServer's responses.

Like most HTTP engines, the Oracle Web Listener allows clients to use the Common Gateway Interface (CGI) to run programs on the server machine to perform special processing and return data to the client.

The Web Request Broker

Unlike other HTTP engines, the Oracle Web Listener provides an interface called the Oracle Web Request Broker (WRB), which allows clients to run programs on the server machine and return data much more efficiently than CGI allows. To do this, the Web Listener passes requests intended for these programs to the WRB Dispatcher, which maintains a pool of processes to which it can assign the requests. See *Introduction to the Web Request Broker* for more information.

CHAPTER

3 The Secure Sockets Layer

The Secure Sockets Layer (SSL) is an emerging standard for secure data transmission over the Internet.

One problem with communicating sensitive information over the Internet is that almost every connection between two computers over a network involves many intermediate steps—a chain of computers that successively receive and forward the information until it reaches its destination. This process, called *routing*, is fundamental to all Internet communication, and any computer in the routing chain has complete access to all the data it receives.

This makes it easy for the unscrupulous to intercept your private conversations, steal your credit card numbers, or illegally obtain confidential or proprietary information.

The Oracle WebServer's implementation of SSL addresses this problem by scrambling data sent from the server to clients (web browser programs) in such a way that the clients can unscramble the information when they receive it. This way, any intermediate computers involved in routing the information see only gibberish that they can't decipher.

This kind of security has three aspects:

- Encryption—a mechanism for scrambling and unscrambling data.

- Authentication—a mechanism by which the one party proves its identity to another party.
- Data integrity—a mechanism for verifying that all of the data transmitted, and *only* the data transmitted, is received correctly.

Encryption

A traditional encryption system, called a *secret-key* system, uses a single large number called a *key* both to scramble (encrypt) and unscramble (decrypt) messages. Secret-key encryption systems are very fast, but they rely on one party communicating the secret key to another party, often by way of a third party such as a courier, before the two parties can exchange encrypted messages. This makes keys vulnerable to theft or tampering while in transit.

Public-Key Encryption

To avoid this problem, SSL uses a form of encryption called *public-key encryption* to encrypt and decrypt transmitted data. Unlike secret-key encryption systems, a public-key system uses pairs of keys (*key pairs*). One key, called the *public key*, is used to encrypt messages, while the other, called the *private key*, is used to decrypt messages. The two keys are large numbers that are related mathematically in such a way that it takes a very long time to calculate the private key from the public key.

If you want to receive encrypted messages using public-key encryption, you must first run a program that generates a key pair. You must then publish the public key in a public database or directory, and store the private key in a secure location on your computer. *This is critical.* The effectiveness of public-key encryption depends entirely on the secrecy of the private key.

Anyone who wants to send you an encrypted message must look up your public key in a directory, use it encrypt the message, and send you the encrypted message. Only your private key can decrypt the message, so if you have kept your private key secret, no one else can read the message.

Because public key encryption is much slower than secret-key encryption, SSL uses it only when the client first connects to the WebServer to exchange a secret key called a *session key*, which both client and server use to encrypt and decrypt transmitted data.

Authentication

Another application of encryption is authentication. Authentication using public-key encryption involves using a *digital signature*, an electronic proof of identity analogous to a handwritten signature.

If you want to “sign” an electronic document in a verifiable and legally binding way, you must first possess a key pair. You must then run a program that generates a digital signature using the private key and the document itself. You can then attach the digital signature to the document and send it. Anyone who receives this document, together with its digital signature, can then use the your public key to verify your identity, and to verify that the document has not been tampered with.

Certificates and Certifying Authorities

When clients connect to your web site for a transactions that require them to transmit sensitive information, they must be assured that they haven’t connected to an impostor pretending to be you. Clients therefore require your WebServer to authenticate itself before such transactions can proceed.

To authenticate itself, your WebServer must present the client with the proper credentials, called a *certificate*.

When you set up a secure WebServer, you must obtain a certificate from a trusted third-party company called a *certifying authority* (CA).

When you contact a certifying authority to request a certificate, you must provide them with certain legal information about your organization, which they can use to certify that your organization is legitimate and should be certified (see Setting Up a Secure Oracle WebServer in the online documentation).

CHAPTER

4 Oracle WebServer Administration

The Oracle WebServer Manager is a collection of HTML forms you can use to configure your Oracle WebServer, allowing you to:

- Start and stop Oracle7 databases.
- Configure Oracle Web Listener processes.
- Configure the PL/SQL Agent.
- Configure the Web Request Broker.

You can find the WebServer Manager pages at:

http://*yourserver*:*port*/ows-adoc/Intro.html

where *yourserver*:*port* identifies the hostname and port on which your administration server is running. You need your administration username and password to access the WebServer Manager pages.

If you get a “not found” error when you try to access this URL, your WebServer might not be installed correctly. See the *Oracle WebServer Installation Guide* for your platform for more information.

Figure 4-1illustrates the possible navigation paths through the WebServer Manager pages:

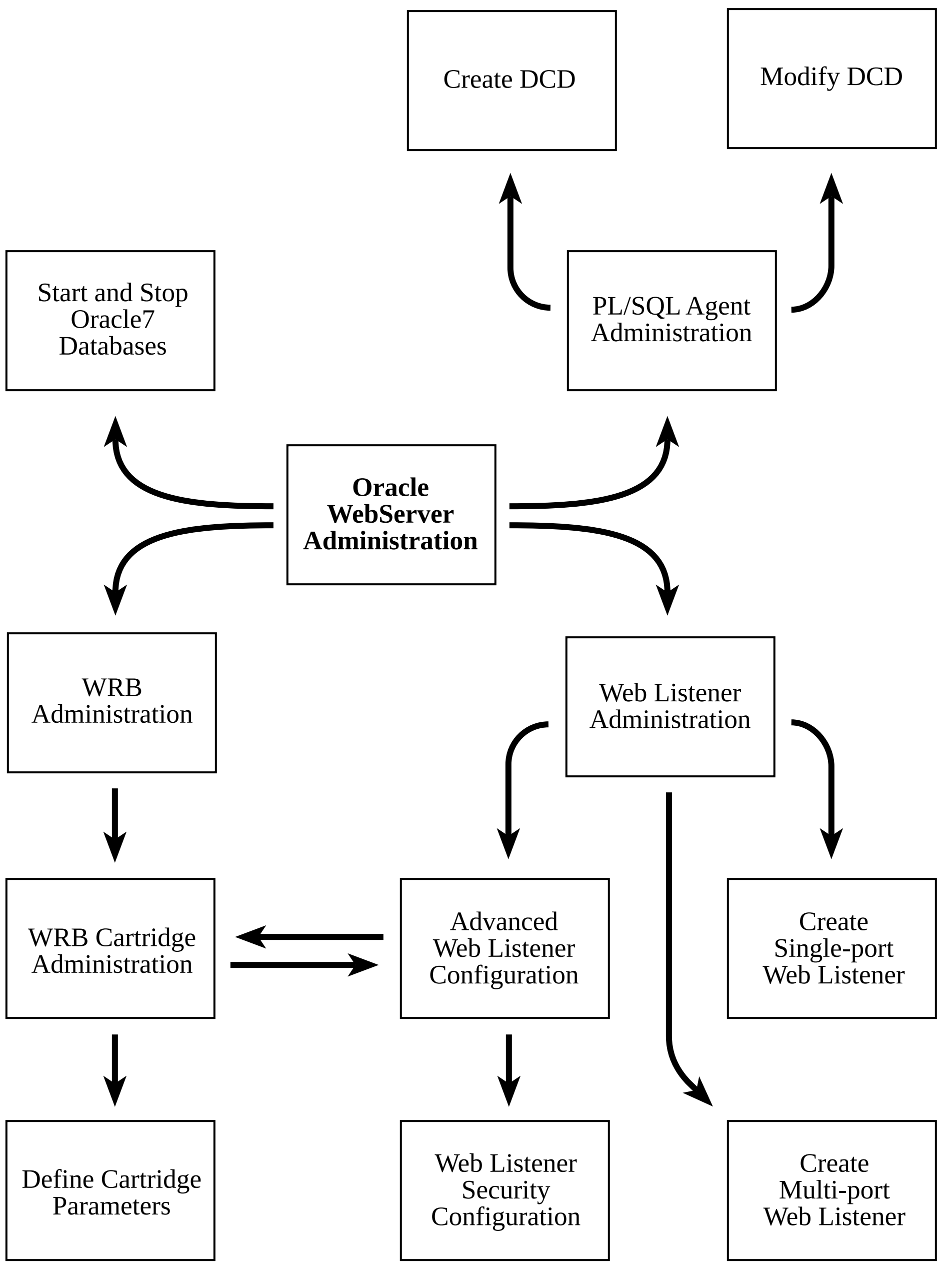
Create DCD
Modify DCD
Start and Stop Oracle7 Databases
PL/SQL Agent Administration
Oracle WebServer Administration
WRB Administration
Web Listener Administration
WRB Cartridge Administration
Advanced Web Listener Configuration
Create Single-port Web Listener
Define Cartridge Parameters
Web Listener Security Configuration
Create Multi-port Web Listener

Figure 4-1: WebServer Manager pages

Starting and Stopping Oracle7 Databases

The *Oracle WebServer Database Administration Form* provides a convenient way to start and stop Oracle7 databases that your WebServer uses. To perform other database administration tasks, see the *Oracle7 Server Administrator's Guide*.

Configuring Oracle Web Listener Processes

The Oracle Web Listener is the Oracle WebServer's HTTP engine, which handles connections from clients (web browsers). You can run several Web Listener processes at once on your WebServer host computer. Using the *Oracle Web Listener Administration Form*, you can create and configure a new Web Listener process, or choose an existing Web Listener process from a list and either modify its configuration, or delete it.

Configuring the PL/SQL Agent

The PL / SQL Agent allows the Oracle Web Listener to access an Oracle7 database in response to an HTTP request, and return the results to the requestor in HTML form. There are two versions of the PL / SQL Agent:

- The PL / SQL Agent WRB Service—uses the Web Request Broker (WRB) interface to communicate with the Web Listener.
- The PL / SQL Agent CGI Program—uses the Common Gateway Interface (CGI) to communicate with the Web Listener.

The PL / SQL Agent WRB Service is the preferred version because it takes advantage of the high performance of the Web Request Broker interface. The PL / SQL Agent CGI Program is provided only for backward compatibility to version 1.0 of the Oracle WebServer.

Configuring the Web Request Broker

The Web Request Broker provides a programming interface for writing WebServer back-end applications. You can use the Web Request Broker Administration Form to perform general Web Request Broker configuration, and to configure specific WRB cartridges.

CHAPTER

5 Overview of the Oracle7 Server, SQL, and PL/SQL

This section provides an overview of the Oracle database, and a guide to its documentation set. This subject cannot be thoroughly covered here. What is provided is a conceptual overview, and enough information on PL/SQL to enable you to write simple programs. You are referred to the Oracle7 Server documentation for further information. The topics covered in this section are as follows:

- The *Oracle7 Server*. A conceptual overview of what a relational database is.
- *SQL*. An introduction to the standard language used to interface to relational databases.
- *PL/SQL*. An introduction to Oracle's extension of SQL that makes it a versatile high-level programming language.

Oracle7 Server

The Oracle7 Server is a Relational Database Management System (RDBMS). That is to say, the job of the Oracle7 Server is to manage data. Users or other processes store, alter, and obtain the data by issuing statements in *SQL* that the RDBMS executes, but they never directly access the data. Having all the information under the control of a single entity ensures, for example, that the information maintains a coherent structure and that simultaneous changes by different users do not interfere with one another.

Database Tables

Saying that Oracle7 is a *Relational* Database Management System implies that all of the data it contains is structured as tables (tables are called “relations” in mathematical jargon).

Here is a simple table, such as you might find in an Oracle database:

CNUM	FNAME	*LNAME*	*ADDRESS*
4005	Julia	Peel	197 Myrtle Court, Brisbane, CA
4007	Terry	Subchak	2121 Oriole Way, Boston, MA
4008	Emilio	Lopez	31D San Bruno Ave. SF, CA
4011	Kerry	Lim	455 32nd St. #45, Brinton, KY

The Customers Table

Each row of this table describes one person, and each column has one type of information about that person. Note the column *cnum*. This is simply a number we generate to distinguish the customers from one another, as names are not necessarily unique. You refer to data in a relational database by its content, not by such things as where it is stored. Therefore, every table must have an identifying group of one or more columns whose values, taken as a set, are always different for every row of the table. This group, in this case the single column *cnum*, is called the *primary key* of the table. Locally generated numbers, as in this example, are a common and easy way to create primary keys.

Foreign Keys

Suppose you wanted to add our customers' phone numbers to your database. Since one person can have multiple phone numbers, these do not fit into your structure well. If you want to include the phone numbers in the Customers table, there are three possibilities, none of them good:

1. You could fit all the phone numbers for a given person into a single column in a single row, in which case it would be difficult to access the phone numbers independently.
2. You could create a column for each type of phone number, in which case you would have to redesign your table each time a new type arose. This also could create an unwieldy number of mostly empty columns.
3. You could enter a new row for each phone number, in which case each such row would consist of redundant information except for the new phone number. This approach would be error-prone and waste space.

The good solution is simply to create a second table, like this:

CNUM	PHONE	TYPE
4005	375-296-8226	home
4005	375-855-3778	beeper
4008	488-255-9011	home
4011	577-936-8554	home
4008	488-633-8591	work

Table 0 - 1 Customers_Phone

Notice that in this table *cnum* is not the primary key; it identifies the customer and therefore is the same for each phone number associated with a given customer. What, then, is the primary key? The combination of *cnum* and *phone*. If you list the same number for the same person twice, you really have made a duplicate entry and should eliminate one anyway.

The *cnum* column does have a special function, however, because it defines the relationship between Customers_Phone and Customers by associating each phone number with a customer. We say that it *references* the *cnum* column in Customers. A group of one or more columns, such as this, that references another group is known as a *foreign key*. The group of columns a foreign key references is called its *parent key* or its *referenced key*. Each foreign key value references a specific row in the table containing the parent key. Clearly, then, all sets of values in the foreign key have to be present once and only once in the parent key (although they may be present any number of times in the foreign key itself, as above) for the reference to be both meaningful and unambiguous. For that reason, the parent key must be either a primary key (the usual case) or another group of columns that is unique, which is known as a *unique key*.

Oracle can make sure that all primary and unique keys stay unique and that all foreign key references are valid; this is called maintaining *referential integrity*. For more information on foreign and parent keys, see Chapter 7 of the *Oracle7 Server Concepts Manual* and “CONSTRAINT clause” in Chapter 4 of the *Oracle7 SQL Reference*.

Users, Connections, Privileges, and Roles

The Oracle7 Server controls which users can do what to its data. To do this, it uses a log-on procedure that is separate from that of the operating system. Once logged on the operating system, you establish a *connection* to the Oracle7 Server with a username and password known to Oracle that may have no relationship to the ones used for you by the operating system. For more information, see "CREATE USER" and "CONNECT" in the *Oracle7 Server SQL Reference*.

The name under which you connect to Oracle7—your Oracle username— is associated with a number of *privileges*, which are the rights to perform various actions. For more information on privileges, see "GRANT" and "REVOKE" in Chapter 4 of the *Oracle7 Server SQL Reference*.

The PL / SQL Agent as an Oracle User

When you access the Oracle7 Server through the WebServer, you use a PL / SQL Agent that already has an established connection to the Server. When a URL causes the Web Listener to invoke the PL / SQL Agent, it associates the PL / SQL Agent with a service based on the URL and / or the domain name of the issuer of the URL. This service determines the Oracle user that the PL / SQL Agent behaves as when executing this request, and thereby controls what that request can do.

SQL

SQL (Structured Query Language) is the language you use to issue instructions to the *Oracle7 Server*. It is, in fact, the standard language used by all major relational database vendors, and Oracle complies at Entry Level with SQL92, the most recent ISO (International Standards Organization) standard. There are several aspects of SQL that may differ from computer languages that you are familiar with, such as the following:

- SQL is non-procedural. In SQL, you tell the Server what to do but not how it is to be done. This frees you from dealing with a lot of detail.
- SQL statements are independent of one another. Although *PL/SQL* addresses this, SQL itself has no conditional or other control-flow statements.
- SQL employs set-at-a-time operation. It operates on arbitrarily large sets of data in a single step.
- SQL uses *Nulls and Three-Valued Logic*. In most languages, Boolean expressions are either TRUE or FALSE. In SQL, they are TRUE, FALSE, or NULL. This will be explained shortly.

Retrieving Data

Suppose you wanted to pull from the Customers table the information on customers named “Peel”. This is called making a *query*. To do it, you could issue the following statement:

```
SELECT *
     FROM Customers
     WHERE LNAME = 'Peel';
```

This produces the following:

```
CNUM      FNAME            LNAME     ADDRESS


4005      Julia            Peel      197 Myrtle Court, Brisbane, CA
```

Oracle interprets the statement as follows: Any number of spaces and / or line breaks are equivalent to one space or line break. These are delimiters, and the extra spaces and line breaks are for readability: all are equivalent “white space”. Likewise, case is not significant, except in literals like the string you are searching for ('Peel').

SELECT is a keyword telling the database that this is a query. All SQL statements begin with keywords. The asterisk means to retrieve all columns; alternatively, you could have listed the desired columns by name, separated by commas. The FROM Customers clause identifies the table from which you want to draw the data.

WHERE LNAME = ‘Peel’ is a predicate. When a SQL statement contains a predicate, Oracle tests the predicate against each row of the table and performs the action (in this case, SELECT) on all rows that make the predicate TRUE. This is an example of set-at-a-time operation. The predicate is optional, but in its absence the operation is performed on the entire table, so that, in this case, the entire table would have been retrieved. The semi-colon is the statement terminator.

Nulls and Three-Valued Logic

With predicates, you should be aware of three-valued logic. In SQL, the basic Boolean values of TRUE and FALSE are supplemented with another: NULL, also called UNKNOWN. This is because SQL acknowledges that data can be incomplete or inapplicable and that the truth value of a predicate may therefore not be knowable. Specifically, a column can contain a *null*, which means that there is no known applicable value. A comparison between two values using relational operators—for example, a = 5—normally is either TRUE or FALSE.

Whenever nulls are compared to other values, however, including other nulls, the Boolean value is neither TRUE nor FALSE but itself NULL.

In most respects, NULL has the same effect as FALSE. The major exception is that, while NOT FALSE = TRUE, NOT NULL = NULL. In other words, if you know that an expression is FALSE, and you negate (take the opposite of) it, then you know that it is TRUE. If you do not know whether it is TRUE or FALSE, and you negate it, you still do not know. In certain cases, three-valued logic can create problems with your programming logic if you have not accounted for it. You can treat nulls specially in SQL with the IS NULL predicate, as explained in Chapter 3 of the *Oracle7 Server SQL Reference*.

Creating Tables

This is how you create tables in SQL. You can use the following SQL statement to create the Customers table:

```
CREATE TABLE Customers
(cnum          integer NOT NULL PRIMARY KEY,
FNAME          char(15) NOT NULL,
LNAME          char(15) NOT NULL,
ADDRESS         varchar2 );
```

After the keywords CREATE TABLE come the table's name and a parenthesized list of its columns with a definition of each. Integer, char, and varchar2 are datatypes: all of the data in a given column is always of the same type (char means a fixed and varchar2 a varying length string). For more information on SQL datatypes, see Chapter 2 of the *Oracle7 Server SQL Reference*.

NOT NULL and PRIMARY KEY are constraints on the columns they follow. They restrict the values you can enter in those columns. Specifically, NOT NULL forbids you from entering nulls in the column. PRIMARY KEY prevents you from entering duplicate values into the column and makes the column eligible to be the parent for some foreign key. For more information, see "CREATE TABLE" and "CONSTRAINT clause" in Chapter 4 of the *Oracle7 Server SQL Reference*.

Ownership and Naming Conventions

Note that when you create a table in SQL, you own it. This means you generally have control over who has access to it, and that it is part of a schema that bears your Oracle username. A *schema* is a named collection of database objects under the control of a single Oracle user. Schemas inherit the names of their owners. When other users refer to an object you have created, they have to precede its name by the schema name followed by a dot (no spaces). SQL utilizes a hierarchical naming convention with the levels of the hierarchy separated by dots. In fact, you sometimes have to precede column names by table names to avoid ambiguity, in which case you also use a dot. The following is an example in the form *schemaname.tablename.columnname:*

```
scott.Customers.LNAME
```

You can simplify references like this by using *synonyms*, which are aliases for tables or other database objects. Synonyms can be *private*, meaning that they are part of your schema and you control their usage, or *public*, meaning that all users can access them. For example, you can create a synonym "Cust" for scott.Customers as follows:

```
CREATE SYNONYM Cust FOR scott.Customers;
```

This would be a private synonym, which is the default. Now you could rewrite the example above like this:

```
Cust.LNAME
```

You still have to refer to the column directly. Synonyms can only be for tables, not table components like columns.

For more information on synonyms, see "CREATE SYNONYM" in Chapter 4 of the *Oracle7 Server SQL Reference*. For more information on SQL naming conventions, see Chapter 2 of the *Oracle7 Server SQL Reference*. For more on schemas, see "CREATE SCHEMA" in Chapter 4 of the *Oracle7 Server SQL Reference*.

Inserting and Manipulating the Data

Which SQL statements determine the actual data content? Chiefly, three —the INSERT statement, the UPDATE statement, and the DELETE statement. INSERT places rows in a table, UPDATE changes the values they contain, and DELETE removes them.

The INSERT Statement

For INSERT, you simply identify the table and its columns and list the values, as follows:

```
INSERT INTO Customers (cnum, FNAME, LNAME)
     VALUES (2004, 'Harry', 'Brighton');
```

This statement inserts a row with a value for every column but ADDRESS. Since you did not, in your CREATE TABLE statement, place a NOT NULL constraint on the ADDRESS column, and since you did not give that column a value here, Oracle sets this column to *null*. If you are inserting a value into every column of the table, and you have the values ordered as the columns are in the table, you can omit the column list. You optionally can put a SELECT statement in place of the VALUES clause of the INSERT statement to retrieve data from elsewhere in the database and duplicate it here. For more information on the INSERT and the SELECT statements, see “INSERT” and “SELECT,” respectively, in Chapter 4 of the *Oracle7 Server SQL Reference*.

The UPDATE Statement

UPDATE is similar to SELECT in that it takes a predicate and operates on all rows that make the predicate TRUE. For example:

```
UPDATE Customers
     SET ADDRESS = null
     WHERE LNAME = 'Subchak';
```

This sets to null all addresses for customers named ‘Subchak’. The SET clause of an UPDATE command can refer to current column values. “Current” in this case means the values in the column before any changes were made by this statement. For more information on the UPDATE statement, see “UPDATE” in Chapter 4 of the *Oracle7 Server SQL Reference*.

The DELETE Statement

DELETE is quite similar to UPDATE. The following statement deletes all rows for customers named ‘Subchak’:

```
DELETE FROM Customers
     WHERE LNAME = 'Subchak';
```

You can only delete entire rows, not individual values. To do the latter, use UPDATE to set the values to null. Be careful with DELETE that you do not omit the predicate; this empties the table. For more information on DELETE, see “DELETE” in Chapter 4 of the *Oracle7 Server SQL Reference*.

Querying Multiple Tables Through Joins

Even though it only retrieves data, SELECT is the most complex statement in SQL. One reason for this is that you can use it to query any number of tables in one statement, correlating the data in various ways. One way to do this is with a *join*, which is a SELECT statement that correlates data from more than one table. A join finds every possible combination of rows, such that one row is taken from each table joined. This means that three tables of ten rows each can produce a thousand rows of output (10 * 10 * 10) when joined. Typically, you use the predicate to filter the output in terms of some relationship. The most common type of join, called a *natural join*, filters the output in terms of the foreign key / parent key relationship explained earlier in this appendix. For example, to see the people in the Customers table coupled with their various phone numbers from the Customers_Phone table, you could enter the following:

```
SELECT a.CNUM, LNAME, FNAME, PHONE, TYPE
     FROM Customers a, Customer_Phone b
     WHERE a.CNUM = b.CNUM;
```

In the above, a and b are *range variables*, also called *correlation variables*. They are simply alternate names for the tables whose names they follow in the FROM clause, so that a = Customers and b = Customers_Phone. You can see that here you need the range variables to distinguish Customers.CNUM from Customers_Phone.CNUM in the SELECT and WHERE clauses. Even when not needed, range variables are often convenient.

Here is the output of the natural join:

```
CNUM     LNAME          FNAME     PHONE            TYPE


4005     Peel           Julia     375-296-8226     home
4005     Peel           Julia     375-855-3778     beeper
4008     Lopez          Emilio    488-255-9011     home
4008     Lopez          Emilio    488-633-8591     work
4011     Lim            Kerry     577-936-8554     home
```

This output represents every combination of rows from the two tables where both rows have the same CNUM value.

Outer Joins

Notice in the preceding example that people from the Customers table who did not have phones (namely, CNUM 4007) were not selected. If a row has no match in the other table, the predicate is never true for that row. Sometimes, you do not want this effect, and you can override it by using an *outer join*. An outer join is a join that includes all of the rows from one of the tables joined, regardless of whether there were matches in the other table. Such a join inserts nulls in the output in whichever columns were taken from the table that failed to provide matches for the outer-joined table. Here is the same query done as an outer join:

```
SELECT a.CNUM, LNAME, FNAME, PHONE, TYPE
     FROM Customers a, Customer_Phone b
      WHERE a.CNUM = b.CNUM (+);
```

This is the output of the above:

```
CNUM          LNAME          FNAME     PHONE          TYPE


4005          Peel           Julia     375-296-8226   home
4005          Peel           Julia     375-855-3778   beeper
```

```
4007          Subchak       Terry     NULL           NULL
4008          Lopez         Emilio    488-255-9011   home
4008          Lopez         Emilio    488-633-8591   work
4011          Lim           Kerry     577-936-8554   home
```

Notice that the only difference in the query is the addition of (+) to the WHERE clause. This follows the table for which nulls are to be inserted. The output from the query, then, includes at least one row for each row of the table that *did not* have (+) appended in the predicate.

You can also use SELECT statements to produce values for processing within queries (these are called subqueries), and you can perform standard set operations (UNION, INTERSECTION) on SELECT statement output. For more information on the SELECT statement, subqueries, and joins, see “SELECT” in Chapter 4 of the *Oracle7 Server SQL Reference*.

Where to Look for More Information

Oracle7 SQL is a very complex subject, and we have been able only to scratch the surface of it here. To make it easier for you to find the specific information you need to perform the task at hand, we provide the following table, which identifies where in the Oracle7 Server documentation set you can find information on specific SQL topics. Unless otherwise noted, find the headings in Chapter 4 of the *Oracle7 Server SQL Reference*.

To Find Out About	**Look Under**
aggregate data (totals, counts, averages, and so on)	SQL Functions in Chapter 3 of the *Oracle7 Server SQL Reference*.
changing user passwords	ALTER USER
connecting to the database	CONNECT
constraints	CONSTRAINT clause; CREATE TABLE; ENABLE clause
controlling user access to objects and user actions	GRANT; REVOKE; CREATE ROLE; SET ROLE; see also Chapters 17 and 18 in the *Oracle7 Server Concepts Manual*
creating databases	CREATE DATABASE
creating users	CREATE USER
functions that change simple values	SQL Functions in Chapter 3 of the *Oracle7 Server SQL Reference*.
linking databases at different locations	CREATE DATABASE LINK; see also “Distributed Databases” in the *Oracle7 Server Concepts Manual*.
making changes to the data permanent	COMMIT; SET TRANSACTION; SAVEPOINT
making SQL statements execute more quickly	CREATE INDEX; see also “Indexes” in the *Oracle7 Server Concepts Manual*.
monitoring database usage	AUDIT
reversing (undoing) changes to the data	ROLLBACK; SET TRANSACTION; SAVEPOINT

PL/SQL

PL/SQL is an application-development language that is a superset of *SQL*, supplementing it with standard programming-language features that include the following:

- block (modular) structure

- flow-control statements and loops
- variables, constants, and types
- structured data
- customized error handling

Another feature of PL / SQL is that it allows you to store compiled code directly in the database. This enables any number of applications or users to share the same functions and procedures. In fact, once a given block of code is loaded into memory, any number of users can use the same copy of it simultaneously (although behavior is as though each user had her own copy), which is useful for the Oracle WebServer. PL / SQL also enables you to define triggers, which are subprograms that the database executes automatically in response to specified events.

Unlike SQL, PL / SQL is not an industry standard, but is an exclusive product of Oracle Corporation.

The remainder of this section covers the following PL / SQL topics:

- *Basic Structure and Syntax*
- *The DECLARE Section*
- *The EXECUTABLE Section*
- *The EXCEPTION Section*
- *Storing Procedures and Functions in the Database*
- *Database Triggers*

Note: For the sake of efficiency, PL / SQL code is compiled prior to runtime. It cannot refer at compile time to objects that do not yet exist, and, for that reason, the one part of SQL that PL / SQL does not include is DDL (Data Definition Language)—the statements, such as CREATE TABLE, that create the database and the objects it contains. However, you can work around this by using the package DBMS_SQL, included with the server, to generate the DDL code itself dynamically at runtime. For more information, see “Using DDL and Dynamic SQL” in the *PL/SQL User’s Guide and Reference*.

Basic Structure and Syntax

PL / SQL, like many programming languages, groups statements into units called *blocks*. These can either be named, in which case they are called *subprograms*, or unnamed, in which case they are *anonymous blocks*. Subprograms can be either functions or procedures. The difference between these, as in most languages, is that a function is used in an expression and returns a value to that expression, while a procedure is invoked as a standalone statement and passes values to the calling program only through parameters. Subprograms can be nested within one another and can be grouped in larger units called *packages*.

A block has three parts:

- *The DECLARE Section*. This is where you define local variables, constants, types, exceptions, and nested subprograms. PL / SQL has a forward declaration, but you can use it only for subprograms. Therefore, you must define all variables, constants, and types before referencing them. For more information on forward declarations, see “Declaring Subprograms” in the *PL/SQL User’s Guide and Reference*.
- *The EXECUTABLE Section*. This is the actual code that the block executes. This is the only part of the block that must always be present.
- *The EXCEPTION Section*. This is a section for handling runtime errors and warnings.

These divisions are explained further in the sections that follow.

The DECLARE Section

The DECLARE section begins with the keyword DECLARE and ends when the keyword BEGIN signals the arrival of the EXECUTABLE section. You can declare types, constants, variables, exceptions, and cursors in any order, as long as they are declared before they are referenced in another definition. You declare subprograms last. A semi-colon terminates each definition.

Datatypes

PL / SQL provides a number of predefined datatypes for variables and constants. It also enables you to define your own types, which are subtypes of the predefined types. The types fall into the following three categories:

- Scalar. These include all string, number, and binary types. All of the SQL datatypes, which are the datatypes that you can store in the database, fall into this category. To find out about these datatypes, see “Datatypes” in the *Oracle7 Server SQL Reference.*

- Composite. These are structured datatypes, which is to say data structures that have components you can address independently. The PL / SQL composite types are TABLE (which is distinct from both database and HTML tables) and RECORD. These types are explained later in this appendix.
- Reference. There is one kind of reference datatype—REF CURSOR—which is a pointer to a cursor. Cursors are explained later in this appendix. For more information on the REF CURSOR datatype, see “Using Cursor Variables” in the *PL/SQL User’s Guide and Reference.*

For a list and explanation of all PL / SQL datatypes, see “Datatypes” in the *PL/ SQL User’s Guide and Reference.*

In many cases, you can convert from one datatype to another, either explicitly or automatically. The possible conversions and the procedure involved are explained in the *PL/SQL User’s Guide and Reference* under “Datatype Conversion”.

You can also define a variable so that it inherits its datatype from a database column or from another variable or constant, as explained in the next section.

Declaring Variables

For variables, provide the name, datatype, and any desired attributes, as follows:

```
cnum INTEGER(5) NOT NULL;
```

This declares a five-digit integer called *cnum* that will not accept nulls. The use of case above serves to distinguish keywords from identifiers; PL / SQL is not case-sensitive. NOT NULL is the only SQL constraint that you can use as a PL / SQL attribute.

Note: PL / SQL initializes all variables to null. Therefore, a NOT NULL variable, such as the above, produces an error if referenced before it is assigned a value.

Optionally, you can assign an initial value to the variable when you declare it by following the datatype specification with an assignment, as follows:

```
cnum INTEGER(5) := 254;
```

This sets *cnum* to the initial value of 254. Alternatively, you can use the keyword DEFAULT in place of the assignment operator := to achieve the same effect. For more information on setting defaults, see “Declarations” in the *PL/SQL User’s Guide and Reference.*

Inheriting Datatypes

To have the variable inherit the datatype of a database column or of another variable, use the %TYPE attribute in place of a declared datatype, as follows:

```
snum cnum%TYPE;
```

This means that *snum* inherits the datatype of *cnum*. You can inherit datatypes from database columns in the same way, by using the notation *tablename.columnname* in place of the variable name. Normally, you do this if the variable in question is to place values in or retrieve them from the column. The advantages are that you need not know the exact datatype the column uses and that you need not change your code if the datatype of that column changes. If you do not own the table containing the column, precede the *tablename* with the *schemaname*, as described under *Ownership and Naming Conventions*. For more information on %TYPE assignments, see “Declarations” in the *PL/SQL User's Guide and Reference.*

Declaring Constants

You declare constants the same way as variables, except for the addition of the keyword CONSTANT and the mandatory assignment of a value. Constants do not take attributes other than the value. An example follows:

```
interest CONSTANT REAL(5,2) := 759.32;
```

Defining Types

User-defined types in PL / SQL are subtypes of existing datatypes. They provide you with the ability to rename types and to constrain them by specifying for your subtype lengths, maximum lengths, scales, or precisions, as appropriate to the standard datatype on which the subtype is based. For more information on the datatype parameters, see “Datatypes” in Chapter 2 of the *Oracle7 Server SQL Reference*. For more information on PL / SQL datatypes, see “Datatypes” in the *PL/SQL User's Guide and Reference*. You can also use the %TYPE attribute in defining a subtype. Here is an example:

```
SUBTYPE shortnum IS INTEGER(3);
```

This defines SHORTNUM as a 3-digit version of INTEGER. For more information see “User-Defined Subtypes” in the *PL/SQL User's Guide and Reference.*

Scope and Visibility

Nested subprograms, defined in the DECLARE section, can be called from either of the other sections, but only from within the same block where they are defined or within blocks contained in that block. Variables, constants, types, and subprograms defined within a block are local to that block, and their definitions are not meaningful outside of it. Objects that are local to a block may be used by subprograms contained at any level of nesting in that same block. Such objects are global to the block that calls them.

The area of a program within which an object can be used is called the object's scope. An object's scope is distinct from its visibility. The former is the area of the program that can reference the object; the latter is the, generally smaller, portion that can reference it without qualification.

Qualification is used to override the default resolution of ambiguous references. An ambiguous reference can arise because objects or subprograms contained in different blocks can have the same names, even if they have overlapping scopes. When this happens, the reference by default means the object most local in scope—in other words, the first one PL/SQL finds by starting in the current block and working out to the enclosing ones. Qualification is the method used to override this. It is similar to the system of qualification used for database objects, as explained under *Ownership and Naming Conventions*. To qualify an object's name, precede it with the name of the subprogram where it is declared, followed by a dot, as follows:

```
relocate.transmit(245, destination);
```

This invokes a procedure called *transmit* declared in some subprogram called *relocate*. The subprogram *relocate* must be global to the block from which it is called.

Data Structures

PL/SQL provides two structured datatypes: TABLE and RECORD. It also provides a data structure called a cursor that holds the results of queries. Cursors are different from the other two in that you declare variables and constants to be of type TABLE or RECORD just as you would any other datatype. Cursors, on the other hand, have their own syntax and their own operations. Explanations of these types follow:

PL/SQL Tables

These are somewhat similar to database tables, except that they always consist of two columns: a column of values and a primary key. This also makes them similar to one-dimensional arrays, with the primary key functioning as the array index. Like SQL tables, PL/SQL tables have no fixed allocation of rows, but grow dynamically. One of their main uses is to enable you to pass entire columns of values as parameters to subprograms. With a set of such parameters, you can pass an entire table. The primary key is always of type BINARY_INTEGER, and the values can be of any scalar type.

You declare objects of type TABLE in two stages:

1. You declare a subtype using the following syntax:

   ```
   TYPE type_name IS TABLE OF
        datatype_spec
        [ NOT NULL ]
        INDEX BY BINARY INTEGER;
   ```

   Where datatype_spec means the following:

   ```
   datatype | variablename%TYPE | tablename.columname%TYPE
   ```

   In other words, you can either specify the type of values directly or use the %TYPE attribute (explained under *Declaring Variables*) to inherit the datatype from an existing variable or database column.

2. You assign objects to this subtype in the usual way. You cannot assign initial values to tables, so the first reference to the table in the EXECUTABLE section must provide it at least one value.

When you reference PL/SQL tables, you use an array-like syntax of the form:

```
column_value(primary_key_value)
```

In other words, the third row (value) of a table called “Employees” would be referenced as follows:

```
Employees(3)
```

You can use these as ordinary expressions. For example, to assign a value to a table row, use the following syntax:

```
Employees(3) := 'Marsha';
```

For more information, see “PL/SQL Tables” in the *PL/SQL User’s Guide and Reference*.

Records

As in many languages, these are data structures that contain one or more fields. Each record of a given type contains the same group of fields with different values. Each field has a datatype, which can be RECORD. In other words, you can nest records, creating data structures of arbitrary complexity. As with tables, you declare records by first declaring a subtype, using the following syntax:

```
TYPE record_type IS RECORD
     (fieldname datatype[, fieldname datatype]...);
```

The second line of the above indicates a parenthesized, comma-separated, list of fieldnames followed by datatype specifications. The datatype specifications can be direct or be inherited using the %TYPE attribute, as shown for TABLE and as explained under *Declaring Variables*.

You can also define a record type that automatically mirrors the structure of a database table or of a cursor, so that each record of the type corresponds to a row, and each field in the record corresponds to a column. To do this, use the %ROWTYPE attribute with a table or cursor name in the same way you would the %TYPE attribute with a variable,or column. The fields of the record inherit the column names and datatypes from the cursor or table. For more information, see “Records” and “%ROWTYPE Attribute” in the *PL/SQL User’s Guide and Reference*.

Cursors

A cursor is a data structure that holds the results of a query (a SELECT statement) for processing by other statements. Since the output of any query has the structure of a table, you can think of a cursor as a temporary table whose content is the output of the query.

When you declare a cursor, you associate it with the desired query. When you want to use that cursor, you open it, executing the associated query and filling the cursor with its results. You then fetch each row of the query’s output in turn for processing by other statements in the program. You can also use a cursor to update a table’s contents. To do this, use a FOR UPDATE clause to lock the rows in the table. See “Using FOR UPDATE” in the *PL/SQL User’s Guide and Reference* for more information. Sometimes, you may need to use cursor variables, which are not associated with a query until runtime. This is a form of dynamic SQL.

For more information on cursor variables, see “Using Dynamic SQL” in the *Oracle7 Server Application Developers Guide* and “Cursor Variables” in the *PL/SQL User’s Guide and Reference.*

For more information on cursors in general, see “Cursors” in the *PL/SQL User’s Guide and Reference*. See also “DECLARE CURSOR,” “OPEN”, and “FETCH” in the *Oracle7 Server SQL Reference.*

You can simplify some cursor operations by using cursor FOR loops. For more information on these, see “Using Cursor FOR Loops” in the *PL/SQL User’s Guide and Reference.*

Exceptions

You also use the DECLARE section to define your own error conditions, called “exceptions”. Explanation of this is deferred until the “EXCEPTION Section” portion of this appendix.

Declaring Subprograms

You must place all subprogram declarations at the end of the declare section, following all variable, constant, type, and exception declarations for the block. The syntax is as follows:

```
PROCEDURE procedure_name (param_name datatype, param_name datatype...)
     IS
     {local declarations}
     BEGIN {executable code}
     EXCEPTION
     END;
```

Note: For subprograms, the keyword DECLARE is omitted before the local declarations. Place local declarations before the keyword BEGIN, as shown.

The names you give the parameters in the declaration are the names that the procedure itself uses to refer to them. These are called the *formal parameters*. When the procedure is invoked, different variables or constants may be used to pass values to or from the formal parameters; these are called the *actual parameters*.

When calling the procedure, you can use each parameter for input of a value to the procedure, output of a value from it, or both. These correspond to the three *parameter modes*: IN, OUT, and IN / OUT. For more information, see “Parameter Modes” in the *PL/SQL User’s Guide and Reference*.

When you call the procedure, you can match the actual to the formal parameters either implicitly, by passing them in the same order they are given in the declaration, or explicitly, by naming the formal followed by the actual parameter as shown:

```
transmit(destination => address);
```

This invokes a procedure called *transmit*, assigning the value of *address* as the actual parameter for the formal parameter *destination*. This implies that the parameter *destination* is used within the transmit procedure and that the parameter *address* is used outside of it. Usually, it is good programming practice to use different names for matching formal and actual parameters. For more information on this, see “Positional and Named Notation” in the *PL/SQL User’s Guide and Reference*.

Functions are the same, except for the addition of a return value, specified as follows:

```
FUNCTION function_name (param_name, param_name datatype...)
     RETURN datatype IS
     {local declarations}
     BEGIN {executable code}
     EXCEPTION {local exception handlers}
     END;
```

Again, line breaks are only for readability. A RETURN statement in the executable section actually determines what the return value is. This consists of the keyword RETURN followed by an expression. When the function executes the RETURN statement, it terminates and passes the value of that expression to whichever statement called it in the containing block.

You can also use the RETURN statement without an expression in a procedure to force the procedure to exit.

For more information on procedures and functions, see “Declaring Subprograms” in the *PL/SQL User’s Guide and Reference*.

The EXECUTABLE Section

The executable section is the main body of code. It consists primarily of SQL statements, flow control statements, and assignments. SQL statements are explained earlier in this appendix; assignments and flow-control statements are explained in the sections that follow.

Assignments

The assignment operator is :=. For example, the following statement assigns the value 45 to the variable a:

```
a := 45;
```

Character strings should be set off with single quotes (’) as in all expressions. An example follows:

```
FNAME := ’Clair’;
```

There are other examples of assignments in other parts of this appendix.

Flow Control

PL/SQL supports the following kinds of flow-control statements:

- IF statements. These execute a group of one or more statements based on whether a condition is TRUE.
- Basic loops. These repeatedly execute a group of one or more statements until an EXIT statement is reached.
- FOR loops. These repeatedly execute a group of one or more statements a given number of times or until an EXIT statement is reached.
- WHILE loops. These repeatedly execute a group of one or more statements until a particular condition is met or an EXIT statement is reached.
- GOTO statements. These pass execution directly to another point in the code, exiting loops and enclosing blocks as necessary. Use these sparsely, as they make code difficult to read and debug.

If you know other programming languages, you probably are familiar with most or all of these types of statements. The following sections describe the PL/SQL versions of them in greater detail. For more information on any of these, see "Control Structures" in the *PL/SQL User's Guide and Reference*.

You can nest flow control statements within one another to any level of complexity.

IF Statements

These are similar to the IF statement in many other languages, except that they use predicates, which are three-valued Boolean expressions like the SQL predicates discussed earlier in this appendix. In most respects, a Boolean NULL behaves like a Boolean FALSE, except that negation does not make it positive, but leaves it NULL.

The IF statement has the following forms:

```
IF <condition> THEN <statement-list>;
     END IF;
```

If the condition following IF is TRUE, PL/SQL executes the statements in the list following THEN. A semicolon terminates this list. END IF (not ENDIF) is mandatory and terminates the entire IF statement. Here is an example:

```
IF balance > 500 THEN send_bill(customer);
END IF;
```

We are assuming that send_bill is a procedure taking a single parameter.

```
IF <condition> THEN <statement-list>;
     ELSE <statement-list>;
     END IF;
```

This is the same as the preceding statement, except that, if that condition is FALSE or NULL, PL/SQL executes the statement list following ELSE instead of that following THEN.

```
IF <condition> THEN <statement-list>;
     ELSIF <condition> THEN <statement-list>;
     ELSIF <condition> THEN <statement-list>;.....
     ELSE <statement-list>;
END IF;
```

You can include any number of ELSIF (not ELSEIF) conditions. Each is tested only if the IF condition and all preceding ELSIF conditions are FALSE or NULL. As soon as PL/SQL finds an IF or ELSIF condition that is TRUE, it executes the associated THEN statement list and skips ahead to END IF. The ELSE clause is optional, but, if included, must come last. It is executed if all preceding IF and ELSIF conditions are FALSE or NULL.

NULL Statements

If you do not want an action to be taken for a given condition, you can use the NULL statement, which is not to be confused with database nulls, Boolean NULLs, or the SQL predicate IS NULL. The syntax of this statement is simply:

```
NULL;
```

The statement performs no action, but fulfills the syntax requirement that a statement list must follow every THEN keyword. In some cases, you can also use it to increase the readability of your code. For more information on the NULL statement, see “NULL Statement” in the *PL/SQL User’s Guide and Reference.*

Basic Loops

A basic loop is a loop that keeps repeating until an EXIT statement is reached. The EXIT statement must be within the loop itself. If no EXIT (or GOTO) statement ever executes, the loop is infinite. An example follows:

```
credit := 0;
LOOP
     IF c = 5 THEN EXIT;
     END IF;
     credit := credit + 1;
END LOOP;
```

This loop keeps incrementing credit until it reaches 5 and then exits. An alternative to placing an exit statement inside an IF statement is to use the EXIT-WHEN syntax, as follows:

```
EXIT WHEN credit = 5;
```

This is equivalent to the earlier IF statement.

Note: The EXIT statement cannot be the last statement in a PL / SQL block. If you want to exit a PL / SQL block before its normal end is reached, use the RETURN statement. For more information, see “RETURN Statement” in the *PL/SQL User’s Guide and Reference*.

FOR Loops

A FOR loop, as in most languages, repeats a group of statements a given number of times. The following FOR loop is equivalent to the example used for basic loops, except that it also changes a variable called interest.

```
FOR credit IN 1..5 LOOP
     interest := interest * 1.2;
END LOOP;
```

The numbers used to specify the range (in this case, 1 and 5) can be variables, so you can let the number of iterations of the loop be determined at runtime if you wish.

WHILE Loops

A WHILE loop repeats a group of statements until a condition is met. Here is a WHILE loop that is the equivalent of the preceding example:

```
credit := 1;
WHILE credit <= 5 LOOP
     interest := interest * 1.2;
     credit := credit + 1;
END LOOP;
```

Unlike some languages, PL / SQL has no structure, such as REPEAT-UNTIL, that forces a LOOP to execute at least once. You can create this effect, however, using either basic or WHILE loops and setting a variable to a value that will trigger the loop, as in the above example. For more information on loops, see “Iterative Control” in the *PL/SQL User’s Guide and Reference*.

GOTO Statements

A GOTO statement immediately transfers execution to another point in the program. The point in the program where the statement is to arrive must be preceded by a label. A label is an identifier for a location in the code. It must be unique within its scope and must be enclosed in double angle brackets, as follows:

```
<<this_is_a_label>>
```

You only use the brackets at the target itself, not in the GOTO statement that references it, so a GOTO statement transferring execution to the above label would be:

```
GOTO this_is_a_label;
```

Note: An EXIT statement can also take a label, if that label indicates the beginning of a loop enclosing the EXIT statement. You can use this to exit several nested loops at once. See “Loop Labels” in the *PL/SQL User’s Guide and Reference* for more information.

A GOTO statement is subject to the following restrictions:

- It must branch to an executable statement, not, for example, an END.
- It cannot branch to a point within the body of IF or a LOOP statement, unless it is contained in the body of that statement itself.
- It cannot branch to a subprogram or enclosing block of the present block (with one exception, explained shortly).
- It cannot branch from one IF statement clause to another. That is to say, it cannot jump between THEN, ELSIF, and ELSE clauses that are part of the same IF statement.
- It cannot branch from the EXCEPTION section to the EXECUTABLE section of the same block.
- It can, however, branch from the EXCEPTION section of a block to the EXECUTABLE section of an enclosing block, which is the exception to the third rule above.

The EXCEPTION Section

The EXCEPTION section follows the END that matches the BEGIN of the EXECUTABLE section and begins with the keyword EXCEPTION. It contains code that responds to runtime errors. An *exception* is a specific kind of runtime error. When that kind of error occurs, you say that the exception is *raised.* An *exception handler* is a body of code designed to handle a particular exception or group of exceptions. Exception handlers, like the rest of the code, are operative only once the code is compiled and therefore can do nothing about compilation errors.

There are two basic kinds of exceptions: predefined and user-defined. The predefined exceptions are provided by PL/SQL in a package called STANDARD. They correspond to various runtime problems that are known to arise often—for example, dividing by zero or running out of memory. These are listed in the *PL/SQL User's Guide and Reference* under "Predefined Exceptions".

The Oracle Server can distinguish between and track many more kinds of errors than the limited set that STANDARD predefines. Each of Oracle's hundreds of messages are identified with a number, and STANDARD has simply provided labels for a few of the common ones. You can deal with the other messages in either or both of two ways:

- You can define your own exception labels for specified Oracle messages using a pragma (a compiler directive). This procedure will be explained shortly.
- You can define a handler for the default exception OTHERS. Within that handler, you can identify the specific error by accessing the built-in functions SQLCODE and SQLERRM, which contain, respectively, the numeric code and a prose description of the message.

You can also define your own exceptions as will be shown. It is usually better, however, to use Oracle exceptions where possible, because then the conditions are tested automatically when each statement is executed, and an exception is raised if the error occurs.

Declaring Exceptions

PL/SQL predefined exceptions, of course, need not be declared. You declare user-defined exceptions or user-defined labels for Oracle messages in the DECLARE section, similarly to variables. An example follows:

```
customer_deceased EXCEPTION;
```

In other words, an identifier you choose followed by the keyword EXCEPTION. Notice that all this declaration has done is provide a name. The program still has no idea when this exception should be raised. In fact, there is at this point no way of telling if this is to be a user-defined exception or simply a label for an Oracle message.

Labeling Oracle Messages

If a previously-declared exception is to be a label for an Oracle error, you must define it as such with a second statement in the DECLARE section, as follows:

```
PRAGMA EXCEPTION_INIT (exception_name, Oracle_error_number);
```

A PRAGMA is a instruction for the compiler, and EXCEPTION_INIT is the type of PRAGMA. This tells the compiler to associate the given exception name with the given Oracle error number. This is the same number to which SQLCODE is set when the error occurs. The advantage of this over defining your own error condition is that you pass the responsibility for determining when the error has occurred and raising the exception to Oracle. You can find the numeric codes and explanations for Oracle messages in *Oracle7 Server Messages*.

User-Defined Exceptions

If the declared condition is not to be a label for an Oracle error, but a user-defined error, you do not need to put another statement referring to it in the DECLARE section. In the EXECUTABLE section, however, you must test the situation you intend the exception to handle whenever appropriate and raise the condition manually, if needed. Here is an example:

```
IF cnum < 0 THEN RAISE customer_deceased;
```

You can also use the RAISE statement to force the raising of predefined exceptions. For more information, see “Error Handling” in the *PL/SQL User's Guide and Reference*.

Handling Exceptions

Once an exception is raised, whether explicitly with a RAISE statement or automatically by Oracle, execution passes to the EXCEPTION section of the block, where the various exception handlers reside. If a handler for the raised exception is not found in the current block, enclosing blocks are searched until one is found. If PL/SQL finds an OTHERS handler in any block, execution passes to that handler. An OTHERS handler must be the last handler in its block. If no handler for an exception is found, Oracle raises an unhandled exception error. Note: this does not automatically roll back (undo) changes made by the subprogram, which might leave the database in an undesirable intermediate state.

This is the syntax of an exception handler:

```
WHEN exception_condition THEN statement_list;
```

The exception is the identifier for the raised condition. If desired, you can specify multiple exceptions for the same handler, separated by the keyword OR. The exception can be either one the package STANDARD provided or one you declared. The statement list does what is appropriate to handle the error—writing information about it to a file, for example—and arranges to exit the block gracefully if possible. Although exceptions do not necessarily force program termination, they do force the program to exit the current block. You cannot override this with a GOTO statement. You can use a GOTO within an exception handler, but only if its destination is some enclosing block.

Note: If you have an error prone statement and want execution to continue following this statement, even when an exception occurs, put the statement, including the appropriate exception handlers, in its own block, so that the current block becomes the enclosing block.

Note: If an exception occurs in the DECLARE section or the EXCEPTION section itself, local exception handlers cannot address it; execution passes automatically to the EXCEPTION section of the enclosing block.

Storing Procedures and Functions in the Database

To have a procedure or function stored as a database object, you issue a CREATE PROCEDURE or a CREATE FUNCTION statement directly to the server using SQL*PLUS or Server Manager. The easy way to do this is to use your ordinary text editor to produce the CREATE statement and then to load it as a script. This process is explained under “Creating Stored Procedures and Functions” in the *Oracle7 Server Application Developers Guide*. This approach is recommended because you often create entire groups of procedures and functions together. These groups are called “packages” and are explained later in this appendix.

The syntax for these statements is slightly different than that used to declare subprograms in PL/SQL, as the following example shows:

```
CREATE PROCEDURE fire_employee (empno INTEGER) IS
    BEGIN
          DELETE FROM Employees WHERE enum = empno;
    END;
```

As you can see, the main difference is the addition of the keyword CREATE. You also have the option of replacing the keyword IS with AS, which does not affect the meaning. To replace an existing procedure of the same name with this procedure (as you frequently may need to do during development and testing), you can use CREATE OR REPLACE instead of simply CREATE. This destroys the old version, if any, without warning.

Privileges Required

A stored procedure or function (for the rest of this discussion, “procedure” shall mean “procedure or function” unless otherwise indicated or clear from context) is a database object like a table. It resides in a schema, and its use is controlled by privileges. To create a procedure and have it compile successfully, you must meet the following conditions:

- If the procedure is to be in your own schema, you must have the CREATE PROCEDURE or the CREATE ANY PROCEDURE system privilege. These privileges apply as well to functions.
- If the procedure is to be in a schema you do not own, you must have the CREATE ANY PROCEDURE system privilege.
- You must have the object privileges necessary to perform all operations contained in the procedure. You must have these privileges as a user, not through roles. If your privileges change after you have created the procedure, the procedure may no longer be executable.

To enable others to use the procedure, grant them the EXECUTE privilege on it using the SQL statement GRANT (see "GRANT" in Chapter 4 of the *Oracle7 Server SQL Reference*). When these users execute the procedure, they do so under your privileges, not their own. Therefore, you do not have to grant them the privileges to perform these actions outside the control of the procedure, which is a useful security feature. To enable all users to use the procedure, grant EXECUTE to PUBLIC. The following example permits all users to execute a procedure called show_product.

```
GRANT EXECUTE ON show_product TO PUBLIC;
```

Of course, the public normally does not execute such a procedure directly. This statement enables you to use the procedure in your PL/SQL code that is to be publicly executable. If multiple users access the same procedure simultaneously, each gets his own instance. This means that the setting of variables and other activities by different users do not affect one another.

For more information on privileges and roles, see "GRANT" in Chapter 4 of the *Oracle7 Server SQL Reference*. There are three versions of GRANT listed—one each for object privileges, system privileges, and roles.

For more information on storing procedures and functions in the database, see "Storing Procedures and Functions" in the *Oracle7 Server Application Developers Guide* and see "CREATE FUNCTION" and "CREATE PROCEDURE" in the *Oracle7 Server SQL Reference*.

Packages

A package is a group of related PL/SQL objects (variables, constants, types, and cursors) and subprograms that is stored in the database as a unit. Being a database object, a package resides in a schema, and its use is controlled by privileges. Among its differences from regular PL/SQL programs are that a package as such does not do anything. It is a collection of subprograms and objects, at least some of which are accessible to applications outside of it. It is the subprograms in the package that contain the executable code. A package has the following two parts:

- The package specification is the public interface to the package. It declares all objects and subprograms that are to be accessible from outside the package. Packages do not take parameters, so these constitute the entire public interface.
- The package body is the internal portion of the package. It contains all objects and subprograms that are to be local to the package. It also contains definitions of the public cursors and subprograms. The package specification declares but does not define these.

One of the advantages of using packages is that the package specification is independent of the body. You can change the body and, so long as it still matches the specification, no changes to other code are needed, nor will any other references become invalid.

Packages cannot be nested, but they can call one another's public subprograms and reference one another's public objects.

Instantiation of Packages

It is important to realize that a package is instantiated once for a given user session. That is to say, the values of all variables and constants, as well as the contents and state of all cursors, in a package, once set, persist for the duration of the session, even if you exit the package. When you reenter the package, these objects retain the values and state they had before, unless they are explicitly reinitialized. Of course, another user has another session and therefore another set of values. Nonetheless, a global reinitialization of a package's objects for you does not take place until you disconnect from the database.

There is an exception, however. When one package calls another, execution of the second has a dependency on the first. If the first is invalidated, for example because its creator loses a privilege that the package requires, the second, while not necessarily invalidated, becomes deinstantiated. That is to say, all its objects are reinitialized.

Note: In PL/SQL, stored procedures and packages are automatically recompiled if changes to the database mandate it. For example, a change to the datatype of a column can automatically cascade to a variable referencing that column if the former is declared with the %TYPE attribute, but that change requires that the PL/SQL procedure declaring that variable be recompiled. So long as the PL/SQL code as written is still valid, the recompilation occurs automatically and invisibly to the user.

To create a package, you use the SQL statement CREATE PACKAGE for the specification and CREATE PACKAGE BODY for the body. You must create the specification first. Sometimes, a package may consist of only public variables, types, and constants, in which case no body is necessary. Generally, however, you use both parts.

Note: Before you can create a package, the special user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script may vary according to your operating system. Contact your database administrator if you are not sure this script has been run.

Creating the Package Specification

The syntax of the CREATE PACKAGE statement is as follows:

```
CREATE [OR REPLACE] PACKAGE package_name IS
    {PL/SQL declarations}
     END;
```

The optional OR REPLACE clause operates just as it does for stored procedure. The PL/SQL declarations are as outlined under *The DECLARE Section*, except that the keyword DECLARE is not used and that the subprogram and cursor declarations are incomplete. For subprograms, you provide only the name, parameters, and, in the case of functions, the datatype of the return value. For cursors, provide the name and a new item called the return type. This approach hides the implementation of these objects from the public while making the objects themselves accessible.

The syntax for declaring a cursor with a return type is as follows:

```
CURSOR c1 IS RETURN return_type;
```

The return type is always some sort of record type that provides a description of the cursor's output. The structure of this record is to mirror the structure of the cursor's rows. You can specify it using any of the following:

- A record subtype previously defined and in scope. For more information, see *Records*.
- A type inherited from such a record subtype using the %TYPE attribute. For more information, see *Declaring Variables*.
- A type inherited from a table, most likely the table the cursor queries, using the %ROWTYPE attribute. For more information, see *Records*.
- A type inherited from a cursor using the %ROWTYPE attribute. For more information, see *Records*.

For more information, see CREATE PACKAGE in Chapter 4 of the *Oracle7 Server SQL Reference*, "Packages" in the *PL/SQL User's Guide and Reference*, and "Using Procedures and Packages" in the *Oracle7 Server Application Developers Guide*.

Creating the Package Body

To create the package body, use the CREATE PACKAGE BODY statement. The syntax is as follows:

```
CREATE [OR REPLACE] PACKAGE BODY package_name IS
    {PL/SQL declarations}
     END;
```

Since a package as such does not do anything, the PL / SQL code still consists only of a DECLARE section with the keyword DECLARE omitted. It is the subprograms within the package that contain the executable code. Variables, constants, types, and cursors declared directly (in other words, not within a subprogram) in the declare section have a global scope within the package body. Variables, constants, and types already declared in the package specification are public and should not be declared again here.

Public cursors and subprograms, however, must be declared again here, as their declarations in the specification is incomplete. This time the declarations must include the PL / SQL code (in the case of subprograms) or the query (in the case of cursors) that is to be executed. For subprograms, the parameter list must match that given in the package specification word for word (except for differences in white space). This means, for example, that you cannot specify a datatype directly in the specification and use the %TYPE attribute to specify it in the body.

You can create an initialization section at the end of the package body. This is a body of executable code—chiefly assignments—enclosed with the keywords BEGIN and END. Use this to initialize constants and variables that are global to the package, since otherwise they could be initialized only within subprograms, and you have no control of the order in which subprograms are called by outside applications. This initialization is performed only once per session.

For more information, see CREATE PACKAGE BODY in the *Oracle7 Server SQL Reference*, "Packages" in the *PL/SQL User's Guide and Reference*, and "Using Procedures and Packages" in the *Oracle7 Server Application Developers Guide*.

Overloading Subprograms

Within a package, subprogram names need not be unique, even at the same level of scope. There can be multiple like-named subprograms in the same declare section, provided that the parameters that they take differ in number, order, or datatype and that, when the procedures are called, the values passed by the calling procedure (the actual parameters) match or can be automatically converted to the datatypes specified in the declaration (the formal parameters). To find out which datatypes PL/SQL can convert automatically, look under "Datatype Conversion" in the *PL/SQL User's Guide and Reference.*

The reason this is permitted is so you can overload subprograms. Overloading permits you to have several versions of a procedure that are conceptually similar but behave differently with different parameters. This is one of the properties of object-oriented programming. For more information on overloading, see "Overloading" in the *PL/SQL User's Guide and Reference.*

Database Triggers

Triggers are blocks of PL/SQL code that execute automatically in response to events. Database triggers reside in the database and respond to changes in the data. They are not to be confused with application triggers, which reside in applications and are beyond the scope of this discussion. Database triggers are a technology that for the most part has superseded application triggers.

You create triggers as you do stored procedures and packages, by using your text editor to write scripts that create them and then using SQL*Plus or Server Manager to run these scripts. A trigger is like a package in that:

- It takes no parameters as such. It refers to, responds to, and possibly affects the data in the database.
- It cannot be directly called like a procedure. To fire (execute) a trigger, you must make the database change to which it responds. If you only want to test the trigger, you can rollback (undo) the database change that you made after the trigger fires.

Triggers can be classified in three ways:

- INSERT triggers, UPDATE triggers, and DELETE triggers. This is a classification based on the statement to which the trigger responds. The categories are not mutually exclusive, meaning one trigger can respond to any or all of these statements.
- Row triggers and statement triggers. Any of the above statements can affect any number of rows in a table at once. A row trigger is fired once for each row affected. A statement trigger is fired once for each statement, however many rows it affects.

- BEFORE triggers and AFTER triggers. This specifies whether the trigger is fired before or after the data modification occurs.

As you can see, all three of these classifications apply to all triggers, so that there are, for example, BEFORE DELETE OR INSERT statement triggers and AFTER UPDATE row triggers.

Creating Triggers

The syntax of the CREATE TRIGGER statement is as follows:

```
CREATE [OR REPLACE] TRIGGER trigger_name
     BEFORE | AFTER
     DELETE | INSERT | UPDATE [OF column_list]
     ON table_name
     [ FOR EACH ROW [ WHEN predicate ] ]
     {PL/SQL block};
```

In the above, square brackets ([ ]) enclose optional elements. Vertical bars (|) indicate that what precedes may be replaced by what follows.

In other words, you must specify the following:

- A trigger name. This is used to alter or drop the trigger. The trigger name must be unique within the schema.
- BEFORE or AFTER. This specifies whether this is a BEFORE or AFTER trigger.
- INSERT, UPDATE, or DELETE. This specifies the type of statement that fires the trigger. If it is UPDATE, you optionally can specify a list of one or more columns, and only updates to those columns fire the trigger. In such a list, separate the column names with commas and spaces. You may specify this clause more than once for triggers that are to respond to multiple statements; if you do, separate the occurrences with the keyword OR surrounded by white space.
- ON table_name. This identifies the table with which the trigger is associated.
- PL/SQL Block. This is an anonymous PL/SQL block containing the code the trigger executes.

You optionally can specify the following:

- OR REPLACE. This has the usual effect.
- FOR EACH ROW [WHEN predicate]. This identifies the trigger as a row trigger. If omitted, the trigger is a statement trigger. Even if this clause is included, the WHEN clause remains optional. The WHEN clause contains

a SQL (not a PL/SQL) predicate that is tested against each row the triggering statement alters. If the values in that row make the predicate TRUE, the trigger is fired; else it is not. If the WHEN clause is omitted, the trigger is fired for each altered row.

Here is an example:

```
CREATE TRIGGER give_bonus
     AFTER UPDATE OF sales
     ON salespeople
     FOR EACH ROW WHEN sales > 8000.00
     BEGIN
     UPDATE salescommissions SET bonus = bonus + 150.00;
     END;
```

This creates a row trigger called give_bonus. Every time the sales column of the salespeople table is updated, the trigger checks to see if it is over 8000.00. If so, it executes the PL/SQL block, consisting in this case of a single SQL statement that increments the bonus column in the salescommissions table by 150.00.

Privileges Required

To create a trigger in your own schema, you must have the CREATE TRIGGER system privilege and one of the following must be true:

- You own the table associated with the trigger.
- You have the ALTER privilege on the table associated with the trigger.
- You have the ALTER ANY TABLE system privilege.

To create a trigger in another user's schema, you must have the CREATE ANY TRIGGER system privilege. To create such a trigger, you precede the trigger name in the CREATE TRIGGER statement with the name of the schema wherein it will reside, using the conventional dot notation.

Referring to Altered and Unaltered States

You can use the correlation variables OLD and NEW in the PL/SQL block to refer to values in the table before and after the triggering statement had its effect. Simply precede the column names with these variables using the dot notation.

If these names are not suitable, you can define others using the REFERENCING clause of the CREATE TRIGGER statement, which is omitted from the syntax diagram above for the sake of simplicity. For more information on this clause, see CREATE TRIGGER in the *Oracle7 Server SQL Reference*.

Note: if a trigger raises an unhandled exception, its execution fails and the statement that triggered it is rolled back if necessary. This enables you to use triggers to define complex constraints. If the effects of the trigger have caused a change in the value of package body variables, however, this change is not reversed. You should try to design your packages to spot this eventuality. For more information, see “Using Database Triggers” in the *Oracle7 Server Application Developers Guide.*

Enabling and Disabling Triggers

Just because a trigger exists does not mean it is in effect. If the trigger is disabled, it does not fire. By default, all triggers are enabled when created, but you can disable a trigger using the ALTER TRIGGER statement. To do this, the trigger must be in your schema, or you must have the ALTER ANY TRIGGER system privilege. Here is the syntax:

```
ALTER TRIGGER trigger_name DISABLE;
```

Later you can enable the trigger again by issuing the same statement with ENABLE in place of DISABLE. The ALTER TRIGGER statement does not alter the trigger in any other way. To do that you must replace the trigger with a new version using CREATE OR REPLACE TRIGGER. For more information on enabling triggers, see ALTER TRIGGER in the *Oracle7 Server SQL Reference.*

For more information on triggers generally, see “Using Database Triggers” in the *Oracle7 Server Application Developer’s Guide* and CREATE TRIGGER and DROP TRIGGER in the *Oracle7 Server SQL Reference.*

Index