ORACLE®

Oracle WebServer User's Guide

If you find any errors, omissions, or have any suggestions on how the information in this manual can be improved, please e-mail ipdoc@us.oracle.comOracle WebServer Documentation.

Release Number 2.0

Document Revision 2.0

Part Number A23646_1

Primary Authors: , Kennan Rossi

Contributors: Charles Prael, John Zussman

Preface

The Oracle WebServer User's Guide is part of the Oracle WebServer documentation set, which contains:

- This book, the *Oracle WebServer User's Guide*
- The Oracle WebServer online documentation
- The *Oracle WebServer Quick Reference*

The online documentation is provided in both `html` and `pdf` formats. You can view the `pdf` files using the Adobe Acrobat Reader provided. To view the `html` files, you must provide use your own web browser program, which must support tables, but need not support frames.

This Preface discusses this Guide's:

- Organization
- Typographic conventions
- Related documents

How this Guide is Organized

- Chapter 1, "Oracle WebServer Concepts" defines and explains WebServer terms and concepts, and provides background information. This chapter gives an orientation to the product.
- Chapter 2, "Using the Oracle WebServer Manager" provides a brief introduction to the Oracle WebServer Manager and lists the tasks you can perform with it.

- Chapter 3, “Setting Up a Secure Oracle WebServer” describes how to configure your Oracle WebServer to accept secure connections using the Secure Sockets Layer (SSL).
- Chapter 4, “Developing Applications for the Oracle WebServer” describes in detail how to develop applications to run on the Oracle WebServer.
- Chapter 5, “The PL/SQL Developer’s Toolkit Reference” provides a detailed reference for the PL/SQL procedures you can use to write programs that generate HTML documents dynamically.
- Appendix A, “Glossary” provides alookup reference for terms and concepts used throughout this book.
- Appendix 6, “Oracle WebServer Messages” provides a lookup reference for errors and messages that the Oracle WebServer can display to the user or log to files.
- Appendix B, “Overview of the Oracle7 Server, SQL, and PL/SQL” summarizes the capabilities of the Oracle7 database server and provides a syntax summary of the SQL and PL/SQL languages.
- Appendix C, “Introduction To HTML” summarizes the capabilities of HTML, gives usage examples, and provides a brief syntax reference.
- The Web Request Broker API is an engineering specification that defines the programming interface for writing back-end WebServer applications using the Web Request Broker. This work in progress is included for your convenience.

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:

```
[...fixup]
```

Related Documents

Part No.	Document Title
A23646-1	Oracle WebServer online documentation
A44047-1	Oracle WebServer Quick Reference

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

Contents

Oracle WebServer Concepts. 11
Overview 12
The Web Listener. 13
The Secure Sockets Layer 17
The Web Server Manager. 19
The CGI Interface 20
The Web Request Broker (WRB) 21
PL/SQL Agent. 22
Java 23
LiveHTML 25

Using the Oracle WebServer Manager. 27

Setting Up a Secure Oracle WebServer 31

Developing Applications for the Oracle WebServer 37
Application Development - an Overview 37
Server Extensions 38
The PL/SQL Agent 40
The Java™ Interpreter 52
The LiveHTML Interpreter 68

The PL/SQL Developer's Toolkit Reference. 71
Installing the Oracle WebServer Developer's Toolkit 73

Procedure and Function Reference 74
Head Related Tags 79
Body Tags 82
Frame Tags 98
List Tags 100
Character Format Tags 106
Physical Format Tags 110
Form Tags 111
Table Tags 120
OWA_UTIL Package 123
OWA_PATTERN Package 131
OWA_TEXT 142
OWA_IMAGE 145
OWA_COOKIE 146

Oracle WebServer Messages 151
00001 - 00600 Generic Oracle WebServer Configuration Messages 151
5000 - 5499 : Oracle Web Agent errors 153
5500 - 5599 : Oracle Web Agent Administration errors and messages 155
5600 - 5699 : Oracle Web Database Administration errors 157
5700 - 5799 : Oracle Web Listener Configuration errors and messages 158
5900 - 5999 : Oracle WebServer Registration errors and messages 160
7500 - 7599 : Oracle Web Server Proxy errors and messages 160
Java Web Developer's Toolkit Messages 160

Glossary 163

Overview of the Oracle7 Server, SQL, and PL/SQL 177
Oracle7 Server 177
SQL 180
PL/SQL 187
Basic Structure and Syntax 187
The EXECUTABLE Section 195
The EXCEPTION Section 199
Storing Procedures and Functions in the Database 201
Database Triggers 207

Introduction To HTML 211
What is HTML? 212
Getting Started 213
Document Structure 214

Body Tags 214
List Tags 218
Hypertext Linking 220
Reviewing Changes to Your HTML Document 225
Adding Style to Your HTML Document 225
Special HTML Tags 226
Tables 228
Forms 230
Creating Your Own HTML Document 235
More Information about HTML 236

CHAPTER

1

Oracle WebServer Concepts

This section offers a conceptual overview of the Oracle WebServer. It is intended to give you a context for the more specific task-oriented information that follows. The subsequent sections of this book are specifically targeted at those who manage Oracle WebServer sites (WebServer Administrators) and those who write application programs for them (application developers). This section covers:

- *Overview*.
- *The Web Listener*. This is the portion of the WebServer that interfaces to the local network or the World Wide Web.
- *The Secure Sockets Layer*. Introduction to SSL public key encryption.
- *The Web Server Manager*. The set of Web pages you can use to perform most WebServer administration.
- *The CGI Interface*. The standard Web mechanism for executing applications on a Web server.
- *The Web Request Broker (WRB)*. The core of the WebServer. An asynchronous request broker with an open API that Oracle WebServer uses to execute applications on the server.

- The *PL/SQL Agent*. The program the Oracle WebServer uses to execute procedures written in PL/SQL, Oracle's application development language, on the Oracle7 Server.
- *Java.* A new language for developing distributed network applications. Oracle WebServer enables you to execute Java either on the WebServer itself or on
- *LiveHTML.* A way to embed dynamic content in Web pages. This content can be either other Web pages or the output of scripts run by the Operating System. LiveHTML is an Oracle extension of the NCSA standard Server Side Includes functionality.

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 Server, the leading database 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. The combination of a cartridge and the WRB API is known as a *WRB Service.* Currently, Oracle WebServer support three kinds of WRB Services:

- PL/SQL cartridges. These execute stored PL/SQL procedures to generate HTML dynamically using Oracle data.
- Java™ cartridges. These execute Java code on the Server.
- LiveHTML. These embed Web pages within one another, and store in Web pages the output of scripts executed by the Operating System.

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 *The Web Request Broker (WRB)* for more information.

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*).

The Web Server Manager

To help you manage your Web site, the Oracle WebServer provides a set of Web pages that you can use to perform most common administration tasks. These pages are simply an easy way to edit the configuration files that the WebServer uses, so you can always use another tool to edit those files directly, although Oracle discourages this.

The WebServer Manager can be used from any Web browser. It has the following sections:

- *The Listener Pages*

- *The WRB Pages*
- *The PL/SQL Agent Pages*
- *The Oracle7 Server Pages*

The Listener Pages

The largest group of pages deal with administering the *Web Listener*. For the most part, however, these pages are just forms where you fill in the values for various configuration parameters. The exception is the security pages, where you specify which sorts of security schemes will be used, including the following:

- If you select Restriction, the IP addresses or domain names that are granted various levels of access.
- If you select Authentication, the user names and passwords that are granted various levels of access.

The WRB Pages

These pages let you specify the actual and virtual directories for WRB cartridges, as well as the number of WRBX's to assign to each.

The PL/SQL Agent Pages

Administering the *PL/SQL Agent* means administering the *Database Connection Descriptors (DCDs)* that it uses to establish its identity when communicating with the database.

The Oracle7 Server Pages

The *Oracle7 Server* is an extremely sophisticated product, and it is properly administered using Oracle Server Manager or directly through the SQL language. The pages provided here enable you to do a few basic things, such as start up and shut down database instances, and browse database contents.

The CGI Interface

The Common Gateway Interface (CGI) is the standard technique used by an HTTP server to execute a program that generates HTML output. Using CGI, you can run *PL/SQL*, and thus interface to the Oracle7 Server, from Internet servers

that do not support the *WRB*. This technique provides dynamic content rather than static content from files on disk. Oracle WebServer is fully compliant with CGI version 1.1.

When the Web Listener recognizes an incoming URL as a request to execute a CGI application, it spawns a separate process to perform the operation (*The Web Request Broker (WRB)* circumvents this need to spawn a new process for each request, thereby improving performance). The Web Listener passes the URL to this process and also maintains communication with it through standard input and output. Therefore, the CGI process can get the input it needs from the URL itself and / or the standard input. It sends its output back to the Listener through the standard output, and the Listener transmits it in turn to the client's Web browser.

The fact that CGI applications spawn a new process each time they are used is costly in terms of performance. For this among other reasons, Oracle recommends that you use the WRB instead.

The Web Request Broker (WRB)

The WRB (Web Request Broker) is an asynchronous request handler with an API (Application Program Interface) that enables it to interface dynamically and seamlessly to various back-end technologies called "WRB Services". It consists of a WRB Dispatcher that allocates Service requests among several running processes known as the WRBX's. The result is that the Listener is free to receive and validate URLs coming in, while each request is handed off to a process that executes it in the background. The WRB API is designed so that third parties can write their own cartridges to extend the WebServer.

Whenever it receives a URL that calls for the WRB, the Web Listener passes execution of the request to the *WRB Dispatcher* or simply *Dispatcher*. The Dispatcher maintains communication with a pool of processes called *WRB Executable Engines (WRBX's)*. Each WRBX interfaces, through the WRB API, to a WRB *cartridge*, which can be of the following types:

- *The PL/SQL Agent*. This cartridge executes PL / SQL commands stored in the database. It is better optimized for database access than the Java cartridge, but doesn't have all of Java's functionality.
- *The Java™ Interpreter*. This cartridge lets you execute Java on the server to generate dynamic Web pages. You can also execute PL / SQL from within Java using this cartridge.

- *The LiveHTML Interpreter*. This cartridge is Oracle's implementation and extension of the industry-standard Server Side Includes functionality. *LiveHTML* enables you to include in your Web pages the output of any program that your Operating System can execute.

The combination of a cartridge and its WRB API constitute a *WRB Service*. WRBX's are instances of WRB Services, so that there are three WRB Services corresponding to the three WRB cartridges, while the WRBX's are created and destroyed as needed to handle the workload.

The Dispatcher creates and maintains the WRBX's. The data passed to a WRBX consists of:

- The URL triggering the request.
- The desired language of the result.
- The desired character set of the result (how the language is encoded, for example, ISO or Unicode)
- CGI environment variables, which allow WRB cartridges to be run by applications written for *CGI*.
- If the request involves the use of the PL/SQL Agent, which of the *Database Connection Descriptors (DCDs)* to use.

The Dispatcher continually adjusts the load by controlling how many WRBX's are running at a given time, subject to certain parameters. These parameters are set by the WebServer Administrator, who decides the maximum and minimum number of WRBX's running. The Dispatcher creates new WRBX's as needed and connects them to the appropriate WRB Services. The Dispatcher keeps track of which WRBX's are executing requests and which are free, and periodically checks the WRBX's to see how long they have been idle. If a WRBX's idle time is beyond the maximum set by the WebServer Administrator, that WRBX is killed. The WRBX's assigned to a given WRB Service always have requests assigned in the same order, so that traffic will be concentrated in a small number of them, and most of them will not be idle. The WRBX's are single-threaded processes that communicate with the WRB Dispatcher using the dataflow mechanism appropriate to the Operating System (such as a pipe in Unix).

PL/SQL Agent

WRBX's interfaced to the PL/SQL Agent connect to the database when they start up, so that the execution of requests can proceed more quickly. The WRBX's connect to the database schemas specified in their configuration files.

Specifying the Use of PL/SQL Agents

If the URL for a given request contains the string “owa”, properly placed, the Listener fulfils the request using the PL/SQL Agent. Whether this is done through the *WRB* or through *CGI* depends on how the WebServer Administrator has configured the directory mappings. The PL/SQL Agent executes application code written in PL/SQL and returns the output in HTML form for the Web Listener to output as a Web page.

When a URL is received for the PL/SQL Agent, it contains a *DCD* (Database Connection Descriptor). The DCD determines both the database access privileges the PL/SQL Agent has when executing this request and the schema (portion of the database) that it accesses.

PL/SQL procedures are stored in the database. The PL/SQL Agent invokes these by issuing commands to the database, which then performs the actual execution and sends the output and status messages back to the PL/SQL Agent.

Oracle WebServer also provides you with Java classes that can invoke PL/SQL. For more information, see Java or the Java Interpreter.

The PL/SQL Developer’s Toolkit

To make it easier for you to develop Web applications using Oracle data, Oracle WebServer provides you a group of *PL/SQL* packages that you can use to easily generate Web pages from data stored in an Oracle database. These packages are called the PL/SQL Developer’s Toolkit. The intent is for you to create PL/SQL procedures that access and process the Oracle data you wish to place on the Web. From within these procedures, you call the PL/SQL Developer’s Toolkit procedures you need to create the HTML you want. You store the procedures you write in the database, just as other PL/SQL packages, including the tookit, are stored. You also design your Web pages, including the dynamically-generated ones, to produce URLs that call the PL/SQL procedures you want in response to specified user actions. Having your code executed within the database brings many performance, security, and portability benefits. For more information on the PL/SQL Developer’s Toolkit, see *The PL/SQL Developer’s Toolkit Reference*.

Java

Java is an object-oriented language for creating distributed applications on the Internet or other networks. Modules of Java code known as “applets” can be

downloaded from the Internet or a local network in real time and locally executed. Java applets themselves can call and execute other applets, so that a Java application as executed on the user's machine can be constructed "on the fly" from a repertoire of standard parts that reside on the net. You might call this the "building block" approach to programming. Among the important features of Java are the following:

- It is extremely portable. Java code is compiled to a form known as "bytecode". This is a sort of generalized computer code that is not executable by any particular machine, but is recognized by the "Java Virtual Machine". It is as bytecode that Java applets transverse the net. The Java Virtual Machine (VM) resides on the computer where the applet is to be executed and converts the bytecode to the native code for that machine. Currently, Java VMs exist or are planned for all current versions of Windows, Solaris, MacOS, OS/2, Linux, Amiga, and other platforms.
- It is fully object-oriented. Languages like C++ that add object-oriented features to non-object-oriented languages. Java applets do not allow violation of object-oriented principles such as "encapsulation".
- It uses a C-like syntax. This makes the language easier for C programmers to learn.
- It is multi-threaded, in effect executing several chains of control flow concurrently. The Java language itself provides tools for managing the threads, rather than relying exclusively on the OS.
- It prohibits direct memory manipulation. In Java, there are no pointers and no direct memory allocation. This eliminates a rich source of C's functionality, and an even richer source of its bugs.
- You can embed calls to Java applets in Web pages, and the applet will be executed by the browser, provided it is Java-enabled. Most major browsers plan to support Java.

Note: Though powerful, Java is a young technology. Oracle WebServer supports it because of its rich features and wide acceptance. However, you should be aware that it may be somewhat less stable than more mature technologies.

Client vs. Server Side Java

Oracle WebServer supports the use of Java either on the client, which is to say any Java-enabled Web browser, or on the server. Code to be executed on the client is for the most part extracted and manipulated like other data. The best way to handle such code is to store it in the OS file system and extract it in real time.

You can also execute Java as a *WRB* cartridge on the WebServer itself. You might want to do this, for example, to perform graphical manipulation for which PL/SQL is ill-suited. For example, you can combine several graphics from the database into a single image. Each region of the image would be a separate button that the user can click, and each button clicked would produce a different effect. In HTML, this is called an "image map". Using Java on the server, you could generate such image maps dynamically, with the components of the image being based on the results of a database query.

To execute Java on the server, you use the WRB API to interface directly to the *Java Interpreter* residing in the WebServer. This interpreter finds and executes the Java code and returns the results, through the WRB interface, to the Web Listener.

Using PL/SQL Within Java

Since *PL/SQL* code is actually part of the database, you can call it from within Java, which enables you to create applications that combine the strengths of both languages. Because PL/SQL execution takes places in the database, doing this does not hinder the portability of the application. A PL/SQL application can execute without modification on any platform where the *Oracle7 Server* runs, just as a Java applet can execute without modification on any platform that has a Java Virtual Machine (VM).

LiveHTML

LiveHTML is Oracle's implementation and extension of the standard Server Side Include functionality defined by the NCSA. *The LiveHTML Interpreter* enables you to include dynamic content in otherwise static Web pages. At the point in your Web page where you want to interject dynamic content, you place a tag that points to one of the following:

- A static Web page.
- Another LiveHTML Web page.
- A script that is executed on the server and outputs HTML. This script may but need not conform to the *CGI* standard.
- A system variable, for example: FMODDATE.

You can use a variable to determine at runtime the Web page, variable, or script to which the tag points. This enables you to have a Web page that selects dynamically from among any number of static Web pages or scripts, based, for

example, on values a user provides in an HTML form. The result is a dynamic Web page built of static Web page components, variables, and HTML output from scripts.

A Web page that is to use LiveHTML must be parsed by the WebServer. For this reason, it differs slightly from ordinary Web pages written in HTML, which the WebServer simply delivers to the browser. To have the LiveHTML tags executed on the server, you must use the WebServer Manager to specify that a given Web Listener is to parse files for LiveHTML You have the option of having the Listener parse all files or just those with certain extensions.

Enabling users to execute scripts on the server can create security and other risks. For this reason, you can specify that a specific Listener allows only "crippled" includes. This means that LiveHTML parsed by that Listener will be able only to call static HTML, environment variables, or other server parsable files, not executable scripts.

You frequently use LiveHTML when you have standard components, such as menus, that you want on many pages. If desired, you can use LiveHTML to run the PL/SQL Agent under *CGI* and thereby incorporate dynamic Oracle data in hardcoded Web pages.

CHAPTER

2 Using the Oracle WebServer Manager

The Oracle WebServer Manager is a collection of HTML forms you can use to configure your Oracle WebServer. These forms allow you to:

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

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.

Oracle WebServer Manager Tasks

The following list is a summary of the configuration tasks you can perform using the Oracle WebServer Manager. Following one of these task links will take you to the form where you can perform the task:

Oracle7 Database Administration

- Starting up an Oracle7 database
- Shutting down an Oracle7 database

Oracle Web Listener Administration

- Creating a new Web Listener
- Modifying a Web Listener configuration
- Configuring Web Listener network parameters
- Configuring Web Listener logging parameters
- Configuring Web Listener user and group parameters
- Configuring Web Listener virtual directory mappings
- Configuring Web Listener file caching
- Defining Web Listener file extensions for language formats
- Defining Web Listener file extensions for MIME types
- Defining Web Listener file extensions for encoding formats
- Configuring Web Listener security
- Controlling access to virtual files and directories

- Configuring port security (SSL)
- Starting a Web Listener process
- Stopping a Web Listener process
- Deleting a Web Listener
- Enabling Web Listener proxy behavior
- Enabling LiveHTML parsing for a Web Listener

PL/SQL Agent Administration

- Creating a new Database Connection Descriptor (DCD)
- Creating a default DCD
- Creating a DCD for database administration
- Creating a new DCD from an existing configuration
- Modifying a DCD
- Deleting a DCD

Web Request Broker Administration

- Configuring the Web Request Broker
- Configuring a new WRB cartridge
- Modifying a WRB cartridge configuration
- Deleting a WRB cartridge

CHAPTER

3

Setting Up a Secure Oracle WebServer

You can up an Oracle WebServer process to accept secure connections on a particular TCP/IP port by configuring it to use the Secure Sockets Layer (SSL) on that port. SSL is an emerging standard for encrypted data transmission—see *The Secure Sockets Layer* for an introduction to the terms and concepts involved in this kind of security.

To set up your Oracle WebServer to use SSL, you must do the following:

1. Generate a certificate request.
2. Request a certificate from a Certifying Authority (CA).
3. Physically secure and prepare your WebServer host machine.
4. Install the certificate granted you by the CA.
5. Activate SSL on at least one WebServer port.

Generating a Certificate Request

To generate a certificate request, run the interactive utility `genreq` and enter the information for which it prompts you. When the prompt specifies a default value, you can just press return to enter that value, or enter a different value if you prefer. To run `genreq`, do the following:

1. Type `genreq` to start the utility.

2. Type `G` to begin creating a certificate request.
3. When prompted, type a password, used in generating the private key. Just choose a random string of characters—you need only remember this string long enough to repeat it in the next step.
4. Retype the password for confirmation.
5. Choose the public exponent you want to use in generating the key pair.
6. Enter the size in bits of the modulus you want to use in generating the key pair. The default size is 768 bits and the maximum is 1024 bits. A modulus size between these two values is recommended.

   For versions of `genreq` sold outside the USA, the maximum (and default) modulus size is 512 bits.
7. Choose one of three methods for generating a random seed to use in generating the key pair:
   - F—`genreq` prompts you to enter the full pathname of a file in your local file system. This can be any file that is at least 256 bytes in size, does not contain any secret information, and has contents that cannot easily be guessed.
   - K—`genreq` prompts you to enter random keystrokes. `genreq` uses the variation in time between keystrokes to generate the seed. Don't use the keyboard's autorepeat capability, and don't wait longer than two seconds between keystrokes. `genreq` prompts you when you have typed enough keystrokes. You must delete any unused characters typed after this prompt.
   - B—genreq prompts you to enter both a file name and random keystrokes. This option is recommended.
8. Enter the name of a file in which to store your WebServer's distinguished name. You can choose the default, or enter any filename with a `.der` extension. `genreq` creates this file in the current directory, though you may later move it to any convenient location.
9. Enter the name of a file in which to store your WebServer's private key. You can choose the default, or enter any filename with a `.der` extension. `genreq` creates this file in the current directory, though you may later move it to any convenient location.
10. Enter the name of a file in which to store the certificate request. You can choose the default, or enter any filename with a `.pkc` extension.
11. Enter the requested identification information for you organization:

- Common Name—the fully qualified host name of your organization's Internet point of presence as defined by the Domain Name Service (DNS), for example, `www.oracle.com`.
- Email Address—the email address where the CA can contact you.
- Organization—the official, legal name of your company or organization. Most CAs require you to verify this name by providing official documents, such as a business license.
- Organizational Unit—(optional) the name of the group, division, or other unit of your organization responsible for your Internet presence, or an informal or shortened name for your organization.
- Locality—(optional) the city, principality, or country where your organization is located.
- State or Province—the full name of the state or province where your organization is located. Most CAs won't accept abbreviations.
- Country—the two-character ISO-format abbreviation for the country where your organization is located. The country code for the United States is "US."

  When you have entered all the requested information, `genreq` responds with `Thank you`, and processes the data you have entered. When it is finished, it outputs `done` and returns you to the main menu.

12. Type `Q` to quit the program.

Requesting a Certificate

To request a certificate, email the request generated by `genreq` to a certifying authority.

The certification process can take time, from a few days to several weeks. The more organized and complete your paperwork, the better your chances are for quick certification.

Preparing Your WebServer Host Machine

For your WebServer to be secure as advertised to clients, you must make sure that no unauthorized person has access to your WebServer's host machine. Here are some suggestions:

- Place the machine in a locked server room.
- Limit distribution of keys or combinations to the server room to a few trusted individuals.

- Set up a secure area of the machine's file system that can be accessed only by the root user. This is where you will store your private key and your certificate when you receive it.
- Set a secure root password on the machine, using at least six characters and mixing numbers, legal punctuation marks, and mixed-case letters. Try not to use a character string that is a proper name or a word in any language. Change passwords frequently, and never write a password on paper.
- Strictly limit the programs that are installed and allowed to run on the machine.
- Limit TCP/IP connections to the machine to port 443, the default port for secure connections. Disable all other ports.

To learn how to implement these suggestions on your WebServer machine, see your machine's operating system reference manual.

Installing Your Certificate

When you receive your certificate from the CA, you must use the Oracle Web Server Manager to install it.

1. Use your email reader to save the message from the CA containing the certificate to a file with a `.der` extension, such as `cert.der`.
2. Use a text editor to remove the header information before the BEGIN CERTIFICATE line and the footer information after the END CERTIFICATE line. *Do not* delete the BEGIN CERTIFICATE and END CERTIFICATE lines themselves.
3. Go to the *Oracle WebServer Administration Form*.
4. Follow the link to the *Oracle Web Listener Administration Form*.
5. Follow the link to the *Oracle Web Listener Advanced Configuration Form*.
6. Follow the link to the *Oracle Web Listener Configure Security Form*.
7. Go to the Secure Sockets Layer section of the form and follow the instructions.

Activating SSL on a Port

Once you've installed your certificate on your WebServer machine and configured SSL on your WebServer, you must activate SSL on at least one port:

1. Go to the *Oracle Web Listener Advanced Configuration Form*.
2. Go to the Addresses and Ports section.
3. Set the Security pull-down menu to SSL in the entry for at least one port.

Note: For increased security, it is best to activate SSL only on port 443.

4 Developing Applications for the Oracle WebServer

This section covers the development of applications for the WebServer. It has the following sections:

- *Application Development - an Overview*
- *Server Extensions*
- *The PL/SQL Agent*
- *The Java™ Interpreter*
- *The LiveHTML Interpreter*

Application Development - an Overview

Applications developed for the Oracle WebServer have two general types of components:

- Web pages, whether statically coded or generated at runtime from PL/SQL or Java. These constitute both the user interface and the final product. Using LiveHTML, these can reference applications executed on the server when the page is accessed.
- Code executed on the server. There are several kinds of code that can be so executed, but all of them eventually produce HTML that is sent to the browser as Web pages.

The Web pages you create will themselves generate URLs that determine what other Web pages are retrieved and what code is executed on the server. Therefore, you must write your Web pages to generate URLs that contain information that the server needs to find the desired Web pages or execute the

desired code. These are the general ways your Web pages can produce such URLs:

- They can be explicitly given as part of hypertext anchors. In this case, the URL is transmitted whenever the user triggers the hypertext link.
- They can come from a user typing them directly into a search dialog box.
- They can come from HTML forms.

From the viewpoint of the client, all of these techniques specify the retrieval of a Web page. What determines whether a Web page is simply fetched from the file system or generated by some code executed on the server is the location specified in the URL. The WebServer understands that some locations contain files and others reference WRB Services or CGI programs. It is the configuration of the WebServer that determines which is which. You could say that the server extensions s appear to the client as “virtual files.”

Server Extensions

There are three basic techniques for executing applications on the server to dynamically generate Web pages:

- *The PL/SQL Agent*
- *The Java™ Interpreter*
- *The LiveHTML Interpreter*.

The first two are languages that the WebServer can cause to be executed. The last is an HTML technique for passing execution to another process at a certain point in the scanning of a Web page, and embedding the output of that process in the calling page. LiveHTML can also execute programs, but its programs are written in some language, such as Perl for Unix or Visual Basic for Windows NT, that is executed by the OS itself rather than the WebServer.

Generally speaking, you can combine these techniques. For example, the Java Interpreter can execute PL/SQL on the database.

The PL/SQL Agent and the Java Interpreter can use either of two interfaces: WRB or CGI. Which interface is used is determined by the mapping of the “file” the URL requests; this mapping is part of the WebServer configuration.

How URLs Specify Applications

Once the Listener has determined that a URL specifies that an application is to be executed on the server, it interprets the URL to extract path information and arguments to be passed to the application on start-up.

URLs that specify applications are split into three different parts:

- The virtual path
- The extra path information
- The query string

The syntax is as follows:

```
virtual_path extra_path_information?query_string
```

Here are the explanations of the syntax components:

virtual path

This is similar to a path you would use to access a regular document or image. That is, it is a pathname that identifies the application that you want executed.

extra path information

This is optional additional information embedded in the URL after the pathname. This consists primarily of various environment variables that you can use to pass information to the application.

query string

This is another optional part of the URL. This is used to directly supply parameter values (as opposed to environment variable values) to the requested application. The parameters are specified in the following form:

```
<parameter name>=<value>
```

Example of a URL Invoking an Application

This section provides an example of a URL that invokes a PL/SQL Agent to access the database. Here is the URL in question:

```
http://www.nhl.com:8080/ows-bin/nhl/owa/hockey_pass?person=Gretzky
```

1. The substring http://www.nhl.com:8080 in the above URL signals the Web browser to connect to the www.nhl.com host's port 8080 using the HTTP protocol.
2. When the Oracle Web Listener that is running on www.nhl.com receives the request, the substring /ows-bin/nhl/owa signals the Web Listener to connect to the PL/SQL Agent instead of returning a file to the browser as

it normally would have done with a static HTML document.

3. The URL is processed as follows:

   The Listener is configured so that the string /ows-bin/nhl/owa causes it to invoke the WRB Dispatcher. The WRB Dispatcher understands “owa” to mean it should pass execution to a WRBX interfaced to the PL/SQL Service. It passes the WRBX the following information in the form of CGI environment variables (For the sake of compatibility, WRB understands CGI environment variables):

   - The string /ows-bin/nhl/owa is passed as the environment variable SCRIPT_NAME. The PL/SQL Agent parses the SCRIPT_NAME to extract the DCD, which is nhl The DCD is passed as the directory name immediately preceding /owa, but is actually a file containing database connection information; owa is the name of the PL/SQL Agent itself. The use of DCDs is specific to the PL/SQL Agent.
   - The string /hockey_pass is passed as the environment variable PATH_INFO. This indicates that hockey_pass is the specific application to be executed. This application is a PL/SQL procedure stored in the database, that the PL/SQL Agent executes.
   - The string “person=Gretzky” is passed as the environment variable QUERY_STRING. This indicates that the value “Gretzky” is to be passed to the hockey_pass application for the parameter person. This will correspond in name and be compatible in datatype to a PL/SQL parameter used in the application.

The PL/SQL Agent

The WebServer employs the PL/SQL Agent to execute PL/SQL procedures stored in the database. If you install the PL/SQL Developer's Toolkit with your PL/SQL Agent, you can use a set of predefined PL/SQL packages to generate HTML formatted output, leaving you free to focus on the logic of your application. Even using the PL/SQL Developer's Toolkit, you need a conceptual understanding of HTML. For example, you must know at what points in your page anchors are necessary, although you needn't write the actual code for the anchors. The PL/SQL Agent takes care of interfacing to the needed environment variables and generating the HTML code.

Database Connection Descriptors (DCDs)

Whenever a URL invokes the PL/SQL Agent, it specifies a DCD. Creating and maintaining the DCDs is the responsibility of the WebServer administrator. As an application developer, all you need be concerned with is generating URLs that specify the correct DCD to achieve the result you want. The following is the information the DCD provides:

- *username.*
- *password.*
- *ORACLE_HOME.*
- *ORACLE_SID.*
- *SQL*Net V2 Service Name or Connect String.*
- *owa_err_page.*
- *owa_valid_ports.*
- *owa_log_dir.*
- *owa_nls_lang.*

username. All SQL and PL/SQL statements are executed by the database under the auspices of some database user. This name identifies that user. The username determines the schema (logical section of a database) that the PL/SQL Agent connects to, the actions it can perform, the resources (disk space and so on) it can use, and the level of monitoring of its activities that the database performs.

password. This is provided if you want to password protect the DCD, ensuring that only certain people connect as the user identified by username. Keep in mind, however, that the users who connect to the database as username still will be able to do only what the application gives them the ability to specify through URLs.

ORACLE_HOME. This is the root of the Oracle7 Server code tree in the OS file system.

ORACLE_SID. If the PL/SQL Agent is to connect to a local database, this determines that database. If the WRB is used, the request will be handed off to a process connected already to this database, if possible

SQL*Net V2 Service Name or Connect String. If the PL/SQL Agent is to connect to a remote database, this determines that database.

owa_err_page. This is the pathname of the HTML document that the PL/SQL Agent is to return to the client's browser when an error occurs in the PL/SQL procedure that the PL/SQL Agent invoked. This is the actual path as

understood by the OS, not the virtual path as configured in the Web Listener

owa_valid_ports. The valid Web Listener network ports to which the PL/SQL Agent will respond. The network port is, of course, specified in the URL. You must make sure that the URLs your applications generate provide port numbers that are valid for the WRB Service or CGI program they specify.

owa_log_dir. The directory where the PL/SQL Agent writes its error file. The error file differs from the page in that it is for the WebServer administrator's attention, rather than the clients.

owa_nls_lang. The NLS_LANG of the Oracle7 database to which the PL/SQL Agent connects. If not specified, the PL/SQL Agent administration program looks up the database NLS_LANG when the service is submitted. The NLS_LANG indicates the language to be used.

How the PL/SQL Agent Uses Environment Variables

Note: These are CGI standard environment variables. However, the PL/SQL Agent can use them whether it is invoked through *CGI* or through the Web Request Broker (WRB).

The PL/SQL Agent uses the environment variables shown below:

t

Variable	Contains
REQUEST_METHOD	GET or POST
PATH_INFO	the name of PL/SQL procedure to invoke
SCRIPT_NAME	contains the DCD the PL/SQL Agent is to use when logging on to Oracle7
QUERY_STRING	parameters to the PL/SQL procedure (for GET method only. POST method parameters are passed via standard input.)

CGI Variables Used by the Oracle PL/SQL Agent

Passing Parameters to PL/SQL

A PL/SQL procedure usually requires parameters in order to execute and generate the appropriate HTML document. The following section discusses several key concepts and tips that a PL/SQL developer should understand with respect to how parameters get passed to the specified PL/SQL routine.

These key concepts and tips are:

- *Getting Parameters from the Web Browser to the PL/SQL Agent*
- *Passing Parameters Using an HTML Form*
- *Providing Default Parameter Values*
- *Multivalued Parameters*
- *Overloading Procedures*

Getting Parameters from the Web Browser to the PL/SQL Agent

Depending on the REQUEST_METHOD used, parameters are passed from the Web Browser to the Web Listener to the PL/SQL Agent in one of two ways:

- Through the QUERY_STRING environment variable. If the GET method is used by the Web browser, the Web Listener passes the parameters to the PL/SQL Agent in this environment variable.
- Through standard input. If the POST method is used, the Web Listener passes the parameters to the PL/SQL Agent using standard input.

It is transparent to the PL/SQL procedure that is the actual consumer of these parameter(s) which method is used to pass the parameters from the Web Listener to the PL/SQL Agent and which protocol, the WRB or CGI, is used. This is an important feature of the Oracle PL/SQL Agent: the PL/SQL programmer need not be aware of whether GET or POST is used and need not be concerned with parsing either the QUERY_STRING environment variable or standard input. Thus, the PL/SQL programmer can concentrate on what he or she knows best: developing the logic to extract data from the Oracle database, based on pre-parsed parameters passed by the Oracle PL/SQL Agent.

It is recommended that you use POST whenever possible. GET is the method used for links and non-form URLs. For HTML forms, one has a choice. Because the GET method uses operating system environment variables, there are limits on the length of the QUERY_STRING.

Passing Parameters Using an HTML Form

The following example is analogous to the one in the previous section, except that it uses an HTML form that employs the POST REQUEST_METHOD.

```
<FORM METHOD="POST" ACTION="http://www.nhl.com:8080/ows-bin/nhl/owa/
hockey_pass">
```

Please type the name of the person you wish to search for:

```
<INPUT TYPE="text" NAME="person"><P> To submit the query, press this
button: <INPUT TYPE="submit" VALUE="Submit Query">. <P> </FORM>
```

The above form does the same thing as the previous example, except that instead of populating the QUERY_STRING environment variable with "person=Gretzky," the Web Listener writes "person=Gretzky" to standard input.

Note that the name of the HTML input variable, in this case "person", has to be the same as the PL/SQL parameter it is to match.

The PL/SQL procedure that is the recipient of the above parameters follows:

```
create or replace procedure hockey_pass (person in varchar2) is
    n_assists integer;
 begin
    select num_assists into n_assists
       from hockey_stats
    where name=person;
    htp.print(person||' has '||to_char(n_assists)||' assists this
season');
 end;
```

Providing Default Parameter Values

If you cannot guarantee that a value will be passed from a Web Browser for a particular PL/SQL procedure parameter, then you should give the parameter a default value. For example:

```
create or replace procedure showvals(a in varchar2 DEFAULT NULL,
                                     b in varchar2 DEFAULT NULL) is
begin
    htp.print('a = '||a||htp.br);
    htp.print('b = '||b||htp.br);
 end;
```

Suppose the PL/SQL Agent receives a request to call procedure showvals with no value for "a" and the value of 'Hello' for "b", and there was no DEFAULT NULL clause in the procedure's definition. Then the request would generate an error with the following message:

```
OWS-05111: Agent : no procedure matches this call
   OWA SERVICE: test_service
```

```
   PROCEDURE: showvals
  PARAMETERS:   ==========   B:    Hello
```

By “defaulting” the parameters, the above request would properly output:

```
a = <BR>
b = Hello<BR>
```

which to the end user would look like:

```
a =
b = Hello
```

Multivalued Parameters

Generally, you need not be concerned with the order in which the Oracle PL/SQL Agent receives parameters from a URL. The only case where it might be relevant is when passing multiple values for the same form field. In this case, all the values for that form field should be together, and if the order of the values in that field is significant, that order must be preserved in the URL.

There are a number of instances where you can have such multiple values for the same HTML variable. The HTML tag “SELECT” allows users to select from a set of possible values for an HTML form field. If the SIZE parameter of that SELECT tag is greater than one, the form allows multiple selection, and the user can choose to select more than one value. For example, if you ask a user to indicate her hobbies, she may well have more than one. In this case, you pass the multiple values to a single PL/SQL parameter, which must be a PL/SQL table. A PL/SQL table is a data structure similar to an array.

Example of a Multivalued Field

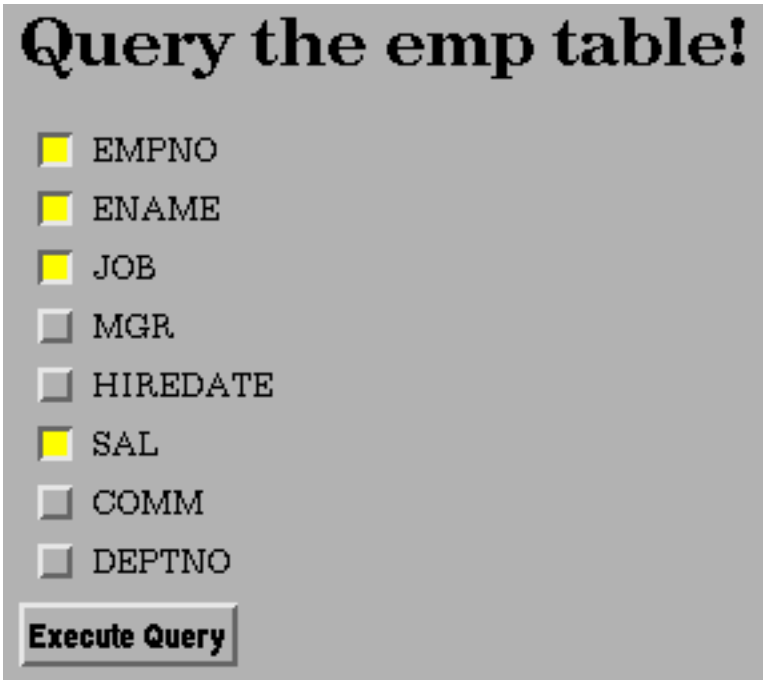
Another case where one has a set of values corresponding to a single form field is shown in this example:

QUERY_FORM prints an HTML page with all the columns for the specified table. Invoke the procedure from a Web Browser with a URL like: http://yourhost:port_num/service_name/owa/query_form?the_table=emp

```
create or replace procedure query_form(the_table in varchar2) is
     cursor cols is
    select column_name
          from user_tab_columns
    where table_name = upper(the_table);
 begin
    htp.htmlOpen;
    htp.headOpen;
```

```
   htp.htitle('Query the '||the_table||' table!');
   htp.headClose;
   htp.bodyOpen;
   -- Use owa_util.get_owa_service path to automatically retrieve
   htp.formOpen(owa_util.get_owa_service_path||'do_query');
   -- Put in the table as a hidden field to pass on to do_query
   htp.formHidden('the_table', the_table);
   -- Put in a dummy value, as we cannot DEFAULT NULL a PL/SQL table.
   htp.formHidden('COLS', 'dummy');
   for crec in cols loop
      -- Create a checkbox for each column. The form field name
      -- will be COLS and the value will be the given column name
.     -- Will need to use a PL/SQL table to retrieve a set of
      -- values like this. Can use the owa_util.ident_arr type
         -- since the columns are identifiers.
      htp.formCheckbox('COLS',crec.column_name);
      htp.print(crec.column_name);
      htp.nl;
   end loop;
   -- Pass a NULL field name for the Submit field; that way, a
   -- name/value pair is  not sent in.  Wouldn't want to do this
    -- if there were multiple submit buttons.
   htp.formSubmit(NULL, 'Execute Query');
      htp.formClose;
   htp.bodyClose;
   htp.htmlClose;
end;
```

Invoking this procedure brings up a page that looks like this:



In this example, the user has already selected to query the EMPNO, ENAME, JOB, and SAL columns:

Here is a procedure to process this form submission:

```
-- DO_QUERY executes the query on the specified columns and
 -- tables.The OWA_UTIL.IDENT_ARR datatype is defined as:
 --  -- type ident_arr is table of varchar2(30) index by binary_integer
  create or replace procedure do_query(the_table in varchar2,
                                                                    cols
in owa_util.ident_arr) is
    column_list varchar2(32000);
    col_counter integer;
    ignore      boolean;
begin
 -- For PL/SQL tables, have to just loop through until you hit
 -- no_data_found. Start the counter at 2 since we put in
 -- a dummy hidden field.
    col_counter := 2;
```

```
   loop
       -- build a comma-delimited list of columns
      column_list := column_list||cols(col_counter)||',';
       col_counter := col_counter + 1;
    end loop;
exception
    when no_data_found
    then
        -- strip out the last trailing comma
        column_list := substr(column_list,1,length(column_list)-1);
         -- print the table - assumes HTML table support
        ignore := owa_util.tablePrint(the_table,
 'BORDER',
OWA_UTIL.HTML_TABLE,  column_list);
 end;
```

Then, after selecting the “Execute Query” button, the user would see the following:

EMPNO	ENAME	JOB	SAL
7369	SMITH	CLERK	800
7499	ALLEN	SALESMAN	1600
7521	WARD	SALESMAN	1250
7566	JONES	MANAGER	2975
7654	MARTIN	SALESMAN	1250
7698	BLAKE	MANAGER	2850
7782	CLARK	MANAGER	2450
7788	SCOTT	ANALYST	3000
7839	KING	PRESIDENT	5000
7844	TURNER	SALESMAN	1500
7876	ADAMS	CLERK	1100
7900	JAMES	CLERK	950
7902	FORD	ANALYST	3000
7934	MILLER	CLERK	1300

If you cannot guarantee that at least one value will be submitted for the PL/SQL table, it is a good idea to use a hidden place-holder variable as the first value. The reason is that you cannot provide a default value for a PL/SQL table, and a call to this procedure with just one argument (the_table) would cause the PL/SQL Agent to generate an error.

Note that the PL/SQL Agent can only pass parameters to PL/SQL tables that have a base type of VARCHAR2. This should not provide a significant limitation, as the PL/SQL type VARCHAR2 is the largest PL/SQL datatype, with a maximum length of 32K (32767 bytes). The values can then be explicitly converted to NUMBER, DATE, or LONG within the stored procedure (using TO_NUMBER or TO_DATE - no conversion needed for LONGs).

Overloading Procedures

As explained in the *Overview of the Oracle7 Server, SQL, and PL/SQL* under *Overloading Subprograms*, PL/SQL allows you to overload procedures and functions that are in PL/SQL packages. In overloading, multiple procedures or

functions have the same name, but are distinguished by the fact that they take different parameters. For example:

```
create or replace package overload is
     procedure proc1(charval in varchar2);
     procedure proc1(numval  in number);
end;
 create or replace package body overload is
     procedure proc1(charval in varchar2) is
     begin
        htp.print('The character value is '||charval);
     end;
      procedure proc1(numval  in number);
        htp.print('The number value is '||numval);
     end; end;
```

Note: The PL/SQL Agent can use this functionality with the following restriction: if two procedures take the same number of parameters, those parameters must differ in name as well as datatype (normally a difference in datatype suffices to distinguish the parameters). For example:

```
create or replace package overload is
     procedure proc1(val in varchar2);
     procedure proc1(val in number);
end;
```

If executed directly from SQL, this would be acceptable, but when the PL/SQL Agent attempts to determine which procedure to call, it is not able to distinguish between the two and will generate an error.

This limitation is imposed by the lack of HTML form datatypes

Oracle PL/SQL Agent Error Handling

There are two types of errors that the Oracle PL/SQL Agent handles:

- *Application Errors*. These are the responsibility of the application developer.
- *System Errors*. These are usually the responsibility of the WebServer administrator, although they also occur when application errors are not handled.

Application Errors

Application errors are specific to the PL/SQL application. All PL/SQL procedures you write should have their own exception handling (see *Handling Exceptions*) that produces the appropriate output in HTML form.

Because the PL/SQL Agent does not read the HTML output to determine its content, and properly handled exceptions themselves generate HTML error messages, handled exceptions are transparent. As far as the PL/SQL Agent is concerned, if the PL/SQL code generates HTML output, the operation was successful. The user will see whatever handled exception message is generated by the PL/SQL procedure.

System Errors

System errors are detected by the Oracle PL/SQL Agent itself. These are errors that occur when the PL/SQL Agent is unable to launch the PL/SQL procedure or when a PL/SQL exception is not handled by the stored procedure, causing the exception to be propagated back to the PL/SQL Agent as a system error. This causes a standard HTML error document to be returned to the browser.

For example, if the Oracle PL/SQL Agent cannot make a connection to the Oracle7 Server, the PL/SQL procedure cannot run and a system error occurs. The PL/SQL Agent then returns a default error message to the browser or returns a customized HTML error page (if one was previously configured as part of the DCD using the OWA_ERR_PAGE parameter).

The Java™ Interpreter

The Java Interpreter is a part of the Oracle WebServer that interprets and executes Java and sends the HTML output to the client's browser as a Web page. The Java code executed on the server also includes a type wrapper for Java applets, so that you can store these in the database, retrieve them through the Java Interpreter, and send them to the client for execution, as applets embedded in a Web page. You can also store and retrieve Java applets for execution on the client using the PL/SQL Agent by simply treating the applets as data.

Oracle WebServer provides a set of Java classes to enable you to access the database and to generate HTML dynamically. These are called the Java Web Developer's Toolkit™. Using these Java classes, you can make PL/SQL calls from within your Java application, effectively combining the strengths of the two languages.

Generally speaking, you should use Java to handle multimedia operations and to interface to objects on the net, and you should use PL/SQL for interfacing to the database. Whether this means using the PL/SQL Agent or the Java Interpreter will largely be determined by the following:

- Whether you want to pull all of your data from the database (as the PL/SQL Agent does) or whether you also want to access other data services such as the OS file system.
- The balance of database-intensive to net or multimedia intensive operations in your application.

To use the Java Web Developer's Toolkit, import the following into your Java code:

- *oracle.html*.* contains the objects for dynamic HTML generation.
- *oracle.rdbms*.* contains the objects for database access.
- *oracle.plsql*.* contains the objects for PL/SQL access.

Obviously, if you only want to perform some of these functions, you need only import some of these objects. For more information, see *Database Access from Java*, *Example of Java Database Access*, *Dynamic HTML from Java*, and *Java Dynamic HTML Examples*.

Database Access from Java

Using Java to call *PL/SQL* circumvents the *PL/SQL Agent*, and this has a number of consequences, among them the following:

- PL/SQL Agent processes invoked through the WRB (but not CGI) connect to the database automatically at startup. Java processes connect to the database when necessary. This gives a significant performance advantage to the PL/SQL Agent.
- The PL/SQL Agent uses *Database Connection Descriptors (DCDs)* to control the privileges an application runs under and the database schema to which it connects in a generalized and application-independent way. In Java, database connections are coded into the application.

Nonetheless, Java can do many things that PL/SQL cannot, like access local files and manipulate multimedia objects. If you need this functionality executed on the server, the Java Interpreter is the way to go.

Creating Package Wrappers

The Java Interpreter interfaces to the Oracle7 Server by running *PL/SQL packages* or standalone PL/SQL procedures and functions. Each package an application is to run must have a *package wrapper*, which is a Java class containing methods to call that package's procedures and functions. Standalone procedures and functions are all wrapped in a single wrapper. Once you have identified or created the PL/SQL packages your applications needs, you can create the package wrappers for them by running the *pl2java* utility as follows:

```
pl2java [flags] username/password[@connect-string] packagename...
```

This utility creates a wrapper class for each package given as an argument to the command. When your application is run, it creates an instance of this class to interface to the package. If you have standalone procedures or functions in your applications, run the *pl2java* utility without any package names, but using the class flag as explained below. This will create a single class wrapper for all the standalone procedures and functions you use.

Here are the component definitions:

flags

Options that control how the wrappers will be created. These are explained below.

username

The name of the Oracle database user that owns the packages.

password

The password for the Oracle user identified by *username*.

connect-string

The string that identifies the local or remote database where the packages

are located. For local databases, this is the Oracle SID, as described in the *Oracle7 Server Administrator's Guide.* For remote databases, this is the SQL*Net Connect String, as described in *Understanding SQL*Net.*

package name...

A list of all the packages that your application references in the schema identified by username. To wrap standalone procedures and functions you must omit this component and must use the class flag to name the class wrapper that will be created. You should not include the containing schemas in the package names. It is good practice to keep all the packages, procedures, and functions you want to use in one schema.

flags

All of the flags that *pl2java* uses are optional, except, under certain conditions, class. Here are the descriptions of the flags:

-help

Provides help information.

-d <dir>

Sets the directory where the wrapper classes will be stored. The default is the current directory.

-package <packagename>

Sets the Java package to which the wrapper classes belong.

-class <class>

Sets the Java class to which the wrappers will belong. If the *pl2java* utility is run against packages, this flag is optional. Java classes based on packages inherit by default the names of the packages they encapsulate.This flag can override that default, but it only applies to the first package named in the command. If the wrappers are being created for standalone procedures and functions, then this flag is mandatory, and all procedures and functions named in the command are grouped into the single class named by this flag.

The names of the classes follow the capitalization given in the command. Since PL/SQL is not case-sensitive, this capitalization need not follow that actually given in the PL/SQL code itself.

There are certain PL/SQL datatypes that the *pl2java* utility cannot encapsulate. These are shown below, along with the recommended substitutes, if any:

Disallowed PL/SQL Datatype	**Substitute PL/SQL Datatype**
POSITIVE	BINARY INTEGER
PL/SQL table of BINARY INTEGER, NATURAL or POSITIVE	PL/SQL table of NUMBER
PL/SQL table of LONG	PL/SQL table of CHAR or VARCHAR2
PL/SQL table of BOOLEAN	PL/SQL table of NUMBER, *treat 0 as false, 1 as true*
ROWID	*none*
MSLABEL	*none*
PL/SQL table of ROWID	*none*
PL/SQL table of MSLABEL	*none*

Application Structure

Your Java program will be a Java class with a public static method called "main", which takes an array of strings, and various methods that are part of the wrapper classes. All the methods in the *Session* class throw *ServerException*. This exception is triggered when a database error occurs during the execution of the PL/SQL procedure or function. When invoking these methods, you should handle the exception appropriately.

The steps your application must go through are as follows:

1. It must create an object of type *Session* to handle the database connection. All of the operations performed during this session must be called from within this object. When the object exits, the database connection is marked for termination by the Java garbage collector. Since connecting to the database takes time, try to minimize the number of connections by grouping into one *Session* object all of the operations that involve a given schema.
2. For each PL/SQL package used in the application, it must create one instance of the packagewrapper subtype created for that PL/SQL package by the *pl2java* utility.

3. For each parameter of a PL/SQL package, it must create an instance of the Java variable that matches that parameter. All classes that encapsulate PL/SQL values have *toString*() methods. Therefore, you can concatenate the PL/SQL values directly in Java strings using Java's "+" concatenation operation.
4. Unlike Java and most languages, but like SQL, PL/SQL uses *Nulls and Three-Valued Logic* to deal with missing information. NULLs should be dealt with differently than known values. When you access a NULL from Java, it throws the *NullValueException* runtime exception. Therefore, you should account for this whenever a PL/SQL value may be NULL. You can do this in either of the following two ways:
   - Using the *isNull* method to check directly whether the value is NULL.
   - Enclosing the operation in a "try {...} catch (...) {...}" block that traps the *NullValueException*.

You must invoke the *Session setProperty* method to set parameters such as ORACLE_HOME that the database session requires. For more information on this method, refer to the standard Java hypertext reference included with this document.

You should try to logoff from the database explicitly when the session is no longer needed. Although Java garbage collection disconnects the session when the *Session* object is exited, this does not necessarily happen right away. Java does not perform garbage collection until resources are low or the program idles. Therefore, it is better to disconnect the session to free up the database connection resource immediately.

Example of Java Database Access

This section outlines a Java program that uses a PL/SQL package to look up all employees from a database and generates a report in HTML. In our example, you have an "EMP" database table and an "*Employee*" PL/SQL package that looks up employee information from that table. This table and package are created by user "scott" with password "tiger" in database "HR_DB".

Here is the SQL and PL/SQL source code that defines the objects used by this example.

```
CREATE TABLE EMP (
  emp_nameVARCHAR2(30) NOT NULL,
  emp_numberNUMBER(10),
  emp_deptVARCHAR2(30) NOT NULL
);
CREATE OR REPLACE package Employee as
  type string_table is table of varchar2(30) index by binary_integer;
```

```
  type number_table is table of number(10) index by binary_integer;

  function count_employees(
    dept_name   in  varchar2
  ) return number;

  procedure list_employees(
    dept_name   in varchar2,
    employee_name   out   string_table,
    employee_no  out  number_table
  );

end;

CREATE OR REPLACE package body Employee as

  function count_employees(
    dept_name  in  varchar2
  ) return number as
    employee_count  number;
  begin

    select count(*)
    into   employee_count
    from   EMP
    where  EMP_DEPT = dept_name;

    return employee_count;

  end;

  procedure list_employees(
    dept_name        in varchar2,
    employee_name    out string_table,
    employee_no      out  number_table
  ) as
    inumber;
    cursor employee_rec(dept_name varchar2) is
    select EMP_NAME, EMP_NUMBER
    from   EMP
    where  EMP_DEPT = dept_name;
  begin

    i := 1;
    for employee in employee_rec(dept_name) loop
      employee_name(i) := employee.EMP_NAME;
      employee_no(i)   := employee.EMP_NUMBER;
      i := i + 1;
    end loop;

  end;

end;
```

For reference, the commands themselves are considered SQL, but the code inside the packages is PL / SQL. Since PL / SQL is a superset of SQL, you can think of it all as PL / SQL.

This example uses the *oracle.html* package to generate dynamic HTML. That package is documented under *Dynamic HTML from Java*.

The main program

In this example, the program is called *EmployeeReport*. This program will be saved in a file called *EmployeeReport.java*.

```
import oracle.html.*;

public class EmployeeReport {

  public static void main (String args[]) {

    HtmlHead hd = new HtmlHead(”Employee Listing”);
    HtmlBody bd = new HtmlBody();
    HtmlPage hp = new HtmlPage(hd, bd);
    hp.printHeader();
    hp.print();

  }
}
```

Generating the Java Wrapper Class for the *Employee* Package

To access the *Employee* package, you must generate a Java wrapper class for it. This encapsulates the package as a Java class. The functions and procedures in the package will appear as methods in the wrapper class under the same names.

PL/SQL datatypes are encapsulated by Java objects. For example, a PL/SQL VARCHAR2 string is encapsulated by a Java *PStringBuffer* object, and a PL/SQL number is encapsulated by a Java *PDouble* object.

To generate the Java class wrapper for *Employee* package, execute the following command at a command prompt:

```
pl2java scott/tiger@HR_DB Employee
```

This generates the class wrapper *Employee.class* in the current directory. The class will have 1 constructor and 2 methods for the *Employee* package:

```
public class Employee {

  public Employee(Session session) { ... }

  public PDouble count_employees(PStringBuffer dept_name)
    throws ServerException { ... }

  public void list_employees(PStringBuffer dept_name,
     PStringBuffer employee_name[],
     PDouble       employee_number[])
    throws ServerException { ... }
}
```

Notice that the constructor takes a *Session* object as a parameter to encapsulate the database connection. This is because each time a PL/SQL package, procedure, or function is accessed in a session, it is instantiated. In effect, a copy is created that is the private property of that session. This gives the package a stable state for that session, while leaving it free to have another state when called by another session.

Overriding Default Value Sizes

When a PL/SQL function returns a value whose size is variable - for example VARCHAR2, LONG, RAW, or LONG RAW - the size of the value is set by

default to 255 bytes. In the wrapper class, you may change the default size by setting the following variable for the PL/SQL function in question:

```
<function name>_<overload number>_return_length
```

The overload number is the number of other functions that exist with the same name as this one. To find out about overloading of functions in PL/SQL, see *Overloading Subprograms*. For more information on overload numbers specifically, see the *PL/SQL User's Guide and Reference*. You can find out what the overload number of a function is by using the Oracle7 Server standard package *dbms_describe*, as covered in the *Oracle7 Server Administrator's Guide*. For non-overloaded functions, the overload number is 0.

For example, assume the following PL/SQL package:

```
CREATE OR REPLACE package Employee as
  function employee_name (
    employee_numberinnumber
  ) return varchar2;

END;
```

By default, the length of the return value is 255 bytes. You can change this using the code shown below:

```
public class EmployeeReport {
  public static void main(String args[]) {
    Session session = new Session("scott", "tiger", "HR_DB");
    Employee employee = new Employee(session);
    PDouble pEmployeeNumber = new PDouble((double)12345);
    // Set VARCHAR2 return length for function employee_name to 50
employee.employee_name_0_return_length = 50;
    System.out.println("Employee ID: " + pEmployeeNumber +
       " name: " + employee.employee_name(pEmployeeNumber));   }
}
```

Similarly, when a PL/SQL function returns a PL/SQL table, its default length is 40. You can change that length by changing the following variable in the wrapper class:

```
<function name>_<overload number>_return_arraylength
```

Making a connection to the database

The first thing that you have to do in the *EmployeeReport* program is to connect to a database. You do this by using the following code to create a *Session* object:

```
import oracle.html.*;
import oracle.rdbms.*;// ADD: import Oracle classes which deal //with
database

public class EmployeeReport {

  public static void main (String args[]) {

    HtmlHead hd = new HtmlHead("Employee Listing");
    HtmlBody bd = new HtmlBody();
```

```
    HtmlPage hp = new HtmlPage(hd, bd);
    hp.printHeader();


    // ADD: defines Oracle session properties like ORACLE_HOME
    Session.setProperty(”ORACLE_HOME”, ”/user/oracle”);
    Session.setProperty(”TNS_ADMIN”, ”/user/oracle/network/admin”);

    // ADD: creates a database session and logon
    Session session;
    try {
      session = new Session(”scott”, ”tiger”, ”HR_DB”);
    } catch (ServerException e) {
      bd.addItem(new SimpleItem(”Logon fails: ” + e.getSqlerrm()));
      hp.print();
      return;
    }
    hp.print()
  }
}
```

To handle any errors raised, put the operation in a “try {...} catch (...) {... }” block and trap any *ServerException*s.

Invoking the *Employee* package

To invoke the *Employee* package, you need to create a new instance of the corresponding wrapper class. Then you can call the procedures and functions in the package by invoking the methods in the wrapper class. Add the following code to the program:

```
import oracle.html.*;
import oracle.rdbms.*;
import oracle.plsql.*;// ADD: import Oracle classes which deal //with PL/
SQL data types

public class EmployeeReport {

    public static void main (String args[]) {

    HtmlHead hd = new HtmlHead(”Employee Listing”);
    HtmlBody bd = new HtmlBody();
    HtmlPage hp = new HtmlPage(hd, bd);
    hp.printHeader();

    Session.setProperty(”ORACLE_HOME”, ”/user/oracle”);
    Session.setProperty(”TNS_ADMIN”, ”/user/oracle/network/admin”);

    Session session;
    try {
      session = new Session(”scott”, ”tiger”, ”HR_DB”);
    } catch (ServerException e) {
      bd.addItem(new SimpleItem(”Logon fails: ” + e.getSqlerrm()));
      hp.print();
      return;
    }

    // ADD: create a new instance of Employee package
    Employee employee = new Employee(session);

    // ADD: find the department name from the input parameter
    String deptName = null;
    if ((args.length < 1) || !args[0].startsWith(”DEPT=”)) {
      bd.addItem(new SimpleItem(”No department name given”));
      hp.print();
```

```
      return;
    } else {
      deptName = args[0].substring(5);
    }

    // ADD: create objects to encapsulate PL/SQL values that are
    // used as parameters
    PStringBuffer pDeptName = new PStringBuffer(30, deptName);
    PStringBuffer pEmployeeName[];
    PDouble       pEmployeeNumber[];
    PDouble  pEmployeeCount;

    // ADD: print report header
    bd.addItem("Department " + pDeptName + ":")
      .addItem(SimpleItem.Paragraph);

    // ADD: call Employee package to count the number of employees in
    // the department
    try {
      pEmployeeCount = employee.count_employees(pDeptName);
    } catch (ServerException e) {
      bd.addItem("Fail to retrieve employee information for department " +
 deptName + ": " + e.getSqlerrm());
      hp.print();
      return;
    }
    int employeeCount = (int)pEmployeeCount.doubleValue();
    if (employeeCount == 0) {
      bd.addItem("No employee found under department " + deptName);
      hp.print();
      return;
    }

    // ADD: allocate the arrays for employee names and numbers
    pEmployeeName     = new PStringBuffer[employeeCount];
    pEmployeeNumber   = new PDouble[employeeCount];

    // ADD: allocate the buffers to retrieve employee information
    for(int i = 0; i < employeeCount; i++) {
      // max length of employee name is 30 (characters)
      pEmployeeName[i]   = new PStringBuffer(30);
      pEmployeeNumber[i] = new PDouble();
    }

    // ADD: call Employee package to look up employees in the dept
    try {
      employee.list_employees(pDeptName, pEmployeeName, pEmployeeNumber);
    } catch (ServerException e) {
      bd.addItem("Fail to retrieve employee information for department " +
 deptName + ": " + e.getSqlerrm());
      hp.print();
      return;
    }

    // ADD: generate report
    DynamicTable tab = new DynamicTable(2);
    TableRow row = new TableRow();
    row.addCell(new TableHeaderCell("Employee Name"))
       .addCell(new TableHeaderCell("Employee Number"));
    tab.addRow(row);

    for (int i = 0; i < employeeCount; i++) {
      row = new TableRow();
      if (pEmployeeNumber[i].isNull())
row.addCell(new TableDataCell(pEmployeeName[i].toString()))
   .addCell(new TableDataCell("new employee"));
      else
row.addCell(new TableDataCell(pEmployeeName[i].toString()))
   .addCell(new TableDataCell(pEmployeeNumber[i].toString()));
      tab.addRow(row);
```

```
    }
    hp.addItem(tab);
    hp.print();

    // ADD: logoff from database
    try {
      session.logoff();
    } catch (ServerException e);
  }
}
```

There are a few aspects of the preceding code worth pointing out:

1. You must create an instance of Employee package before you can invoke the procedures and functions in it. When initiating the package, you need to specify the database session where the instance of the package is to be created.
2. Before you invoke a PL/SQL procedure or function, you have to create Java instance variables for the PL/SQL values that are to be used as parameters (like "pDeptName" in the above example). PL/SQL tables map to Java arrays. Remember to allocate the array as well as the individual elements in the array.
3. When you access a PL/SQL value, remember that the value may be NULL unless database constraints prevent this. If the value is NULL, and you try to retrieve it using the value methods (such as doubleValue() of PDouble), it throws a *NullValueException* runtime condition. It is better to ensure the value is not NULL before retrieving the value or to try the *NullValueException*.
4. All classes that encapsulate PL/SQL values have *toString*() methods and therefore can be concatenated. For an example of this, see the report-header-generation section above.
5. You should try to logoff from the database explicitly when the session is no longer needed. When a session is no longer needed, the session is disconnected when Java performs garbage collection. Java does not, however, guarantee that any garbage objects will be collected immediately when they become garbage. In fact, Java's garbage collector waits until the program idles, which for a busy Web site could be infrequent, or until resources are low, before it collects garbage objects. Therefore, it is better to issue a disconnect statement to free up database resources explicitly.

Dynamic HTML from Java

To generate dynamic HTML from within Java, you create various objects that use the interface *IHtmlItem*. All classes that generate dynamic HTML implement this interface, which has two simple methods: *toHTML* and *print*. Both of these methods produce the content of the object as HTML, but *toHTML* returns it as a string, whereas *print* sends it to system output. Therefore, you use *toHTML* when

you want the resulting HTML to be further processed by another method and use *print* when you want to output it. In effect, you build your Web page in a buffer with *toHTML* and flush the buffer with *print.*

The *oracle.html* package provides a standard set of classes based on HTML2, HTML3, and popular browser-specific extensions. You are not limited to these, however. You can easily create your own customizable HTML classes by deriving them from the *CompoundItem* or *Container* classes. The *oracle.html* package also has the intelligence to generate output that is optimized for the browser at hand. For example, a browser that does not support tables will get table data in the form of preformatted strings.

In some cases, interfaces have been used to specify the attributes of HTML tags. This was done to simplify cases where tag assignments can be complex or where similar arguments are used by several types of tags.

If you have a body of HTML you want to use repeatedly, you can encapsulate it in an object of class *Compounditem* and thereafter treat it as a single *HTMLitem.*

The HTML tags that you can dynamically generate using the supplied objects are listed below:

HTML FEATURE	JAVA OBJECTS THAT GENERATE
headings	HTMLHead
page breaks	HTMLPage
main body of page	HTMLBody
Comments	Comment
links	Link
anchors	Anchor
client- side Java applets	Applet
checkboxes	CheckBox
forms	Form
lists	Java classes exist for various types
frames	Frameset; Frame
hidden fields	Hidden
GIF graphics	Image
select options	Select; Option
passwords	PasswordField
radio buttons	Radio
tables	Table; DynamicTable; TableRow; TableCell; TableDataCell; TableHeaderCell; TableRowCell
text areas	TextArea; TextField

As you can see, the Java objects that generate the main structural HTML tags begin with HTML; others are chiefly named for the tags they generate.

The general procedure is to use the first three objects to define the basic structure of your generated Web page to then to use the *AddItem* method to add *HTMLItem*s to the body.

Java Dynamic HTML Examples

Here are examples of how to dynamically generate some HTML text using The Java Interpreter.

The following is a basic Java program that produces an HTML page whose title and content are both the famous “Hello World!”:

```
import oracle.html.*;

public class HelloWorld {

public static void main (String args[]) {

  // Create an HtmlHead Object titled ”Hello World!”
  HtmlHead hd = new HtmlHead(”Hello World!”);

  // Create an HtmlBody Object
  HtmlBody bd = new HtmlBody();

  // Create an HtmlPage Object
  HtmlPage hp = new HtmlPage(hd, bd);

  // Adds a simple string ”Hello World” in this page
  bd.addItem(”Hello World!”);

  // Print out the content of this Page
  hp.print();
}

}
```

The following Java code creates an HTML anchor:

```
// Creates an anchor
Anchor anchor = new Anchor(”expire_date”, new SimpleItem(”Expire Date: 02/
96”));
```

The following Java code creates an HTML form:

```
// Create a form object
Form form = new Form(”GET”, ”http://www.myhom.com/wrb/doit”);

// Create a TextField object and add it to the form
form.addItem(new TextField(”textfield”));

// Add the form object to the HtmlBody object
bd.addItem(form);
```

The following Java code creates an HTML table. To make this example realistic, we have added some user-defined functions:

```
// Some user-defined functions
Product product = getFirstProduct();

// create a dynamic table with 2 columns
DynamicTable tab = new DynamicTable(2);

// create the rows and add them to the table
TableRow rows[] = new TableRow[NUM_ROWS];

for (int i=0; i< NUM_ROWS; i++) {
  // allocate TableRow
  rows[i] = new TableRow();
  // populate row with data
  rows[i].addCell(new TableHeaderCell(product.getProductID()))
         .addCell(new TableDataCell(product.getProductDescription()))
```

```
  // add them to Table
  tab.addRow(rows[i]);
}
```

The following Java code creates an HTML menu:

```
// Create a MenuList Object
MenuList menulist = new MenuList();
// Add new items to the list
menulist.addItem(new SimpleItem("Menu Item 1"))
        .addItem(new SimpleItem("Menu Item 2"));
// Add the list object to the body
bd.addItem(menulist);
```

The following Java code creates an HTML definition list, encapsulated in a *Container* object:

```
// Creates a new Container Object
Container dterms = new Container();
dterms.addItem(new SimpleItem("DefTerm1.1"));
dterms.addItem(new SimpleItem("DefTerm1.2"));
DefinitionList dl = new DefinitionList();
// Creates a new Definition List Object, note the first argument
dl.addDef(dterms, new SimpleItem("Definition1"));
```

The following Java code creates an HTML ordered list:

```
// Create a OrderedList Object
OrderedList orderedlist = new OrderedList();
// Add new items to the list
orderedlist.addItem(new SimpleItem("Ordered Item 1"))
           .addItem(new SimpleItem("Ordered Item 2"));
// Add the list object to the body (assuming it already exists)
bd.addItem(orderedlist);
```

The following Java code encapsulates a group of HTML tags as a single component that can be used repeatedly in a page:

```
// Create a new CompoundItem object
CompoundItem compoundItem = new CompoundItem();
// Set the default text attribute of all items in Compound Item
// Note that this operation cannot be done with a <b>Container</b>
compoundItem.setItal();
// Add a SimpleItem and a TextArea to the CompoundItem
compoundItem.addItem(new Simpleitem("How are you?").setBold())
            .addItem(new TextArea("response", 30, 10));
// Add the CompoundItem to the body object (assuming it's been created)
bd.addItem(compoundItem);
```

The following Java code specifies an applet of Java bytecode to be included in the Web page output for execution on the client's browser.

```
// Create a new Applet object with the following attributes:
// Applet file name:  "NervousText.class"
// Width of Applet Window:  400
// Height of Applet Window:  75
// Parameter:   Name="text", Value="This is an applet test"
Applet applet = new Applet("NervousText.class", 400, 75);
```

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

The LiveHTML Interpreter

Files that the WebServer is to parse for LiveHTML are internally of the following MIME type:

```
text/x-server-parsed-html
```

The WebServer Administrator normally will create file extensions that are synonyms for this type, and these are what you use in your application code. The default synonym is SHTML. The administrator can also specify HTML as a synonym, in which case all HTML files are parsed for LiveHTML. Unless all your HTML files actually use LiveHTML, this is a bad idea, as it degrades performance. The WebServer Administrator can also specify that some Web Listeners are to allow full LiveHTML, some crippled LiveHTML, and some no LiveHTML at all. An application developer needs to know which Listeners are which, so as to specify the correct port numbers in the URLs. You can find this out by examining the WebServer Manager

LiveHTML code is formatted as SGML comments, so that it is ignored should the file ever find its way to the browser unparsed. The format for LiveHTML codes, therefore, is the following:

```
<!--#command tag1="value1" tag2="value2" -->
```

The tags are arguments to the commands, most of which actually only accept one of the possible tags. The possible commands and their associated tags are as follows:

- *config*. This command sets parameters for how the file or script is to be parsed and therefore is normally the first LiveHTML command in a file. The possible tags are:
  - *errmsg*. This specifies the error message that is sent to the client if an error occurs while parsing this document. Here is an example:

```
<!--#config errmsg="A parse error occurred in the Hockey_Pass file"-->
```

  - *timefmt*. This specifies a date format. LiveHTML files frequently include timestamps. The conventions follow the *strftime* library call supported in most versions of Unix, even if the WebServer is not running on a Unix platform.
  - *sizefmt*. This specifies the format used when displaying a file size. The possibilities are *bytes*, which gives the absolute size in bytes, and *abbrev*, which gives the size in kilobytes or megabytes as appropriate.
  - *cmdecho*. This specifies whether non-*CGI* scripts subsequently executed have their output incorporated into this HTML page. The

possible values are *ON* and *OFF*. ON specifies that the output is included. The default is OFF.

- *cmdprefix*. Specifies a string that will be prepended to each line of the script output.
- *cmdpostfix*. Specifies a string that will be appended to each line of the script output.

- *include*. This command specifies that a file is to be included in the generated HTML page at this point. The file can any of the following:
  - another LiveHTML file like the current one.
  - a regular HTML file.
  - an ASCII file.

  which of these it is determined, as usual, by the extension. The possible tags are:
  - *virtual*. This gives a *virtual path* to the file. The directory mappings for virtual paths are set by the WebServer administrator using WebServer Manager.
  - *file*. This gives a pathname relative to the current directory. References to parent directories or uses of absolute pathnames are forbidden.
- *echo*. This gives the value of an environment variable. This variable is either one of the standard CGI environment variables or one of the LiveHTML extensions, which currently are all standard Server Side Include variables. There is only one tag, *var*, and it must be present. It provides the name of the variable. The LiveHTML environment variables are as follows:
  - *DOCUMENT_NAME*. The current filename.
  - *DOCUMENT_URL*. The virtual path to this file.
  - *QUERY_STRING_UNESCAPED*. If the client sent a query string, this is an unescaped version of it, with all shell-special characters escaped with \.
  - *DATE_LOCAL* The current date and local time zone in the format specified by the most recent *config timefmt* command.
  - *DATE_GMT*. Same as the above, but in Greenwich mean time.
  - *LAST_MODIFIED*. The last modification date of the file, given in the format specified in the last *config timefmt* command.
- *fsize*. This produces the size of the file in the format specified in the most recent *config filesize* command. Tags are the same as for *include*.

- *flastmod.* This produces the last modification date of the file in the format specified in the most recent *config timefmt* command. Tags are the same as for *include.*
- *exec.* This is the command to execute a script. The tags specify whether or not the script is CGI.
  - *cmd.* This specifies a non-CGI script. Execution is passed to the Operating System, and the given string is parsed as though it were entered at a command-line interface. The full path of the script must be given. The non-CGI environment variables specified under *echo* above can be referenced. Whether the output of the script is included in the HTML page that the parser outputs, is determined by the most recently executed *config cmdencho.*
  - *cgi.* This specifies a CGI script. The value given will be the *virtual path* of the script. *URL* locations are automatically converted into HTML anchors.

CHAPTER

5 The PL/SQL Developer's Toolkit Reference

This section describes the hypertext procedures, hypertext functions, and utilities that make up the Oracle WebServer Developer's Toolkit.

One of the main goals of the Oracle PL/SQL Agent is to eliminate the PL/SQL programmer's need to be intimately familiar with World Wide Web technology. To this end, the Oracle WebServer includes a Developer's Toolkit made up of several PL/SQL packages that minimize the programmer's need to know HTML syntax. Although the programmer is still required to have a working knowledge of HTML, by using the Toolkit he or she will not need to hard code the exact syntax of HTML tags into PL/SQL procedures. For instance, a programmer still needs to realize that an anchor tag is needed, but he or she doesn't need to know the exact sequence of characters needed to generate an anchor.

The Oracle WebServer Developer's Toolkit includes the following PL/SQL Packages:

Hypertext Procedures (HTP)

A hypertext procedure generates a line in an HTML document that contains the HTML tag that corresponds to its name. For instance, the htp.anchor procedure generates an anchor tag. The HTP package will be the most commonly used package of the three.

Hypertext Functions (HTF)

A hypertext function returns the HTML tag that corresponds to its name. However, it is not sufficient to call an HTF function on its own because the HTML tag is not passed to the PL/SQL Agent. The output of an HTF function must be passed to htp.print in order to actually be part of an HTML document. Thus, the following line:

```
htp.print(htf.italic('Title'));
```

is functionally equivalent to:

```
htp.italic('Title');
```

Every hypertext function (HTF) has a corresponding hypertext procedure (HTP). Thus, HTF functions are generally used only when the programmer needs to nest calls, such as:

```
htp.header(1,htf.italic('Title'));
```

In this example, 'htf.italic' will generate the following character string:

```
<I>Title</I>
```

This string is then passed to the 'htp.header' procedure and the following line will appear in the HTML document being formatted:

```
<H1><I>Title</I></H1>
```

OWA Utilities (OWA_UTIL)

This is a collection of useful utility procedures and functions. The purposes of these range from printing a signature tag on HTML pages to easy formatting of Oracle tables into HTML.

OWA

This is a set of procedures called only by the Oracle PL/SQL Agent itself. None of the subprograms in this package should be called directly from user-developed PL/SQL.

Pattern Matching Utilities (OWA_PATTERN)

This is a set of procedures and functions you can use to perform string matching and substitution with rich regular expression functionality.

Text Manipulation Utilities (OWA_TEXT)

This is a set of procedures, functions, and datatypes used by OWA_PATTERN for manipulating large data strings. They are externalized so you can use them directly if you wish.

Image Map Utilities (OWA_IMAGE)

This is a set of datatypes and functions for manipulating HTML image maps.

Cookie Utilities (OWA_COOKIE)

This is a set of datatypes procedures, and functions for manipulating HTML cookies.

Installing the Oracle WebServer Developer's Toolkit

To install the Developer's Toolkit, use the Oracle PL/SQL Agent DCD Administration forms. Both the DCD Creation and DCD Modification pages provide a checkbox for installing the Developer's Toolkit. Selecting this button and submitting the form will do the following:

- grant the CONNECT and RESOURCE roles to the OWA database user For more information on these roles or on granting roles, see "GRANT (roles)" in Chapter 4 of the *Oracle7 Server SQL Reference*.
- execute the OWAINS.SQL sql script, which can be found in the ows Administration directory. If run manually, the script should be run from SQL*DBA or from Server Manager. If you want to run it from SQL*Plus, see the header of the script for instructions.

  The OWAINS.SQL script installs all of the Developer's Toolkit packages.

Optimizing Multiple-DCD Installations

If your site has multiple PL/SQL Agent DCDs, you can minimize the amount of storage space used and enhance PL/SQL performance by doing the following:

1. Install the Developer's Toolkit in one database user's schema. This user becomes the toolkit owner.
2. Drop the Developer's Toolkit PL/SQL from the schemas of the OWA database users for other PL/SQL Agent DCDs, if you have already installed them.

   ```
   connect <user> / <password>
   drop package HTF;
   drop package HTP;
   drop package OWA_UTIL;
   drop package OWA;
   ```

3. Grant the system privilege EXECUTE on the PL/SQL packages to OWA database users for other PL/SQL Agent DCDs. For more information on this command, see "GRANT (system privileges)" in Chapter 4 of the

Oracle7 Server SQL Reference.

```
connect <toolkit owner> / <password>
grant execute on HTF to <user>;
grant execute on HTP to <user>;
grant execute on OWA_UTIL to <user>;
grant execute on OWA to <user>;
```

4. Create synonyms for the Developer's Toolkit PL/SQL packages in the schemas of all OWA database users who are specified in PL/SQL Agent DCDs. Synonyms are alternate names that make it possible for the packages to be referred to without being qualified by schema names. For more information, see "Ownership and Naming Conventions" in Chapter of this manual and see "CREATE SYNONYM" in Chapter 4 of the *Oracle7 Server SQL Reference.*

```
connect <user>/<password>
create synonym HTF for <Toolkit owner>.HTF;
create synonym HTP for <Toolkit owner>.HTP;
create synonym OWA_UTIL for <Toolkit owner>.OWA_UTIL;
create synonym OWA for <Toolkit owner>.OWA;
```

Security Note

PL/SQL procedures run with the privileges of the creator of the PL/SQL code. For the Developer's Toolkit, this is only an issue for the owa_util package. Two of the subprograms, showsource and tableprint, access user data. Granting execute privileges on this package to users allows those users to view the tables, views, and stored PL/SQL code of the owner of owa_util.

If this is a security issue for your installation, install the owa_util package separately for each OWA database user.

The scripts to do this are:

- PUBUTIL.SQL
- PRIVUTIL.SQL

Both scripts reside in the OWS Administration directory, and should be run in the order listed.

Procedure and Function Reference

This section describes each procedure and function in the htp, htf, and owa_util packages. Please note that for every htp procedure that generates HTML tags, a corresponding htf function exists with identical parameters. Note that defaulted parameters do not need to be passed.

The description of each procedure or function is broken down into the following parts. Note that items for which no "generates" entry is shown are available only as procedures.

Parameters Passed into Procedures and Functions

All parameters passed into a hypertext procedure or function are of data type varchar2 (varying-length character string), integer, or date. The data type is indicated by the first letter of the parameter's name, "c" for character (varchar2), "n" for number (integer), and "d" for date. For example:

`cname in varchar2`

The "c" in cname indicates a character data type (varchar2).

`nsize in integer`

The "n" in nsize indicates a number data type (integer).

`dbuf in date`

The "d" in dbuf indicates a date data type (date).

As in BNF notation, a vertical bar (|) in the syntax diagram means "or".

Note: Many HTML 3.0 tags have a large number of optional attributes that, if passed as individual parameters to the hypertext procedures or functions would make the calls quite cumbersome. In addition, some browsers support non-standard attributes. Therefore, each hypertext procedure or function that generates an HTML tag has as its last parameter *cattributes*, an optional parameter. This parameter enables you to pass the exact text of the desired HTML attributes to the PL/SQL procedure.

For example, the syntax for htp.em is:

```
htp.em (ctext, cattributes);
```

A call that uses HTML 3.0 attributes might look like the following:

```
htp.em('This is an example','ID="SGML_ID" LANG="en"');
```

This line would generate the following:

```
<EM ID="SGML_ID" LANG="en">This is an example</EM>
```

Print Procedures

The following print procedures are used in conjunction with htf functions to generate a line in the HTML document being constructed. They can also be passed hard-coded text that will appear in the HTML document as-is. The generated line is passed to the PL/SQL Agent, which sends it to standard

output. As documented in the CGI 1.1 specification, the Oracle Web Listener takes the contents of standard output and returns it to the Web browser that requested the dynamic HTML document.

htp.print

Syntax

htp.print (cbuf | dbuf | nbuf);

Purpose

generates a line in an HTML document.

Parameters

cbuf in varchar2 *or*
dbuf in date *or*
nbuf in number

Generates

Generates a line in an HTML document based on the value passed to it.

htp.prn

Alias for htp.print

Syntax

htp.prn (cbuf | dbuf | nbuf);

Purpose

Just like htp.print, but doesn't put a new line at the end of the value submitted.

htp.prints

Syntax

htp.prints (ctext);

Purpose

Generates a line in an HTML document and replaces all occurrences of the following special characters with the shown escape characters. If not replaced, the special characters would be interpreted as HTML control characters, and would produce garbled output.

‘<‘ with ’<‘
‘>’ with ’>‘
‘”‘ with ‘"‘
‘&‘ with ‘&‘

Parameters

ctext in varchar2

Generates

Generates a line in an HTML document based on the value passed to it. This procedure is the same as htp.print or htp.p but first replaces the special characters listed above with escape characters.

htp.ps

Alias for htp.prints

Structure Tags

The following tags are used to identify the major parts of an HTML document.

Note: Although this section shows hypertext procedures (HTP), all of them are also available as hypertext functions (HTF).

htp.htmlOpen

Syntax

htp.htmlOpen;

Purpose

Prints a tag that indicates the beginning of an HTML document

Parameters

none

Generates

```
<HTML>
```

htp.htmlClose

Syntax

htp.htmlClose;

Purpose

Prints a tag that indicates the end of an HTML document

Parameters

none

Generates

```
</HTML>
```

htp.headOpen

Syntax

htp.headOpen;

Purpose

Prints a tag that indicates the beginning of the HTML document head

Parameters

none

Generates

```
<HEAD>
```

htp.headClose

Syntax

htp.headClose;

Purpose

Prints a tag that indicates the end of the HTML document head

Parameters

none

Generates

```
</HEAD>
```

htp.bodyOpen

Syntax

htp.bodyOpen (cbackground, cattributes);

Purpose

Prints the tag that identifies the beginning of the body of an HTML document, and allows you to specify an image as the background of the document

Parameters

cbackground	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<BODY background="cbackground" cattributes>
```

Note: If cbackground and cattributes are NULL, this tag generates <BODY>.

Example

```
htp.bodyOpen ('/img/background.gif');
```
This line produces:
```
<BODY background="background.gif">
```

htp.bodyClose

Syntax

htp.bodyClose;

Purpose

Defines the end of the HTML document body

Parameters

none

Generates

```
</BODY>
```

Head Related Tags

The following procedure tags should be placed between the htp.headOpen and htp.headClose procedure tags.

Note: Although this section shows hypertext procedures (HTP), all of them are also available as hypertext functions (HTF).

htp.title

Syntax

htp.title (ctitle);

Purpose

Prints an HTML tag with the text you pass in as the value of TITLE. Most Web Browsers display the text value enclosed between <TITLE> and </TITLE> at the top of the document viewing window.

Parameters

ctitle in varchar2

Generates

`<TITLE>`*`ctitle`*`</TITLE>`

htp.base

Syntax

htp.base (ctarget, cattributes);

Purpose

Prints an HTML tag that records the URL of the document

Parameters

ctarget	in varchar2 DEFAULT NULL
cattributest	n varchar2 DEFAULT NULL

Generates

Inserts absolute pathname of current document.

htp.isindex

Syntax

htp.isindex (cprompt, curl);

Purpose

Creates a single entry field with a prompting text, such as "*enter value*,"then sends that value to the URL of the page or program.

Parameters

cprompti	n varchar2	DEFAULT NULL
curl	in varchar2	DEFAULT NULL

Generates

```
<ISINDEX PROMPT="cprompt" HREF="curl">
```

htp.linkRel

Syntax

htp.linkRel (crel, curl, ctitle);

Purpose

Prints the HTML tag that gives the relationship described by the hypertext link from the anchor to the target. This is only used when the HREF attribute is present. This tag indicates a relationship between documents, but does not create a link. To do that, use htp.anchor.

Parameters

crel	in varchar2
curl	in varchar2
ctitle	in varchar2DEFAULT NULL

Generates

```
<LINK REL="crel" HREF="curl" TITLE="ctitle">
```

htp.linkRev

Syntax

htp.linkRev (crev, curl, ctitle);

Purpose

Gives the relationship described by the hypertext link from the target to the anchor. This is the opposite of htp.linkRel. This tag indicates a relationship between documents, but does not create a link. To do that, use htp.anchor.

Parameters

crev	in varchar2	
curl	in varchar2	
ctitle	in varchar2	DEFAULT NULL

Generates

```
<LINK REV="crev" HREF="curl" TITLE="ctitle">
```

htp.meta

Syntax

htp.meta (chttp_equiv, cname, ccontent);

Purpose

Prints an HTML tag that identifies and embeds document meta-information that supplies the Web browser with information about the objects returned in HTTP.

Parameters

chttp_equiv	in varchar2
cname	in varchar2
ccontent	in varchar2

Generates

```
<META HTTP-EQUIV="chttp_equiv" NAM ="cname" CONTENT="ccontent">
```

Example

```
htp.meta ('Refresh', NULL, 120);
```

This line produces:

```
<META HTTP-EQUIV="Refresh" CONTENT=120>
```

which on some Web browsers will cause the current URL to be reloaded automatically every 120 seconds.

Body Tags

Body tags are used in the main text of your HTML page. They can format a paragraph, allow you to add hidden Comments to your text, and add images within the body of your HTML text.

Note: Although this section shows hypertext procedures (HTP), all of them are also available as hypertext functions (HTF)

htp.line

Syntax

htp.line (cclear, csrc, cattributes);

Purpose

Prints the HTML tag that generates a line in the HTML document. *csrc* enables you to specify a custom image as the source of the line.

Parameters

cclear	in varchar2	DEFAULT NULL
csrc	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<HR CLEAR="cclear" SRC="csrc" cattributes>
```

htp.hr

Alias for **htp.line**

htp.nl

Syntax

htp.nl (cclear, cattributes);

Purpose

Prints the HTML tag that inserts a new line

Parameters

cclear	in varchar2	DEFAULT NULL,
cattributes	in varchar2	DEFAULT NULL,

Generates

```
<BR CLEAR="cclear" cattributes>
```

htp.br

Alias for **htp.nl**

htp.header

Syntax

htp.header (nsize, cheader, calign, cnowrap, cclear, cattributes);

Purpose

Prints the HTML tag for a heading level, with the value of the heading level assigned in the nsize parameter. Valid levels are 1 through 6.

Parameters

nsize	in integer	
cheader	in varchar2	
calign	in varchar2	DEFAULT NULL
cnowrap	in varchar2	DEFAULT NULL
cclear	in varchar2	DEFAULT NULL
cattributesi	n varchar2	DEFAULT NULL

Generates

```
<Hnsize ALIGN="calign" NOWRAP CLEAR="cclear"
   cattributes>cheader</Hnsize>
```

Example

```
htp.header (1,'Overview');
```

produces `<H1>Overview</H1>`

htp.anchor

Syntax

htp.anchor (curl, ctext, cname, ctarget, cattributes);

Purpose

Prints the HTML tag for an anchor to be the start or end destination of a hypertext link. This anchor can accept several attributes, but either HREF or NAME is required. HREF specifies where to link to. NAME allows this tag to be a target of a hypertext link.

Parameters

curl	in varchar2	
ctext	in varchar2	
cname	in varchar2	DEFAULT NULL
ctarget	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<A HREF="curl" NAME="cname" TARGET = "ctarget" cattributes>ctext</A>
```

htp.mailto

Syntax

htp.mailto (caddress, ctext, cname, cattributes);

Purpose

Prints the HTML tag for an anchor with 'mailto' concatenated ahead of the mail address argument.

Parameters

caddress	in varchar2	
ctext	in varchar2	
cname	in varchar2	
cattributesi	n varchar2	DEFAULT NULL

Generates

```
<A HREF="mailto:caddress" cattributes>ctext</A>
```

Example

```
htp.mailto('pres@white_house.gov','Send Email to the President');
```

prints

```
<A HREF="mailto:pres@white_house.gov">Send Email to the President</A>
```

htp.img

Syntax

htp.img (curl, calign, calt, cismap, cusemap, cattributes);

Purpose

Prints an HTML tag that signals the browser to load an image to be placed into the HTML page. ALT allows you to specify alternate text to be shown while the image is being loaded, or instead of the image if the browser does not support images. The ISMAP attribute indicates that the image is an image map.

Parameters

curl	in varchar2	DEFAULT NULL
calign	in varchar2	DEFAULT NULL
calt	in varchar2	DEFAULT NULL
cismap	in varchar2	DEFAULT NULL
cusemap	in varchar2	DEFAULT NULL
cattributes i	n varchar2	DEFAULT NULL

Generates

```
<IMG SRC="curl" ALIGN="calign" ALT="calt" ISMAP  USEMAP="cusemap"
cattributes>
```

htp.para

Syntax

htp.para;

Purpose

Prints an HTML tag that indicates that the text previous to it should be formatted as a paragraph.

Parameters

none

Generates

```
<P>
```

htp.paragraph

Syntax

htp.paragraph (calign, cnowrap, cclear, cattributes);

Purpose

Prints the same HTML tag as htp.para except that parameters pass in exact alignment, leading, wrapping, and attributes.

Parameters

calign	in varchar2	DEFAULT NULL
cnowrap	in varchar2	DEFAULT NULL
cclear	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<P ALIGN="calign" NOWRAP CLEAR="cclear" cattributes>
```

htp.address

Syntax

htp.address (cvalue, cnowrap, cclear, cattributes);

Purpose

Prints an HTML tag that enables you to specify address, author and signature of document

Parameters

cvalue	in varchar2	
cnowrap	in varchar2	DEFAULT NULL
cclear	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<ADDRESS CLEAR="cclear" NOWRAP cattributes>cvalue</ADDRESS>
```

htp.Comment

Syntax

htp.Comment (ctext);

Purpose

Prints an HTML tag that allows you to store Comments or lines in HTML pages. These Comments are not visible to the end user.

Parameters

ctext in varchar2

Generates

```
<!-- ctext -->
```

htp.preOpen

Syntax

htp.preOpen (cclear, cwidth, cattributes);

Purpose

Prints an HTML tag that indicates the beginning of preformatted text in the body of the HTML page.

Parameters

cclear	in varchar2	DEFAULT NULL
cwidth	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Parameters

```
<PRE CLEAR="cclear" WIDTH="cwidth" cattributes>
```

htp.preClose

Syntax

htp.preClose;

Purpose

Prints an HTML tag that ends the preformatted section of text.

Parameters

none

Generates

```
</PRE>
```

htp.blockquoteOpen

Syntax

htp.blockquoteOpen (cnowrap, cclear, cattributes);

Purpose

Prints an HTML tag that precedes a paragraph of quoted text.

Parameters

cnowrap	in varchar2	DEFAULT NULL
cclear	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL;

Generates

```
<BLOCKQUOTE CLEAR="cclear" NOWRAP cattributes>
```

htp.blockquoteClose

Syntax

htp.blockquoteClose;

Purpose

Ends the <BLOCKQUOTE> section of quoted text.

Parameters

none

Generates

```
</BLOCKQUOTE>
```

htp.base

Syntax

htp.base(ctarget, cattributes);

Purpose

Prints an HTML tag that records the URL of the document. The ctarget attribute establishes a default window name to which all links in this document will be targeted.

Parameters

ctarget	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

BASE HREF="<current URL>" TARGET="target" cattributes>

htp.area

Syntax

htp.area(ccoords, cshape, chref, cnohref, ctarget, cattributes);

Purpose

Prints an HTML tag to specify the shape of a client-side image map region.

Parameters

ccoords	in varchar2	
cshape	in varchar2	DEFAULT NULL
chref	in varchar2	DEFAULT NULL
cnohref	in varchar2	DEFAULT NULL
ctarget	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

<AREA COORDS="ccoords" SHAPE="cshape" HREF="chref" NOHREF
TARGET="ctarget" cattributes>

htp.mapOpen

Syntax

htp.mapOpen(cname, cattributes);

Purpose

Prints an HTML tag to specify a set of regions in a client-side image map.

Parameters

cname	in varchar2
cattributes	in varchar2 DEFAULT NULL

Generates

<MAP NAME="cname" cattributes>

htp.mapClose

Syntax

htp.mapClose;

Purpose

Prints an HTML tag that ends the definition of a client-side image map.

Parameters

none

Generates

</MAP>

htp.bgsound

Syntax

htp.bgsound(csrc, cloop, cattributes);

Purpose

Prints an HTML tag to include background sound for a web page.

Parameters

csrc in varchar2
cloop in varchar2 DEFAULT NULL
cattributesin varchar2 DEFAULT NULL

Generates

<BGSOUND SRC="csrc" LOOP="cloop" cattributes>

htp.div

Syntax

htp.div(calign, cattributes);

Purpose

Prints an HTML tag to create document divisions.

Parameters

calign in varchar2 DEFAULT NULL
cattributesi n varchar2 DEFAULT NULL

Generates

<DIV ALIGN="calign" cattributes>

htp.listingOpen

Syntax

htp.listingOpen;

Purpose

Prints an HTML tag to indicate the beginning of fixed-width text in the body of an HTML page.

Parameters

none

Generates

<LISTING>

htp.listingClose

Syntax

htp.listingClose;

Purpose

Prints an HTML tag to end the fixed-width section of text.

Parameters

none

Generates

</LISTING>

htp.nobr

Syntax

htp.nobr(ctext);

Purpose

Prints an HTML tag to turn off line-breaking with a section of text.

Parameters

ctext in varchar2

Generates

<NOBR>ctext</NOBR>

htp.wbr

Syntax

htp.wbr;

Purpose

Prints an HTML tag to insert a soft linebreak within a section of NOBR text.

Parameters

none

Generates

<WBR>

htp.center

Syntax

htp.center(ctext);

Purpose

Prints a pair of HTML tags to center a section of text within a web page.

Parameters

ctexy in varchar2

Generates

<CENTER>ctext</CENTER>

htp.centerOpen

Syntax

htp.centerOpen;

Purpose

Prints an HTML tag to open a centered section of text within a web page.

Parameters

none

Generates

<CENTER>

htp.centerClose

Syntax

htp.centerClose;

Purpose

Prints an HTML tag to close a centered section of text within a web page.

Parameters

none

Generates

</CENTER>

htp.dfn

Syntax

htp.dfn(ctext);

Purpose

Prints a pair of HTML tags that specify the text they surround is rendered as italics.

Parameters

ctext	in varchar2

Generates

<DFN>ctext</DFN>

htp.big

Syntax

htp.big(ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround is rendered using a big font.

Parameters

ctext	in varchar2
cattributes	in varchar2 DEFAULT NULL

Generates

<BIG cattributes>ctext</BIG>

htp.small

Syntax

htp.small(ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround is rendered using a small font.

Parameters

ctext	in varchar2
cattributes	in varchar2 DEFAULT NULL

Generates

<SMALL cattributes>ctext</SMALL>

htp.sub

Syntax

htp.sub(ctext, calign, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround is rendered as a subscript.

Parameters

ctext	in varchar2
calign	in varchar2 DEFAULT NULL
cattributes	in varchar2 DEFAULT NULL

Generates

_{ctext}

htp.sup

Syntax

htp.sup(ctext, calign, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround is rendered using as a superscript.

Parameters

ctext	in varchar2
calign	in varchar2
cattributes	in varchar2 DEFAULT NULL

Generates

^{ctext}

htp.basefont

Syntax

htp.basefont(nsize);

Purpose

Prints an HTML tag that specifies the base font size for a web page.

Parameters

nsize	in integer

Generates

<BASEFONT SIZE="nsize">

htp.fontOpen

Syntax

htp.fontOpen(ccolor, cface, csize, cattributes);

Purpose

Prints an HTML tag that indicates the beginning of a section of text with the specified font characteristics.

Parameters

ccolor	in varchar2 DEFAULT NULL
cface	in varchar2 DEFAULT NULL
csize	in varchar2 DEFAULT NULL
cattributes	in varchar2 DEFAULT NULL

Generates

<FONT COLOR="ccolor" FACE="cface" SIZE="csize" cattributes>

htp.fontClose

Syntax

htp.fontClose;

Purpose

Prints an HTML tag that indicates the end of a section of text with the specified font characteristics.

Parameters

none

Generates

</FONT>

htp.plaintext

Syntax

htp.plaintext(ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround be rendered with fixed-width type.

Parameters

ctext	in varchar2
cattributes	in varchar2 DEFAULT NULL

Generates

<PLAINTEXT cattributes>ctext</PLAINTEXT>

htp.s

Syntax

htp.s(ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround be rendered in strikethrough type.

Parameters

ctext	in varchar2
cattributes	in varchar2 DEFAULT NULL

Generates

<S cattributes>ctext</S>

htp.strike

Syntax

htp.strike(ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround be rendered in strikethrough type.

Parameters

ctext	in varchar2
cattributesi	n varchar2 DEFAULT NULL

Generates

<STRIKE cattributes>ctext</STRIKE>

Frame Tags

htp.framesetOpen

Syntax

htp.framesetOpen(crows, ccols, cattributes);

Purpose

Prints an HTML tag to open a container of web page frames.

Parameters

crows	in varchar2 DEFAULT NULL
ccols	in varchar2 DEFAULT NULL
cattributes	in varchar2 DEFAULT NULL

Generates

<FRAMESET ROWS="crows" COLS="ccols" cattributes>

htp.framesetClose

Syntax

htp.framesetClose;

Purpose

Prints an HTML tag to close a container of web page frames.

Parameters

none

Generates

</FRAMESET>

htp.frame

Syntax

htp.frame(csrc, cname, cmarginwidth, cmarginheight, cscrolling, cnoresize, cattributes);

Purpose

Prints an HTML tag that defines a single frame within a frameset.

Parameters

csrc	in varchar2
cname	in varchar2 DEFAULT NULL
cmarginwidth	in varchar2 DEFAULT NULL
cmarginheight	in varchar2 DEFAULT NULL
cscrolling	in varchar2 DEFAULT NULL
cnoresizei	n varchar2 DEFAULT NULL
cattributes i	n varchar2 DEFAULT NULL

Generates

<FRAME SRC="csrc" NAME="cname" MARGINWIDTH="cmarginwidth" MARGINHEIGHT="cmarginheight" SCROLLING="cscrolling" NORESIZE cattributes>

htp.noframesOpen

Syntax

htp.noframesOpen;

Purpose

Prints an HTML tag to open a container of content which is viewable by non-Frame-capable web browsers.

Parameters

none

Generates

<NOFRAMES>

htp.noframesClose

Syntax

htp.noframesClose;

Purpose

Prints an HTML tag to close a container of content which is viewable by non-Frame-capable web browsers.

Parameters

none

Generates

</NOFRAMES>

List Tags

List tags allow you to display information in any of the following ways:

- *ordered*: these lists have numbered items
- *unordered*: these lists have bullets to mark each item
- *definition*: these lists alternate a term with its definition

Note: All the hypertext procedures (HTP) shown in this section are also available as hypertext functions (HTF).

htp.listHeader

Syntax

htp.listHeader (ctext, cattributes);

Purpose

Prints an HTML tag at the beginning of the list

Parameters

ctext	in varchar2	
cattributes	in varchar2	DEFAULT NULL

Generates

```
<LH cattributes>ctext</LH>
```

htp.listItem

Syntax

htp.listItem (ctext, cclear, cdingbat, csrc, cattributes);

Purpose

Prints an HTML tag that formats a listed item.

Parameters

ctext	in varchar2	DEFAULT NULL
cclear	in varchar2	DEFAULT NULL
cdingbat	in varchar2	DEFAULT NULL
csrc	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<LI CLEAR="cclear" DINGBAT="cdingbat" SRC="csrc" cattributes>ctext
```

htp.ulistOpen

Syntax

htp.ulistOpen (cclear, cwrap, cdingbat, csrc, cattributes);

Purpose

Prints an HTML tag that is used to open an unordered list that presents listed items separated by white space and marked off by bullets.

Parameters

cclear	in varchar2	DEFAULT NULL
cwrap	in varchar2	DEFAULT NULL
cdingbat	in varchar2	DEFAULT NULL
csrc	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<UL CLEAR="cclear" WRAP="cwrap" DINGBAT="cdingbat" SRC="csrc"
cattributes>
```

htp.ulistClose

Syntax

htp.ulistClose;

Purpose

Prints an HTML tag that ends the unordered list.

Parameters

none

Generates

```
</UL>
```

htp.olistOpen

Syntax

htp.olistOpen (cclear, cwrap, cattributes);

Purpose

Prints an HTML tag that is used to open an ordered list that presents listed items marked off with numbers.

Parameters

cclear	in varchar2	DEFAULT NULL
cwrap	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<OL CLEAR="cclear" WRAP="cwrap" cattributes>
```

htp.olistClose

Syntax

htp.olistClose;

Purpose

Prints an HTML tag that ends an ordered list.

Parameters

none

Generates

```
</OL>
```

htp.dlistOpen

Syntax

htp.dlistOpen (cclear, cattributes);

Purpose

Prints an HTML tag that starts a definition list

Parameters

cclear	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<DL CLEAR="cclear" cattributes>
```

htp.dlistClose

Syntax

htp.dlistClose

Purpose

Prints an HTML tag that Ends a definition list

Parameters

none

Generates

```
</DL>
```

htp.dlistDef

Syntax

htp.dlistDef (ctext, cclear, cattributes);

Purpose

Prints an HTML tag that is used to insert terms, and their corresponding definitions in an indented list format. The htp.dlistTerm must immediately follow this tag.

Parameters

ctext	in varchar2	DEFAULT NULL
clear	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<DD CLEAR="cclear" cattributes>ctext
```

htp.dlistTerm

Syntax

htp.dlistTerm (ctext, cclear, cattributes);

Purpose

Prints an HTML tag used to insert the definition term inside the definition list. This tag must immediately follow the htp.dlistDef.

Parameters

ctext	in varchar2	DEFAULT NULL
cclear	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<DT CLEAR="cclear" cattributes>ctext
```

htp.menulistOpen

Syntax

htp.menulistOpen;

Purpose

Prints an HTML tag that begins a list that presents one line per item, and appears more compact than an unordered list. The htp.listItem will follow this tag.

Parameters

none

Generates

```
<MENU>
```

htp.menulistClose

Syntax

htp.menulistClose;

Purpose

Prints an HTML tag that ends a menu list.

Paramenters

none

Generates

```
</MENU>
```

htp.dirlistOpen

Syntax

htp.dirlistOpen;

Purpose

Prints an HTML tag that begins a directory list. This presents information in a list of items that contain up to 20 characters. Items in this list are typically arranged in columns, typically 24 characters wide. The <LI> or htp.listItem must appear directly after you use this tag.

Parameters

none

Generates

```
<DIR>
```

htp.dirlistClose

Syntax

htp.dirlistClose;

Purpose

Prints an HTML tag that closes the directory list tag, htp.dirlistOpen.

Parameters

none

Generates

```
</DIR>
```

Character Format Tags

The character format tags are used to specify or alter the appearance of the marked text. Character format tags have opening and closing elements, and affect only the text that they surround.

Character format tags give hints to the browser as to how a character or character string should appear, but each browser determines its actual appearance. Essentially, they place text into categories such that all text in a given category is given the same special treatment, but the browser determines what that

treatment is. For example, the HTML string `<STRONG>Here is some text</STRONG>` might appear as bold in some browsers, or might flash instead.

If a specific text attribute, such as bold is desired, a physical format tag may be necessary. See the section, "Physical Format Tags," for more information.

Note: All the hypertext procedures (HTP) shown in this section are also available as hypertext functions (HTF).

htp.cite

Syntax

htp.cite (ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround as a citation. Usually rendered as italics.

Parameters

ctext	in varchar2
cattributes	in varchar2DEFAULT NULL

Generates

`<CITE cattributes>ctext</CITE>`

htp.code

Syntax

htp.code (ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround as an example of code output. Usually rendered in monospace format, e.g. Courier.

Parameters

ctext	in varchar2
cattributes	in varchar2DEFAULT NULL

Generates

`<CODE cattributes>ctext</CODE>`

htp.emphasis

Syntax

htp.emphasis (ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround as requiring typographic emphasis. This tag is equivalent to htp.em. Usually rendered as italics.

Parameters

ctext	in varchar2
cattributes	in varchar2DEFAULT NULL

Generates

```
<EM cattributes>ctext</EM>
```

htp.em

Alias for htp.emphasis

htp.keyboard

Syntax

htp.keyboard (ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround as text typed in by the user, which usually is rendered as monospace. This tag is equivalent to htp.kbd.

Parameters

ctext	in varchar2
cattributes	in varchar2DEFAULT NULL

Generates

```
<KBD cattributes>ctext</KBD>
```

htp.kbd

Alias for htp.keyboard

htp.sample

Syntax

htp.sample (ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround as a sequence of literal characters that must be typed in the exact sequence in which they appear. Usually rendered as monospace font.

Parameters

ctext	in varchar2
cattributesi	n varchar2DEFAULT NULL

Generates

```
<SAMP cattributes>ctext</SAMP>
```

htp.strong

Syntax

htp.strong (ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround as needing strong typographic emphasis. Usually rendered as bold.

Parameters

ctext	in varchar2
cattributes	in varchar2DEFAULT NULL

Generates

```
<STRONG cattributes>ctext</STRONG>
```

htp.variable

Syntax

htp.variable (ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround as a variable name, or a variable that might be entered by the user. Usually rendered as italics.

Parameters

ctext	in varchar2	
cattributes	in varchar2	DEFAULT NULL

Generates

```
<VAR cattributes>ctext</VAR>
```

Physical Format Tags

The physical format tags are used to specify the format of the marked text.

Note: All the hypertext procedures (HTP) shown in this section are also available as hypertext functions (HTF).

htp.bold

Syntax

htp.bold (ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround is to be rendered as boldface.

Parameters

ctext	in varchar2	
cattributes	in varchar2	DEFAULT NULL

Generates

```
<B cattributes>ctext</B>
```

htp.italic

Syntax

htp.italic (ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround is to be rendered as italics.

Parameters

ctext	in varchar2	
cattributesi	n varchar2	DEFAULT NULL

Generates

```
<I cattributes>ctext</I>
```

htp.teletype

Syntax

htp.teletype (ctext, cattributes);

Purpose

Prints a pair of HTML tags that specify the text they surround is to be rendered in a fixed width typewriter font, e.g. Courier.

Parameters

ctext	in varchar2	
cattributesi	n varchar2	DEFAULT NULL

Generates

```
<TT cattributes>ctext</TT>
```

Form Tags

The form tags are used to create and manipulate an HTML form. Forms are used to allow interactive data exchange between a Web Browser and a CGI program.

Forms can have the following types of elements:

- Input: used for a large variety of types of input fields, for example:
  - single line text
  - single line password fields
  - checkboxes
  - radio buttons
  - submit buttons
- Text area: used to create a multi-line input field.

- Select: used to allow the user to chose one or more of a set of alternatives described by textual labels. Usually rendered as a pulldown, pop up, or a fixed size list.

Note: All the hypertext procedures (HTP) shown in this section are also available as hypertext functions (HTF).

htp.formOpen

Syntax

htp.formOpen (curl, cmethod, ctarget, cenctype, cattributes);

Purpose

Prints an HTML tag that starts the form. The *curl* value is required and is the URL of the CGI script, normally owa, to which the contents of the Form will be sent. The method is either ”GET” or ”POST.”

Parameters

curl	in varchar2
cmethod	in varchar2DEFAULT ’POST
ctarget	in varchar2
cenctype	in varchar2DEFAULT NULL
cattributes	in varchar2DEFAULT NULL

Generates

```
<FORM ACTION="curl" METHOD="cmethod" TARGET="ctarget" ENCTYPE="cenctype"
cattributes>
```

htp.formClose

Syntax

htp.formClose;

Purpose

Prints an HTML tag that closes the <FORM> tag

Parameters

none

Generates

```
</FORM>
```

htp.formCheckbox

Syntax

htp.formCheckbox (cname, cvalue, cchecked, cattributes);

Purpose

Prints an HTML tag that inserts a checkbox which the user can toggle off or on.

Parameters

cname	in varchar2	
cvalue	in varchar2	DEFAULT 'on'
cchecked	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<INPUT TYPE="checkbox" NAME="cname" VALUE="cvalue" CHECKED cattributes>
```

htp.formHidden

Syntax

htp.formHidden (cname, cvalue, cattributes);

Purpose

Prints an HTML tag that sends the content of a field along with a submitted form. The field is not visible to the end user.

Parameters

cname	in varchar2	
cvalue	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<INPUT TYPE="hidden" NAME="cname" VALUE="cvalue" cattributes>
```

htp.formImage

Syntax

htp.formImage (cname, csrc, calign, cattributes);

Purpose

Prints an HTML tag that creates an image field that can be clicked on causing the Form to be immediately submitted. The coordinates of the selected point are measured in pixels, and returned (along with other contents of the form) in two name/value pairs. The x-coordinate is submitted under the name of the field with ".x" appended, and the y-coordinate with the ".y" appended. Any value attribute is ignored. The image itself is specified by the CSRC attribute.

Parameters

cname	in varchar2	
csrc	in varchar2	
calign	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL;

Generates

```
<INPUT TYPE="image" NAME="cname" SRC="csrc" ALIGN="calign" cattributes>
```

htp.formPassword

Syntax

htp.formPassword (cname, csize, cmaxlength, cvalue, cattributes);

Purpose

Prints an HTML tag that creates a single line text entry field. Text will not be displayed as it is entered. When the user enters a password, characters are represented by asterisks on single line text entry field.

Parameters

cname	in varchar2	
csize	in varchar2	
cmaxlength	in varchar2	DEFAULT NULL
cvalue	in varchar2	DEFAULT NULL
cattributes	i n varchar2	DEFAULT NULL

Generates

```
<INPUT TYPE="password" NAME="cname" SIZE="csize" MAXLENGTH="cmaxlength"
VALUE="cvalue" cattributes>
```

htp.formRadio

Syntax

htp.formRadio (cname, cvalue, cchecked, cattributes);

Purpose

Prints an HTML tag that inserts a radio button on the HTML Form. Used to create a set of radio buttons, each representing a different value, only one of which will be toggled on by the user. Each radio button field should have the same name. Only the selected radio button will generate a name/value pair in submitted data area. This will require an explicit VALUE attribute.

Parameters

cname	in varchar2	
cvalue	in varchar2	
cchecked	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<INPUT TYPE="radio" NAME="cname" VALUE="cvalue" CHECKED cattributes>
```

htp.formReset

Syntax

htp.formReset (cvalue, cattributes);

Purpose

Prints an HTML tag that creates a RESET button that, when selected, resets all the form fields to their initial values.

Parameters

cvalue	in varchar2	DEFAULT 'Reset'
cattributes	in varchar2	DEFAULT NULL

Generates

```
<INPUT TYPE="reset" VALUE="cvalue" cattributes>
```

htp.formSubmit

Syntax

htp.formSubmit (cname, cvalue, cattributes);

Purpose

Prints an HTML tag that creates a button that, when selected, submits the form. If a SUBMIT button is selected to submit the Form, and that button has a name attribute specified, the submit button then contributes a name/value pair to the submitted data.

Parameters

cname in varchar2DEFAULT NULL
cvalue in varchar2DEFAULT 'Submit'
cattributes in varchar2DEFAULT NULL

Generates

```
<INPUT TYPE="submit" NAME="cname" VALUE="cvalue" cattributes>
```

htp.formText

Syntax

htp.formText (cname, csize, cmaxlength, cvalue, cattributes);

Purpose

Prints an HTML tag that creates a field for a single line of text.

Parameters

cname	in varchar2	
csize	in varchar2	DEFAULT NULL
cmaxlength	in varchar2	DEFAULT NULL
cvalue	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<INPUT TYPE="text" NAME="cname" SIZE="csize" MAXLENGTH="cmaxlength"
VALUE="cvalue" cattributes>
```

htp.formSelectOpen

Syntax

htp.formSelectOpen (cname, cprompt, nsize, cattributes);

Purpose

Prints an HTML tag that begins a Select list of alternatives. Contains the attribute NAME which specifies the name that will be submitted as a name/value pair.

Parameters

cname	in varchar2	
cprompt	in varchar2	DEFAULT NULL
nsize	in integer	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

*c*prompt <SELECT NAME="cname" SIZE="nsize" cattributes>

Example

```
htp.formSelectOpen('greatest_player';
      'Pick the greatest player:');
htp.formSelectOption('Messier');
htp.formSelectOption('Howe');
htp.formSelectOption('Hull');.
htp.formSelectOption('Gretzky');.
htp.formSelectClose;

generates:

Pick the greatest player:
 <SELECT NAME="greatest_player">
 <OPTION>Messier
 <OPTION>Howe
 <OPTION>Hull
 <OPTION>Gretzky
</SELECT>
```

Note: See htp.formSelectOption and htp.formSelectClose.

htp.formSelectOption

Syntax

htp.formSelectOption (cvalue, cselected, cattributes);

Purpose

Prints an HTML tag that represents one choice in the Select element.

Parameters

cvalue	in varchar2	
cselected	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

`<OPTION SELECTED` *`cattributes>cvalue`*
See example for htp.formSelectOpen.

Example

See htp.formSelectOpen.

htp.formSelectClose

Syntax

htp.formSelectClose;

Purpose

Prints an HTML tag that ends a Select list of alternatives.

Parameters

none

Generates

`</SELECT>`

Example

See htp.formSelectOpen.

htp.formTextarea

Syntax

htp.formTextarea (cname, nrows, ncolumns, calign, cattributes);

Purpose

Prints an HTML tag that creates a text field that has no predefined text in the text area. Used to enable the user to enter several lines of text.

Parameters

cname	in varchar2	
nrows	in integer	
ncolumns	in integer	
calign	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<TEXTAREA NAME="cname" ROWS="nrows" COLS="ncolumns" ALIGN="calign"
cattributes></TEXTAREA>
```

htp.formTextareaOpen

Syntax

htp.formTextareaOpen (cname, nrows, ncolumns, calign, cwrap, cattributes);

Prints an HTML tag that opens a text area where you can insert predefined text that will always appear in the text field.

Parameters

cname	in varchar2	
nrows	in integer	
ncolumns	in integer	
calign	in varchar2	DEFAULT NULL
cwrap	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<TEXTAREA NAME="cname" ROWS="nrows" COLS="ncolumns" ALIGN="calign" WRAP =
"cwrap" cattributes>
```

htp.formTextareaClose

Syntax

htp.formTextareaClose;

Purpose

Prints an HTML tag that ends TextArea field

Parameters

none

Generates

```
</TEXTAREA>
```

Table Tags

The Table tags allow the user to insert tables and manipulate the size and columns of the table in a document.

Note: All the hypertext procedures (HTP) shown in this section are also available as hypertext functions (HTF).

htp.tableOpen

Syntax

htp.tableOpen (cborder, calign, cnowrap, cclear, cattributes);

Purpose

Prints an HTML tag that begins an HTML table.

Parameters

cborder	in varchar2	DEFAULT NULL
calign	in varchar2	DEFAULT NULL
cnowrap	in varchar2	DEFAULT NULL
cclear	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL;

Generates

```
<TABLE "cborder" NOWRAP ALIGN="calign" CLEAR="cclear" cattributes>
```

htp.tableClose

Syntax

htp.tableClose;

Purpose

Prints an HTML tag that ends an HTML table.

Parameters

none

Generates

```
</TABLE>
```

htp.tableCaption

Syntax

htp.tableCaption (ccaption, calign, cattributes);

Purpose

Prints an HTML tag that places a caption in the inserted table.

Parameters

ccaption	in varchar2	
calign	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<CAPTION ALIGN="calign" cattributes>ccaption</CAPTION>
```

htp.tableRowOpen

Syntax

htp.tableRowOpen (calign, cvalign,cdp, cnowrap, cattributes);

Purpose

Prints an HTML tag that inserts a row tag into a table.

Parameters

calign	in varchar2	DEFAULT NULL
cvalign	in varchar2	DEFAULT NULL
cdp	in varchar2	DEFAULT NULL
cnowrap	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<TR ALIGN="calign" VALIGN="cvalign" DP="cdp" NOWRAP catttributes>
```

htp.tableRowClose

Syntax

htp.tableRowClose;

Purpose

Prints an HTML tag that ends a row in a table.

Parameters

none

Generates

```
</TR>
```

htp.tableHeader

Syntax

htp.tableHeader (cvalue, calign, cdp, cnowrap, crowspan, ccolspan, cattributes);

Purpose

Prints an HTML tag that inserts a table header.

Parameters

cvalue	in varchar2	DEFAULT NULL
calign	in varchar2	DEFAULT NULL
cdp	in varchar2	DEFAULT NULL
cnowrap	in varchar2	DEFAULT NULL
crowspan	in varchar2	DEFAULT NULL
ccolspan	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<TH ALIGN="calign" DP="cdp" ROWSPAN="crowspan" COLSPAN="ccolspan" NOWRAP
cattributes>cvalue</TH>
```

htp.tableData

Syntax

htp.tableData (cvalue, calign, cdp, crowspan, ccolspan, cnowrap, cattributes);

Purpose

Prints an HTML tag that inserts data into the rows and columns of a selected table.

Parameters

cvalue	in varchar2	DEFAULT NULL
calign	in varchar2	DEFAULT NULL
cdp	in varchar2	DEFAULT NULL
crowspan	in varchar2	DEFAULT NULL
ccolspan	in varchar2	DEFAULT NULL
cnowrap	in varchar2	DEFAULT NULL
cattributes	in varchar2	DEFAULT NULL

Generates

```
<TD ALIGN="calign" DP="cdp" ROWSPAN="crowspan" COLSPAN="ccolspan" NOWRAP
cattributes>cvalue</TD>
```

OWA_UTIL Package

The OWA_UTIL Package is a set of useful utility procedures built on top of hypertext functions and hypertext procedures.

owa_util.signature

Syntax

owa_util.signature;

Purpose

Prints an HTML line followed by a signature line on the HTML document.

Parameters

none

Prints a signature line in the HTML document that might look like the following: "This page was produced by the **Oracle PL/SQL Agent** on August 9, 1995 09:30"

owa_util.signature (cname)

Syntax

owa_util.signature (cname);

Purpose

Allows the programmer to create a signature line on the bottom of the HTML document that has a hypertext link to view the PL/SQL source for that procedure.

Parameters

cname in varchar2

Generates

Prints a signature line on the bottom of the HTML document that has a link to the actual PL/SQL source for that procedure. The link calls the procedure `showsource`. The line would look like the following:

Generates

"This page was produced by the **Oracle PL/SQL Agent** on 6/14/95 09:30"
View PL/SQL Source

owa_util.showsource (cname)

Syntax

owa_util.showsource (cname);

Purpose

Prints the source of the specified PL/SQL procedure, function, or package. If a procedure or function which belongs to a package is specified, then the entire package is displayed.

Parameters

cname in varchar2

Generates

Generates the source code of the specified PL/SQL procedure.

owa_util.showpage

Syntax

owa_util.showpage;

Purpose

This procedure allows a user to view the HTML output of a PL/SQL procedure call from SQL*Plus, SQL*DBA, or Oracle Server Manager. The PL/SQL procedure must use HTP and/or HTF to generate the HTML page, and owa_util.showpage must be issued after the PL/SQL procedure has been called and before any other HTP or HTF subprograms are directly or indirectly called. This method is useful for generating pages filled with static data.

Purpose

Note that this procedure uses dbms_output and thus is limited to 255 characters per line and an overall buffer size of 1,000,000 bytes.

Parameters

none

Generates

One can use this procedure to generate static pages in SQL*Plus that can then be accessed as a standard HTML page. For example:

Generates

```
SQL>set serveroutput on
SQL>spool gretzky.html
SQL>execute hockey.pass('Gretzky")
SQL>execute owa_util.showpage
SQL>exit
```

Generates

This would generate an HTML page which could be accessed from Web clients.

owa_util.get_cgi_env(function)

Syntax

owa_util.get_cgi_env(param_name in varchar2);

Purpose

Allows programmer to retrieve the value of the specified CGI environment variable in the PL/SQL procedure.

Purpose

Note that param_name is case-insensitive.

PurposeParameters

param_name in varchar2

Returns

Returns value of specified CGI environment variable for PL/SQL procedure. If the value is not set, returns *null*

owa_util.print_cgi_env

Syntax

owa_util.print_cgi_env;

Purpose

enables programmer to print all of the CGI environment variables made available by the PL/SQL Agent to the PL/SQL procedures. This utility is good for testing purposes.

Parameters

none

Generates

Prints CGI environment variables made available by the PL/SQL Agent to the PL/SQL procedures.

owa_util.mime_header

Syntax

owa_util.mime_header(ccontent_type);

Purpose

Enables programmer to change the default MIME header that the PL/SQL Agent returns. This *must* come before any htp.print or htp.prn calls in order to signal the PL/SQL Agent not to use the default.

Parameters

ccontent_type in varchar2

Generates

Content-type: <ccontent_type>\n\n

owa_util.get_owa_service_path (Function)

Syntax

owa_util.get_owa_service_path;

Purpose

Returns the name of the currently active path with its full virtual path, plus the currently active DCD. For example, a call to get_owa_service_path could return `/ows-bin/myservice/owa/`.

Parameters

none

Returns

The DCD path. The datatype is varchar2.

owa_util.tableprint

Syntax

owa_util.tablePrint;

Purpose

Enables programmers to print Oracle tables as either preformatted or HTML tables, depending upon Web browser capabilities. Note that RAW COLUMNS are supported, however LONG RAW are not. References to LONG RAW columns will print the result 'Not Printable'. In this case, `cattributes` is the second, rather than the last, parameter.

Parameters

ctable in varchar2
cattributesin varchar2DEFAULT NULL
ntable_typein integer DEFAULT
HTML_TABLE
ccolumns in varchar2 DEFAULT '*'
cclauses in varchar2 DEFAULT NULL
ccol_aliasesin varchar2 DEFAULT NULL

nrow_min in number DEFAULT 0
nrow_max in number DEFAULT NULL

Parameters

Note that ntable_type can be either owa_util.html_table or owa_util.pre_table.

Generates

Prints out either a preformatted or HTML table.

Returns

True or False as to whether there are more rows available beyond the nrow_max requested.

PurposeExample

For browsers that don't support HTML tables, create the following procedure:

```
create or replace procedure showemps is
  ignore_more boolean;
begin
  ignore_more := owa_util.tablePrint('emp', 'BORDER', OWA_UTIL.PRE_TABLE);
end;
 and requesting a URL like this example: http://myhost:8080/ows-bin/hr/owa/
showemps

returns to the client:
<PRE>
------------------------------------------------------------------------
| EMPNO |  ENAME |    JOB    |  MGR |  HIREDATE |  SAL | COMM | DEPTNO |
------------------------------------------------------------------------
| 7369  | SMITH  | CLERK     | 7902 | 17-DEC-80 | 800  |      | 20     |
| 7499  | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 | 1600 | 300  | 30     |
| 7521  | WARD   | SALESMAN  | 7698 | 22-FEB-81 | 1250 | 500  | 30     |
| 7566  | JONES  | MANAGER   | 7839 | 02-APR-81 | 2975 |      | 20     |
| 7654  | MARTIN | SALESMAN  | 7698 | 28-SEP-81 | 1250 | 1400 | 30     |
| 7698  | BLAKE  | MANAGER   | 7839 | 01-MAY-81 | 2850 |      | 30     |
| 7782  | CLARK  | MANAGER   | 7839 | 09-JUN-81 | 2450 |      | 10     |
| 7788  | SCOTT  | ANALYST   | 7566 | 09-DEC-82 | 3000 |      | 20     |
| 7839  | KING   | PRESIDENT |      | 17-NOV-81 | 5000 |      | 10     |
| 7844  | TURNER | SALESMAN  | 7698 | 08-SEP-81 | 1500 | 0    | 30     |
| 7876  | ADAMS  | CLERK     | 7788 | 12-JAN-83 | 1100 |      | 20     |
| 7900  | JAMES  | CLERK     | 7698 | 03-DEC-81 | 950  |      | 30     |
| 7902  | FORD   | ANALYST   | 7566 | 03-DEC-81 | 3000 |      | 20     |
| 7934  | MILLER | CLERK     | 7782 | 23-JAN-82 | 1300 |      | 10     |
------------------------------------------------------------------------
</PRE>
```

To view just the employees in department 10, and only their employee ids, names, and salaries, create the following procedure:

```
create or replace procedure showemps_10 is
   ignore_more boolean;
begin
 ignore_more := owa_util.tablePrint
('EMP', 'BORDER', OWA_UTIL.PRE_TABLE,
       'empno, ename, sal',
       'where deptno=10 order by empno',
       'Employee Number, Name, Salary');
end;
```

A request for a URL like `http://myhost:8080/ows-bin/hr/owa/showemps_10` would return the following to the client:

```
<PRE>
-------------------------------------
| Employee Number |  Name  | Salary |
-------------------------------------
| 7782            | CLARK  | 2450   |
| 7839            | KING   | 5000   |
| 7934            | MILLER | 1300   |
-------------------------------------
</PRE>
```

For browsers that do support HTML tables, to view the department table in an HTML table, create the following procedure:

```
create or replace procedure showdept is
 ignore_more boolean;
begin
   ignore_more := owa_util.tablePrint('dept', 'BORDER');
end;
```

A request for a URL like `http://myhost:8080/ows-bin/hr/owa/showdept` would return the following to the client:

```
<TABLE BORDER>
<TR>
<TH>DEPTNO</TH>
<TH>DNAME</TH>
<TH>LOC</TH>
</TR>
<TR>
<TD ALIGN="LEFT">10</TD>
<TD ALIGN="LEFT">ACCOUNTING</TD>
<TD ALIGN="LEFT">NEW YORK</TD>
</TR>
<TR>
<TD ALIGN="LEFT">20</TD>
<TD ALIGN="LEFT">RESEARCH</TD>
<TD ALIGN="LEFT">DALLAS</TD>
</TR>
<TR>
<TD ALIGN="LEFT">30</TD>
<TD ALIGN="LEFT">SALES</TD>
<TD ALIGN="LEFT">CHICAGO</TD>
</TR>
<TR>
<TD ALIGN="LEFT">40</TD>
<TD ALIGN="LEFT">OPERATIONS</TD>
<TD ALIGN="LEFT">BOSTON</TD>
</TR>
</TABLE>
```

which a Web browser can format to look like this:

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO

Customized Extensions to HTP Packages

The design of the hypertext procedure and function packages allows you to use customized extensions. Therefore, as the HTML standard changes, you can add new functionality similar to the hypertext procedure and function packages to reflect those changes.

Here is an example of customized packages using non-standard <BLINK> and imaginary <SHOUT>tags:

```
create package nsf as
function blink(cbuf in varchar2) return varchar2;
function shout(cbuf in varchar2) return varchar2;
end;
create package body nsf as
function blink(cbuf in varchar2) return varchar2 is
begin return ('<BLINK>' || cbuf || '</BLINK>');
end;
function shout(cbuf in varchar2) return varchar2 is
begin return ('<SHOUT>' || cbuf || '</SHOUT>');
end;
end;

create package nsp as
procedure blink(cbuf  in varchar2);
procedure shout(cbuf  in varchar2);
end;
create package body nsp as
procedure blink(cbuf  in varchar2) is
begin htp.print(nsf.blink(cbuf)); end;
procedure shout(cbuf  in varchar2) is
begin htp.print(nsf.shout(cbuf)); end;
end;
```

Now you can begin to use these procedures and functions in your own procedure.

```
create procedure nonstandard as
begin
nsp.blink('Gee this hurts my eyes!');
htp.print('And I might ' || nsf.shout('get mad!'));
end;
```

For more examples of using the Developer's Toolkit, see *Passing Parameters to PL/SQL*

OWA_PATTERN Package

The OWA_PATTERN package enables you to do sophisticated string manipulation using regular expressions. OWA_PATTERN provides the following three operations:

- MATCH. This determines whether a regular expression exists in a string. This is a function that returns TRUE or FALSE.
- AMATCH. This is a more sophisticated variation on MATCH that lets you specify *where* in the string the match has to occur. This is a function that returns as an integer the end of the location in the string where the regular expression was found. If the regular expression is not found, it returns 0.
- CHANGE. This lets you replace the portion of the string that matched the regular expression with a new string. CHANGE can be either a procedure or a function. If a function, it returns the number of times the regular expression was found and replaced.

There are also operations that these operations use, but that you can also use directly. These are explained shortly.

The OWA_PATTERN operations all use the following three parameters:

- *line*. This is the target to be examined for a match. Despite the name, it can be more than one line of text or can be a PL/SQL table(see *PL/SQL Tables*) of type multi_line
- *pat*. This is the regular expression the functions attempts to locate in *line*. This regular expression uses the special tokens explained shortly. Note: in CHANGE, this is called *from_str*.
- *flags*. These are arguments that control how the search is to be performed.

Some of the operations take additional parameters as well.

Regular Expressions

You specify a regular expression by creating the string you want to match interspersed with various wildcard tokens and quantifiers. The wildcard tokens all match something other than themselves, and the quantifiers modify the meaning of tokens or of literals by specifying such things as how often each is to be applied.

Tokens

The wildcard tokens that are supported are as follows:

^ Matches newline or the beginning of the target.

$ Matches newline or the end of the target.

\n Matches newline.

. Matches any character except newline.

\t Matches tab.

\d Matches digits [0-9]

\D Matches non-digits [not 0-9]

\w Matches word characters (alphanumeric) [0-9, a-z, A-Z or _]

\W Matches non-word characters [not 0-9, a-z, A-Z or _]

\s Matches whitespace characters [blank, tab, or newline]

\S Matches non-whitespace characters [not blank, tab, or newline]

\b Matches "word" boundaries (between \w and \W)

\x<HEX> Matches the value in the current character set of the two hexidecimal digits.

\<OCT> Matches the value in the current character set of the two or three octal digits.

\ Followed by any character not covered by another case matches that character.

& Applies only to CHANGE. This causes the string that matched the regular expression to be included in the string that replaces it. This differs from the other tokens in that it specifies how a target is changed rather than how it is matched. This is explained further under CHANGE.

Quantifiers

Any of the above tokens except & can have its meaning extended by any of the following quantifiers. You can also apply these quantifiers to literals.

? 0 or 1 occurrence(s)

* 0 or more occurrences

+ 1 or more occurrence(s)

{n} Exactly *n* occurrences

(n,} At least *n* occurrences

{n,m} At least *n*, but not more than *m*, occurrences

Flags

In addition to targets and regular expressions, the OWA_PATTERN functions and procedures can use flags to affect how they are interpreted. The recognized flags are as follows:

i This indicates a case-insensitive search.

g This applies only to CHANGE. It indicates a global replace. That is to say, all portions of the target that match the regular expression are replaced.

Datatypes

The following special datatype is used by OWA_PATTERN.

pattern

A PL/SQL table (see *PL/SQL Tables*) of 4 byte VARCHAR2 strings, indexed by BINARY INTEGER. This is an alternative way to store your regular expression than in simple VARCHAR2 strings. The advantages of this is that you can use a pattern as both an input and output parameter. Thus, you can pass the same regular expression to several subsequent OWA_PATTERN function calls, and it only has to be parsed once.

Note: The following datatypes are used by OWA_PATTERN, but are part of the OWA_TEXT package. For information on these, see the section on OWA_TEXT: *owa_text.vc_array, owa_text.multi_line, owa_text.int_array, owa_text.row_list.*

Using MATCH, AMATCH, and CHANGE

Here is an example of MATCH.

```
MATCH ('BATMAN', 'Bat.*', i);
```

This is how the function is interpreted: BATMAN is the target where we are searching for the regular expression. Bat.* is the regular expression we are attempting to find. The period (.) indicates any character other than newline, and the asterisk (*) indicates any 0 or more of such. Therefore, this regular expression specifies that a matching target consists of 'Bat', followed by any set of characters neither ending in nor including a newline (which does not match the period). The *i* at the end is a flag indicating that case is to be ignored in the search.

This would return TRUE, indicating that a match had been found.

Note that, if multiple overlapping strings can match the regular expression, OWA_PATTERN takes the longest match.

Summaries of OWA_PATTERN Functions

MATCH, AMATCH, and CHANGE are overloaded. That is to say, there are several versions of each, distinguished by the parameters they take. Specifically, there are six versions of MATCH, and four each of AMATCH and CHANGE. This section provides a summary of all versions; the following section provides a reference on each version.

MATCH is a function that returns TRUE or FALSE depending on whether a match was found. Here is a summary of the versions of MATCH:

- The target can be either a simple VARCHAR2 string of less than 32K or a multi_line. A multi-line is described under "OWA_TEXT Datatypes". You can create a multi_line from a long string using the *stream2multi* function described under OWA_TEXT. If a multi_line is used, there is a parameter called *rlist*, after the regular expression, but before the flags. This is a list of the chunks where matches were found. Use of a VARCHAR2 implies use of one of the first four versions of this function. Use of a multi_line implies use of version 5 or 6.
- The regular expression can be either a VARCHAR2 string or a *pattern*. You can create a *pattern* from a string using the *getpat* function described later in this section. If you use a pattern, you are using one of the even-number versions of the function.
- If the line is a string and not a multi_line, then you can add an optional output parameter called *backrefs*. This goes after the regular expression, but before the flags. You cannot use *backrefs* if you pass a multi_line to MATCH, because this is the same place in the parameter list that the *rlist* parameter would go. The *backrefs* parameter is a row_list that holds each string in the target that was matched by a sequence of tokens in the regular expression. If you use *backrefs*, you are using the third or fourth version of the function.
- Given the above, you can determine which version you are using as follows: if you are using a multi_line, it is version 5 or 6. If you are using backrefs, it is version 3 or 4. Otherwise, it is version 1 or 2. Once you have determined the pair, you determine the actual version by whether the regular expression is a VARCHAR2 string (odd-numbered) or a pattern (even-numbered).

AMATCH is a function giving a number that indicates the number of characters from the beginning of the target to the end of the first match found (AMATCH stops searching after the first match). If no match is found, it returns 0. Here is a summary of the versions of AMATCH:

- Following the target, but preceding the regular expression, is the input parameter *from_loc*. This indicates how many characters from the beginning of the target the search should commence.
- The regular expression can be either a VARCHAR2 string or a pattern. You can create a pattern from a string using the *getpat* function described later in this section.. If you use a pattern, you are using one of the even-number versions of the function.
- After the regular expression, but before the flags, you can add an optional output parameter called "backrefs". This is a PL/SQL table (see *PL/SQL Tables*) that will hold each string in the target that was matched by a sequence of tokens in the regular expression. If you use *backrefs*, you are using the third or fourth version of the function.
- Given the above, you can determine which version you are using as follows: If you are using backrefs, it is version 3 or 4. Otherwise, it is version 1 or 2. Once you have determined the pair, you determine the actual version by whether the regular expression is a VARCHAR2 string (odd-numbered) or a pattern (even-numbered).

CHANGE can be either a procedure or a function, depending on how it is invoked. If a function, it returns the number of changes made. If the flag 'g' is not used, this number can be only 0 or 1. Here is a summary of the versions of CHANGE

- The target can be either a simple VARCHAR2 string of less than 32K or a multi_line.
- Unlike MATCH and AMATCH, the regular expression can only be a VARCHAR2 string, not a pattern.
- Following the regular expression is the string that is to replace it. This string can use the token ampersand (&), which indicates that the portion of the target that matched the regular expression is to be included in the expression that replaces it. For example:

```
CHANGE('Cats in pajamas','C.+in', '& red ')
```

The regular expression matches the substring 'Cats in'. It then replaces this string with '& red'. & indicates 'Cats in', since that's what matched the regular expression. Thus, this procedure replaces the string 'Cats in pajamas' with 'Cats in red pajamas'. Of course, we used a literal here for clarity. In actuality, 'Cats in pajamas' would be the value of a variable, and that value would be changed. Were this a function rather than a procedure, the value it would return would not be Cats in red pajamas' but 1, indicating that a single substitution had been made.

- The flag 'g' indicates that all matching portions of the target are to be replaced by the regular expression. Otherwise, only the first match is replaced.
- The way to distinguish the versions is by whether the target is a string (version 1 or 2) or a multi_line (version 3 or 4). Within these pairs, odd numbers are functions, and even procedures.

Here are the formal descriptions of the OWA_PATTERN procedures and functions.

Procedures and Functions

owa_pattern.match (version 1)

Syntax

owa_pattern.match(line, pat, flags)

Purpose

enables programmers to search a string for a pattern using regular expressions.

Parameters

line	in	varchar2
pat	in	varchar2
flags	in	varchar2 DEFAULT NULL

Generates

Boolean indicating whether match was found.

owa_pattern.match (version 2)

Syntax

owa_pattern.match(line, pat, flags)

Purpose

enables programmers to search a string for a pattern using regular expression

Parameters

line	in	varchar2
pat	in out	pattern
flags	in	varchar2 DEFAULT NULL

Generates

Boolean indicating whether match was found.

owa_pattern.match (version 3)

Syntax

owa_pattern.match(line, pat, backrefs, flags)

Purpose

enables programmers to search a string for a pattern using regular expression

Parameters

line	in	varchar2
pat	in	varchar2
backrefs	out	owa_text.vc_arr
flags	in	varchar2 DEFAULT NULL

Generates

Boolean indicating whether match was found.

owa_pattern.match (version 4)

Syntax

owa_pattern.match(line, pat, backrefs, flags)

Purpose

enables programmers to search a string for a pattern using regular expression

Parameters

line	in	varchar2
pat	in out	pattern
backrefs	out	owa_text.vc_arr
flags	in	varchar2 DEFAULT NULL

Generates

Boolean indicating whether match was found.

owa_pattern.match (version 5)

Syntax

owa_pattern.match(mline, pat, rlist, flags)

Purpose

enables programmers to search a string for a pattern using regular expression

Parameters

mline	inowa_text.multi_line	
pat	in	varchar2
rlist	out	owa_text.row_list
flags	in	varchar2 DEFAULT NULL

Generates

Boolean indicating whether match was found.

owa_pattern.match (version 6)

Syntax

owa_pattern.match(mline, pat, rlist, flags)

Purpose

enables programmers to search a string for a pattern using regular expression

Parameters

mline	in	owa_text.multi_line
pat	in out	pattern
rlist	out	owa_text.row_list
flags	in	varchar2 DEFAULT NULL

Generates

Boolean indicating whether match was found.

owa_pattern.amatch (version 1)

Syntax

owa_pattern.match(line, from_loc, pat, flags)

enables programmers to search a string for a pattern using regular expressions.

Parameters

line	in	varchar2
from_loc	ini	nteger
pat	in	varchar2
flags	in	varchar2 DEFAULT NULL

Generates

Location (in number of characters from the beginning) of the end of the match. 0 if none such.

owa_pattern.amatch (version 2)

Syntax

owa_pattern.match(line, from_loc, pat, flags)

Purpose

enables programmers to search a string for a pattern using regular expression

Parameters

line	in	varchar2
from_loc	in	integer
pat	in out	pattern
flags	in	varchar2 DEFAULT NULL

Generates

Location (in number of characters from the beginning) of the end of the match. 0 if none such.

owa_pattern.amatch (version 3)

Syntax

owa_pattern.match(line, from_loc, pat, backrefs, flags)

Purpose

enables programmers to search a string for a pattern using regular expression

Parameters

line	in	varchar2
from_loc	in	integer
pat	in	varchar2

backrefs	out	owa_text.vc_arr
flags	in	varchar2 DEFAULT NULL

Generates

Location (in number of characters from the beginning) of the end of the match. 0 if none such.

owa_pattern.amatch (version 4)

Syntax

owa_pattern.match(line, from_loc, pat, backrefs, flags)

Purpose

enables programmers to search a string for a pattern using regular expression

Parameters

line	in	varchar2
from_loc	in	integer
pat	in out	pattern
backrefs	out	owa_text.vc_arr
flags	in	varchar2 DEFAULT NULL

Generates

Location (in number of characters from the beginning) of the end of the match. 0 if none such.

owa_pattern.change (version 1)

Syntax

owa_pattern.match(line, from_str, to_str, flags)

Purpose

This version is a function. It enables programmers to search a string for a pattern and replace it.

Parameters

line	in out	varchar2
from_str	in	varchar2
to_str	in	varchar2
flags	in	varchar2 DEFAULT NULL

Generates

Revises line parameter. Function outputs number of substitutions made.

owa_pattern.change (version 2)

Syntax

owa_pattern.match(line, from_str, to_str, flags)

Purpose

This version is a procedure. It enables programmers to search a string for a pattern and replace it.

Parameters

line	in out	varchar2
from_str	in	varchar2
to_srt	in	varchar2
flags	in	varchar2 DEFAULT NULL

Generates

Revises line parameter.

owa_pattern.change (version 3)

Syntax

owa_pattern.match(mline, from_str, to_str, backrefs, flags)

Purpose

This version is a function. It enables programmers to search a multi_line for a pattern and replace itenables programmers to search a string for a pattern using regular expression

Parameters

mline	in out	multi_line	
from_str	in	varchar2	
to_srt	in	varchar2	
flags	in	varchar2	DEFAULT NULL

Generates

Revises mline parameter. Function outputs number of substitutions made.

owa_pattern.change (version 4)

Syntax

owa_pattern.match(mline, from_str, to_str, flags)

Purpose

This version is a procedure. It enables programmers to search a multi_line for a pattern and replace it.

Parameters

mline	in out	multi_line
from_str	in	varchar2
to_srt	in	varchar2
flags	in	varchar2 DEFAULT NULL

Generates

Revises mline parameter.

owa_pattern.getpat

Syntax

owa_pattern.getpat(arg, pat)

Purpose

This converts a VARCHAR2 string into the special datatype pattern. This datatypes is explained under OWA_PATTERN Datatypes.

Parameters

arg	in	varchar2
pat	in out	pattern

Generates

pattern.

OWA_TEXT

The OWA_TEXT package is chiefly used by OWA_PATTERN, but the functions are externalized so that you can use them directly if desired.

Datatypes

vc_arrayowa_text.

A PL/SQL table of 32K VARCHAR2 strings, indexed by BINARY INTEGER. This is a component of *owa_text.multi_line.*

multi_lineowa_text.

A record with the following three fields:

nt_arrayowa_text.i

A PL/SQL table of INTEGER indexed by BINARY INTEGER. This is a component of *owa_text.row_list*

ow_listowa_text.r

A record with the following fields:

Procedures and Functions

owa_text.stream2multi

Syntax

owa_text.stream2multistream, mline)

Purpose

Converts a long string to a multi_line.

Parameters

srtream	in	varchar2
mline	out	multi_line

Generates

multi_line

owa_text.add2multi

Syntax

owa_text.add2multi(stream, mline, continue)

Purpose

Adds more content to a multi_line. The continue parameter specifies whether to begin appending within the previous final chunk (assuming it is less than 32K) or to start a new chunk.

Parameters

stream	in	varchar2
mline	out	multi_line
continue	in	boolean DEFAULT TRUE

owa_text.new_row_list

Syntax

owa_text.new_row_list(rlist)

Purpose

Can be a procedure or a function. If a function it takes no parameters and outputs a new, initially empty, row_list. If a procedure, it places into the rlist parameter a new, initially empty, row_list.

Parameters

rlist	in	row_list

Generates

Actual output generated by htp.print.

owa_text.print_multi

Syntax

owa_text.print_multi(mline)

Purpose

Uses htp.print to print the multi_line.

Parameters

mline	in	multi_line

Generates

Actual output generated by htp.print.

owa_text.print_row_list

Syntax

owa_text.print_row_list(rlist)

Purpose

Uses htp.print to print the row_list.

Parameters

rlist	in	row_list

Generates

Actual output generated by htp.print.

OWA_IMAGE

OWA_IMAGE is a package providing simple functions that are for handling image maps.

Datatypes

point

This is a table of 32K VARCHAR2 strings indexed by BINARY INTEGER.

Procedures and Functions

owa_image.get_x

Syntax

owa_image.get_x(p)

Purpose

This is a function that produces the X coordinate of the image map.

Parameters

in	point

Generates

X coordinate as integer.

owa_image.get_y

Syntax

owa_image.get_Y(p)

Purpose

This is a function that produces the Y coordinate of the image map.

Parameters

in point

Generates

Y coordinate as integer.

OWA_COOKIE

OWA_COOKIE is a package that provides wrappers so that you can send cookies to and get them from the client's browser. Cookies are strings that are opaque to the client, but that maintain state throughout the client's session, or longer if an expiration date is included.

Datatypes

cookie

Since the HTTP standard is that cookie names can be overloaded, that is, multiple values can be associated with the same cookie name, this is a PL/SQL RECORD holding all values associated with a given cookie name. The fields are as follows:

```
name      varchar2(4K)
vals       vc_arr
numvals    integer
```

Note: vc_arr is defined in the OWA_TEXT package.

Procedures and Functions

owa_cookie.send

Syntax

owa_cookie.send(name, value, expires, path, domain, secure)

Purpose

This is a procedure that transmits a cookie to the client. This procedure must occur in the context of an OWA procedure's HTTP header output.

Parameters

name	in	varchar2
value	in	varchar2
expires	in	date DEFAULT NULL
path	in	varchar2 DEFAULT NULL
domain	in	varchar2 DEFAULT NULL
secure	in	varchar2 DEFAULT NULL

Generates

The following HTML code:

```
Set-Cookie: <name>=<value> expires=<expires> path=<path> domain=<domain>
secure
```

Any defaulted options are omitted.

owa_cookie.get

Syntax

owa_cookie.get(name)

Purpose

This is a function that converts the string to a cookie

Parameters

name	in	varchar2

Generates

A cookie

owa_cookie.get_all

Syntax

owa_cookie.get_all(names, vals, num_vals)

Purpose

This is a procedure that returns all cookie name/value pairs from the client's browser that are associated with your domain.

Parameters

names	out	vc_arr
vals	out	vc_arr
num_vals	out	integer

Generates

Arrays of the names and of the values in the order received, and the count of the combinations.

owa_cookie.remove

Syntax

owa_cookie.remove(name, value, path)

Purpose

This is a procedure that forces a cookie to expire immediately. This output of this procedure must be embedded in an HTML header.

Parameters

name	in	varchar2
value	in	varchar2
path	in	varchar2 DEFAULT NULL

Generates

The following HTML code:

```
Set-Cookie: <name>=<value> expires=01-JAN-1990 path=<path>
```

CHAPTER

6 Oracle WebServer Messages

This appendix contains messages you may receive if you encounter errors while using Oracle WebServer. If you get an error message, look in this appendix to find the probable cause and suggestions of what to do.

00001 - 00600 Generic Oracle WebServer Configuration Messages

OWS—00000, *"utility* : normal, successful completion"

Cause: Normal exit

Action: None

OWS—00001, *"utility* : unable to open file *filename"*

Cause: The file may not exist or may not be readable to the signalling program.

Action: Verify the file's existence in the file-system. Check file and directory permissions and ownership, as well as the

ownership of the signalling process.

OWS—00002, "*utility* : error reading file *filename*"

Cause: Usually an operating system error. If the problem had been file permissions or non-existence, error 1 would have been signalled.

Action: Check your operating system log for errors.

OWS—00003, "*utility* : error writing file *filename*"

Cause: Usually an operating system error. If the problem had been file permissions or non-existence, error 1 would have been signalled.

Action: Check your operating system log for errors.

OWS—00010, "*utility* : file format error when reading *filename*"

Cause: The signalling program expected a specific file format, which it did not find.

Action: Analyze the file for incorrect formatting.

OWS—00011, "Installation not completed. Root.sh must be run"

Cause: The signalling program checked if root.sh completed successfully.

Action: Run root.sh as indicated by your installation guide.

OWS—00020, "*utility* : out of memory when requesting *number* bytes"

Cause: The operating system was not able to supply the requested amount of memory.

Action: Reduce the number of running programs on the machine and retry the operation.

OWS—00600, "*utility* : internal error, arguments [*arg1*][*arg2*][*arg3*][*arg4*][*arg5*]"

Cause: This is the generic internal error number for NDW program exceptions. This indicates that a process has encountered an exceptional condition.

Action: Report as a bug - the first argument is the internal error number

5000 - 5499 : Oracle Web Agent errors.

OWS—05100, "Agent : unable to connect due to Oracle error *number*"

Cause: The Oracle Web Agent was unable to connect to the specified Database due to an Oracle error.

Action: See the corresponding Oracle error. Common errors are: invalid username / password - check the Web Agent configuration file; Oracle not available - start up the database.

OWS—05101, "Agent : execution failed due to Oracle error *number*"

Action: See the Oracle error on the error stack.

OWS—05102, "Agent : initialization failed due to error *number*"

Action: See the Oracle error on the error stack.

OWS—05103, "Agent : error *number* during initialization"

Action: See the Oracle error on the error stack.

OWS—05104, "Agent : unable to attach to host database: [*ORACLE_HOME*], [*ORACLE_SID*], [*SQL*Net V2 Service*]"

Action: See the Oracle error on the error stack.

OWS—05110, "Agent : no stored procedure specified to call"

Cause: OWA requires the PATH_INFO environment variable to be set. PATH_INFO is set based on information passed through the URL after the OWA executable name and before the "?" or end of the URL

Action: specify a stored procedure name after the OWA executable name in the URL.

OWS—05111, "Agent : no stored procedure matches this call with the arguments passed"

Cause: The procedure and parameters passed to OWA don't match any existing stored procedures.

Action: verify that there is a stored procedure (not function) with no RECORD parameters which matches the parameters passed. If no value was passed for a parameter, be sure that the stored procedure was declared to have a default value for that parameter. Check for typographical errors.

OWS—05112, "Agent : too many procedures matches this call"

Cause: The procedure and parameters passed to OWA match too many existing stored procedures.

Action: This should only happen if there are overloaded stored procedures, and that in two different versions of the procedure, a given parameter is declared in one as a scalar and the other as a PL/SQL table.

OWS—05150, "Agent : http server Error - environment variable *variable* is NULL or non-existent"

Cause: The http server or other invoking application did not set the listed environment variable.

Action: Contact your http server or other invoking application vendor.

OWS—05151, "Agent : Port *number* not in list of valid ports : *number-list*"

Cause: The network port that the invoking http server is listening on is not a valid port for this OWA Service.

Action: Add this port to the list of valid ports for this Service, or access this service through a http server listening on a valid port.

OWS—05200, "Agent : service *service-name* not found in configuration file"

Cause: The piece of the URL just before the Web Agent executable name did not correspond to a Web Agent Service in the OWA configuration file.

Action: Correct the URL, or create the desired service.

OWS—05201, "Agent : service parameter *parameter* not found for service *service-name*"

Cause: A parameter required by the Oracle Web Agent was not found in the configuration file along with the specified service.

Action: Modify the Web Agent service configuration using the Web Agent administration form.

5500 - 5599 : Oracle Web Agent Administration errors and messages

OWS—05504, "Service *service-name* successfully written. Verify the PL/SQL output in *filename*"

Cause: The Web Agent Administration program successfully wrote the specified service and performed all actions requested. The program, however, cannot detect errors which may have occurred while installing the PL/SQL.

Action: Check the output in the file specified and verify that there are not unexpected error messages.

OWS—05521, "Service *service-name* submission failed because the passwords submitted do not match"

Cause: The password and the "confirm password" field do not match.

Action: Retype both passwords and re-submit the form.

OWS—05522, "Service *service-name* submission failed because password identification was selected and the submitted password was NULL"

Cause: The form submitted indicated that the Web Agent service should use a password to connect to the database, however no password was provided.

Action: Select operating system authentication for this service, or provide a password for the give OWA Database User.

OWS—05523, "Service *service-name* submission failed because value for mandatory parameter *parameter* is NULL"

Action: Fill in the specified field and re-submit the form.

OWS—05524, "Service *service-name* submission failed because both *parameter1* and *parameter2* are NULL"

Cause: One of the two parameters listed must not be null.

Action: Fill in one of the listed fields and re-submit the form.

OWS—05525, "Service *service-name* submission failed because a service with the same name already exists"

Action: Choose another name for the service you are attempting to create, or delete the existing service.

OWS—05526, "Service *service-name* submission failed due to error *number*"

Action: See the associated error and take the appropriate action.

OWS—05527, "Service *service-name* submission failed because the *PL/SQLAgent-or-DBA* Username and Password are invalid"

Cause: Either the OWA Database User name and password, or the DBA username and password, is not a valid username / password combination for the specified database.

Action: Correct the username and password and re-submit the form.

OWS—05528, "Service *service-name* submission failed because the DBA User submitted does not have *privilege* privileges"

Action: Choose a different Oracle username and password for the DBA user submitted, or have the specified privilege granted to this user by another DBA.

OWS—05529, "Service *service-name* to modify does not exist"

Cause: The user chose to modify an existing Web Agent service, however that service does not exist. This error should not occur unless the service to modify was deleted from a different form after this service modification form was generated.

Action: Create a new service.

OWS—05530, "Service *service-name* submission failed because the NLS Language could not be determined"

Cause: The Web Agent Administration package was unable to lookup the NLS Language for the specified database.

Action: Specify the value explicitly or submit a valid dba username and password.

OWS—05531, "Service *service-name* submission failed because neither Server Manager, nor SQL*DBA could be run"

Cause: The Web Agent Administration package was unable to install the WebServer Developer's Toolkit which must be done from Server Manager (line mode) or SQL*DBA.

Action: Install either Server Manager or SQL*DBA in the ORACLE_HOME where the Web Agent administration is installed.

5600 - 5699 : Oracle Web Database Administration errors

OWS—05610, "Startup of database *database* failed due to error *number*"

Action: See the associated error and take the appropriate action.

OWS—05611, "Shutdown of database *database* failed due to error *number*"

Action: See the associated error and take the appropriate action.

OWS—05620, "DB Admin : no database selected"

Cause: A button was pressed to startup or shutdown the database, but no database had been selected.

Action: Return to the administration screen and select a database.

OWS—05621, "DB Admin : *database* is already running."

Cause: A request was made to start up a database, but that database is already running.

Action: If you would like to restart the database, shut it down first or choose "Startup Force".

OWS—05622, "DB Admin : *database* is already shut down."

Cause: A request was made to shut down a database, but that database is already shut down.

OWS—05623, "DB Admin : parameter file *filename* is larger than maximum size *number* bytes"

Cause: The initialization file listed was larger than the maximum allowable size.

Action: Reduce the size of the initialization file.

5700 - 5799 : Oracle Web Listener Configuration errors and messages

OWS—05710, "Value submitted for *parameter* must not be NULL"

Action: Type a value for the specified parameter and re-submit the form.

OWS—05711, "Port *number* is already in use by the Oracle Web Administration Server"

Cause: The port specified for service creation or modification is already in use by the Oracle WebServer Administration Server.

Action: Choose another port to run one of the Web Listeners on.

OWS—05712, "The Oracle Web Listener *listener-name* is already configured to run on port *number*"

Cause: The port specified for service creation or modification is already in use by another Oracle Web Listener.

Action: Choose another port to run one of the Web Listeners on.

OWS—05713, "Value for *parameter* must be between *minimum* and *maximum*"

Action: Specify a value between the two limits and resubmit the form.

OWS—05714, "For ports less than *number*, the effective userid must be root"

Cause: Ports below that specified require superuser privileges to run a Web Listener on it.

Action: Configure the Web Listener to run on a unrestricted port, or set the effective userid for this Web Listener the superuser.

OWS—05715, "An Oracle Web Listener named *listener-name* already exists"

Cause: Listener names must be unique

Action: Name the submitted listener something else.

OWS—05716, "Port Number *number* is duplicated in the Addresses and Ports list"

Cause: Port numbers must be unique

Action: Remove duplication and resubmit the form.

OWS—05717, "Fill out form has unknown input field format"

Cause: Oracle admbin/*.html files have probably been tampered with.

Action: Restore original files.

OWS—05718, "Configuration information for application *app-name* not found"

Cause: There is no correct information for the said application in the configuration file.

Action: Add required information to the configuration file.

OWS—05719, "Required environment variable OWS_APPFILE not set"

Action: Set OWS_APPFILE to the full path of the cartridge configuration file (Only for WS2.0 Beta).

OWS—05725, "The Web Listener *listener-name* failed to start:"

Action: See the associated error message.

OWS—05726, "The Web Listener *listener-name* failed to reload:"

Action: See the associated error message.

OWS—05727, "The Web Listener *listener-name* was not stopped:"

Action: See the associated error message.

5900 - 5999 : Oracle WebServer Registration errors and messages

OWS—05901, “Please enter a value for field *field*.”

Cause: The specified field is a required field for registration.

Action: Type a value into the specified field and re-submit the form.

7500 - 7599 : Oracle Web Server Proxy errors and messages

OWS—07500, “Proxy switch could not be accessed.”

Cause: Error while reading or writing proxy switch value

Action: Check configuration file location / format.

OWS—07501, “Proxy Administration Form has invalid input values”

Cause: Form is being submitted with invalid field values

Action: Check and correct field values

Java Web Developer’s Toolkit Messages

Message: [*Method*]: Invalid [*Argument*] Value(s)

Exception: `oracle.lang.HtmlRuntimeException`

Cause: The client programmer has supplied incorrect argument value(s) for the named [***Method***].

Action: Change the argument values.

Message: [*Method*]: Circular Reference

Exception: `oracle.lang.HtmlRuntimeException`

Cause: The client programmer has attempted to add a Container / CompoundItem to itself.

Action: Remove that statement from the client application program.

Message: [*Method*]: File ([*filename*]) Not Found

Exception: `oracle.lang.HtmlRuntimeException`

Cause: This package / method cannot find the named file in the system's path.

Action: Provide the correct file / path name and / or change the system path.

Message: [*Method*]: IO Exception caught

Exception: `oracle.lang.HtmlRuntimeException`

Cause: An IO Exception has occurred.

Action: Check if the disk storage device is full or if there is a sharing violation.

APPENDIX

A Glossary

Administration DCD

The Database Connection Descriptor (DCD) used by the administration server to manage PL/SQL Agent database access.

Administration Server

A collection of special instances of WebServer components that a WebServer administrator uses to configure and maintain the WebServer.

Applet

A Java term for small programs that can be dynamically imported into Web pages or applications as needed. Generally, applets are imported from the Internet or another computer network.

Application Developer

A person who writes programs that the WebServer and/or the Oracle7 Server executes.

Authentication

a. A security scheme that requires a client to enter a user name and password to access certain files provided by the WebServer.

b. The practice of "signing" an electronic document in a legally binding way using digital signatures.

Base Directory

The directory name to which URL-encoded pathnames addressed to this

port are to be appended. For example, if the base directory is `/public_html`, the URL `http://www.blob.com/file` is converted to `http://www.blob.com/public_html/file`.

Basic Authentication

An authentication scheme that does not encrypt passwords when sending them over the Internet. Basic authentication is much less secure than digest authentication. See also *Digest Authentication* and *Authentication*.

Bytecode

Java code is interpreted in two steps. First, it is converted from source code (the Java code as written) to bytecode, which is a form executable by any platform on which Java runs. When the bytecode is executed, it is converted from bytecode to the native code for the platform in question.

CA

See *Certifying Authority (CA)*.

Certificate

A file provided by a certifying authority (CA) and installed on a WebServer machine that the WebServer uses to authenticate itself to clients requesting secure connections.

Certifying Authority (CA)

A trusted third-party company that provides certificates to legitimate organizations that request them.

CGI

See *Common Gateway Interface (CGI)*.

Character Set

A set of characters used to write a human language or group of languages, as defined by RFC 1521.

Client

A process, such as a Web browser, that interfaces to one or more users, sends requests to a server, and presents the results of those requests to the users. See also *Server*.

Common Gateway Interface (CGI)

The industry-standard technique for running applications on a web server. Oracle WebServer supports this standard, but also offers the Web Request Broker as a superior alternative.

Common Logfile Format

An industry standard format for transaction log files. The Web Listener uses this format to log transactions.

Configuration Directory

A directory in which a Web Listener process stores its configuration file.

Cookie

Information inserted by the server into the client's browser to track what the client has been doing. This can either expire when the user exits the browser or expire at the date the creator of the cookie specifies.

Database

A structured collection of information and the program that manages such. The Oracle7 Server is a relational database, which is the prevalent type.

Database Connection Descriptor (DCD)

A file that specifies information such as how the PL/SQL Agent is to connect to the Oracle7 Server to fulfil an HTTP request. A URL that requires the use of the PL/SQL Agent includes the name of the DCD it is to use. The information in the DCD includes the username (which also specifies the schema and the privileges), password, connect-string, error log file, standard error message, and the language to be used.

Data Integrity

A mechanism that uses digital signatures to ensure that transmitted data is not tampered with.

DCD

See *Database Connection Descriptor (DCD)*.

Default Character Set

The character set the Web Listener uses in interpreting a file that uses an unrecognized character set.

Default DCD

The Database Connection Descriptor(DCD) that the PL/SQL Agent uses to access a database in response to a request that does not specify an available DCD.

Default MIME Type

The Multipurpose Internet Mail Extensions (MIME) type that the Web Listener uses in interpreting requested files of an unsupported MIME type. See also *MIME Type*.

Digest Authentication

An authentication scheme that encrypts passwords before sending them over the Internet, unlike basic authentication. See also *Authentication*.

Digital Signature

A "signature" attached to an electronic document that reliably identifies the author or sender, and guarantees that the document has not been tampered with.

Directory Indexing

The practice of returning a directory listing when a request URL resolves to a directory that does not contain the default initial file. Directory listings can sometimes help clients learn the correct spelling of filenames they want to request.

Directory Mapping

The practice of defining a virtual file system.

Dispatcher

See *WRB Dispatcher*.

DNS Resolution

The practice of determining a computer's DNS (Domain Name Service) host name from its IP address.

Document Root

The file-system directory that serves as the root of the Web Listener's virtual file system.

Domain

See *Domain Name Service (DNS)*.

Domain-based Restriction

A restriction scheme that allows only machines within specified DNS domains to access certain files. See also *Restriction*.

Domain Name Service (DNS)

The mechanism that divides the Internet into separate, hierarchical groups called domains, identified by unique alphanumeric names, such as `us.oracle.com`. DNS identifies each computer within a domain by a unique host name. For example, a computer named `hal` in the `us.oracle.com` domain would be uniquely identified on the Internet as `hal.us.oracle.com`

Encoding

An algorithm used to alter a file's format, such as compression. You can use the WebServer manager to define the encodings that each Web Listener process recognizes.

Encryption

The practice of scrambling (encrypting) data in such a way that only an intended recipient can unscramble (decrypt) and read the data. See also *Public-Key Encryption* and *Secret-Key Encryption*.

Error File

A file to which a Web Listener process logs errors. There is one error file for each Web Listener process.

Exception

A runtime occurrence in PL/SQL or Java that requires special handling and may indicate an error.

File Caching

The practice of leaving files open (resident in memory) so the WebServer can provide them to clients quickly. You can use the WebServer Manager to specify files to be cached.

Filename Extension

A short alphanumeric suffix attached to a filename, following a dot "." that represents the file's format. The WebServer uses filename extensions to identify several kinds of file formats, including MIME types and encodings. A file may have several extensions.

File Protection

The practice of assigning an authentication or restriction scheme to control access to a specific file or group of files.

Firewall Machine

A computer that regulates access to computers on a local area network from outside, and regulates access to outside computers from within the local area network.

Foreign Key

See *Key*.

genreq

A utility you can use to generate a request for a certificate. You can then submit the generated request to a certifying authority (CA).

Host Name

An alphanumeric character string that uniquely identifies a computer within a DNS domain.

HTTP

See *HyperText Transfer Protocol (HTTP)*.

HTTPS

Secure HTTP—a version of HTTP with provisions for secure data transmission. See *HyperText Transfer Protocol (HTTP)*.

HyperText Markup Language (HTML)

A format for encoding hypertext documents that may contain text, graphics, and references to programs, and references to other hypertext documents.

HyperText Transfer Protocol (HTTP)

The protocol that clients use to issue requests for documents to the WebServer.

Image Map

Graphic in a Web page that specifies several URL's, each associated with a specified region of the single image.

Info File

A file to which a Web Listener process logs its transactions on a particular port. There is one info file for each port on which the Web Listener process accepts connections. The info file is in Common Logfile Format.

Initial File

The name of the HTML file that the WebServer returns by default when a request URL specifies only a directory name.

IP Address

A four-part number with no more than three digits in each part that uniquely identifies a computer on the Internet; the number format is defined by the Internet Protocol (IP).

IP-based Restriction

A restriction scheme that allows only machines within specified groups of IP addresses to access certain files. See also *Restriction*.

Java

Language developed by Sun Microsystems and used by Oracle WebServer. This language is fully object-oriented, extremely portable, and

optimized for creating distributed applications on the Internet or other computer networks. Oracle WebServer can execute Java directly and can send Java programs called applets to the client's browser for execution there.

Java Interpreter

In general, a program that interprets and executes Java bytecode independently of a Web browser. In Oracle WebServer, this refers specifically to the Java Interpreter that WebServer provides for the purpose of executing Java on the server.

Java Web Developer's Toolkit

A group of Java classes provided with the Oracle WebServer SDK to make it easier for you to interface to the Oracle7 Server and generate dynamic HTML using Java.

Key

a. A large number used in encrypting data. See also *Private Key* and *Public Key*.

b. A unique identifier used in a relational database, called a "primary key."

c. A reference to a primary key, called a "foreign key."

Key Pair

A pair of mathematically related keys (a public key and a private key) associated with a user, used in public-key encryption.

Language Identifier

A two-character alphanumeric string that identifies a human language, as defined by RFC 1766.

Listener

Portion of the WebServer that receives HTTP requests. You can use the Oracle WebServer Manager to create multiple Web Listener processes and assign each to accept connections on a different set of ports.

Listener Configuration

A collective name for the parameters that control the behavior of a Web Listener process. You use the WebServer Manager to maintain listener configurations.

Listener Name

An alphanumeric string no more than six characters long that uniquely identifies a Web Listener process.

Listener PID File

A one-line text file that contains an ASCII representation of the process ID (PID) of a Web Listener process.

LiveHTML

Oracle's extension of the industry-standard Server Side Includes (SSI) functionality. LiveHTML files supplement HTML with instructions that WebServer executes before transmitting the page. These instruction specify material that is to be included in the generated page. Said material can include other Web pages, environment variables, and the output of programs executed on the Server. The programs may, but need not, conform to the CGI standard.

Local Database

For the WebServer, a database that runs on the same machine as the WebServer.

Log File Directory

A directory in which the PL/SQL Agent stores log files for a particular Database Connection Descriptor (DCD).

Memory Mapping

For the WebServer, the practice of mapping an open file directly into the address space of a Web Listener process. This speeds file access, and allows multiple clients to access the same file simultaneously without making a separate copy for each client.

MIME Type

A file format defined by the Multipurpose Internet Mail Extensions standard. You can review the RFCs that define MIME at `http://www.oac.uci.edu/indiv/ehood/MIME/MIME.html`.

NULL

a. A marker is the database for the absence of data.

b. The logical result of the comparison of a database NULL with any value.

c. A statement is PL/SQL that does nothing, but functions as a placeholder.

Oracle7 Server

The leading database product in the world. A program for sophisticated high-level management of information. See also *Database* and *Server*.

ORACLE_HOME

Environment Variable that indicates the root of the Oracle7 Server code tree.

Oracle Web Agent (OWA)

A term from an earlier release of this product. OWA is now the PL/SQL Agent. For the sake of compatibility, the string "owa" is still used in URLs to specify that execution is to go to the PL/SQL Agent.

Overloading

Overloaded procedures and functions (in PL/SQL) or methods (in Java) have the same name but take different parameters (PL/SQL) or are contained in different classes (Java) and do similar but not identical things.

Package

A group of PL/SQL functions and procedures.

Parsable File

A file located on the WebServer that contains codes that the server interprets prior to transmission of the file as a Web page. This is part of the LiveHTML functionality. See also *LiveHTML*.

PL/SQL

Oracle's proprietary extension to the SQL language. PL/SQL adds procedural and other constructs to SQL that make it suitable for writing applications.

PL/SQL Agent

A server extension that interfaces to the Oracle7 Server using PL/SQL and dynamically derives and outputs HTML. The PL/SQL Agent can be invoked either as a WRB Service or as a CGI program.

PL/SQL Developer's Toolkit

A bundle of PL/SQL packages, provided with the Oracle WebServer SDK, that make it easier to generate HTML using PL/SQL. Applications written for either the PL/SQL Agent or the Java Interpreter can use these packages.

Port

A number that TCP uses to route transmitted data to and from a particular program.

Preferred Language

The language the Web Listener uses when handling a request for a file available in more than one language, if the request doesn't specify a

language. Language identifiers are defined by RFC 1766.

Primary Key

See *Key*.

Private Key

A key known only to one user, used to decrypt data encrypted with the user's public key. See also *Public-Key Encryption*.

Proxy Server

An HTTP engine, such as the Web Listener, that clients inside a firewall can use to access web sites outside the firewall.

Public Key

A key known to all users, used to encrypt data in such a way that only a specific user can decrypt it. See also *Private Key* and *Public-Key Encryption*.

Public-Key Encryption

A form of encryption that uses a key pair (a public key and a private key) to encrypt and decrypt data.

Query String

Optional portion of a URL that specifies parameters to be passed to some server extension.

Realm

A group of users and groups assigned by an authentication scheme to regulate access to specific files or directories.

Remote Database

For the WebServer, a database running on a different machine from the WebServer, which the WebServer accesses over the network.

Restriction

A security scheme that restricts access to files provided by the WebServer to client machines within certain groups of IP addresses or DNS domains.

Routing

The process of directing data from one machine on the Internet to another by way of intermediate machines.

Secret-Key Encryption

A form of encryption that uses a single key both to encrypt and to decrypt a document. Secret-key encryption is much faster that public-key encryption, but is more vulnerable to attack.

Secure Sockets Layer (SSL)

An emerging standard for secure transmission of hypertext documents over the Internet using secure HTTP (HTTPS).

Server

A process that executes requests on behalf of another process (the client) whose main purpose is to interface to the user. There are two types of servers relevant to this product. The first is the Oracle7 Server, which is a *database* server dedicating to performing data management duties on behalf of clients using any number of possible interfaces. The other is the Oracle WebServer itself, which is a *web* server, dedicating to answering requests that come in through the HyperText Transfer Protocol (http). This product is a web server that utilizes a database server.

Server Extension

Generic term for external programs executed by the Oracle WebServer whether through the WRB or CGI. Server Extensions are either WRB Services or CGI programs.

Server-Parsable

See *Parsable File*.

Server Side Includes (SSI)

Industry-standard term for LiveHTML. See *LiveHTML*.

Session Key

A secret key used by SSL to encrypt data transmitted over a secure connection. The client generates the session key after the WebServer authenticates itself and communicates it to the WebServer using public-key encryption.

Socket

The combination of an IP address and a port number.

SQL (Structured Query Language)

The industry-standard language for interfacing to relational databases such as the Oracle7 Server. See also *PL/SQL*.

SSI

See *Server Side Includes (SSI)* and *LiveHTML*.

SSL

See *Secure Sockets Layer (SSL)*.

Table

a. In HTML, a table is a way of presenting information to the user.

b. In SQL, a table is the basic way that data is structured, regardless of how it is presented to the user.

c. In PL/SQL, special kinds of SQL tables are used to function, effectively, as dynamic arrays.

Transmission Control Protocol (TCP)

The underlying communication protocol that the WebServer and its clients use to communicate HTTP requests.

Uniform Resource Locator (URL)

The text-string format clients use to encode requests to the WebServer.

User Directory

The subdirectory of a user's home directory in which the Web Listener searches by default for files when the user's home directory appears in the request URL.

Varchar2

A standard datatype of the Oracle7 Server. A variable-length string.

Virtual Machine (VM)

The mechanism the Java language uses to achieve its high portability. Java bytecode is executable by any Java Virtual Machine running on any actual machine. The VM converts the bytecode to the native code for the machine at hand.

Virtual File System

A mapping that associates the pathnames used in request URLs to the file system maintained by the host machine's operating system.

Virtual Pathname

A synonym that the virtual file system maps to a file stored in the file system maintained by the host machine's operating system.

Web Listener

See *Listener*.

Web Request Broker (WRB)

The core of the Oracle WebServer architecture. The Web Request Broker passes http requests that require the running of Server programs to various processes (WRB Executable Engines or WRBX's) that continuously run and await such requests. The WRB also includes an open

API, so you can run your own Server programs under it. The WRB is a more efficient alternative to the industry-standard CGI, but it can process requests that use CGI environment variables.

WebServer Administrator

Person in charge of configuring and running the Oracle WebServer.

WebServer Manager

A collection of utilities and HTML forms you can use to configure and maintain the WebServer installed on your computer.

Wrapper

A Java class that encapsulates another kind of object, possibly external to Java itself, such as a PL/SQL package.

WRB

See *Web Request Broker (WRB).*

WRB API

An open API used by the Web Request Broker (WRB).

WRB Cartridge

A program that is executed on the WebServer through the WRB API.

WRB Dispatcher

Portion of the WRB that distributes requests to running processes.

WRB Executable Engine (WRBX)

One of a pool of processes that the WRB maintains continuously, so that HTTP requests requiring the execution of programs are not slowed down by the performance cost of spawning a new process. WRBX's are associated with WRB Cartridges and are created and destroyed according to workload requirements.

WRB Service

A particular server extension to be run through the WRB. A WRB Service is the combination of a WRB Cartridge with its interface, the WRB API. The Oracle WebServer comes with three WRB Services: the PL/SQL Agent, the Java Interpreter, and the LIveHTML Interpreter. Using the WRB API, you can supplement these with your own.

APPENDIX

B

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

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     FNAMEL     NAME     ADDRESS
4005      JuliaP       eel       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
(cnuminteger NOT NULL PRIMARY KEY,
FNAMEchar(15) NOT NULL,
LNAMEchar(15) NOT NULL,
ADDRESSvarchar2 );
```

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 variable*s, also call*ed 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     LNAM     EFNAME     PHONE           TYPE
4005     Peel     Julia      375-296-8226h   ome
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:

```
CNUMLNAMEFNAMEPHONETYPE
4005PeelJulia375-296-8226home
4005PeelJulia375-855-3778beeper
4007SubchakTerryNULLNULL
4008LopezEmilio488-255-9011home
4008LopezEmilio488-633-8591work
4011LimKerry577-936-8554home
```

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

Table 0 - 2 Guide to Further Information on SQL

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.

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 "Naming Conventions" elsewhere in this appendix. 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.columnname%TYPE
```

In other words, you can either specify the type of values directly or use the %TYPE attribute (explained under "Declaring Variables", elsewhere in this appendix) 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", elsewhere in this appendix.

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 (parameter_name datatype, parameter_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 (parameter_name, parameter_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/SL code as written is still valid, the recompilation occurs automatically and invisibly to the user.

Creating Packages

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 procedures, as explained elsewhere in this appendix. The PL/SQL declarations are as outlined under DECLARE SECTION elsewhere in this appendix, 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" elsewhere in this appendix.
- A type inherited from such a record subtype using the %TYPE attribute. For more information, see "Declaring Variables" elsewhere in this appendix.
- A type inherited from a table, most likely the table the cursor queries, using the %ROWTYPE attribute. For more information, see "Records" elsewhere in this appendix.
- A type inherited from a cursor using the %ROWTYPE attribute. For more information, see "Records" elsewhere in this appendix.

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*.

APPENDIX

C Introduction To HTML

This appendix discusses the basic concepts of HyperText Markup Language, or HTML. This appendix is an introduction to HTML and provides important information on how to set up, format, and define HTML documents.

This appendix covers the following topics:

- *What is HTML?*
- *Getting Started*
- *Document Structure*
- *Body Tags*
- *List Tags*
- *Hypertext Linking*
- *Reviewing Changes to Your HTML Document*
- *Adding Style to Your HTML Document*
- *Special HTML Tags*
- *Tables*
- *Forms*
- *Creating Your Own HTML Document*
- *More Information about HTML*

Note: This appendix contains examples of formatted elements, such as underlined text. Some of these examples may not appear correctly on all on-line viewing systems.

What is HTML?

Hypertext Markup Language (HTML) is the standard language used for creating hypermedia documents on the Web. HTML documents can be viewed by many different Web browsers, of varying abilities, in a simple, portable way. When a document is coded in HTML, a browser. can interpret the HTML to identify the elements of the document and to render it. The use of HTML allows documents to be formatted for presentation using fonts and line justification appropriate for the system on which it is displayed.

Most documents have common elements, such as a title, paragraphs, or lists. Using HTML *tags* you can label these elements as you are writing. HTML tags provide the browser with a minimum of presentation information, while keeping the integrity of information in the document. All the reader needs is a formatting tool, a Web browser, which interprets the HTML tags and produces an on-screen display that approximates the intent of the document creator.

With most methods of documentation, the writer of a document has strict control over the look and feel of a document. With HTML, the *reader* (subject to the capabilities of the Web browser) has control over the look and feel of a document. HTML allows you to mark titles or paragraphs with HTML instructions or tags, and then leaves the interpretation of these tags up to the browser. For example, one browser may indent the beginning of each paragraph, and another may leave only a blank line. The user of a particular browser may also have some control over the specific fonts used.

HTML tags can be divided into two main categories:

- tags that define how the body of the document is to be displayed by the browser
- tags that define information about the document such as the title

Remember, the power of HTML is that your document can be viewed on a variety of browsers, on most platforms, and formatted to suit any reader.

How Are HTML Documents Created?

HTML documents can be created using any text editor (emacs, textedit, or vi on UNIX machines; DOS and Macintosh machines have a variety of simple text editors or HTML-specific programs). Choose the editor you are most comfortable with to write your HTML document.

For example, HTML editors such as SoftQuad's "HoTMetaL" allow the creation of HTML documents graphically in "what you see is what you get" (WYSIWYG) mode. In addition, many traditional word processing packages have add-ons or integrated HTML output capabilities.

To create dynamic pages that retrieve information from an Oracle7 Server, you can generate HTML using the PL/SQL utility packages provided by the Developer's Toolkit. See "Appendix B: The PL/SQL Developer's Toolkit," for more information.

Getting Started

All HTML tags begin with a < (left angle bracket), and end with a > (right angle bracket). There is usually a beginning tag and an ending tag.

An example is the title tag which surrounds the text that is designated as the document's title:

```
<TITLE>All the Hockey Greats</TITLE>
```

Tags are usually paired as follows:

```
<TITLE> and </TITLE>
```

The ending tag looks like the beginning tag except that a forward slash precedes the text within the bracket. In this example, the tag <TITLE> tells the Web browser to use a title format, and the </TITLE> tells the browser that the title heading is complete.

A few tags, such as <P>, which is a paragraph delimiter, do not need an end tag, but most do.

HTML is not case sensitive; therefore, the previous tags could look like this:

```
<title>All the Hockey Greats</title>
```

The convention used in this document is to capitalize all HTML format tags.

Note: Extra spaces, tabs, or returns that you have added by hand are highly discouraged, and will be lost. HTML only interprets tabs, extra spaces, and

returns enclosed by the <PRE> </PRE> tags. See “Preformatted Tag” for more information on this HTML option.

Document Structure

When a browser receives a document, it determines how it should be interpreted. The very first tag you need in your HTML document is the <HTML> structure tag. This declares that the content of your document is written with HTML. A minimal HTML document would look like this:

```
<HTML>...the content of the document...
</HTML>
```

Head Tag

The head tag can be used right after the HTML declaration, or not at all in your document. This tag represents the prologue to the rest of the file. Avoid putting any text into the document <HEAD> tag. This tag is placed immediately before and after the <TITLE> tag, as shown in the following example:

```
<HTML>
<HEAD>
<TITLE>All the Hockey Greats</TITLE>
</HEAD>
```

Note: Technically, the start and end tags for <HTML>, <HEAD>, and <BODY> are not needed. However, they are recommended because the head and body structure tags allow a browser to determine certain properties of a document, such as the <TITLE>, without having to parse, or go through the whole document.

Title Tag

Most browsers display the contents of the <TITLE> tag in the title bar of the window containing the document, and in the bookmark file of the browser if it supports one. The title, surrounded by <TITLE> and </TITLE> tags, is placed between the Head tags, as shown above. The title of a document does not appear in the contents of the document window, however. You must separately indicate it as a heading inside the body of the document if you want it to appear there.

Body Tags

The body tags specifically identify the body components in an HTML document. The body of an HTML document can contain links, text, and formatting information inside of the <BODY> and </BODY> tags.

Body Tag

The body of the document should be marked off with the <BODY> and </BODY> tags. This is the part of the document that is displayed as the page of text and graphics on your Web browser.

Heading Levels

When writing an HTML document, organize the text by heading levels to reflect its structure and organization. The first heading would be level 1, the next sub heading level 2, and so on. Most browsers recognize up to six heading levels, with six distinct styles. Heading levels above 6 are indistinguishable from one another.

The largest heading is a level 1 heading. The syntax of the head 1 is as follows:

```
<H1>Hockey Greats on Offense</H1>
```

Other headings can be created as follows:

```
<Hx>Text here</Hx>
```

where *x* is a number between 1 and 6 specifying the heading level. For example, if your next heading level is a level 3, the syntax would look like the following:

```
<H3>Hockey Defense </H3>
```

Paragraph Tag

Unlike most word processors, HTML usually ignores carriage returns. Word wrapping can occur at any point in your source file. Therefore, paragraphs must be separated with the <P> tag. If you do not separate your paragraphs with the <P> tag, your document will look like one long paragraph.

Preformatted Tag

The preformatted tag, <PRE>, allows you to present text formatted specifically to a screen. The preformatted text ends at the closing </PRE> tag. Within the preformatted text:

- Line breaks move to the next line
- The <P> tag is moved to the next line
- Horizontal tabs move in multiples of eight
- A fixed width font is used for all characters after the <PRE> tag.

Avoid using tags that define paragraph formatting, such as headings or address, within the <PRE> tags. They will have no effect.

Let's incorporate some of the previous examples to show what the document looks like with a title, a couple of heading level tags, and a few paragraphs:

```
<HTML>
<HEAD>
<TITLE>All the Hockey Greats</TITLE>
</HEAD>
<BODY>
<H1>All the Hockey Greats Before 1970</H1>
<H2>The Original Six Teams</H2>
This section deals with all of the hockey legends before the expansion.<P>
The game was very different for these players.<P>
There is no way New Jersey would have won the Stanley Cup. New Jersey just
would not have had the talent to do it.<P>
Chicago would still be on top.<P>
All would be well.<P>
</BODY>
</HTML>
```

The result would display something like this:

All The Hockey Greats Before 1970

The Original Six Teams

This section deals with all of the hockey legends before the expansion.

The game was very different for these players.

There is no way New Jersey would have won the Stanley Cup. New Jersey just would not have had the talent to do it.

Chicago would still be on top.

All would be well.

Note: The title, "All the Hockey Greats," would not show within the document itself. On most browsers it would be displayed in the title bar.

Forced Line Breaks

The
 tag forces a line to break. The best example of the use of this tag is for formatting addresses, or some other sequence of lines where you don't want the browser to add extra spacing. For example:

```
Sandy's Super Sundaes<BR>
123 Main Street<BR>
Anytown, USA<BR>
```

BlockQuote

The <BLOCKQUOTE> tag is used to contain text quoted from another source. The quote will be indented approximately 8 spaces. For example:

```
My favorite hockey saying is<P>
<BLOCKQUOTE>
```

```
Today is a great day for hockey.
</BLOCKQUOTE>
But I'm not sure if Bob Johnson really said it like that.</P>
```

This would appear something like the following:

My favorite hockey saying is

> Today is a great day for hockey.

But I'm not sure if Bob Johnson really said it like that.

Summary of Basic HTML Tags

The following table lists the basic HTML tags and their corresponding values:

Opening	***Closing***	***Definition***
<HTML>	</HTML>	An entire HTML document
<HEAD>	</HEAD>	The prologue of the document
<TITLE>	</TITLE>	Title of the document
<BODY>	</BODY>	Content of the document
<H1>	</H2>	First level heading
<H2>	</H2>	Second level heading
<H3>	</H3>	Third level heading
<H4>	</H4>	Fourth level heading
<H5>	</H5>	Fifth level heading
<H6>	</H6>	Sixth level heading
<P>		Paragraph
<PRE>	</PRE>	Preformatted text
 		Forced line break
<BLOCKQUOTE>	</BLOCKQUOTE>	Text quoted from another source

The previous section provides all you need to know to get started with HTML. At this point you can write a simple document in HTML. However, the following sections show you how to enhance your HTML pages to present information in many different fashions.

List Tags

There are three basic lists in HTML:

ordered

These lists have numbered items.

unordered

These lists have bullets to mark each item.

definition

These lists alternate a term with its definition

You can create nested lists with indents using the ordered or unordered tags. Simply place a second list (complete with its own start and end tags) within the first lists enclosing tags. Whether the nested list uses the same markers (numbers or bullets,) depends on the browser; some track the number of nests you use and change the markers of each successive nesting to blocks or other symbols. See the following example in the section “Nested Lists”.

Ordered Lists

In an ordered list, the browser automatically inserts numbers. Therefore, if you insert or delete an item in your ordered list, the numbers will reflect the change automatically.

An ordered list begins with <OL> and ends with </OL>. The individual list items are started with the <LI> tag. The following is an example of an ordered list:

```
<OL>
<LI>Gordie Howe
<LI>Rocket Richard
<LI>Howie Morenz
</OL>
```

Unordered Lists

In an unordered list the browser typically uses bullets or dashes to indicate the items in your list. (Each browser has its own way of indicating an unordered list)

An unordered list begins with <UL> and ends with </UL>. The following is an example of an unordered list:

```
<UL>
```

```
<LI>Gordie Howe
<LI>Rocket Richard
<LI>Howie Morenz
</UL>
```

Nested Lists

Here is an example of how to nest a list within another:

```
<HTML>
<HEAD>
<TITLE>All the Hockey Greats</TITLE>
</HEAD>
<BODY>
<H1>Hockey Greats before Expansion</H1>
<H2>The Original Six Teams</H2>
This section deals with all of the hockey legends before the expansion.<P>
<UL>
<LI>Gordie Howe
<LI>Rocket Richard
<LI>Howie Morenz
<UL>
<LI>great player
<LI>good stickhandler
</UL>
<LI>Bobby Orr
</UL>
</BODY>
</HTML>
```

Here's what the example would look like:

Hockey Greats before Expansion

The Original Six Teams

This section deals with all of the hockey legends before the expansion.

- Gordie Howe
- Rocket Richard
- Howie Morenz
    - great player
    - good stickhandler
- Bobby Orr

Note: When you create nested lists using HTML, you do not need to indent the nested HTML components. You may wish to do so for clarity, however.

Definition Lists

The definition list tags <DL>...<D / DL> enclose both the defined term (identified with the <DT> tag), and the definition of that term (identified with the <DD> tag). Most browsers format the definition on a separate line from the term. The following is an example of a definition list:

```
<DL>
<DT>Slapshot:
<DD>A shot used to drill the goalie at speeds up to 100 mph.
<DT>Wristshot:
<DD>A shot used to scare the goalie after the slapshot.
</DL>
```

The output looks like this:

Slapshot:
A shot used to drill the goalie at speeds up to 100 mph.

Wristshot:
A shot used to scare the goalie after the slapshot.

Hypertext Linking

Hypertext linking is the key characteristic that makes the Web appealing to users. By adding hypertext links, called *anchors* in your HTML document, you can create a highly intuitive information flow and guide the user directly to the information he or she needs.

Anchors have a standard format that allows any Web browser to interpret a link and perform the proper function (called a method) for that type of link. Links can refer to other documents, specific locations within the same document, or can perform operations, such as retrieving a file using FTP for display by the browser. URLs can refer to a specific location by an absolute pathname, or can be relative to the current document, which is more convenient when managing a large site.

Note: You can use hypertext links to navigate through a document or to move from document to document. However, HTML does not support returning you to the anchor point of a link within a document. If you use a hypertext link within a document, and then use the Back button, you do not return to the anchor, but to the previous point that you reached through a link.

What is a URL?

HTML uses what are called Uniform Resource Locators (URLs) to represent hypermedia links and links to network services within documents. The first part of the URL before the colon specifies the access method. The part of the URL after the colon is interpreted specifically according to the access method. In general two forward slashes after the colon indicate a machine name.

The general format of a URL is:

`method://machine-name/path/foo.html`

The following example would fetch the document `index.html` from the server `www.acme.com` using the HTTP protocol.

```
http://www.acme.com/index.html
```

A Uniform Resource Locator (URL) has the following format:

`method://servername:port/pathname#anchor`

The components of the URL are as follows:

method

is the name of the operation that is performed to interpret this URL. The most common methods are:

file

Read a file from the local disk. The filename is interpreted on the *user*'s machine, so this can be used to display any file that resides on the user's disk.

file

For example: `file:/home/jjones/jjones.html` displays the file `jjones.html` from the directory `/home/jjones` on the local machine.

http

Access a page over the network by way of the HTTP protocol. (This is the most common method, usually used to get an HTML document.)

http

For example: `http://www.acme.com/`
accesses Acme's home page.

mailto

Activate a mail session to the specified username and host.

mailto

For example: `mailto:jjones@us.acme.com` mails a message to jjones if the browser supports mail creation. Note that the `mailto` method does not require double forward slashes after the colon.

ftp

Retrieve a file using anonymous FTP from a server.

For example, `ftp://`*`hostname/directory/filename`*

servername

is optional and indicates the full hostname of a machine connected to the network. For example, `www.oracle.com` is the fully qualified hostname of Oracle's web server. If a servername is not specified, the URL is a *relative* link, and it is assumed that the file is on the same server that was used to display the current page. An IP address may be used instead of a hostname, although it is not recommended to build content with embedded IP addresses.

port

is the TCP port number that the web server is running on. The default is 80 if `:port` is not specified. This parameter is not used in most URLs.

pathname

is the relative or absolute pathname of the document being accessed by this URL. Web servers can be configured to interpret certain pathnames differently. For instance, CGI applications work by configuring the HTTP server to recognize that files within certain directories should be executed instead of being returned to the browser.

pathname

For example: `http://www.acme.com/index.html`

pathname

In this example, an HTTP connection is made to `index.html`, which is the name of the file to be accessed on the server `www.acme.com` using port 80 (the default). The file could have a full UNIX-style pathname to indicate a document contained in a lower level directory. If there is no directory component in the pathname, the document must be located in the server's document root directory which is configured by the server administrator. If the pathname part of the URL is missing, many servers provide a directory listing of the document root directory or access a specific 'top level' file (usually `index.html`).

#anchor

The named anchor points to a specific location within an HTML page. In addition to specifying a document name, specifying #anchor will cause most browsers to move the top of the display to the point referred to by the anchor. These anchor names are inserted into documents with the NAME tag as explained under "Linking to Sections on Pages" later in this appendix.

Structure of an Anchor Link within a Document

So far, we've discussed what a URL looks like. To cause a link to be displayed for the user to access, an anchor link must be embedded in the document text. The HTML syntax which allows this is:

```
<A HREF="URL">text_to_be_displayed_highlighted</A>
```

The <A HREF="*URL*"> tag opens the anchor link and the </A> tag closes it. All the text between those tags is displayed highlighted in some way by a web browser. A common technique is to display it underlined and in blue or some other user-selected contrasting color.

The URL part of the tag refers to the text of the URL reference as defined in the previous section. The URL text does not appear on the user's screen; it is only used when the user activates the link, usually by clicking on it with the mouse.

An example HTML segment:

```
For interesting products see
<A HREF="http://www.acme.com">Acme's home page.</A>
```

This line would produce on the user's screen:

For interesting products see **Acme's home page**.

Graphics within HTML Documents

One of the most compelling features of the Web is the ability to embed references to graphics and other data types within a document using the <IMG...ISMAP> tags. This adds a very lively character to your pages and makes them visually interesting.

There are two ways to use graphics from within an HTML document. The first is by embedding them within the document itself, so the user's screen will display the graphics within the context of the other elements of your document (such as explanatory text). This is the most common technique used by HTML designers and is called an "inline image". The syntax for specifying this is:

```
<IMG SRC="URL" ALT="text"
   ALIGN["top"|"middle"|"bottom"|"texttop"] ISMAP>
```

The elements in this syntax statement are as follows:

URL

is the same syntax as any other URL, as explained above. This is the way the browser accesses the actual image data file, which should be in a format supported by the browser. Currently GIF and JPEG formats are supported by most browsers. Specifying the URL is required.

ALT="text"

will cause the string *text* to be displayed if the browser is incapable of displaying images, or if image display is turned off. This is a way of 'labelling' the image the user would be seeing if image display was turned on. ALT is an optional keyword; if it is missing, no text will be displayed if images are turned off on the browser. Most browsers put some sort of icon on the screen to indicate an image would normally be there. Using the ALT tag is recommended so that your page is compatible with text-only browsers such as Lynx.

ALIGN

is used with one of the keywords to tell the browser where to place the next block of text. This allows a certain amount of creativity in the layout of your page. If this is not specified, most browsers put the image on the left side of the screen and fill in following text to the right of it.

ISMAP

tells the server that this image is a bitmap and allows the user to click on a location on the image to cause a URL to be accessed directly. Image maps are an advanced HTML feature and require server side configuration to function.

For example, the following line will cause the file `logo.gif` to be fetched from server `www.oracle.com` and the text `Oracle Logo` to be displayed if the user has graphic display turned off.

```
<IMG SRC="http://www.oracle.com/logo.gif" ALT="Oracle Logo">
```

Linking to Sections on Pages

You can link to a different area or section of your document by using a hidden reference marker to that specific section. This provides a quick way to move through sections of the document without having to scroll up or down. Once you click on that link, the hidden reference places you in the section, and the Browser presents the hidden marker line as the first line on the screen.

To create a link, follow these steps:

1. Create a named anchor marker in each section title that you might want to jump to. The HTML syntax is as follows:

```
<A NAME="named_anchor"> Text_to_link_to</A>
```

2. In the following example, a hidden reference marker is placed in a Head2 level:

```
<H2><A NAME="intro">Introduction to Hockey</A></H2>
```

3. Create the link by entering:

```
<A HREF="#named_anchor">Text</A>
```

4. For example, the link might appear as:

```
<A HREF="#intro">Gordie Howe</A>
```

The "#" symbol instructs your web browser to look through the HTML document for a named anchor called "intro".

When the user clicks on "Gordie Howe," the browser displays the heading "Introduction to Hockey" at the top of the screen.

Note: As specified earlier in the URL syntax, a link to a section can appear in the same document or in another document. The example here describes a link to a specific section of the same document.

Reviewing Changes to Your HTML Document

At some point, you will want to view the changes you have made to your HTML document. The following steps show you how to view your changes:

1. After you have edited your HTML file in your favorite text editor, save the file.
2. If you are currently displaying the document in a browser, you will need to reload it to see the changes. Select File -->Reload on the browser window. (In some browsers, reload will be under another menu item.)

The browser will read in the new file information and display the file with the changes you have made.

Adding Style to Your HTML Document

HTML tags offer several text styles so that you can emphasize any text in your document. The following is a short list of the most often used tag styles:

- bold
- italics
- underline
- mono spaced (typewriter style)

You can also use combinations of styles (for example, bold and italics)

Style	Element or Tag	Result
Bold	<B>I want this text bold</B>	**I want this text bold**
Italics	<I>I want this text italics</I>	*I want this text italics*
Underlined	<U>I want this underlined</U>	I want this underlined
Mono spaced(typewriter)	<TT>I want this text typed</TT>	`I want this text typed`

You can add style to text that appears anywhere in the HTML document. Combinations of styles can also be used, provided you assign all the closing tags needed.

```
<I>Hockey</I> is <B>Life</B>.<P>
```

becomes a paragraph that looks like this:

Hockey is **Life**.

The style tags surround the words they affect, in conjunction with other tags such as headings.

Be careful how many style tags you use on one page. If you use too many, the text may become difficult to read.

Special HTML Tags

The following tags or characters are options provided to make your HTML document more robust.

Address Tag

The <ADDRESS> tag is used to specify the author of a particular HTML document, and a way to contact the author (such as an email address). The syntax is as follows:

```
<ADDRESS>
address_of_author
</ADDRESS>
```

Escape Sequences

The following ASCII characters have special meaning within HTML and cannot be used in regular text:

- left angle bracket <
- right angle bracket >
- ampersand &
- double quote ”

To use these characters you must use these escape sequences:

For	**Use**
<	<
>	>
&	&amp
"	"

Table C-1:

There are several more escape sequences to support accented characters, such as the umlaut or the tilde. A full list of supported characters can be found in any number of reference manuals on HTML syntax. See also the list of on-line HTML references at the end of this appendix.

Tables

Tables in HTML organize data by row and column. Tables can contain a wide range of content, such as headers, lists, paragraphs, or figures. They can include any element or tag in HTML.

Cells can be merged across rows or columns.

Basic Table Tags

This section describes the basic table tags and their meanings.

Table: `<TABLE>...</TABLE>`

This is the main wrapper for all the other table tags. Other table tags can be ignored if they aren't wrapped inside of the <TABLE></TABLE> tags. By default, tables have no borders. Borders are added if the BORDER attribute is specified. See the next section "Basic Table Attributes".

Table Row: `<TR></TR>`

The number of rows in a table is specified by how many <TR> tags are contained within it. <TR> can have both the ALIGN and VALIGN attributes, which if specified become the default alignments for all cells in this row. See the next section, "Basic Table Attributes".

Table Data: `<TD></TD>`

This specifies a standard table data cell. Table data cells must only appear within table rows. Each row need not have the same number of cells specified, because short rows will be padded with blank cells on the right. A cell can contain any of the HTML tags normally present in the body of an HTML document. The default alignment of table data is ALIGN=left and VALIGN=middle. These alignments can be overridden by any alignments specified in the containing <TR> tag.

Note: Row alignments are overridden by any attributes assigned to a cell.

By default, lines inside of table cells can be broken up to fit within the overall cell width. Use the NOWRAP attribute described in the section, "Basic Table Attributes," to prevent line breaking for that cell.

Table Header: `<TH></TH>`

The table header cells are identical to data cells in all respects, except that header cells are in a bold font and have a default ALIGN=center.

Caption: `<CAPTION>... </CAPTION>`

This optional tag represents the caption for a table. The <CAPTION> tags should appear inside the <TABLE></TABLE> tags but not inside table rows or cells. The default alignment for the <CAPTION> tag is ALIGN=top but can be explicitly set to ALIGN=bottom. Like table cells, any document body HTML tags can appear in a caption. Captions are always horizontally centered with respect to the table, and they may have their lines broken to fit within the width of the table.

Basic Table Attributes

This section lists the basic table attributes and their meanings.

BORDER: This attribute appears in the Table tag. If present, borders are drawn around all table cells. If absent, there are no borders. By default space is left for borders, so a table has the same width with or without the border attribute.

ALIGN: If the ALIGN attribute appears inside a <CAPTION> </CAPTION> tag, it controls whether the caption appears above or below the table. It can have the values top or bottom. The default attribute is ALIGN=top.

When appearing inside a <TR>, <TH>, or <TD> tag, ALIGN controls whether text inside the table cell(s) is aligned to the left side of the cell, the right side of the cell, or centered within the cell. Values are left, center, and right.

VALIGN: The VALIGN attribute appears inside a <TR>, <TH>, or <TD> tag. This attribute controls whether text inside the table cell(s) is aligned to the top of the cell, the bottom of the cell, or vertically centered within the cell. It can also specify that all the cells in the row should be vertically aligned to the same baseline. Values are top, middle, bottom, and baseline.

NOWRAP: If the NOWRAP attribute appears in any table cell, the lines within this cell cannot be broken to fit the width of the cell. Be cautious in use of this attribute as it can result in excessively wide cells.

COLSPAN: The COLSPAN attribute can appear in any table cell and specifies how many columns of the table a specified cell should span. The default COLSPAN for any cell is 1.

ROWSPAN: The ROWSPAN attribute can appear in any table cell and specifies how many rows of the table this cell should span. The default ROWSPAN for any cell is 1. A span that extends into rows that were not specified with a <TR> tag will be truncated.

COLSPEC: The COLSPEC attribute can be used when needed to exert control over column widths, either by setting explicit widths or by specifying relative widths. Specify the table width explicitly or as a fraction of the current margins.

Example Table

The following is a table using some of the table tags and attributes we have previously discussed:

```
<TABLE BORDER>
<CAPTION ALIGN=bottom> Table #1 </CAPTION>
<TR><TD ROWSPAN=2> </TD> <TH COLSPAN=2>Average</TH></TR>
<TR><TH>Height</TH><TH>Weight</TH></TR>
<TR><TD>Males</TD><TD ALIGN=center> 69 </TD>
<TD ALIGN=center>150 </TD></TR>
<TR><TD>Females</TD><TD ALIGN=center> 64 </TD>
<TD ALIGN=center>130 </TD></TR>
</TABLE>
```

The table will look like this:

	Average	
	Height	**Weight**
Males	69	150
Females	64	130

Table #1

Forms

HTML pages can be formatted in any fashion, but remain read-only. The HTML *form* feature brings the added advantage of being interactive. An HTML form lets the Web user enter Comments and specify database search criteria.

When a form is interpreted by a Web browser, a special graphical user interface (GUI) screen is created with text entry fields, buttons, checkboxes, pull-down menus, and scrolling lists. When the Web user fills out the form and presses a button indicating the form should be *submitted*, the information on the form is sent to an HTTP server for processing by a CGI (Common Gateway Interface) program. For more information on CGI, see Chapter 3, “The Oracle Web Listener”.

When you write a form, each of your input items has an <INPUT> tag. When the user places data in these items in the form, that information is encoded into the form data and is known as the “value”.

All form elements have Name and Value attributes. Data are sent as NAME=VALUE pairs, separated by ampersands (&), where name is given in the NAME attribute, and value is given in the VALUE attribute, or replaced by the user.

This section illustrates the basic use of HTML forms. Forms can be used for simple table searches or complex queries to relational databases.

Forms Syntax

All forms begin with <FORM> and end with </FORM>.

The syntax is as follows:

```
<FORM METHOD=”get|post” ACTION=”URL”>Form_elements_and_other_HTML
</FORM>
```

METHOD

The request method supplies the data to the program. There are two request methods that can be used to access your forms. Depending on which request method you use, you will receive the encoded results of the form in a different way.

- **GET**: Information from a form is appended onto the end of the URL being requested. Your CGI program receives the encoded form input in the environment variable QUERY_STRING. Use of the GET method is discouraged.
- **POST**: This request method transmits all form input information immediately after the requested URL. Your CGI program will receive the encoded form input on standard input. The server will *not* send you an EOF on the end of the data; instead use the environment variable CONTENT_LENGTH to determine how much data you should read from standard input. This is the preferred method.

ACTION

ACTION specifies the URL being requested from the form. This URL will almost always point to a CGI script to decode the form results. If you are referring to a script on the same server as the form, you can use a relative URL.

TEXTAREA

The <TEXTAREA> tag is used to allow a user to enter more than one line of text in a form. The following is an example of the TEXTAREA tag:

```
<TEXTAREA NAME=”address” ROWS=14 COLS=60>
Chicago Blackhawks
1800 Madison Ave
Chicago, Il 60612
</TEXTAREA>
```

The attributes included within the <TEXTAREA> tag are used to initialize the field's value. The </TEXTAREA> tag is always required even if the field is initially blank.

The following are attributes of <TEXTAREA> and determine the visible dimensions of the field in characters:

- **NAME**-user defined name
- **ROWS**-height in characters of TEXTAREA
- **COLS**-width in characters of TEXTAREA

If you want text to appear within the text area, enter it between the start and end <TEXTAREA>tags.

INPUT

The <INPUT> tag allows you to input a single word or line of text, with a default width of 20 characters. It is usually preceded with some descriptive text.

The following are attributes of <INPUT>:

- **CHECKED**-When present indicates that a checkbox or radio button is selected.
- **MAXLENGTH**-The maximum number of characters that will be accepted as input. This limits the number of characters a user can type into the field. The form will give an error beep if the user tries to enter too many characters. This can be greater that specified by SIZE, in which case the field will scroll appropriately. The default is unlimited.
- **NAME**-Symbolic field name used when transferring the form's contents. This attribute is always needed and should uniquely identify this field.
- **SIZE**-Specifies how large an area to allocate, in characters, on the screen.
- **SRC**-A URL specifying an image for use only with TYPE=IMAGE.

- **TYPE**-Defines the type of data the field accepts. Defaults to free text. Several types of fields can be defined with the TYPE attribute:

  CHECKBOX

  Used for simple Boolean attributes, or for attributes that can take multiple values at the same time. The latter is represented by a number of checkbox fields each of which has the same NAME. Each occurrence of a checkbox is either ON or OFF. For a multi-valued attribute, there is a checkbox for each attribute, indicating whether the attribute applies. The various attributes are associated with one another by the fact that each attribute checkbox uses the same name. Each selected checkbox generates a separate NAME=VALUE pair in the submitted data, even if this results in duplicate names. The default value for CHECKBOX is ON.

  HIDDEN

  No field is presented to the user, but the content of the field is sent with the submitted form. This value may be used to transmit state information about client-server interaction, or for password information.

  IMAGE

  An image field upon which you can click with a pointing device, causing the form to be immediately submitted. The coordinates of the selected point are measured in pixel units from the upper left corner of the image, and are returned (along with the other contents of the form) in two NAME=VALUE pairs. The x-coordinate is submitted under the name of the field with .x appended, and the y-coordinate is submitted under the name of the field with .y appended. Any value attribute is ignored. The image itself is specified by the SRC attribute, exactly as for the <IMG> tag.

  PASSWORD

  Same as TEXT attribute, except that password text is not displayed as it is entered.

  RADIO

  For attributes which can take a single value from a set of alternatives. Each radio button field in the group should be given the same NAME. Only the selected radio button in the group generates a NAME=VALUE pair in the submitted data. Radio buttons require an explicit VALUE attribute.

  RESET

  This is a button that when pressed resets the form's fields to their specified initial values.

SUBMIT

This button, when pressed, submits the form. You can use the VALUE attribute to provide a non-editable label to be displayed on the button. The default label is application-specific. The NAME attribute, if used, passes a name=value pair along with the submitted form.

TEXT

Single line text entry fields. Use in conjunction with the SIZE and MAXLENGTH attributes. Use the TEXTAREA tag for text fields that can accept multiple lines.

- **VALUE**-Assigns an initial default value for the field, or the value when checked for checkboxes and radio buttons. This attribute is required for radio buttons.

Form Selection Menus

There are three types of selection menu tag for forms:

- **Select**: user selects from a fixed set of values represented by the *option* tag. This tag is usually displayed with a pull down menu.
- **Select single**: same as Select, but display is presented with a window with three items displayed at once. If there are more than three options, the window will have a scroll bar.
- **Select multiple**: allows multiple items to be selected from the menu.

SELECT

The SELECT tag allows the user to select a value from a fixed list. This is usually presented as a pull down menu.

The SELECT tag has one or more options between the start<SELECT> and end</SELECT> tag. By default the first option is displayed in the menu. The following is an example of a <SELECT> tag:

```
<FORM>
<SELECT NAME=group>
<OPTION>Gretzky
<OPTION>Messier
<OPTION>Coffey
</SELECT>
</FORM>
```

SELECT SINGLE

The SELECT SINGLE tag is the same as the SELECT tag, but options are displayed in a window with three items shown at once. If there are more than three options, the window will have a scroll bar. The SIZE tag within the SELECT tag specifies how many options will be shown in the window. The following is an example of a <SELECT SINGLE> tag:

```
<FORM>
<SELECT SINGLE NAME=group SIZE=3>
<OPTION>Gretzky
<OPTION>Messier
<OPTION>Coffey
<OPTION>Kurri
</SELECT>
</FORM>
```

In this example, the first three names would appear in the window, and a scroll bar would scroll to the last name.

SELECT MULTIPLE

The SELECT MULTIPLE tag is the same as the SELECT SINGLE tag, but the user can select more than one option in the window. The SIZE tag specifies how many lines appear in the window, and the MULTIPLE tag specifies how many options can be selected.

The following is an example of SELECT MULTIPLE:

```
<FORM>
<SELECT MULTIPLE NAME=group SiZE=3MULTIPLE=2>
<OPTION>Gretzky
<OPTION>Messier
<OPTION>Coffey
<OPTION>Kurri
</SELECT>
</FORM>
```

Note: On some browsers, it may be necessary to hold down the CONTROL or SHIFT key to select multiple items.

If multiple items are selected, they each get passed to the server with the same name. The decoding script has to be able to recognize multiple values associated with the same name.

Creating Your Own HTML Document

Now that most of HTML has been demystified, you can create your own HTML documents. Use any browser to view the HTML document you have created.

The following is an example of an HTML document that you can type word for word, or modify as you wish:

```
<HTML>
<HEAD>
<TITLE>Chicago Blackhawks: A Love Story</TITLE>
</HEAD>
<BODY>Chicago Blackhawks: A Love Story
<P>This is a story about my beloved Blackhawks. Year after painful year
they amass teams that could potentially win a Stanley Cup. But year after
insidiously painful year, they manage to lose in the first or second round
of the playoffs. If you have any suggestions as to how to reconcile this
evil, many fans would be grateful. Still we live on to love the Blackhawks.
<H2>Some Players We Have Loved</H2>
<UL>
<LI><A HREF="Hull.html">Bobby Hull</A>
<LI><A HREF="Makita.html">Stan Makita</A>
<LI><A HREF="Espo.html">Tony Esposito</A>
<LI><A HREF="Jr.html">Jeremy Roenick</A>
</UL>
<H2>Some Players We Love to Hate</H2>
<OL>
<LI><A HREF="Clark.html">Wendall Clark</A>
<LI><A HREF="Gilmour.html">Doug Gilmour</A>
<LI><A HREF="Domi.html">Tie Domi</A>
<LI><A HREF="cheapshot.html">Ulf Samuellson</A>
</OL>
<H2>List of Love Letters to Our Team</H2>
<UL>
<LI><A HREF="http://www.love2hawks.com/">Love Letters</A>
<LI><A HREF="http://www.fromNHL.com/">Letters from NHL</A>
</UL>
<HR><A HREF="Write">Write</A>to me.
Click below to send me Comments.
<BR> <A HREF="mailto:oldstadium@madison.com">
<I>crazed4hawks, oldstadium@madison.com</I>
</A>
</BODY>
</HTML>
```

More Information about HTML

For online information on HTML, the following URL sites provide a wealth of information:

`http://www.ncsa.uiuc.edu/demoweb/html-primer.html`

`http://union.ncsa.uiuc.edu/HyperNews/get/www/html.html`

`http://fire.clarkson.edu/doc/html/htut.html`

`http://ugweb.cs.ualberta.ca/~gerald/guild/html.html`

%ROWTYPE attribute, 192
%TYPE attribute, 189, 192

A

aliases
for database objects, 183
for database tables, 185
ALTER TRIGGER statement (SQL), 209
anchors
structure of, 223
used in hypertext linking, 220
applets
defined, 23
embedded in Web pages, 24
wrapping in server-side Java, 52
application developers, 11
applications
components of, 37
specifying, 40
structure of in Java, 55
arrays
implemented as tables in PL/SQL, 191
assignments
initial, 189
values to parameters (PL/SQL), 194
values to variables (PL/SQL), 195
authentication, 15
automatic recompilation (PL/SQL), 204

B

basic authentication, 15
blocks (PL/SQL), 187
Boolean logic
IF statements and (PL/SQL), 196
Three-Valued in SQL, 181
browsers
Java-enabled, 24
bytecode, 24

C

caching
files, 15
cattributes, 75
cattributes parameter
use in passing exact text, 75
certifying authorities, 19
CGI, 20
environment variables, 42, 69, 125
DCDs
how specified, 40
executing from LiveHTML, 70
LiveHTML can call, 26
changes
viewing in HTML document, 225
Common Gateway Interface (CGI), 20
compiler directives (PL/SQL), 200
compression, 16
Connect-String, 41
constants
declaring (PL/SQL), 190
constraints (SQL), 182
cookies, 146
correlation variables
in SQL statements, 185
in triggers, 209
CREATE FUNCTION statement (SQL), 201
CREATE PACKAGE BODY statement (SQL), 205
CREATE PACKAGE statement (SQL), 204
CREATE PROCEDURE privilege (SQL), 202
CREATE PROCEDURE statement (SQL), 201
CREATE TABLE statement (SQL), 182
crippled LiveHTML, 68
cursor variables (PL/SQL), 193
cursors (PL/SQL), 193
in package specifications, 204

D

data structures
in PL/SQL, 191
databases
changing data content of, 183
connecting to, 41, 53
connecting to in Java, 55
constraining, 182
creating objects in using PL/SQL, 187
creating tables in, 182
database triggers, 207
interfacing to, 180
missing data in, 181
nulls used in, 181
referencing objects in, 182
removing data from, 184
retrieving data from, 181
storing code in, 187, 201, 207
datatypes
composite, 188
converting, 189
in OWA_PATTERN, 133
inheritance of (PL/SQL), 189
OWA_TEXT uses, 143
PL/SQL, 188
PL/SQL Agent limitations, 49
PL/SQL Developer's Toolkit, 75
PL/SQL vs. Java, 55
pointer, 189
RECORD (PL/SQL), 192
scalar, 188
SQL, 182
structured, 188
TABLE (PL/SQL), 191
user-defined, 190
PL/SQL records, 192
PL/SQL tables, 191
DATE_GMT, 69
DATE_LOCAL, 69
DBMS_SQL package, 187
DBMSSTDX.SQL script, 204
DCDs, 53
format of, 41
how specified
CGI
variables used by WRB, 22
optimizing for multiple, 73
specified in URLs
URLs
DCDs used in, 23
declarations
constants (PL/SQL), 190
subprograms (PL/SQL), 193
types (PL/SQL), 190
variable (PL/SQL), 189
DECLARE section (PL/SQL), 188
DELETE statement (SQL), 184
DevelopersToolkit
proceduresandfunctions', 71
DevelopersToolkit', 71
digest authentication, 15
digital signatures, 19
DNS, 14
DOCUMENT_NAME, 69
DOCUMENT_URL, 69
domain names, 14
domains
restricting access to specified
IP addresses
restricting access to specified, 16
dot notation
for database objects, 182
for PL/SQL objects and subprograms, 189, 191

E

encryption
communications, 17
password, 15
environment variables, 69
retrieving, 125

error handling
Java, 55
LiveHTML, 68
Oracle7, 199
PL/SQL, 199
PL/SQL Agent, 50
triggers and, 209
exceptions (PL/SQL), 199
declared in DECLARE section, 193
declaring, 200
predefined, 199
user-defined, 200
EXECUTABLE section (PL/SQL), 195
EXECUTE privilege (SQL), 202
EXIT statement (PL/SQL), 198
extensions
filename, 16
extra path information, 39

F

fields
in PL/SQL records, 192
file formats
MIME types, 16
file sharing, 15
files
access control, 15
caching, 15
compression of, 16
filename extesions, 16
formats of
language, 16
firewalls, 14
flow control (PL/SQL), 195
FOR loops (PL/SQL), 197
FOR UPDATE clause (PL/SQL), 193
forced line breaks, 216
foreign keys
joins using, 184
Form tags, 232
Input, 232
TextArea, 232
form tags, 111
Forms, 211, 230
selection menus, 234
syntax, 231
forms
multivalued parameters, 45
parameter passing through, 43
frame tags
HTML frame tags, 98
FTP, 222
functions, 193
grouped into packages, 203
return values of (PL/SQL), 194
storing in database, 201

G

GET, 42–43
GOTO statement (PL/SQL), 198
error handling and, 201
graphics, 145
image maps, 25
within HTML documents, 223

H

host names, 14
HTML
basic concepts, 211–212
case sensitivity, 213
comments in, 68
dynamically generating
Web pages

dynamic, 40
extending, 130
form tags, 111
introduction, 211
Java generates dynamic, 62
list tags, 100
print tags, 75
structure tags, 77
table tags, 120
HTML Tags
ADDRESS, 227
BLOCKQUOTE, 216
BODY, 211, 214
BOLD, 225
CAPTION, 229
definition list, 220
escape sequence, 227
getting started, 211, 213
HEAD, 214
heading level, 215
I, 225
lists, 211, 218
mono spaced, 225
nested list, 219
ordered list, 218
P, 215
PRE, 215
TABLE, 228
TD, 228
TH, 228
TITLE, 214
TR, 228
U, 225
unordered list, 218
HTTP, 221
HyperText Markup Language, 211
HyperText Procedures
body tags, 82
character format tags, 106
definition of, 71
form tags, 111
head-related tags, 79
list tags, 100
physical format tags, 110
printing procedures, 75
structural tags (HTML), 77
table tags, 120

I

IF statement (PL/SQL), 196
image maps
defined, 25
dynamic, 25
imagemaps
generating dynamically, 145
initialization section (PL/SQL packages), 206
input parameters, 194
INSERT statement (SQL), 183
IP addresses, 13
ISMAP attribute, 224
iwa, 73

J

Java
application structure, 55
client side, 24
database connections, 55
error handling, 55
generating HTML from, 62
NULL handling in, 56
overview of, 23
PL/SQL combined with, 25
PL/SQL datatypes in, 55
proscribes direct memory management,

24
server side, 24
server-side, 52
wrappers for PL/SQL, 53
joins, 184
natural, 184
outer, 185

L

labels
PL/SQL, 198
languages
file formats and, 16
specified for WRBXs, 22
LAST_MODIFIED, 69
links
to other documents, 211, 220
to other sections, 224
list tags, 100
definition, 220
nested, 219
ordered, 218
types of, 211, 218
unordered, 218
Listener<$startpage>, 13
LiveHTML
environment variables, 69
executing programs from, 70
overview of, 25
tags, 68
loops (PL/SQL), 197
exiting nested, 198
FOR, 197
WHILE, 198

M

mailto, 221–222
memory
management of, 15
proscribed in Java, 24
multi_line, 143

N

naming conventions
PL/SQL, 191
tables, 192
SQL, 182
natural joins, 185
NLS
file formats and, 16
NOT NULL constraint (SQL), 182
PL/SQL use of, 189
nt_array, 143
NULL statement (PL/SQL), 197
nulls
as Boolean values, 181
database contains, 181
differentiated from NULL statement, 197
outer joins generate, 185
prohibiting, 182, 189
SQL handling of, 181
UPDATE statement (SQL) generates, 184
variables initialized to (PL/SQL), 189

O

objects
ownership of database, 182
oracle.html(Java class), 52
oracle.plsql(Java class), 52
oracle.rdbms(Java class), 52
ORACLE_HOME, 41
outer joins, 185
output parameters, 194
overloading
PL/SQL procedures, 49
ow_list, 143

OWA Utilities package
description of, 72
OWA_COOKIE, 146
OWA_IMAGE, 145
OWA_PATTERN, 131
OWA_TEXT, 142
OWA_UTIL Package, 123
OWAINS.SQL, 73
OWAINS.SQL sql script
using to install DevelopersToolkit', 73

P

packages, 203
bodies, 205
creating, 204
specification, 204
parameters
actual (PL/SQL), 194
formal (PL/SQL), 194
in PL/SQL, 194
in PL/SQL packages, 206
modes of (PL/SQL), 194
multivalued, 45
passing through forms, 43
parent keys, 179
PATH_INFO, 40, 42
PL/SQL
blocks, 187
datatypes
Java encapsulation of, 55
declarations, 188
DECLARE section, 188
embedding dynamic output in static Web pages, 26
EXCEPTION section, 199
EXECUTABLE section, 195
flow control in, 195
generating dynamically, 187
GOTO restrictions, 198
Java combined with, 25
overloading subprograms, 206
parameter passing, 40, 43
procedures
overloading, 49
scope of objects and subprograms, 190
sections of, 187
subprograms, 187
user-defined datatypes in, 190
PL/SQL Agent, 40
database access of, 23
datatype limitations, 49
environment variables, 42
error handling, 50
Java circumvents, 52
PL/SQL Agents, 22
connect to database, 53
PL/SQL Developer's Toolkit
ownership of, 73
PL/SQL Developers Toolkit
installation, 73
PL/SQL Developers' Toolkit
datatypes used, 75
PL/SQL packages
Java wrappers for, 53
PL/SQL tables
datatype limitations, 49
pl2java, 53
ports, 13
specifying in URLs, 39
POST, 42–43
pragmas (PL/SQL), 200
predicates
in IF statements (PL/SQL), 196
in SQL, 181

primary keys
enforcing, 182
print tags, 75
private synonyms, 183
privileges
object, 180
required to create triggers, 209
stored procedures require, 202
stored procedures use, 202
system, 180
PRIVUTIL.SQL, 74
procedures
declaring, 193
grouped into packages, 203
storing in database, 201
procedures and functions
getting started, 74
proxy servers, 14
public key encryption
encryption
public key, 18
public synonyms, 183
PUBUTIL.SQL, 74

Q

queries (SQL), 181
multiple tables used in, 184
stored in cursors (PL/SQL), 193
within other queries (subqueries), 185
query strings, 39
QUERY_FORM, 45
QUERY_STRING, 42
QUERY_STRING_UNESCAPED, 69

R

RAISE statement (PL/SQL), 200
range variables, 185
records
PL/SQL datatype, 192
ref cursors (PL/SQL), 189
regular expessions, 131
reload, 225
REQUEST_METHOD, 42–43
restriction, 16
RETURN statement (PL/SQL), 194
return values
cursor (PL/SQL packages), 204
functions (PL/SQL), 194

S

schemas, 182
scope
of PL/SQL objects and subprograms, 190
PL/SQL packages and, 205
SCRIPT_NAME, 40, 42
security
certifying authorities, 19
digital signatures, 19
encryption, 17
file access, 15
PL/SQL Agent database access, 23
proxy Internet connections, 14
public key encryption, 18
session keys
session keys
encryption
session keys, 18
SSL, 17
SELECT statement (SQL), 181
server extensions, 38
Server Side Includes
same as LiveHTML, 25
SGML comments, 68
SHTML, 68
sockets
defined, 14

SQL, 180
dynamic, 187
predicates, 181
standards, 180
Three-Valued Logic (TRUE, FALSE, NULL) in, 181
SQLCODE, 199
SQLERRM, 199
SSL
Overview of, 17
STANDARD package (PL/SQL), 199
stored procedures, 201
string matching, 131
structure tags, 77
subprograms (PL/SQL), 187
declaring, 193
error handling and, 201
GOTO restrictions, 199
overloading, 206
parameters of, 191
resolving ambiguous references, 191
scope and visibility of, 190
subqueries (SQL), 185
subtypes (PL/SQL), 190
using to create PL/SQL records, 192
using to create PL/SQL tables, 191
synonyms
for database objects, 183

T

Table attributes
Align, 229
Border, 229
Colspan, 229
Colspec, 230
Nowrap, 229
Rowspan, 229
Valign, 229
table tags, 120
Tables
in HTML, 211, 228
tables
database
aliases for, 183, 185
creating, 182
stored in cursors (PL/SQL), 193
HTML, 45
ownership of, 182
passing as parameters in PL/SQL, 191
PL/SQL
referencing, 192
PL/SQL datatype, 191
TCP/IP, 13
text/x-server-parsed-html, 68
text-only browsers, 224
Three-Valued Logic (TRUE, FALSE, NULL), 181
addressing in Java, 56
triggers, 207
altering, 207
creating, 207
enabling and disabling, 209

U

Uniform Resource Locator (URL), 221, 223–224
unique keys, 179
UPDATE statement (SQL), 184
URL, 211, 220, 223
URLs
GET vs. POST, 43
interpretation of, 39
parameter ordering, 45
specifying PL/SQL Agent with, 22
specifying ports in, 39
specifying secure connections with, 14
use of "owa" in, 22
use virtual file names, 14
used in applications, 37
Used, v
usernames
specified in DCDs

security
DCD DCD access control, 41
users
database
own schemas, 182

V

variables
assigning values to (PL/SQL), 195
cursor, 189, 193
declaring (PL/SQL), 189
in triggers, 209
initializing
in PL/SQL packages, 206
instantiation in PL/SQL packages, 204
vc_array, 143
virtual file systems
defined, 14
Virtual Machines (VMs), 24
virtual paths, 39
visibility
of PL/SQL objects and subprograms, 190
PL/SQL packages and, 205

W

Web Agent
optimizing multiple services, 73
Web Listener<$startpage>, 13
Web Listeners
control WRBX load, 22
file type negotiation, 16
interpret URLs, 12
memory management, 15
Overview of, 13
Web pages
applications and, 37
dynamic, 25, 38
from Java, 62
, 62
dynamic data embedded in static
LiveHTML
PL/SQL Agent can be called from
PL/SQL Agent
LiveHTML can call, 26
embedded
Web pages
dynamic
including in LiveHTML, 68
Web Request Broker (WRB), 21
WebServer
executing Java on, 24
Overview of, 12
WebServer Administrators, 11
WebServer DevelopersToolkit
installing', 73
securityconsiderations', 74
usingOWAINS.SQLtoinstall', 73
WebServer Manager
overview of, 19
WHILE loops (PL/SQL), 198
WRB, 21
CGI environment variables and, 42
overview of, 12
WRB cartridges
defined, 12
WRB Dispatchers
how invoked, 40
WRB Services
defined, 12
WRBXs
allocation of requests to, 22
connect to database, 53
connecting to database, 22
data passed to, 22
defined, 12
single-threaded, 22