

Using Oracle Web Application Server™ Cartridges

Release 3.0.1

ORACLE®

Enabling the Information Age



If you have not read this copyright page, you should read it in its entirety. If you have read this page, you can go to the Table of Contents.

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

Primary Authors: Francisco Abedrabbo, Martin Gruber, Kenman Rossi, Livingston Schneider

Contributors: Seshu Adunuthula, Mala Anand, Tony Casacuberta, Mike Freedman, Magnus Lonnroth, Raymond Ng, Murugan Palaniappan, Robert Pang, Ankur Sharma

This software was not developed for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It is the customer's responsibility to take all appropriate measures to ensure the safe use of such applications if the programs are used for such purposes.

This software/documentation contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited.

If this software/documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).

Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

If this software/documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with "Restricted Rights", as defined in FAR 52.227-14, Rights in Data - General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error-free.

REGISTERED TRADEMARKS of Oracle Corporation:

CASE Designer, CASE Dictionary, CASE Exchange, CASE Workshops, CoAuthor, ConText, Cooperative Development Environment, Cooperative Server Technology, Datalogix, Easy*SQL, Express, GEMMS, NLS*WorkBench, Oracle, Oracle Alert, Oracle Application Object Library, Oracle Book, Oracle Card, Oracle ConText, Oracle Financials, Oracle Glue, Oracle Leasing, Oracle Media Objects, Oracle Media Server, Oracle Power Objects, Oracle Press, Oracle Procedural Gateway, Oracle Secure Network Services, Oracle Transparent Gateway, OracleWare, Pro*Ada, Pro*COBOL, Pro*FORTRAN, Pro*Pascal, Pro*PL/I, Pro*Rexx, Secure Network Services, SQL*Connect, SQL*Forms, SQL*Loader, SQL*Menu, SQL*Module, SQL*Net, SQL*Plus, SQL*Report.

NON-REGISTERED TRADEMARKS of Oracle Corporation:

Advanced Networking Option, Advanced Replication Option, AIM Advantage, Alexandria, Alliance Online, Application Agent, Architected Best in Class, Athenia, Better Decisions Made Simple, CASE Generator, Charlotte, CDM Advantage, Content/2000, Corporate Planner Option, Database Server, DDE Manager, Des40, Designer/2000, Developer/2000, Discoverer, Dynamic Discovery Option, Easy*Query, Enabling the Information Age, End User Layer, Gist, Global Accounting Engine, Hyper*SQL, Intelligent Data Manager, Internet Video Server, InterOffice, J/SQL, Live HTML, Media Talk, Network Computing Architecture, Object Marketplace, ODP Pulse, ODP Techwire, Open/2000, Oracle Access, Oracle Access Manager, Oracle Accounts Receivable, Oracle Advanced Benefits, Oracle Agents, Oracle Application Display Manager, Oracle Applications, Oracle Applications Window Manager, Oracle Assets, Oracle Automotive, Oracle BASIC, Oracle Bills of Material, Oracle Bookbatch, Oracle BookBuilder, Oracle Browser, Oracle Business Analysis, Oracle Business Manager, Oracle Call Interface, Oracle Capacity, Oracle CASE, Oracle CDD/Administrator, Oracle CDD/Repository, Oracle Clinical, Oracle CODASYL DBMS, Oracle Cooperative Applications, Oracle Cost Management, Oracle Data Browser, Oracle Data Query, Oracle Departmental Server, Oracle DEVCONNECT, Oracle Developer Programme Pulse, Oracle Digital Library Solutions Framework, Oracle Documents, Oracle EDI Ex*tender, Oracle EDI Gateway, Oracle Energy, Oracle Engineering, Oracle Enterprise Interface Manager, Oracle Enterprise Manager, Oracle Enterprise Manager Performance Pack, Oracle Expert, Oracle Expert Option, Oracle Express Administrator, Oracle Express Analyzer, Oracle Express Server, Oracle Financial Analyzer, Oracle Financial Controller, Oracle Forms, Oracle Forms Generator, Oracle Foundation, Oracle GEMMS, Oracle General Ledger, Oracle Government Financials, Oracle Government General Ledger, Oracle Government Human Resources, Oracle Government Payables, Oracle Government Payroll, Oracle Government Purchasing, Oracle Government Receivables, Oracle Government Revenue Accounting, Oracle Graphical Schema Editor, Oracle Graphics, Oracle Human Resource Management Systems, Oracle Human Resources, Oracle Illustrated, Oracle Illustrated Series, Oracle Imaging, Oracle Incident, Oracle Industries, Oracle Installer, Oracle InstantSQL, Oracle Integrator, Oracle Internet Commerce, Oracle Internet Server, Oracle InterOffice, Oracle InterOffice Client, Oracle InterOffice Manager, Oracle InterOffice Server, Oracle Inventory, Oracle Magazine, Oracle Magazine Interactive, Oracle Manufacturing, Oracle Master Scheduling, Oracle Master Scheduling/ MRP, Oracle Media Data Store, Oracle Media Library, Oracle Mission Control, Oracle Mobile Agents, Oracle Module Language, Oracle MRP, Oracle MultiProtocol Interchange, Oracle Names, Oracle NetSolutions, Oracle Network Manager, Oracle Newsroom Manager, Oracle Object Marketplace, Oracle Objects, Oracle Office, Oracle Office Directory, Oracle Office Mail, Oracle Office Manager, Oracle Office Scheduler, Oracle Online, Oracle Open Client Adapter, Oracle Open Gateways, Oracle Open World, Oracle Order Entry, Oracle Parallel Server [or Oracle7 Parallel Server], Oracle Payables, Oracle Payroll, Oracle Personal Time and Expense, Oracle Planner Workbench, Oracle PowerBrowser, Oracle Procedure Builder, Oracle Process Modeller, Oracle Product Configurator, Oracle Project Accounting, Oracle Project Billing, Oracle Project Costing, Oracle Projects, Oracle Public Sector, Oracle Purchasing, Oracle Quality, Oracle RALLY, Oracle Rdb7, Oracle Receivables, Oracle Release Management, Oracle Replication Manager, Oracle Replication Services, Oracle Reports, Oracle Reports Generator, Oracle Repository Administrator, Oracle Revenue Accounting, Oracle RMU, Oracle Sales Analysis, Oracle Sales Analyzer, Oracle Sales and Compensation, Oracle Sales and Marketing, Oracle Sales Brief, Oracle Sales Compensation, Oracle Server Generator, Oracle Server Manager, Oracle Smart Video, Oracle Store, Oracle System Sizer, Oracle SQL*Tutor, Oracle SQL/Services, Oracle Supplier Scheduling, Oracle Supply Chain Planning, Oracle SupportNotes, Oracle Systems Designer,

Oracle Systems Modeller, Oracle Terminal, Oracle Text Server, Oracle TextServer3, Oracle Toolkit, Oracle TRACE, Oracle TRACE Collector, Oracle TRACE Option, Oracle Training Administration, Oracle Translation Manager, Oracle Universal Database, Oracle Upstream, Oracle Video Client, Oracle Video Server, Oracle Web Customers, Oracle Web Employees, Oracle Web Suppliers, Oracle WebServer, Oracle Work in Process, Oracle Workflow, Oracle Workgroup Server [or Oracle7 Workgroup Server], Oracle*Mail, Oracle7, Oracle7 Enterprise Backup Utility, Oracle7 Server, Oracle7 Spatial Data Option, Oracle8, Oracle 64 Bit Option, Oracle/2000, PC Express, Personal Express, Personal Oracle [or Personal Oracle7], Personal Oracle Lite, PJM Advantage, PL/SQL, Profit, ProREXX, Pro*C, Pro*C/C++, Pro*REXX, Programmer/2000, ProRexx, RDB7, Report Card, Security Without Compromise, Server/2000, Services/2000, Set-top/2000, Smart Application Client, Smart-Box, SmartCharts, SmartClient, SmartHints, SmartLayout, SmartSpring, SmartStandards, SmartTab, SmartTriggers, SQL*TextRetrieval, SQL*VDM, SupportAssistant, SupportNotes, SupportNews, The Oracle Network Builder, Trusted Oracle, Trusted Oracle7, Tutor, Video Client, Video Server, Web Request Broker, Workgroup/2000, World/2000.

SERVICE MARKS of Oracle Corporation:

BAP, Business Alliance Programme, CASE*Method, Cooperative Applications Initiative, International Oracle User's Group, International Oracle User's Week, IOUG, IOUW, Migration Technology Initiative, OOW, Operations Readiness Assessment, Oracle Alliance Program, Oracle Bronze, Oracle Business Alliance Programme, Oracle Consulting Services, Oracle Education, Oracle Gold, Oracle Master, Oracle Mercury, Oracle Metals, Oracle Platinum, Oracle Service, Oracle Silver, Oracle Sterling, Oracle SupportFax, Real Time Support Services, Systems Management Tools Initiative, Warehouse Technology Initiative, Web System Initiative.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Adobe and Acrobat are trademarks of Adobe Systems Incorporated.

Java is a trademark of Sun Microsystems Incorporated

Netscape Navigator is a trademark of Netscape Corporation.

Oracle, SQL*Forms, SQL*DBA, SQL*Loader, SQL*Net and SQL*Plus are registered trademarks of Oracle Corporation.

PL/SQL, Oracle7, Web Request Broker, LiveHTML, Web Access Manager, Oracle Browser, Oracle Web Application Server, Web Agent, Web Desktop, and Web Listener are trademarks of Oracle Corporation.

Alpha and Beta Draft Documentation Alpha and Beta Draft documentation are considered to be in prerelease status. This documentation is intended for demonstration and preliminary use only, and we expect that you may encounter some errors, ranging from typographical errors to data inaccuracies. This documentation is subject to change without notice, and it may not be specific to the hardware on which you are using the software. Please be advised that Oracle Corporation does not warrant prerelease documentation and will not be responsible for any loss, costs, or damages incurred due to the use of this documentation.

NOTICE

Copyright 1995 by: Massachusetts Institute of Technology (MIT), INRIA.

This W3C software is being provided by the copyright holders under the following license. By obtaining, using and/or copying this software, you agree that you have read, understand, and will comply with the following terms and conditions.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee or royalty is hereby granted, provided that the full text of this NOTICE appears on ALL copies of the software and documentation or portions thereof, including modifications, that you make.

THIS SOFTWARE IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, COPYRIGHT HOLDERS MAKE NO REPRESENTATION OR WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS. COPYRIGHT HOLDERS WILL BEAR NO LIABILITY FOR ANY USE OF THIS SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

CERN ACKNOWLEDGEMENT

This product includes computer software created and made available by CERN. This acknowledgment shall be mentioned in full in any product which includes the CERN computer software included herein or parts thereof.

Oracle Web Application Server 3.0 contains encryption and/or authentication engines from RSA Data Security, Inc. Copyright 1996 RSA Data Security, Inc. All rights reserved.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Contents

Chapter 1	Using the PL/SQL Cartridge	1-1
	Overview	1-3
	Tutorial	1-5
	Packages Overview	1-9
	Invocation	1-14
	Life Cycle of the PL/SQL Cartridge	1-15
	Supported Data Types	1-16
	Overloading	1-16
	Variables with Multiple Values	1-17
	NLS Extensions	1-20
	PL/SQL Cartridge and Applets	1-21
	Transactions	1-21
	Sessions/Cookies	1-24
	Customized Extensions to HTP and HTF Packages	1-24
	String Matching and Manipulation	1-25
	ICX	1-28
	Error-Reporting Levels	1-29
	Authentication and Security	1-29
	Dynamic Username/Password Authentication	1-30
	Custom Authentication	1-31
	Performance	1-33
	Troubleshooting	1-34
Chapter 2	Using the Java Cartridge	2-1
	Overview	2-2
	Developing Web Applications in Java	2-4
	Tutorial	2-6
	Developer's Guide	2-8
	Troubleshooting and Debugging	2-34
	Examples	2-37
Chapter 3	Using the LiveHTML Cartridge	3-1
	Overview	3-1
	LiveHTML Commands	3-3
	LiveHTML Examples	3-7
Chapter 4	Using the Perl Cartridge	4-1
	Overview	4-2
	Tutorial	4-3

Configuration	4-6
Invocation	4-8
Writing Perl Scripts for the Perl Cartridge	4-9
Developing Perl Extension Modules	4-12
Troubleshooting	4-12

Appendix A

The http and htf Packages	A-1
http.address	A-5
http.anchor, http.anchor2	A-6
http.appletopen, http.appletclose	A-7
http.area	A-8
http.base	A-9
http.basefont	A-10
http.bgsound	A-11
http.big	A-12
http.blockquoteOpen, http.blockquoteClose	A-13
http.bodyOpen, http.bodyClose	A-14
http.bold	A-15
http.center	A-16
http.centerOpen, http.centerClose	A-17
http.cite	A-18
http.code	A-19
http.comment	A-20
http.dfn	A-21
http.dirlistOpen, http.dirlistClose	A-22
http.div	A-23
http.dlistOpen, http.dlistClose	A-24
http.dlistDef	A-25
http.dlistTerm	A-26
http.emphasis, http.em	A-27
htf.escape_sc	A-28
htf.escape_url	A-29
http.fontOpen, http.fontClose	A-30
http.formCheckbox	A-31
http.formOpen, http.formClose	A-32
http.formHidden	A-33
http.formImage	A-34
http.formPassword	A-35
http.formRadio	A-36
http.formReset	A-37
http.formSelectOpen, http.formSelectClose	A-38
http.formSelectOption	A-39
http.formSubmit	A-40
http.formText	A-41
http.formTextarea, http.formTextarea2	A-42
http.formTextareaOpen, http.formTextareaOpen2, http.formTextareaClose	A-43
http.frame	A-44
http.framesetOpen, http.framesetClose	A-45
http.headOpen, http.headClose	A-46
http.header	A-47
http.htmlOpen, http.htmlClose	A-48
http.img, http.img2	A-49
http.isindex	A-50
http.italic	A-51
http.keyboard, http.kbd	A-52

http.line, http.hr	A-53
http.linkRel	A-54
http.linkRev	A-55
http.listHeader.	A-56
http.listingOpen, http.listingClose	A-57
http.listItem	A-58
http.mailto	A-59
http.mapOpen, http.mapClose.	A-60
http.menulistOpen, http.menulistClose	A-61
http.meta	A-62
http.nl, http.br.	A-63
http.nobr.	A-64
http.noframesOpen, http.noframesClose	A-65
http.olistOpen, http.olistClose	A-66
http.para, http.paragraph	A-67
http.param	A-68
http.plaintext	A-69
http.preOpen, http.preClose.	A-70
http.print, http.prn.	A-71
http.prints, http.ps	A-72
http.s	A-73
http.sample	A-74
http.script	A-75
http.small	A-76
http.strike	A-77
http.strong	A-78
http.style.	A-79
http.sub.	A-80
http.sup.	A-81
http.tableCaption	A-82
http.tableData	A-83
http.tableHeader	A-84
http.tableOpen, http.tableClose	A-85
http.tableRowOpen, http.tableRowClose.	A-86
http.teletype.	A-87
http.title	A-88
http.ulistOpen, http.ulistClose.	A-89
http.underline	A-90
http.variable.	A-91
http.wbr	A-92

Appendix B	The owa_cookie Package.	B-1
	owa_cookie.cookie data type	B-2
	owa_cookie.get function.	B-3
	owa_cookie.get_all procedure	B-4
	owa_cookie.remove procedure	B-5
	owa_cookie.send procedure.	B-6

Appendix C	The owa_image Package	C-1
	owa_image.NULL_POINT package variable	C-2
	owa_image.point data type	C-3
	owa_image.get_x function	C-4
	owa_image.get_y function	C-5

Appendix D	The owa_opt_lock Package	D-1
	owa_opt_lock.vcArray data type	D-2
	owa_opt_lock.checksum function	D-3
	owa_opt_lock.get_rowid function	D-4
	owa_opt_lock.store_values procedure	D-5
	owa_opt_lock.verify_values function	D-6
Appendix E	The owa_pattern Package	E-1
	owa_pattern.amatch function	E-2
	owa_pattern.change function and procedure	E-4
	owa_pattern.getpat procedure	E-6
	owa_pattern.match function	E-7
	owa_pattern.pattern data type	E-9
Appendix F	The owa_sec Package	F-1
	owa_sec.get_client_hostname function	F-2
	owa_sec.get_client_ip function	F-3
	owa_sec.get_password function	F-4
	owa_sec.get_user_id function	F-5
	owa_sec.set_authorization procedure	F-6
	owa_sec.set_protection_realm procedure	F-7
Appendix G	The owa_text Package	G-1
	owa_text.add2multi procedure	G-2
	owa_text.multi_line data type	G-3
	owa_text.new_row_list	G-4
	owa_text.print_multi procedure	G-5
	owa_text.print_row_list procedure	G-6
	owa_text.row_list data type	G-7
	owa_text.stream2multi procedure	G-8
	owa_text.vc_arr data type	G-9
Appendix H	The owa_util Package	H-1
	owa_util.bind_variables function	H-3
	owa_util.calendarprint procedure	H-4
	owa_util.cellsprint procedure	H-5
	owa_util.choose_date procedure	H-6
	owa_util.dateType data type	H-8
	owa_util.get_cgi_env function	H-9
	owa_util.get_owa_service_path function	H-10
	owa_util.get_procedure function	H-11
	owa_util.http_header_close procedure	H-12
	owa_util.ident_arr data type	H-13
	owa_util.ip_address data type	H-14
	owa_util.listprint procedure	H-15
	owa_util.mime_header procedure	H-16
	owa_util.print_cgi_env procedure	H-17
	owa_util.redirect_url procedure	H-18
	owa_util.showpage procedure	H-19
	owa_util.showsource procedure	H-20
	owa_util.signature procedure	H-21
	owa_util.status_line procedure	H-22

owa_util.tablePrint function	H-23
owa_util.todate function.	H-26
owa_util.who_called_me procedure.	H-27

1

Using the PL/SQL Cartridge

The PL/SQL Cartridge provides the environment for developing Web applications as PL/SQL procedures stored in an Oracle database server. (PL/SQL is Oracle Corporation's procedural language extension to SQL, the standard data access language for relational databases.)

Contents

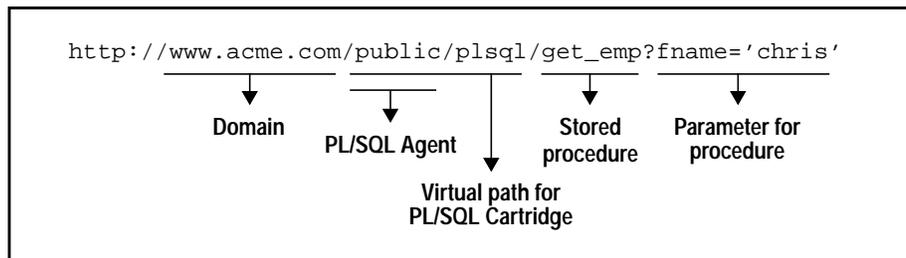
- [Overview](#)
- [Tutorial](#)
- [Packages Overview](#)
- [Invocation](#)
- [Life Cycle of the PL/SQL Cartridge](#)
- [Supported Data Types](#)
- [Overloading](#)
- [Variables with Multiple Values](#)
- [NLS Extensions](#)
- [PL/SQL Cartridge and Applets](#)
- [Transactions](#)
- [Sessions/Cookies](#)
- [Customized Extensions to HTP and HTF Packages](#)
- [String Matching and Manipulation](#)
- [ICX](#)
- [Error-Reporting Levels](#)
- [Authentication and Security](#)
- [Dynamic Username/Password Authentication](#)

- [Custom Authentication](#)
- [Regular Expressions](#)
- [Performance](#)
- [Troubleshooting](#)

Overview

The PL/SQL Cartridge enables Web users to connect to Oracle7 database servers. In each HTTP request for the PL/SQL Cartridge, the URL specifies the PL/SQL Agent (which contains connection information) and the name of the stored procedure to run. The URL can also contain values for any parameters required by the stored procedure. Figure 1-1 shows the parts of a URL:

Figure 1-1: Breakdown of a URL for the PL/SQL Cartridge



The following events occur when the Web Application Server receives a request (see Figure 1-2):

1. The Listener component of the Web Application Server receives the request from a client, and determines who should handle it. In this case, it forwards the request to the Web Request Broker (WRB) since the request is for a cartridge.
2. The WRB routes the request to an available PL/SQL Cartridge.
3. The PL/SQL Cartridge retrieves the name of the PL/SQL Agent from the request, and uses the agent's configuration values to determine to which database server to connect and how to set up the PL/SQL client configuration. You can define many PL/SQL Agents, each with different configuration information.
4. Using the PL/SQL Agent's configuration values, the PL/SQL Cartridge connects to the database, prepares the call parameters, and invokes the procedure in the database.
5. The procedure generates the HTML page, which can include dynamic data accessed from tables in the database as well as static data.
6. The output from the procedure is returned via the response buffer back to the PL/SQL Cartridge and the client.

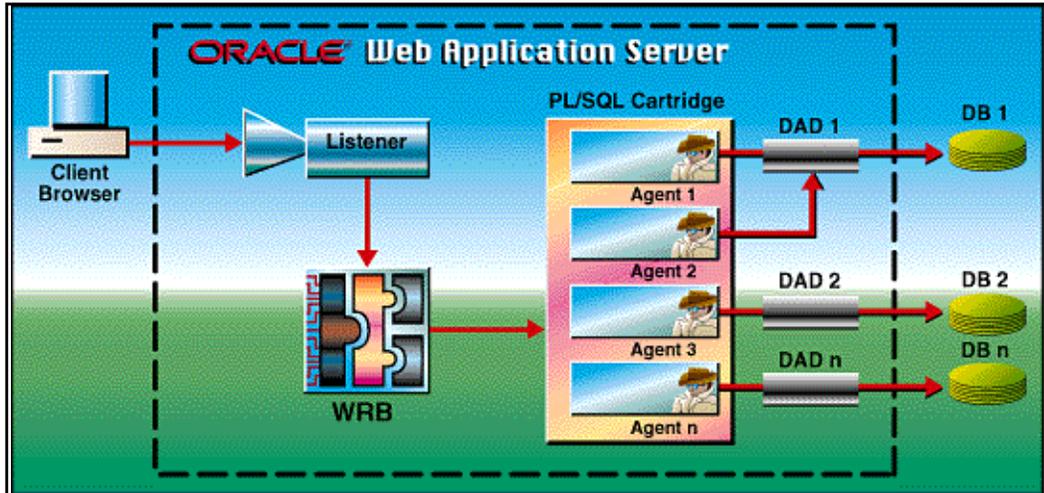
When connecting to a database, the PL/SQL Cartridge uses configuration information from two sources: a PL/SQL Agent and a Database Access Descriptor (DAD). Like the PL/SQL Agent, a DAD is a named set of configuration values used for database access. Each PL/SQL Agent is associated with a DAD.

A DAD specifies information such as the database name or the SQL*Net V2 service name, the ORACLE_HOME directory, and NLS configuration information such as language, sort type, and date language. You can also specify username and password information in a DAD; if they are not specified, the user will be prompted to enter a username and password when the URL is invoked.

The PL/SQL Agent specifies information such as which DAD to use, how much error information to return if an error occurs, a list of authorized ports that it can use, and transaction parameters.

The connection information is divided into PL/SQL Agents and DADs so that multiple agents can use the same DAD. This enables you to define a DAD for each database to which you want to connect, since it is the DAD that specifies the database. The only difference is in the configuration of the PL/SQL Agent. For example, the level of error information that is returned and transaction parameters.

Figure 1-2: Connecting to an Oracle7 Server using the PL/SQL Cartridge



The Web Application Server comes with the Web Application Server Manager, which is a set of HTML forms you use to configure the PL/SQL Cartridge, the PL/SQL Agent, and the DAD. On these forms you enter information such as virtual paths for the PL/SQL Cartridge, the SQL*Net V2 service name for the DAD, and the error reporting level for the PL/SQL Agent.

When you configure the PL/SQL Cartridge, you install packages that help generate HTML pages. These packages define procedures, functions, and data types that you can use in your stored procedures.

When designing your Web applications, you must consider security issues. You have to design your application such that unauthorized users do not have access to the application, and authorized users can run the application only in the proper context. See “[Authentication and Security](#)” for details.

Note: To users familiar with the PL/SQL Cartridge prior to Web Application Server version 3.0: Prior to version 3.0, the PL/SQL Agent configuration information was kept with the DAD information, and this combined information was called a Database Connection Descriptor (DCD). In version 3.0, this configuration information has been separated into PL/SQL Agent and DAD because other cartridges needed to use configurable connect information (DADs) independent of the configuration information specific to the PL/SQL Cartridge.

Tutorial

This section provides a step-by-step guide on creating and invoking a simple Web application. The application is a stored procedure that calls functions and procedures defined in the PL/SQL Cartridge packages to display the contents of a database table as an HTML table.

This tutorial steps you through the following tasks:

1. [Installing the PL/SQL Cartridge Packages and Creating a DAD](#)
2. [Configuring the PL/SQL Agent](#)
3. [Checking the Virtual Path Mapping](#)
4. [Stopping and Restarting the Listener](#)
5. [Creating and Loading the Stored Procedure onto the Database](#)
6. [Creating an HTML Page to Invoke the Procedure](#)

This tutorial assumes the following:

- You can log in as the “admin” user for the Web Application Server. This is required because you will be adding new settings to the configuration of the server.
- You have the “scott” schema on your database. This tutorial installs the PL/SQL Cartridge packages in this schema. If you don’t have the “scott” schema, you can use an existing schema on your database, or you can create the “scott” schema using the “CREATE SCHEMA” command.

A schema can be thought of as a user account: it is a collection of database objects such as tables, views, procedures, and functions, and each object in the schema can access other objects in the same schema.

1. Installing the PL/SQL Cartridge Packages and Creating a DAD

Before you can use the PL/SQL Cartridge, you need to install the PL/SQL Cartridge packages in the schema from which you will be running the procedure. This tutorial uses the “scott” schema.

To install the packages, you create a Database Access Descriptor (DAD). A DAD specifies connection information such as the database to which you want to connect, and the username and password to use to log into the database. You will use this same DAD later to run your stored procedure.

1. Start up your browser and go to the Web Application Server home page. The URL looks like:

```
http://your_machine_name/
```
2. Click Web Application Server Manager to go to the Web Application Server Administration home page.
3. Click Oracle Web Application Server to see more Administration options.
4. Click DAD Administration to go to the Database Access Descriptor page.
5. Click Create New DAD.

6. On the Create New DAD page, fill in these fields:

Field	Value	Description
DAD Name	scotts	The name that identifies the DAD
Database User	scott	The schema name
Identified by	Password	Information on the database user is stored in the database
Database User Password and Confirm Password	tiger	The password for scott
ORACLE_HOME	Example: /private/app/ oracle/product /7.3.2	The directory that contains the files for your Oracle7 database server
ORACLE_SID or SQL*Net V2 Service		The name of the database to which to connect Specify the name of ORACLE_SID if the database server is running on your machine. Otherwise, use the SQL*Net connect string (for example, wdk7322).
Database Role	Default	This field is used in the context for Content Service only.
NLS fields	Leave blank.	
Store the user name and password in the DAD	Select this option.	

Table 1-1: values for DAD

7. Click the Submit New DAD button.

2. Configuring the PL/SQL Agent

After you have created a DAD and installed the packages, you need to create a PL/SQL Agent to associate with the DAD. A PL/SQL Agent uses the connection information in the associated DAD to connect to the database server, and it also specifies the amount of error information to return to the client.

1. Click the Cartridge button on the bottom of the page to go to the Cartridge Administration page.
2. Click PLSQL to go to the PL/SQL Agent Administration page.
3. Click Create New PL/SQL Agent.

4. Enter these values for the fields:

Field	Value	Description
Name of PL/SQL Agent	agentScott	The name that identifies the PL/SQL Agent
Name of DAD to be used	scotts	The DAD to associate with the PL/SQL Agent
Protect PL/SQL Agent	TRUE	
Authorized Ports	Example: 7777	The port at which your Listener is listening
HTML Error Page	Leave blank.	
Error Level	Leave blank.	
DAD Username	Leave blank.	
DAD Password	Leave blank.	

5. Click Submit New Agent.

This could take some time as the packages are loaded into scott's schema.

3. Checking the Virtual Path Mapping

After you have created a DAD and a PL/SQL Agent, the Web Application Server creates a new virtual path mapping to the new PL/SQL Agent. The Dispatcher will direct URLs that specify this virtual path to your DAD. You can change this mapping if you like.

To see the new virtual path:

1. Click the WRB button on the bottom of the page to go to the Web Request Broker Administration page.
2. Click Applications and Directories on the left side.
3. There should be a row that contains `\agentScott\plsql` in the "Virtual Path" column, `PLSQL` in the "App" column, and `%ORAWEB_HOME%\bin` in the "Physical Path" column.

The first element, `agentScott`, names the PL/SQL Agent to use for this request. The second element, `PLSQL`, merely indicates to the user that the request is for the PL/SQL Cartridge.

`PLSQL` is the symbol for the PL/SQL Cartridge. The virtual paths for the `PLSQL` rows cause the Web Request Broker to send the request to the PL/SQL Cartridge.

The physical path specifies the directory where the Web Request Broker looks for shared libraries to load the cartridge.

4. Stopping and Restarting the Listener

After reconfiguring the Web Application Server, you have to stop and restart the Web Listener for the new configuration to take effect.

1. Click the Listener button on the bottom of the page to go to the Oracle Web Listener Administration page.
2. Click Stop to stop the Listener process.
3. Click Start to restart the Listener process.

5. Creating and Loading the Stored Procedure onto the Database

To create and load the `current_users` procedure (defined below) onto the database, save the text of the procedure in a file, and then run Oracle Server Manager to read and execute the statements in the file.

1. Type the following lines and save it in a file called `current_users.sql`. The `current_users` procedure retrieves the contents of the `all_users` table and formats it as an HTML table.

```
create or replace procedure current_users
AS
    ignore boolean;
BEGIN
    http.htmlopen;
    http.headopen;
    http.title('Current Users');
    http.headclose;
    http.bodyopen;
    http.header(1, 'Current Users');
    ignore := owa_util.tableprint('all_users');
    http.bodyclose;
    http.htmlclose;
END;
/
show errors
```

This procedure uses functions and procedures in the `http` and `owa_util` packages to generate the HTML page. For example, the `http.htmlopen` procedure generates the string `<html>`, and `http.title('Current Users')` generates `<title>Current Users</title>`.

The [owa_util.tablePrint function](#) queries the specified database table, and formats the contents as an HTML table.

2. Start up Server Manager in line mode. `ORACLE_HOME` is the directory that contains the Oracle7 database server files.

```
% $ORACLE_HOME/bin/svrmgr1 (UNIX)
> %ORACLE_HOME%\bin\svr,gr1 (MT)
```

3. Connect to the database as “scott”. The password is “tiger”.

```
SVRMGR> connect scott/tiger
```

4. Load the `current_users` stored procedure from the `current_users.sql` file. You need to provide the full path to the file if you started up Server Manager from a directory different than the one containing the `current_users.sql` file.

```
SVRMGR> @
Name of script file: current_users.sql
```

5. Exit Server Manager.

```
SVRMGR> exit
```

6. Creating an HTML Page to Invoke the Procedure

To run the `current_users` procedure, type in the following URL in your browser:

```
http://your_machine_name/agentScott/plsql/current_users
```

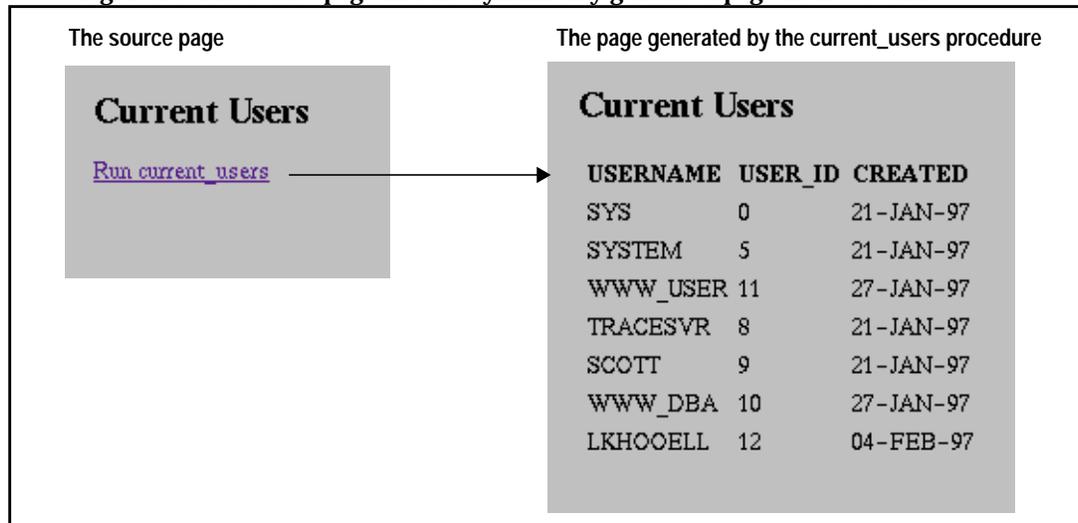
It is more common, however, to invoke the procedure from an HTML page. For example, the following HTML page has a link that calls the URL.

```
<HTML>
<HEAD>
<title>Current Users</title>
</HEAD>

<BODY>
<H1>Current Users</H1>
<p><a
href="http://hal.us.oracle.com:9999/agentScott/plsql/current_users">Ru
n current_users</a>
</BODY>
</HTML>
```

Figure 1-3 shows the source page (the page containing the link that invokes the stored procedure), and the page that is generated by the `current_users` stored procedure.

Figure 1-3: The source page and the dynamically generated page in the tutorial



Packages Overview

Before you can use the PL/SQL Cartridge, you need to load the packages listed in this section into the database schemas from which you want to run the procedures. The packages define data types, functions, and procedures that are used by the cartridge,

and you can use some of these in your Web application. The functions and procedures help you generate dynamic HTML pages that contain data retrieved from the database.

The packages are designed to be used in two ways:

- Store HTML pages in the database and update the dynamic parts when the page is requested.
- Generate the entire page dynamically from scratch.

Installing the PL/SQL Web Toolkit

To install the PL/SQL Web Toolkit, use the PL/SQL Agent administration forms. For details on these forms, see “[Web Application Server Manager](#)”. The installation procedure:

- Grants the CONNECT and RESOURCE roles to the database user. For more information on roles, see “GRANT (roles)” in Chapter 4 of the *Oracle7 Server SQL Reference*.
- Executes the `$ORAWEB_HOME/admin/owains.sql` script, which installs the packages in the PL/SQL Web Toolkit. 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 packages are installed in the user’s schema. This ensures that the user cannot use the subprograms in the packages to access data in another user’s schema.

htf and http Packages

The **htp** (hypertext procedures) package contains procedures that generate HTML tags, or text surrounded by HTML tags. For instance, the `htp.anchor` procedure generates the HTML anchor tag (<A>). The **htp** procedures can be grouped into the following categories:

- Structure procedures set up the major parts of the HTML document.
- Head-related procedures are used in the <HEAD> section of your HTML document. The HTML tags generated by these procedures should be placed between the tags generated by the `htp.headOpen` and `htp.headClose`.
- Print procedures are used with **htf** functions to generate a string in the HTML document being constructed. They can also be passed hard-coded text that appears in the HTML document as-is. The generated text is passed to the PL/SQL Agent, which sends it to the user’s browser.

You can use these procedures to generate “custom” HTML, that is, HTML that is not in the official specification. You enter the desired HTML markup and text that you want to generate by passing it to the procedures in the text buffers. This places the text verbatim in the web page you are generating.

- Body procedures are used in the <BODY> section of your HTML document. They can format paragraphs, and allow you to add comments, anchors, and images to your document.
- List procedures allow you to display information in lists:
 - *Ordered lists* have numbered items
 - *Unordered lists* have bullets to mark each item

- *Definition lists* alternate a term with its definition
- Character procedures 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 `Here is some text` might appear bold in some browsers, and italics in others.

If a specific text attribute, such as bold is desired, a physical format tag may be necessary.

- Physical markup procedures specify the format of the marked text. For example, you can explicitly direct the browser to render the text in a specific font size.
- Form procedures are used to create and manipulate HTML forms, which allow interactive data exchange between a web browser and a CGI program or WRB cartridge. Forms can have the following types of elements:
 - Input elements, which are used for a large variety of types of input fields. Examples of input elements include single line text fields, single line password fields, checkboxes, radio buttons, and submit buttons.
 - Text area elements, which are used to create a multi-line input field.
 - Select elements, which are used to allow the user to choose one or more of a set of alternatives described by textual labels. Usually rendered as a pull-down, pop up, or a fixed size list.
- Table procedures insert HTML tables in a document.

The **htf** (hypertext functions) package contains the function version of the procedures in the **htp** package. The function versions in the PL/SQL Web Toolkit do not directly generate output in your Web page. Instead, they pass their output as return values to the statements that invoked them.

To print the output of **htf** functions, call them from within the `htp.print` procedure, which simply prints its parameter values to the generated Web page. The advantage of using the functions is that you can nest calls.

Note: To look up **htf** functions, see the entry for the corresponding **htp** procedures. The string listed under “Generates” is the return value of the function.

owa Package

This package contains functions and procedures required by the PL/SQL Cartridge.

owa_init Package

This package contains functions and procedures that initialize the cartridge. It also provides constants that you override to set the time zone used by cookies. Cookies use expiration dates defined in Greenwich Mean Time (GMT). If you are not on GMT, you can specify your time zone using one of these two constants:

- If your time zone is recognized by Oracle, you can specify it directly using `dbms_server_timezone`. The value for this is a string abbreviation for your

time zone. See chapter 3 of the *Oracle7 Server SQL Reference* under “SQL Functions” for a list of recognized time zones.

```
dbms_server_timezone    constant varchar2(3) := 'PST'
```

- If your time zone is not recognized by Oracle, use `dbms_server_gmtdiff` to specify the offset of your time zone from GMT. Specify a positive number if your time zone is ahead of GMT, otherwise negative.

```
dbms_server_gmtdiff    constant number := NULL
```

After making the appropriate changes, you need to reload the package.

owa_sec Package

This package contains functions and procedures used by the cartridge for authenticating requests.

owa_util Package

This package contains utility procedures and functions. It is divided into the following areas:

- **OWA_UTIL HTML Utilities** - The purposes of these range from printing a signature tag on HTML pages to retrieving the values of CGI environment variables and performing URL redirects.
- **OWA_UTIL Dynamic SQL Utilities** - These enable you to produce Web pages with dynamically generated SQL code.
- **OWA_UTIL Date Utilities** - These make it easier to properly handle dates, which are simple strings in HTML, but are properly treated as a data type by the Oracle RDBMS.

owa_text Package

This package contains procedures, functions, and data types used by `OWA_PATTERN` for manipulating large data strings. They are externalized so you can use them directly.

owa_pattern Package

This package contains procedures and functions that you can use to perform string matching and string manipulation with regular expression functionality.

owa_image Package

This package contains data types and functions that you use to get the coordinates of where the user clicked on an image. You use this when the user clicks an image, and the location where the user clicked invokes the PL/SQL Cartridge. Your procedure would look something like:

```
create or replace procedure process_image (my_img in owa_image.point)
    x integer := owa_image.get_x(my_img);
    y integer := owa_image.get_y(my_img);
begin
    /* process the coordinate */
end
```

owa_cookie Package

This package contains data types, procedures, and functions that enable you to send HTTP cookies to and get them from the client's browser. HTTP cookies are opaque strings sent to the browser to maintain state between HTTP calls. State can be maintained throughout the client's session, or longer if an expiration date is included. Your system date is calculated with reference to the information specified in the `owa_init` package.

owa_opt_lock Package

This package contains functions and procedures that enable you to impose database optimistic locking strategies, so as to prevent lost updates.

Since HTTP is a stateless protocol, conventional database locking schemes cannot be used directly. The `owa_opt_lock` package works around this by giving you a choice of two ways of dealing with the lost update problem: the problem caused if a user selects and then attempts to update a row whose value has been changed in the meantime by another user. The two techniques this package provides are as follows:

- The hidden fields method - This stores the previous values in hidden fields in the HTML page. When the update is performed, it checks these values against the current state of the database. This is implemented with the procedure [owa_opt_lock.store_values procedure](#).
- The checksum method - This stores a checksum rather than the values themselves. This is implemented with the [owa_opt_lock.checksum function](#).

Both of these techniques are "optimistic". That is, they do not prevent other users from performing updates, but reject the current update if an intervening update has occurred.

Parameters Passed into Procedures and Functions

All parameters passed into a hypertext procedure or function are of data type VARCHAR2, INTEGER, or DATE. The data type is indicated by the first letter of the parameter's name: "c" for VARCHAR2, "n" for 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.

A vertical bar (|) in the syntax diagram means "or".

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 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"');
```

which would generate the following:

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

Invocation

To invoke the PL/SQL Cartridge, the URL must be in the following format:

```
http://host_and_domain_name[:port]/virtual_path/  
[package.]proc_name[?QUERY_STRING]
```

where:

- *host_and_domain_name* specifies the domain and machine where the Web server is running.
- *port* specifies the port at which the Web server is listening. If omitted, port 80 is assumed.
- *virtual_path* specifies a virtual path mapped to the PL/SQL Cartridge. The first element in the path specifies the PL/SQL Agent to use. For example, if the virtual path is `/public/plsql`, the PL/SQL Agent named `public` will be used to handle this request. At the very least, you need to specify a PL/SQL Agent in the virtual path.
- *package* specifies the package (if any) that contains the procedure. If omitted, the procedure is standalone.
- *proc_name* specifies the stored procedure to run. This cannot be a stored function.
- *QUERY_STRING* specifies parameters (if any) for the stored procedure. The string follows the format of the GET method. For example, multiple parameters are separated with the `&` character, and space characters in the values to be passed in are replaced with the `+` character. If you use HTML forms to generate the string (as opposed to generating the string yourself), the formatting will be done automatically for you.

For example, if a browser sends the following URL:

```
http://www.acme.com:9000/hr/plsql/  
get_emp?fname='john'&lname='doe'&role='office+manager'
```

the web server running on `www.acme.com` and listening at port 9000 would handle the request. When the Listener receives the request, it passes the request to the WRB because it sees that the `/hr/plsql` virtual directory is configured to call the PL/SQL Cartridge. The WRB then starts up an instance of the PL/SQL Cartridge with the PL/SQL Agent named `hr`. The instance connects to the database using the DAD associated with the Agent and runs the `get_emp` stored procedure. The *fname* parameter of the procedure gets the value `john`, the *lname* parameter gets the value `doe`, and the *role* parameter gets the value `office manager`. The space character is put back in before the stored procedure sees the value.

You need not be concerned with the order in which PL/SQL parameters are given in the URL or the HTTP header, since the parameter name is specified. An exception to this rule is when you have multiple parameters of the same name; this might happen if you have an HTML form that contains multiple elements with the same name. In this case, the parameter is passed to the PL/SQL procedure as a PL/SQL table. See [“Variables with Multiple Values”](#) for details.

Life Cycle of the PL/SQL Cartridge

This section describes what the PL/SQL Cartridge does when it receives a request. This section assumes knowledge of the callback functions used by the Web Request Broker (WRB). These functions are described in [“Callback Functions in a Web Cartridge”](#).

You do not need to know the information in this section in order to use the PL/SQL Cartridge. However, this information is useful if you want to optimize the performance of the cartridge, or want to know the architecture of the cartridge.

When a request comes in for the PL/SQL Cartridge, the Init callback function is executed. The Init function:

- loads all the PL/SQL Agents
- establishes connections to unique databases specified by the PL/SQL Agents
- logs on to databases if username/password information is provided in the DAD. Information required for the initialization process (such as database character set and authorization scheme) is also loaded at this time.

The Authorize callback function is executed when the PL/SQL Agent needs to authorize the URL request. The Authorize function:

- checks if the requested object is protected under any authorization schemes or restrictions
- checks the authorization level set in the `owa_sec` package to determine if custom authentication is specified (See [“Authentication and Security”](#) for details on custom authentication.)
 - If custom authentication is specified, the custom PL/SQL function that authenticates the user is executed.
- logs into the database if custom authentication succeeds and the DAD contains username and password information

If the Authorize callback function succeeds, the Exec callback function is called next. The Exec function:

- gets the values of the CGI environment variables
- determines the PL/SQL procedure to run
- determines the parameters for the procedure
- builds PL/SQL scripts that bind the variables, and executes the scripts, which execute the procedure and write the output to the client through the WRB

The Shutdown callback function is called automatically by the WRB. The Shutdown function closes all open connections.

Supported Data Types

Because HTTP supports character streams only, the PL/SQL Cartridge supports the following subset of PL/SQL data types.

- NUMBER
- VARCHAR2
- TABLE OF NUMBER
- TABLE OF VARCHAR2

Records are not supported.

Overloading

PL/SQL supports overloading, where multiple subprograms (procedures or functions) have the same name, but differ in the number, the order, or the data type family of the parameters. When you call an overloaded subprogram, the PL/SQL compiler determines which subprogram to call based on the data types passed. PL/SQL allows you to overload local or packaged subprograms; standalone subprograms cannot be overloaded. See the Oracle7 documentation for more information on PL/SQL overloading.

In addition to PL/SQL's restrictions on overloading, the PL/SQL Cartridge places one more restriction: you must give the parameters different names for overloaded subprograms that have the same number of parameters. The reason for this is that HTML data is not associated with data types, and this makes it impossible for the cartridge to know which version of the subprogram to call. For example, PL/SQL allows you to define the following two procedures, but you will get an error when you try to use them with the PL/SQL Cartridge because the parameter names are the same:

```
-- legal PL/SQL, but not for the PL/SQL Cartridge
CREATE PACKAGE my_pkg AS
    PROCEDURE my_proc (val IN VARCHAR2);
    PROCEDURE my_proc (val IN NUMBER);
END my_pkg;
```

To avoid the error, name the parameters differently. For example:

```
-- legal PL/SQL and also works for the PL/SQL Cartridge
CREATE PACKAGE my_pkg AS
    PROCEDURE my_proc (valvc2 IN VARCHAR2);
    PROCEDURE my_proc (valnum IN NUMBER);
END my_pkg;
```

To invoke the first version of the procedure, the URL looks something like:

```
http://www.acme.com/pub_agent/plsql/my_pkg.my_proc?valvc2=input
```

To invoke the second version of the procedure, the URL looks something like:

```
http://www.acme.com/pub_agent/plsql/my_pkg.my_proc?valnum=34
```

Variables with Multiple Values

A browser can return, to the server, variables that contain multiple values. For example, an HTML form that uses the `SELECT` element allows users to select one or more values from a given set. You can also have different form elements with the same value for the `NAME` attribute, in which case the values for these elements will be returned on the same variable name.

The PL/SQL Cartridge handles multi-value variables by storing the values in a PL/SQL table. This enables you to be flexible about how many values the user can pick, and it makes it easy for you to process the user's selections as a unit. Each value is stored in a row in the PL/SQL table, starting at index 1. The first value (in the order that it appears in the query string) of a variable that has multiple values is placed at index 1, the second value of the same variable is placed at index 2, and so on. If the order of the values in the PL/SQL table is significant in your procedure, you need to determine the order in which the variables appear in the query string.

If you do not have variables with multiple values, you do not have to worry about the order in which the variables appear, because their values are passed to the procedure's parameters by name, not by position.

The PL/SQL tables used as parameters in the PL/SQL Cartridge environment must have a base type of `VARCHAR2`. Oracle7 can convert `VARCHAR2` to other data types such as `NUMBER`, `DATE`, or `LONG`. The maximum length of a `VARCHAR2` variable is 32K.

If you cannot guarantee that at least one value will be submitted for the PL/SQL table (for example, the user can select zero options), use a hidden form element to provide the first value. Not providing a value for the PL/SQL table produces an error, and you cannot provide a default value for a PL/SQL table.

The following example passes a multi-valued parameter into a PL/SQL table. The form contains a `SELECT` element and a set of checkbox elements. Note the two hidden elements: one is used for the `SELECT` element, and the other for the checkboxes. This is the HTML that creates the form:

```
<html>
<head>
<title>Multivalue Example</title>
</head>

<body>
<h1>Multivalue Example</h1>

<p>This form shows how variables with multiple values are
handled by the PL/SQL Cartridge. The form has one SELECT
element and a set of checkbox elements.

<form method="PUT" action="/owa_default_service/owa/dept_machine">

<input type=hidden name="departments" value="no_value">
<input type=hidden name="machines" value="no_value">

<p>Select the departments in which you want to search:
```

```

<p>
<select name="departments" multiple>
  <option>Benefits
  <option>Marketing
  <option>Finance
  <option>Sales
  <option>Engineering
  <option>QA
  <option>Customer Support
</select>

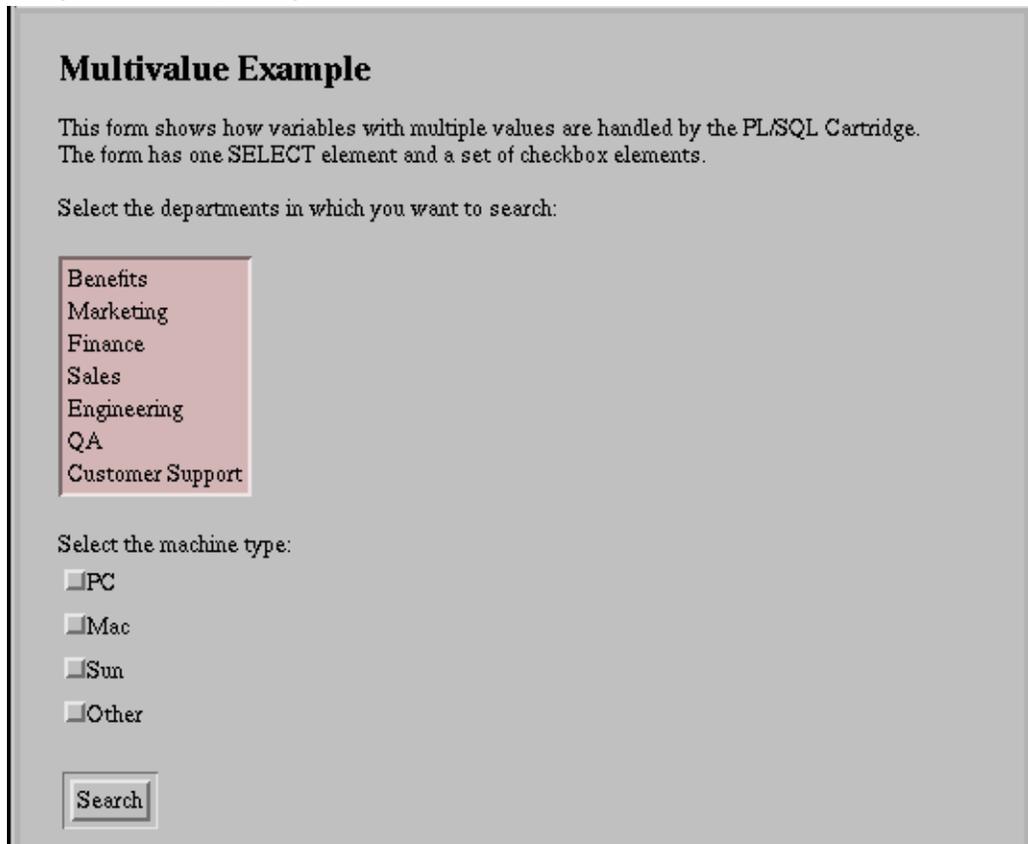
<p>Select the machine type:<br>
<input type="checkbox" name="machines" value="PC">PC<br>
<input type="checkbox" name="machines" value="Mac">Mac<br>
<input type="checkbox" name="machines" value="Sun">Sun<br>
<input type="checkbox" name="machines" value="Other">Other<br>

<p><input type="submit" value="Search">
</body>
</html>

```

Figure 1-4 shows the form as it appears in a browser:

Figure 1-4: Form passing in multiple values



When the user clicks Search, the dept_machine procedure runs on the database server. The procedure simply returns an HTML page that lists the user's selections. Note that the loop counter starts at index 2 because the hidden element values are in

index 1 in the PL/SQL tables. When the procedure prints out the number of rows in the PL/SQL tables, it subtracts one to avoid counting the hidden row.

```
create or replace procedure dept_machine (
    departments IN owa_util.ident_arr,
    machines    IN owa_util.ident_arr )
IS
    counter INTEGER;
    ct      INTEGER;
BEGIN
    http.htmlopen;
    http.headopen;
    http.title('Dept and Machines Results');
    http.headclose;

    http.bodyopen;
    http.header(1, 'Dept and Machines Results');

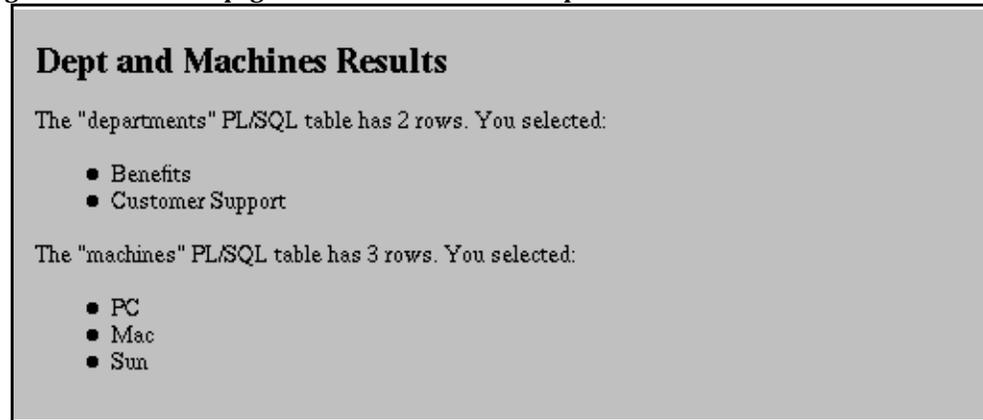
    ct := departments.COUNT - 1;
    http.print('The "departments" PL/SQL table has ` || ct || `
rows. ');
    http.print('You selected:');
    http.ulistOpen;
    FOR counter IN 2 .. departments.COUNT LOOP
        http.listItem(departments(counter));
    END LOOP;
    http.ulistClose;

    ct := machines.COUNT - 1;
    http.print('The "machines" PL/SQL table has ` || ct || `
rows. ');
    http.print('You selected:');
    http.ulistOpen;
    FOR counter IN 2 .. machines.COUNT LOOP
        http.listItem(machines(counter));
    END LOOP;
    http.ulistClose;
    http.paragraph;

    http.bodyclose;
    http.htmlclose;
END;
/
show errors
```

For example, if the user selected “Benefits” and “Customer Support” from the SELECT element, and “PC”, “Mac”, and “Sun” from the checkbox, the procedure returns an HTML page that looks like the following:

Figure 1-5: Generated page from the multivalue example



NLS Extensions

The NLS extensions are part of the DAD configuration, and they provide a flexible infrastructure to request and retrieve values to and from Oracle7 databases in different languages/formats. Even when the database server is configured with other NLS settings, all the conversions are handled implicitly by the database server and the PL/SQL Cartridge.

For example, if you have a database that is configured with US\$ for NLS Currency and you want to present the values in Japanese Yen to the user, all you need to do is to set NLS Currency to Japanese Yen. When the data is retrieved from the database, it will be presented as Japanese Yen.

The PL/SQL Cartridge in Web Application Server version 3.0 supports all the NLS extensions supported by the Oracle7 database server. Versions of PL/SQL Cartridge prior to Web Application Server 3.0 supported only the NLS_LANGUAGE parameter. This parameter is used by the PL/SQL Cartridge to derive the NLS_LANGUAGE, NLS_TERRITORY, and NLS_CHARSET parameters.

Web Application Server version 3.0 supports these NLS extensions:

- NLS_DATE_FORMAT specifies the format to print dates in the client browser.
- NLS_DATE_LANGUAGE specifies the language to print day and month names in the client browser.
- NLS_SORT specifies the type of sort to use when sorting within the database.
- NLS_NUMERIC_CHARACTERS specifies the decimal character and the grouping separator character.
- NLS_CURRENCY specifies the local currency system to print monetary values in the client browser.
- NLS_ISO_CURRENCY specifies the ISO currency symbol.
- NLS_CALENDAR specifies the calendar system to print dates in the client browser.

The new NLS extension parameters are optional. If you do not provide values for these parameters, the default values are derived from the NLS_LANG parameter. For example, if the value of NLS_LANG is AMERICAN_AMERICA.US7ASCII:

- the values for NLS_DATE_LANGUAGE and NLS_SORT are derived from the language part of NLS_LANG, and
- the values for NLS_CURRENCY, NLS_DATE_FORMAT, NLS_ISO_CURRENCY, and NLS_NUMERIC_CHARACTERS are derived from the territory part of NLS_LANG

See the Oracle7 documentation for details and valid values for these parameters.

PL/SQL Cartridge and Applets

When you reference an applet using the APPLET tag in an HTML file, the server looks for the applet class file in the directory containing the HTML file. If the applet class file is in another directory, you use the CODEBASE attribute of the APPLET tag to specify that directory.

When you generate an HTML page from the PL/SQL Cartridge and the page references an applet, you must specify the CODEBASE attribute because the cartridge does not have a concept of a current directory and does not know where to look for the applet class file.

The following example uses [http.appletopen](#) to generate an APPLET tag. It uses the *attributes* parameter to specify the CODEBASE value.

```
http.appletopen('myapplet.class', 100, 200, 'CODEBASE="/applets"')
```

generates

```
<APPLET CODE="myapplet.class" height=100 width=200 CODEBASE="/applets">
```

`/applets` is a virtual path that contains the `myapplet.class` file.

Transactions

Note: The Transaction Service is available only in the Advanced version of the Web Application Server. It is not available in the Standard version.

In versions of the Web Application Server prior to 3.0, a procedure or set of procedures executed through a URL request committed all the changes/transactions done within that sequence of PL/SQL code. This may not be preferred for certain situations. For example, in electronic commerce applications, you want users to be able to add or remove items in their shopping basket without doing a commit every time the users make a change or invoke a new request. The preferred behavior is to show an updated view of the table with the new uncommitted row values, and allow the user to commit or abort the transaction.

Web Application Server 3.0 lets you do that: the PL/SQL Cartridge in this version supports the WRB Transaction Service, which allows you to perform transactions that span several HTTP requests. The Transaction Service is based on the XA open model

transactions defined by the X/Open Company; the PL/SQL Cartridge acts as a transactional model client and the database is used as the resource manager.

How Does It Work?

Using the Transaction Service, you associate URLs with the operations on transactions (begin, commit, and rollback). When a user invokes one of these URLs, the corresponding transaction operation is performed by the Transaction Service.

These URLs invoke stored procedures that display appropriate pages to the user. For example, the begin transaction URL could display to the user a list of items to add to his or her shopping cart, the commit transaction URL could display to the user a list of purchased items, and the rollback transaction URL could display to the user a page that asks if he wants to drop the existing shopping cart and start another one.

Between the begin and the commit or rollback URL, the user would invoke other URLs that call procedures to perform some action on the database. These procedures might or might not be within the transaction boundary. If the URL is within the transaction boundary, the actions performed by that procedure would be committed or rolled back when the transaction ends. If the URL is not within the transaction boundary, it is not affected by the transaction, and the Web Application Server treats it as a regular request (changes made by that URL are committed upon completion).

A transaction boundary is usually a package, in which case all URLs that invoke procedures in the package are considered to be part of the transaction.

The sequence of URLs that are invoked would look like the following:

```
-- begin a transaction
http://host:port/agent_name/plsql/test.txn_begin

-- the first operation in the transaction
http://host:port/agent_name/plsql/test.txn_update1

-- the second operation in the transaction
http://host:port/agent_name/plsql/test.txn_update2

-- some more operations

-- commit the transaction
http://host:port/agent_name/plsql/test.txn_commit
```

In the example above, **test.begin**, **test.update1**, **test.update2**, and **test.commit** are procedures in the **test** package in the database. The **test** package marks the transaction boundary. You can give your procedures any name you like; the names used here are used only for clarification.

Note: In the procedures associated with transactions, including the ones within the transaction boundary, you *must not* call the PL/SQL statements that commit or roll back transactions. If you do, you cannot use the Transaction Service model.

If an error occurs before the commit or rollback transaction, you need to roll back the transaction by calling the URL associated with the rollback transaction. Here is an outline of the **test.update1** procedure:

```
procedure test.update1 (...)
```

```

begin
    -- update some tables here

exception
    when appropriate_exception then
        owa_util.redirect_url("/agent_name/plsql/test.rollback");
end;

```

The [owa_util.redirect_url procedure](#) generates a Location line in the HTTP header. You cannot call the rollback transaction procedure directly from within other procedures.

Configuring the PL/SQL Agent to Use Transaction Service

Specifying the Transaction Service is done at the PL/SQL Agent level. You can have some PL/SQL Agents that use the service, and others that do not.

To use the Transaction Service, you configure a PL/SQL Agent with the following parameters:

Parameter	Description	Example
Transaction Name	The name for this transaction.	txn_demo
Begin Transaction URI	The URI that starts a transaction.	/owa_dba/plsql/txn_demo.begin_url
Commit Transaction URI	The URI that commits a transaction.	/owa_dba/plsql/txn_demo.commit_url
Rollback Transaction URI	The URI that rolls back a transaction.	/owa_dba/plsql/txn_demo.rollback_url
Transaction Timeout	The number of seconds after which the transaction times out.	600 (the default value)
Transactional Boundaries	A list of URIs that belong to the transaction.	/owa_dba/plsql/txn_demo.*

In the example values, the name of the PL/SQL Agent is “owa_dba”.

A PL/SQL Agent switches between regular database connection and the Transaction Service context connection, depending on the URIs. When the URIs fall within the transaction boundary, the PL/SQL Agent uses the Transaction Service context connection to connect to the database.

Note: For transaction service to work for the PL/SQL Cartridge, you need to enable the TRANSACTION service for the cartridge. You can do this from any of these forms in the Web Application Server Manager:

- From the New Cartridge Configuration form or the Update Cartridge Configuration form, select “TRANSACTIONS” from the Services list.
- From the Web Request Broker Administration form, go to the “Applications and Services” section, and enter “TRANSACTIONS” for the cartridges for which you need the Transaction service.

Example

You could design an electronic commerce application that allows users to add items to their shopping carts, and the new values are not committed until the user clicks a Commit button. The example uses the values from the table above.

A transaction begins when the user invokes the URL:

```
http://host:port/owa_dba/plsql/txn_demo.begin_url
```

The **txn_demo.begin_url** procedure could generate an HTML page that displays to the user a list of items to add to his or her shopping cart. When the user adds an item to the shopping cart, the page would invoke a procedure that is within the transaction boundary so that the addition is considered part of the transaction but is not committed (for example, **txn_demo.add_item**); the procedure that is invoked could just add a new row to a table in the database and generate a page that displays the contents of the user's shopping cart. The page would contain buttons that allow the user to commit or roll back the transaction. The commit button would invoke the **txn_demo.commit_url** procedure, which could display to the user what he bought, and the rollback button would invoke the **txn_demo.rollback_url** procedure, which could ask the user if he wants to start another shopping cart.

Sessions/Cookies

Sessions are used by the Web Request Broker to maintain persistent states within cartridges through multiple accesses over a period of time. Since the PL/SQL Cartridge is unique in connecting to the database and all the states are maintained within the database, the concept of sessions does not apply to the PL/SQL Cartridge. Instead, cookies can be used to maintain persistent state variables from the client browser. For information about cookies, see:

- http://home.netscape.com/newsref/std/cookie_spec.html
- <http://www.virtual.net/Projects/Cookies/>

The `owa_cookie` package contains subprograms and data types that you can use to set and get cookie values:

- [owa_cookie.cookie data type](#) contains cookie name-value pairs.
- [owa_cookie.get function](#) gets the value of the specified cookie.
- [owa_cookie.get_all procedure](#) gets all cookie name-value pairs.
- [owa_cookie.remove procedure](#) removes the specified cookie.
- [owa_cookie.send procedure](#) generates a "Set-Cookie" line in the HTTP header.

Customized Extensions to HTP and HTF Packages

The **htp** and **htf** packages allow 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(cbufin      varchar2);
    procedure shout(cbufin     varchar2);
end;

create package body nsp as
    procedure blink(cbufin      varchar2) is
    begin http.print(nsf.blink(cbuf)); end;
    procedure shout(cbufin     varchar2) is
    begin http.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!');
    http.print('And I might ' || nsf.shout('get mad!'));
end;
```

String Matching and Manipulation

The `owa_pattern` package contains procedures and functions that you can use to perform string matching and string manipulation with regular expression functionality. The package provides the following subprograms:

- The [owa_pattern.match function](#) determines whether a regular expression exists in a string. It returns TRUE or FALSE.
- The [owa_pattern.amatch function](#) is a more sophisticated variation of the [owa_pattern.match function](#). It lets you specify where in the string the match has to occur. This function returns the end of the location in the string where the regular expression was found. If the regular expression is not found, it returns 0.
- The [owa_pattern.change function and procedure](#) lets you replace the portion of the string that matched the regular expression with a new string. If you call it as a function, it returns the number of times the regular expression was found and replaced.

These subprograms are overloaded, that is, there are several versions of each, distinguished by the parameters they take. Specifically, there are six versions of `match`, and four each of `match` and `change`.

The subprograms use the following parameters:

- *line* - is the string to be examined for a match. Despite the name, it can be more than one line of text. Its data type is either a `VARCHAR2` or a [owa_text.multi_line](#).

You can create a `multi_line` data type from a string using the [owa_text.stream2multi procedure](#). If a `multi_line` is used, the *rlist* parameter specifies a list of chunks where matches were found.

If the *line* is a string and not a `multi_line`, you can add an optional output parameter called *backrefs*. This parameter is a `row_list` that holds each string in the target that was matched by a sequence of tokens in the regular expression.

- *pat* - is the pattern that the subprograms attempt to locate in *line*. The pattern can contain regular expressions. See “[Regular Expressions](#)” for more information. Note in the [owa_pattern.change function and procedure](#), this parameter is called *from_str*.
- *flags* - specifies if the search is case-sensitive or if substitutions are to be done globally.

owa_pattern.match Function

Here is an example of the [owa_pattern.match function](#):

```
boolean foundMatch;  
foundMatch := owa_pattern.match('KAZOO', 'zoo.*', 'i');
```

This is how the function works: KAZOO is the target where it is searching for the regular expression “`zoo.*`”. The dot indicates any character other than newline, and the asterisk matches 0 or more of the preceding characters. In this case, it matches any character other than the newline.

Therefore, this regular expression specifies that a matching target consists of “zoo”, followed by any set of characters neither ending in nor including a newline (which does not match the period). The “i” is a flag indicating that case is to be ignored in the search.

In this case, the function returns TRUE, which indicates that a match had been found.

owa_pattern.change Function or Procedure

`owa_pattern.change` can be a procedure or a function, depending on how it is invoked. As a function, it returns the number of changes made. If the flag ‘g’ is not used, this number can be only 0 or 1. The “g” flag specifies that all matches are to be replaced by the replacement string. Otherwise, only the first match is replaced.

The replacement string can use the ampersand (&) to indicate that the portion of the target that matched the regular expression is to be included in the expression that replaces it. For example:

```
owa_pattern.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", because that is what matched the regular expression. Thus, this procedure replaces the string 'Cats in pajamas' with 'Cats in red'. If you called this as a function instead of a procedure, it would return 1, indicating that a single substitution had been made.

Regular Expressions

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

Regular expressions for the subprograms in this package can be either a VARCHAR2 or a [owa_pattern.pattern data type](#). You can create a [owa_pattern.pattern data type](#) from a string using the [owa_pattern.getpat procedure](#).

The following wildcard tokens are supported:

Token	Description
^	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 (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 hexadecimal 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

Token	Description
&	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 their meaning extended by any of the following quantifiers. You can also apply these quantifiers to literals.

Quantifier	Description
?	0 or 1 occurrence(s)
*	0 or more occurrences
+	1 or more occurrence(s)
{n}	Exactly <i>n</i> occurrences
{n, }	At least <i>n</i> occurrences
{n, m}	At least <i>n</i> , but not more than <i>m</i> , occurrences

Flags

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

Flag	Description
i	This indicates a case-insensitive search.
g	This applies only to CHANGE. It indicates a global replace. That is, all portions of the target that match the regular expression are replaced.

ICX

If you are running Oracle7 database server version 7.3.3 or later, you can perform ICX (intercartridge exchange) from the PL/SQL Cartridge. ICX enables cartridges to communicate with each other by making HTTP requests. The responses from the ICX calls can be received back by the PL/SQL Cartridge for further processing.

Consult the Oracle7 documentation for details on the ICX calls. For the 7.3.3 release, the information can be found in the “readme” file.

Error-Reporting Levels

In Web Application Server version 3.0, the PL/SQL Cartridge supports three levels of error-reporting. These levels control what the user sees when an error occurs. In earlier versions, the PL/SQL Cartridge displayed a static file in case of errors, and it was not possible to identify the error from the browser.

Now, two more levels of error-reporting are supported. The error levels are configured as part of the PL/SQL Agent. Errors are reported only during the Exec callback function.

Error level	Description
0	<p>Displays a static file in the client browser when an error occurs. Use this level if you do not want the users to see any information about the error.</p> <p>You can specify the file to return to the client. Otherwise, the default file is <code>\$ORAWEB_HOME/admdoc/error.html</code> (UNIX) or <code>%ORACLE_HOME%\admdoc\error.html</code> (NT).</p>
1	<p>Displays the date and time of the error, and also the URL that caused the error. Use this level if you want to provide only minimal information for the user to pass it on to Web site managers or application developers. The site manager or developer can use this information to help diagnose the error in the log file.</p> <p>If this error level is specified, the error page (if specified) is ignored.</p> <p>Example:</p> <p><i>Error occurred while accessing "/owa_dba/owa/myproc" at Mon Jan 6 16:33:32 1997.</i></p>
2	<p>Displays detailed information such as date and time of the error, the URL, the agent name, the procedure that was called, the parameter names and values, the web server error code, and the database error with a call stack. This error level is typically used only while developing or debugging an application.</p> <p>Example:</p> <p><i>Error occurred at Mon Jan 6 16:33:32 1997 OWS-05101: Agent: execution failed due to Oracle error 6564 ORA-06564: object show_stats does not exist ORA-06512: at "SYS.DBMS_DESCRIBE", line 55 ORA-06512: at line 1 OWA SERVICE: OWA_DEFAULT_SERVICE PROCEDURE: show_stats</i></p>

Table 1-2: Error levels in the PL/SQL Cartridge

Authentication and Security

The Web Application Server provides different levels of authentication mechanisms, which are documented in [Security](#). In addition to these mechanisms, the PL/SQL Cartridge provides two levels of authentication mechanisms that are different from the

ones provided by the Web Application Server. The Web Application Server protects the documents, virtual paths, and contents generated from the WRB, while the PL/SQL Cartridge protects the users logging into the database or the PL/SQL web application itself.

Note: Even though PL/SQL procedures are case-insensitive, the WRB protection mechanism is case-sensitive. To protect all packages and procedures you need to protect the virtual path for the PL/SQL Cartridge, for example: /owa_dba/plsql. Notice that this does not specify a particular package or procedure. To specify a single procedure or package, you must use custom authentication, which is described in the section “Custom Authentication” in “Using Web Application Server Cartridges”.

For more information, read the security white paper on Oracle’s web site. This paper describes how to develop secure PL/SQL web applications.

Dynamic Username/Password Authentication

In this scheme, access is controlled by the database itself; this scheme is suitable for applications that do not want to control the access on their own.

In Web Application Server versions prior to 3.0, the PL/SQL Cartridge stored the username/password information used to login to the databases in a configuration file. This file is read during the cartridge’s initialization phase, and the cartridge uses this information to log into the database. This mechanism did not provide any additional level of security other than those from the web server itself.

In Web Application Server version 3.0, the PL/SQL Cartridge does not require developers to store the username/password information in the configuration file. The username and password parameters under the Database Access Descriptor (DAD) section are optional; by leaving these parameters empty, the developer enables users to log into different schemas (database accounts) using the same PL/SQL Agent/DAD combination. Users will be prompted with a dialog box in the client browser to provide username and password information. This prompting happens during the authorization callback, and the user-supplied information will be used to log into the database schema that belongs to the given username/password.

This enables developers to develop applications that access data from different schemas; the schemas correspond to the given username/password.

This scheme alleviates the problem of creating multiple PL/SQL Agent/DAD combinations (DCDs in version 2.0) for multiple users. Note that the PL/SQL application and the Toolkit packages need to be loaded into all the schemas for this method to work. However, the availability and exposure of data from one schema into other is cumbersome and can be insecure.

This is suitable for applications where multiple users with their own database accounts interact through the web applications. For example, for an intranet HR application serving 100 employees within that company, version 2.0 required 100 DCDs to be created with 100 different usernames and passwords, whereas in version 3.0 you just need to create one PL/SQL Agent/DAD combination with no username and password.

Custom Authentication

Custom authentication is suitable for applications that want to control the access within the application itself or for applications that do not have a separate schema for every user who uses the application. For example, if you have a purchasing application accessed by many third-party vendors, you do not need to create a schema for every vendor. All you need to do is to write the application and load it into one schema where all the application users will be logging in. The application authenticates the users in its own level and not within the database level.

Custom authentication needs a static username/password to be stored in the configuration file, and cannot be combined with the dynamic username/password authentication. Custom authentication does not require the application and the Toolkit packages to be loaded in multiple schemas. The PL/SQL Cartridge uses the username/password provided in the DAD to log into the database. Once the login is done, authentication control is passed to the application, and application-level PL/SQL hooks (callback functions) are called. The implementations for these callback functions are left to the application developers. The return value of the callback function determines if the authentication succeeded or failed: if the function returns TRUE, authentication succeeded. If it returns FALSE, authentication failed and the code in the application is not executed. If you specify custom authentication and the callback function does not exist, you will get an error in the **wrb.log** file.

To enable custom authentication, you have to edit the following line in the **privinit.sql** file:

```
owa_sec.set_authorization(OWA_SEC.NO_CHECK)
```

Change the default parameter value of `OWA_SEC.NO_CHECK` to set a different level of authentication. The following table describes the other valid values:

Value for parameter	Access control scope	Callback function
<code>OWA_SEC.NO_CHECK</code>	n/a	n/a
<code>OWA_SEC.GLOBAL</code>	All packages	owa_init.authorize
<code>OWA_SEC.PER_PACKAGE</code>	Specified package	<i>packageName.authorize</i>
<code>OWA_SEC.PER_PACKAGE</code>	Anonymous procedures	authorize

- `OWA_SEC.GLOBAL`

If you set the parameter to `OWA_SEC.GLOBAL`, the **owa_init.authorize** function will be called whenever the PL/SQL Cartridge is invoked. For example, the following **authorize** function verifies that the user logged in as “guest” and specified “welcome” as the password and that the first and second numbers of the IP address of the client are 144 and 25.

```
create or replace package body owa_init is
  -- Global authorize callback function
  -- It is used when the authorization scheme is set to
  -- OWA_SEC.GLOBAL
  function authorize return boolean is
```

```

        ip_address    owa_util.ip_address;
begin
    -- prompt the user for login and password
    owa_sec.set_protection_realm('vendors');
    ip_address := owa_sec.get_client_ip;
    if ((owa_sec.get_user_id = 'guest') and
        (owa_sec.get_password = 'welcome') and
        (ip_address(1) = 144) and (ip_address(2) = 25))
then
        return TRUE;
    else
        return FALSE;
    end if;
end;

begin -- OWA_INIT package initialization
    owa_sec.set_authorization(OWA_SEC.GLOBAL);
end;

```

- **OWA_SEC.PER_PACKAGE**

If you set the parameter to `OWA_SEC.PER_PACKAGE` and the request specifies a procedure within a package, the **authorize** function in the package is invoked. If the procedure is not within a package, the anonymous **authorize** function is called.

For example, if the user invokes a procedure called **foo.print_page**, the **foo.authorize** function is called to authenticate the user.

```

create or replace package foo is
    procedure print_page;
    function authorize return boolean;
end;

create or replace package body foo is
    procedure print_page is
    begin
        htp.print('Hello World');
    end;

    function authorize return boolean is
    begin
        owa_sec.set_protection_realm('Enter Password for
foo');
        if ((owa_sec.get_user_id = 'guest') and
            (owa_sec.get_password = 'welcome') then
            return TRUE;
        else
            return FALSE;
        end if;
    end; -- authorize function
end; -- package body foo

create or replace package body owa_init is

```

```

-- The authorize function in the owa_init package will not be
-- invoked if the authorization level is set at
-- OWA_SEC.PER_PACKAGE.
begin -- OWA_INIT package initialization
    owa_sec.set_authorization(OWA_SEC.PER_PACKAGE);
end;

```

Performance

This section covers performance issues involved in configuring and running the PL/SQL Cartridge. Before addressing the performance issues, you should read “[Life Cycle of the PL/SQL Cartridge](#)” to be familiar with cartridge invocation and execution.

The [Apps] section of the WRB configuration file contains parameters that specify the minimum and maximum number of cartridges that can exist through the life of the system. These Min and Max parameters optimize the performance of the cartridge depending up on the incoming requests. Factors such as the number of requests, frequency of requests, and number of concurrent requests dictate the settings for the Min and Max parameters. The Min value specifies the number of cartridges that are spawned when the WRB starts up.

For example, if a system serves 10,000 requests per day with 50 concurrent requests and at a frequency of one request per minute, then setting Min to 50 and Max to 200 will optimize the performance of the system. The Min parameter should be set to a number slightly greater than or equal to the number of concurrent requests, and the Max value can be derived from the frequency and number of concurrent requests. Note that arbitrarily setting these numbers can lead to higher resource utilization and thus reduce the performance of the system.

When the cartridge starts up, it loads all the agent configurations associated with that cartridge and maintains physical database connections to every unique database server. This process takes some time and consumes the machine’s resources (memory).

For example, if you have 200 agents associated with 100 unique database servers and if it takes 1 second to connect to each database server, then the cartridge initialization will take 100 seconds with resources to maintain 100 database connections. This can degenerate the system when it scales.

To avoid this, you can partition the agents to map to different PL/SQL Cartridges. The Web Application Server version 3.0 allows you to name the cartridges with user-defined names. This gives you flexibility to associate a subset of agents with every cartridge. In the previous example, you could partition 100 agents associated with 50 unique database servers to be invoked through cartridge named PLSQL1 and the other 100 agents through cartridge name PLSQL2. This cuts down the initialization time into half.

The disadvantage of this method is that applications developed for cartridge PLSQL1 cannot be invoked through cartridge PLSQL2 if the agent names/virtual paths are hard coded within applications (as part of URL links and anchors). This can be avoided by using the CGI environment variable “SCRIPT_NAME” instead of hard coding the agent names/virtual paths.

Troubleshooting

- [Problems with Invoking Your PL/SQL Application](#)
- [Looking at Error Messages Generated by the Database](#)
- [Unhandled Exceptions](#)
- [Looking at the HTML Generated by Your PL/SQL Application](#)
- [Setting Tracing Levels](#)

Problems with Invoking Your PL/SQL Application

If your PL/SQL application cannot be invoked:

- Make sure that the PL/SQL Cartridge is registered with the WRB, and the virtual path for your application maps to the PL/SQL Cartridge.
- Make sure that the Web Listener and the WRB are functioning properly. For example, check that you can invoke other PL/SQL applications and other cartridges. You can try invoking the sample PL/SQL applications.
- Make sure that your PL/SQL subprogram that the URL references is a procedure, not a function.

Looking at Error Messages Generated by the Database

You can look at the database error messages that are returned to the user by setting the error-reporting level to the highest value, which is 2.

You set the error levels for each PL/SQL Agent using the Web Application Server Manager. See “[Error-Reporting Levels](#)” for details on the different error levels.

If you have logging turned on for the PL/SQL Cartridge, database error messages are logged to the WRB log file (**wrb.log**). To turn logging on for the cartridge, go to the “Applications and Services” section of the WRB Administration form, and check that the PL/SQL Cartridge has the Logger Service enabled. You should have a line that has `PLSQL` in the App column, and `LOGGER` in the Service Name column.

Unhandled Exceptions

If an error occurs in your PL/SQL application, an exception is thrown. If you do not handle the exception, the error is logged in the log file with the Oracle error stack and an error message is returned to the user. The error-reporting level controls what the user sees. See “[Error-Reporting Levels](#)” for details on the different error levels.

Recall that when a procedure exits with an unhandled exception, PL/SQL does not assign values to OUT parameters and does not roll back database work done by your procedure.

You can avoid unhandled exceptions by coding an OTHERS handler at the top level of your procedure.

Looking at the HTML Generated by Your PL/SQL Application

The PL/SQL Web Toolkit provides the [owa_util.showpage procedure](#), which you can use in Oracle Server Manager to print out the output of your application. The following example prints out the HTML generated by the `current_users` procedure (which was shown in the [Tutorial](#) section).

```
% svrmgr1
Oracle Server Manager Release 2.3.2.0.0 - Production
Copyright (c) Oracle Corporation 1994, 1995. All rights reserved.
Oracle7 Server Release 7.3.2.1.0 - Production Release
With the distributed option
PL/SQL Release 2.3.2.0.0 - Production
SVRMGR> connect scott/tiger
Connected.
SVRMGR> set serveroutput on
Server Output          ON
SVRMGR> execute current_users
Statement processed.
SVRMGR> execute owa_util.showpage
Statement processed.
<HTML>
<HEAD>
<TITLE>Current Users</TITLE>
</HEAD>
<BODY>
<H1>Current Users</H1>
<TABLE >
<TR>
<TH>USERNAME</TH>
<TH>USER_ID</TH>
<TH>CREATED</TH>
</TR>
<TR>
<TD ALIGN="LEFT">SYS</TD>
<TD ALIGN="LEFT">0</TD>
<TD ALIGN="LEFT">21-JAN-97</TD>
</TR>
<TR>
<TD ALIGN="LEFT">SYSTEM</TD>
<TD ALIGN="LEFT">5</TD>
<TD ALIGN="LEFT">21-JAN-97</TD>
</TR>
<TR>
<TD ALIGN="LEFT">WWW_USER</TD>
<TD ALIGN="LEFT">11</TD>
<TD ALIGN="LEFT">27-JAN-97</TD>
</TR>
<TR>
<TD ALIGN="LEFT">TRACESVR</TD>
<TD ALIGN="LEFT">8</TD>
<TD ALIGN="LEFT">21-JAN-97</TD>
</TR>
<TR>
```

```

<TD ALIGN="LEFT">SCOTT</TD>
<TD ALIGN="LEFT">9</TD>
<TD ALIGN="LEFT">21-JAN-97</TD>
</TR>
<TR>
<TD ALIGN="LEFT">WWW_DBA</TD>
<TD ALIGN="LEFT">10</TD>
<TD ALIGN="LEFT">27-JAN-97</TD>
</TR>
</TABLE>
</BODY>
</HTML>

```

Setting Tracing Levels

You can get detailed information about what the PL/SQL Cartridge is doing by increasing the tracing level. You can do this from the Logger Configuration form. In the form, go to the “System Messages Logging” section, and change the value in the Severity Level field.

The tracing messages are printed only to the **wrb.log** file; they are not sent to the user.

The severity levels range from 0 to 15; low values indicate that only errors are logged, while high values indicate that warnings and informative messages are also logged. For example, if you set the severity level to 8, you can see when the cartridge has performed the authentication and execution operations. The following table describes the severity levels:

Meaning	Severity	Recommended usage
Fatal errors (for example, memory errors)	0	0 indicates a core failure or a database error occurred.
Soft errors (for example, non-fatal input/output errors)	1	1 indicates that writing to file or resource failed.
	2	(user-defined)
	3	(user-defined)
Warnings (for example, missing file or missing configuration section)	4	4 indicates a configuration error.
	5	(user-defined)
	6	(user-defined)
Tracings (for example, request has been executed)	7	7 indicates the process has entered the init, terminate, or reload stages.
	8	8 indicates the process has entered the authentication and execution stages.
	9	(user-defined)
	10	(user-defined)

Meaning	Severity	Recommended usage
Debugging (for example, variable logging)	11	11 is used for printing debugging variables.
	12	(user-defined)
	13	(user-defined)
	14	(user-defined)
	15	(user-defined)

2

Using the Java Cartridge

The Java Cartridge provides a runtime environment where client browsers can send requests to Java applications running on the server side.

Contents

- Overview
- Developing Web Applications in Java
- Tutorial
- Developer's Guide
- Troubleshooting and Debugging
- Examples
- Reference

Overview

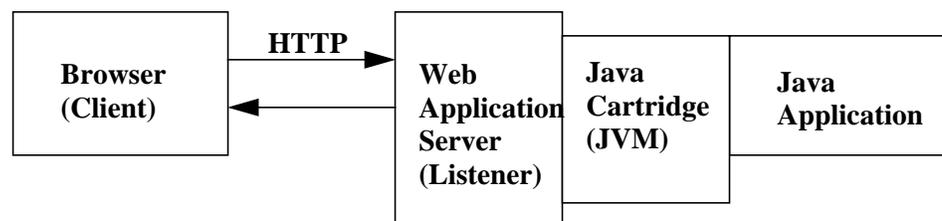
An HTTP request routed to the Java Cartridge is processed by running a Java application which returns the HTTP response. Java applications can take advantage Java functionality, such as:

- Object-oriented programming characteristics (inheritance, encapsulation and polymorphism), which allow code to be written with modular design, and encourage the reuse of code (such as with writing libraries)
- Access to a database (such as with the PL/SQL cartridge)
- Ability to perform other computing tasks such as programmatically inserting data
- A more general-purpose programming language (compared to PL/SQL)
- Platform-independent applications because of the bytecode interpreter approach
- Easy to add custom classes and packages
- Ability to incorporate legacy codes and libraries (in C) by the use of the native method interface

Architecture

When the Listener component of the Web Application Server receives requests for the Java Cartridge from browsers, the Java Cartridge interprets information in the URL to identify the Java application that should handle the request. The Java application then programmatically responds to the client, performs interim tasks, accesses a database, or performs other computing tasks. Ultimately, the Java application sends a response back up through the Java Virtual Machine (JVM) and the Web Application Server to the client.

Figure 2-1: Java Cartridge environment



Java Application vs. Applet

Java programs fall into two categories: applications and applets.

- A Java applet is downloaded from the net and runs in a Web browser. The browser prevents the applet from causing system damage or security breaches. For example, an applet is not allowed to read from the local file system. An applet must be a subclass of the Applet class, which has many entry points such as `init()`, `start()`, `stop()`, and `destroy()`.

- A Java application is typically loaded from the local file system. It is trusted and is not subject to security restrictions. An application's entry point is the `main()` method.

The Java Cartridge supports only Java applications. It does not support Java applets.

Java Cartridge

The Java Cartridge has the following characteristics:

- It uses Sun's Java interpreter, with its class library. The interpreter is converted into a Web Request Broker (WRB) cartridge.
- It gives better performance (no start-up/shut-down per request as in CGI), load balancing, and benefits provided by the WRB.
- It must be invoked by the Web Application Server. It is not an executable and cannot be executed separately from the command line.
- It does not include any Java compiler, debugger, disassembler, or other tool that is needed for Java application development. You need either the Java Development Kit (JDK) from JavaSoft or other hardware vendor, or a Java Integrated Development Environment, such as Microsoft Visual J++ or Symantec Cafe.
- It provides a Java Web Toolkit, which is a set of class libraries and tools for dynamic HTML generation, database access, HTTP access, and WRB access.

Invocation of the Application

The routing of the request from the Web Application Server to the Java Cartridge to the Java application (and back) is determined by information contained in the URL.

Using the HTTP **Get** form for a URL, the Dispatcher and the Listener route the request to the Java Cartridge as follows:

1. The Web Application Server receives the request identified by a URL. For example: **http://mymachine.mydomain/java/HelloWorld**.
2. The WRB of the Web Application Server dispatches the request. The WRB examines the URL and determines which cartridge should handle the request. If the URL is under one of the virtual paths that belong to the Java Cartridge, the WRB dispatches the request to the Java Cartridge. For example: **/java/HelloWorld** is under the virtual path **/java**, which maps to the Java Cartridge.
3. The virtual path settings of the Java Cartridge is in the WRB's configuration, under the **AppDirs** section.
4. The WRB finds a Java Cartridge, if available, or starts one if none is available.
5. The Java Cartridge receives the request, examines the URL, and finds the name of the application (class) to invoke. The name of the class is the tail part of the URL. For example, in **/java/HelloWorld**, **HelloWorld** is the name of the class.
6. The Java Cartridge loads the class. It invokes the class at its entry point, `main()`.
7. The Java Cartridge application generates a response, including both the HTTP response header and response body, and returns it through a special output stream (HtmlStream) using the print method. The Java Cartridge receives the response and returns it to the WRB. The WRB forwards the response to the browser that invoked this request.

Supported Versions of Java

The Java Cartridge is implemented using Java Virtual Machine version 1.0.2 from JavaSoft. You should use JDK 1.0.2 or an IDE based on 1.0.2 for developing Java applications for the Java Cartridge.

Development using other versions such as 1.1 should be possible if you limit your class usage to those found in 1.0.2. However, this has not been tested and is not supported.

Developing Web Applications in Java

- Structure of a Java Web Application
- Designing a Java Web Application with Static HTML Files
- Building the Java Application
- Adding and Running the Application into the Web Application Server

Note: This section assumes that you understand the general request-response approach of a Web application.

Structure of a Java Web Application

When the Java Cartridge invokes a Java Web application, it looks for an entry point in the application. Similar to the `main` function in a C program, the entry point of a Java Application is the `main` method. This method must be `public` and `static`. It takes an array of strings as the only parameter and does not have a return value. This entry point is consistent with that of Java applications invocable by the Java interpreter in the JDK. As an example, a class invocable by the Java Cartridge should have a `main` method similar to:

```
class HelloWorld {
    // The main method must be public and static
    public static void main (String args[]) { ... }
    ...
}
```

You should store the source code for each class using the class name plus the “.java” extension. For example, the `HelloWorld` example class should be stored in **`HelloWorld.java`**.

When a Java application is invoked by a Web request, it can retrieve the request object by invoking the `getRequest` method in the `HTTP` class in the `oracle.owas.wrb.services.http` package. With the request object, you can access information in the HTTP request.

To generate a response back to the client, use the `print` method on an `HtmlStream` object in the `oracle.html` package. An `HtmlStream` object can be obtained by invoking the `Stream` method in `HtmlStream` class.

For most Web requests, the response will be returned in HTML format. You may print the HTML content directly to `HtmlStream`. Oracle’s Java Web Toolkit provides a Dynamic HTML Generation package (in `oracle.html` package) which handles the

details of HTML generation. It also allows you to generate HTML pages in an object-oriented approach.

For more information on handling Web requests and responses, see the “Developer’s Guide”.

Designing a Java Web Application with Static HTML Files

There are two methods of designing a Java Web application:

- Your Java application may generate all the HTML content. You may store the static portions of the HTML content in files.
- You may generate only those portions of the HTML content that are dynamic. When serving a request, you can combine the static portions with the dynamic ones which you generate.

The second approach has the following advantages:

- It gives you better performance because you construct only the portions of your HTML page that change in each request.
- By separating the static portions from the dynamic ones from your HTML pages, you can change the HTML layout of your Web Application without touching your Web application.

The Dynamic HTML Generation package enables you to generate HTML pages by incorporating dynamic HTML content in static HTML pages. Using this feature, you can create the basic skeleton of HTML pages in separate HTML files. A skeleton HTML page contains placeholders that will be replaced with HTML that you dynamically generate with your Web application.

More details of generating HTML with static HTML pages are discussed in the “*Using Dynamic Content*” section.

Building the Java Application

The Java Cartridge only provides the Java runtime environment, specifically the Java Interpreter and a set of class libraries. To develop a Java Web application, you also need a Java compiler and a debugger. These tools are parts of many Java development tools. They include:

- Java Development Kit (JDK) - The basic Java development tool provided by JavaSoft. The JDK for Sun Solaris and Microsoft Windows 95/NT can be downloaded from JavaSoft’s home-page <http://www.javasoft.com>. Other hardware vendors also provide JDK for their platforms.
- Borland’s JBuilder
- Microsoft’s J++
- SunSoft’s Java Workshop
- Symantec’s Cafe

Because the Java Cartridge does not include a debugging environment to develop your Java Web application, you should use your JDK or Java IDE to write and debug your application in the same way you do for your other Java applications.

When you debug your applications in these tools, some of the functionalities in the Java Web Toolkit may be disabled because your Web application is not executed in the WRB environment. In these situations, you may need to insert dummy code in your application to substitute for Java Web Toolkit calls so that your application functions as if it is running in the Java Cartridge.

Once you have completed the non-Java Web Toolkit portions of your application, enter the Java Web Toolkit calls and compile the application. To compile the application using the JDK or Java IDE with Java Web Toolkit, you need to include the Toolkit's library in your environment. Make sure that the following paths are included in the *CLASSPATH* and *LD_LIBRARY_PATH* environment variables:

- The Toolkit class library **\$ORAWEB_HOME/java/oracle.zip** in the class path *CLASSPATH* setting
- The supplementary native libraries **\$ORAWEB_HOME/lib** in the native library path *LD_LIBRARY_PATH* setting

After compilation, your Java Application will be compiled into Java class files with a **.class** extension.

Adding and Running the Application into the Web Application Server

Once your application is compiled, deploy it by copying your Java class file(s) to the physical path of the virtual path under which you want to invoke your Java application. The physical paths are added to the *CLASSPATH* automatically. For example, to invoke your Java application under the virtual path **/java**, copy your classes to **\$ORAWEB_HOME/java** (UNIX) or **%ORAWEB_HOME%\java** (NT), which is the physical path of **/java**.

Make sure that your Web Request Broker and Web Listener have been started. Then, invoke your application by entering the URL of your application in your Web browser. For example: **http://<host>.<domain>:<port>/java/<Java class>**. The **<Java class>** should contain the `main` entry method. You may use virtual paths other than `/java`.

You should see the HTML results generated by your Java Web Application in your browser.

Once a Java Web Application is invoked, the classes that implement the application are cached by the Java Cartridge. This improves the performance of the application because the Java Cartridge does not have to reload the classes in subsequent invocations. If you modify your Java Application, you need to restart the Java Cartridge to reload the classes. Reloading the Java Cartridge does not cause the cartridge to reload the classes.

Tutorial

The Web Application Server does not have built-in debugging facilities, other than using print statements to generate messages on the screen or to a log file. For this reason, Oracle recommends that you build and debug as much of your application outside of the Web Application Server as possible using the JDK APIs, then finish the application using the Java Cartridge APIs inside the Web Application Server. This way, you can use the full debugging of the JDK environment for the standard API parts

of your program. Then move on to adding the Web Application Server portions when all of this is working properly.

Creating your First Java Application

The following is a simple Java application that prints “Hello World” to the standard output. Start by building the application outside of the Web Application Server environment using standard JDK APIs.

Use the editor in the IDE that comes with your Java compiler to enter the Java program shown below.

```
class HelloWorld {
    public static void main (String args[]) {
        System.out.println("Hello World");
    }
}
```

There are two points of interest:

- The program is enclosed by a class definition—`HelloWorld`.
- The body of the program is contained in a method called `main()`. Every Java application requires a `main()` method which is the first method run when the application executes.

Once you have finished entering the source code, save the file. Typically, Java programs are saved using the name of the class they define with the `.java` extension. For our example, save the file as **HelloWorld.java**.

Now compile the source file using the Java compiler.

- In Sun’s JDK, the Java compiler is called **javac**. To compile a Java program, make sure the `javac` program is in your execution path and type:

```
javac HelloWorld.java
```

- In Windows, the Java compiler is part of the IDE. Select the Compile option from the menus.

The program creates a Java bytecode file called **HelloWorld.class** in the same directory as the source file. You can now run the bytecode file using the Java interpreter that comes with your JDK.

- For Sun, the interpreter is **java**. Make sure that `java` is in your path and enter:

```
java HelloWorld
```
- For Windows, select appropriate Run command from the IDE menu.

The string “Hello World” prints to your screen.

Creating an Oracle Java Application

To make this example runnable by the Java Cartridge, add the following functionality to the basic HelloWorld program to produce an HTML page whose title and content are both “Hello World”:

```
import oracle.html.*;
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);
        // Add a simple string "Hello World" in this page
        bd.addItem("Hello World");
        // Print out the content of this page
        hp.printHeader();
        hp.print();
    }
}
```

Notice the following additions:

- The “import oracle.html.*;” statement enables the application to access the Java Cartridge classes. (For C/C++ programmers, this is analogous to #include files).
- The main() method uses the HTML classes to set up an HTML header, body, and page.

As you continue to add functionality to your Java application, you should only test those portions that pertain to the Java Cartridge within the Web Application Server. Test standard JDK methods outside of the Web Application Server to take advantage of the JDK debugger. You will need to comment out those portions of the application that are specific to the Java Cartridge.

Developer’s Guide

- Accessing HTTP Request Information
- Generating HTTP Response Information
- HtmlStream vs. System.out Stream
- Accessing WRB Services
- Accessing a Database
- Security
- Just-In-Time Compiler
- Dynamic HTML Generation
- Class Hierarchy
- HTML Features: A Summary
- Examples

- Extending the oracle.html Package
- Using Dynamic Content
- Extending the Java Cartridge

Accessing HTTP Request Information

To perform various functions, such as checking proper access to a Web application, a Web application may access HTTP request information. Such information is provided by the HTTP class in the `oracle.owas.wrb.services.http` package. It provides methods to access:

- HTTP request headers
- CGI environment values
- URL parameters
- Client's Accept-Charset and Accept-Language

Note: These are part of HTTP 1.1 request headers.

To access HTTP request information, retrieve the current request object with the `getRequest` static method:

```
// Obtains the current request object
HTTP request = HTTP.getRequest();
```

When a client makes an HTTP request, it can specify various attributes in the HTTP request header. To retrieve HTTP request headers, use the `getHeader` method.

CGI is an standard interface that enables an external program to respond to an HTTP request in a Web Application Server. In a Java Cartridge, CGI environment values can be retrieved using the `getCGIEnvironment` method on an HTTP object.

URL parameters are the query-string parameters in URLs in the Form's input encoding format. To access URL parameters, use the `getURLParameter` method.

With HTTP 1.1, a client can specify the national language and character set it can accept in an HTTP request. To determine Accept-Language and Accept-Charset of an HTTP request, use the `getAcceptLanguage` or `getAcceptCharset` methods. A client may list more than one Accept-Language or Accept-Charset, for each it may indicate its preference. Use the `getPreferredLanguage` or `getPreferredAcceptCharset` methods to determine the preferred language and character set.

The following code examples illustrate these methods:

```
// Retrieves User-Agent HTTP header information
String userAgent = request.getHeader("User-Agent");

// Retrieves CGI environment values
String remoteHost = request.getCGIEnvironment("REMOTE_HOST");

// Retrieves URL parameters
String department = request.getURLParameter("DEPT");

// Retrieves the most preferred Accept-Language of the request.
// If no language is specified, use English.
String language = request.getPreferredAcceptLanguage("en");
```

Generating HTTP Response Information

When a Java application responds to an HTTP request, the response should be preceded by HTTP response headers. The headers describe the properties of the response, such as the MIME type, or the character set of the HTML page. Use the `print` method in the `HtmlStream` object to send the headers, which will be returned to the client through `HtmlStream`.

The following sample source code illustrates how to generate HTTP response headers:

```
// Generates the MIME type and charset header of the HTTP response
HtmlStream responseStream = HtmlStream.theStream();
responseStream.println("Content-type: text/html; charset=us-
ascii");

// Ends the response header section with an empty line
responseStream.println();

// Returns the HTML page in the response body
HtmlPage hp = new HtmlPage();
...
hp.print();
```

Notice the generation of an empty line that demarcates the header and the body sections. Also notice that the `HtmlPage` class contains the method `printHeader()` that you can invoke to generate the “content-type: text/html” header.

HtmlStream vs. System.out Stream

In the Java Cartridge, output written to `System.out` stream will be returned to the client. However, the `print` and `println` methods of `System.out` stream are not thread-safe. When two threads write to the output stream at the same time, the output may be garbled. Oracle provides the `HtmlStream` class which synchronizes the output when it is written by multiple threads. You should switch to `HtmlStream` to return output to the client, although `System.out` stream is still supported for backward compatibility.

Accessing WRB Services

Since the Java Cartridge is executed in the WRB environment, Java applications executed by it can access WRB services. The services are provided in native libraries, and Java applications may utilize them in their native methods. For encapsulation purposes, it is a good idea to write Java class wrappers around those services.

To utilize WRB services you need a handle to the WRB context, which can be obtained from the `getWRBContext` static method of the `WRB` class (in the `oracle.owas.wrb` package). It returns the context handle as a Java long (a 64-bit number). Pass the handle to your native method and cast it to a `void*` in C.

The following sample code illustrates how to retrieve WRB context in Java and invoke WRB services in C.

```
// Retrieve WRB context
long wrbContext = WRB.getWRBContext();

// Invokes WRB API to log a message
```

```
logMessage(wrbContext, "This is a test log message");
```

In this example, WRB service is invoked in the native code and is accessed through the Java method with the following prototype:

```
void logMessage(long wrbContext, String message);
```

In the native implementation of the method, the WRB context handle is cast to a void*:

```
void MyClass_logMessage(struct HMyClass *this, int64_t wrbContext,
Hjava_lang_String *message)
{
    void *WRBContext = (void*)wrbContext;
    WRB_LOGwriteMessage(WRBContext, ...);
}
```

The exact prototype of the native C function may be different on some platforms.

Note that `getWRBContext` throws the `WRBNotRunningException` when the method is not invoked in the WRB environment.

As the WRB service is invoked natively, your application may use custom shared libraries which need to be added to the Java Cartridge.

For Logger Service, the Java Web Toolkit provides a Java API to access the service. For other services, you may consider writing your own wrapper classes to encapsulate the services.

Transaction Service

WRB provides the X/Open Transaction Service that enables the execution of a series of operations in a transaction environment where operations can commit or rollback as an atomic operation. During the course of operations when it is necessary to abort a transaction, the Transaction Service guarantees that all operations that have been carried out will be reverted. A common application of the Transaction Service is the manipulation of multiple databases, where such operations should be carried out in a “do-all-or-nothing” manner.

Note that the WRB Transaction Service is not enabled in Java-PL/SQL’s database access. To use the Transaction Service with an Oracle database, you must use it with your own database access mechanism.

Intercartridge Exchange Service

The WRB InterCartridge Exchange Service (ICX) allows an application in one cartridge to invoke an application in a different cartridge and retrieve output from it. With this powerful communication service, you can produce sophisticated Web applications with output from different cartridges, where each one is specialized in handling a certain type of request or producing a certain type of results. For example, you can write a Java application to search for the users logging on to a system, then invoke a PL/SQL Agent to generate a report of the users’ information from the database and include it in the HTML results.

The Web Application Server provides the following cartridges:

- **PL/SQL Cartridge** - for database programming with PL/SQL
- **Java Cartridge** - for general-purpose programming

- **LiveHTML Cartridge** - for HTML with embedded commands
- **Oracle Worlds Cartridge** - for 3-dimensional modeling (VRML)
- **Perl Cartridge** - for system programming
- **ODBC Cartridge** - for database manipulation and query with heterogeneous databases

Content Service

The WRB Content Service provides a framework for a document repository where documents can be stored, retrieved, and shared easily by the cartridges that can publish them on the Web. It also allows cartridge developers to develop tools to manage and administer the repository from the Web. In addition, attributes can be attached to documents and queries can be performed on these attributes.

Logger Service

The WRB provides Logger Service for cartridges to write error, warning or other useful messages to a central log repository (a file system or a database). Web listener and WRB messages are also logged in the same repository. The Web Application Server suite includes a Log Analyzer tool that can be used to analyze the log messages and generate reports for auditing, performance-tuning, or other purposes.

See the Logger APIs section for a description of the severity levels.

Sessions

In the WRB, multiple instances of a cartridge can execute at the same time. When the WRB receives a request from a browser, it randomly picks an instance of the cartridge which is available to handle the request. If the cartridge maintains an application state on behalf of the request and needs any future request from the browser to be dispatched to the same cartridge instance, you can utilize the WRB Session to tie the browser to a particular instance of a cartridge. This effectively maintains a session between a browser and a cartridge instance. To enable the Session for the Java Cartridge, use the Web Application Server.

Accessing a Database

There are a number of ways to access a database from the Java Cartridge.

- For PL/SQL users, the cartridge provides a Java-PL/SQL toolkit that provides tight integration between Java and PL/SQL. Database users can implement their business logic in PL/SQL to capture their business activity and to ensure proper use of the data in their databases.

With a code-generation tool, the Java-PL/SQL toolkit generates Java wrapper classes for PL/SQL packages to invoke PL/SQL stored procedures as easily as invoking methods in Java objects. This powerful tool allows database programmers to leverage their existing investment in PL/SQL with Java development.

- If your application consists primarily of direct SQL data manipulation, the JDBC package may suit your needs better. It provides a standard interface to access database from different vendors.

Java-PL/SQL is the mechanism provided by the Java Web Toolkit that allows you to make PL/SQL calls from within your Java application.

Code Generation with pl2java utility

To provide such tight integration between Java and PL/SQL, the Java Web Toolkit includes a code generation utility, called **pl2java**, that generates Java wrapper classes for PL/SQL stored procedures. A wrapper class is a Java class containing methods to call a PL/SQL package's procedures and functions, and serves as an interface between the two languages. Stand-alone PL/SQL procedures and functions are all wrapped in a single wrapper class.

pl2java is written in Java and can be found under the **\$ORAWEB_HOME/java/bin** (UNIX) or **%ORAWEB_HOME%\java\bin** (NT) directory. To use **pl2java**, you must have the Java Development Kit (JDK) installed and the Java interpreter executable must be in your execution path. Also, **pl2java** requires a PL/SQL stored package installed in the database where your PL/SQL packages are loaded. This package must be installed by the database user SYS. The installation script is **dbpkins.sql** under **\$ORAWEB_HOME/java/sql** (UNIX) or **%ORAWEB_HOME%\java\sql** (NT) directory. (The de-installation script is **dbpkdins.sql** under the same directory.) Check with your database administrator to make sure that the package has been installed.

Note: If you are connecting to Oracle8, use **dbpkins8.sql** instead of **dbpkins.sql**. If you use the wrong installation script, you will get the following error message:

```
Warning: Package Body created with compilation errors
```

To generate the Java wrapper class for your PL/SQL package, invoke **pl2java** from the command prompt:

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

pl2java creates a wrapper class for each PL/SQL package given as an argument to the command. When your application is run, an instance of this class to interface to the package is created. If you have stand-alone procedures or functions in your applications, run **pl2java**, with the **class** flag as explained below. This creates a single wrapper class for all the stand-alone procedures and functions you use.

The following table contains the arguments of **pl2java**:

Argument	Description
username	The name of the Oracle database user that owns the PL/SQL packages
password	The password for the Oracle user identified by username
connect-string	The string that identifies the database where the packages are located. This is the SQL*Net Connect String, as described in Understanding SQL*Net. For local databases, omit this connect-string. Instead, set the ORACLE_SID environment variable to specify the local databases, as described in the Oracle7 Server Administrator's Guide.

Argument	Description
package name...	A list of all the PL/SQL packages that your application references in the schema identified by username. To wrap stand-alone procedures and functions 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.

All of the flags that **pl2java** uses are optional except under certain conditions, class. The following table contains the descriptions of the flags.

Flag	Description
-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 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 the default, but it only applies to the first package named in the command. If the wrappers are being created for stand-alone 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.

PL/SQL Data Type Mapping in Java

One of the tasks in enabling PL/SQL calls in Java is determining the PL/SQL data type mapping in Java, in particular how PL/SQL data types are to be represented in Java. Unlike Java basic data types, a PL/SQL data type can be null, which means that it does not have a value assigned to it. Therefore, a Java data type cannot represent a PL/SQL data type without losing the null property. And it is natural to use Java wrapper classes to represent PL/SQL data types. These wrapper classes, all belonging to the `oracle.plsql` package, are derived from the base `PValue` base class, which encapsulates the null attribute of PL/SQL values. Each individually derived wrapper class represents one or more related PL/SQL data type. The following table shows a list of Java wrapper classes of different PL/SQL data types:

PL/SQL data type	Java wrapper class
BINARY_INTEGER (+ NATURAL, POSITIVE)	PInteger

PL/SQL data type	Java wrapper class
NUMBER (+ DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NUMERIC, REAL, SMALLINT)	PDouble
CHAR(n) (+ CHARACTER, STRING)	PStringBuffer
VARCHAR2(n) (+ VARCHAR)	PStringBuffer
LONG	PStringBuffer
RAW (n)	PByteArray
LONG RAW	PByteArray
BOOLEAN	PBoolean
DATE	PDate
PL/SQL table	Java array (not a class)

When you pass a value to a PL/SQL call, you create a wrapper object for that PL/SQL data type and store the value in it. When the call returns with an output parameter, you retrieve the value from the wrapper object.

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

Disallowed PL/SQL data type	Substitute PL/SQL data type
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

PL/SQL Procedure Mapping in Java Wrapper Classes

The Java wrapper classes generated by `pl2java` provide a convenient way to invoke PL/SQL stored procedures in Java. For each PL/SQL package you specify, `pl2java` generates a Java wrapper class, which contains a wrapper method for each procedure or function in the PL/SQL package. The prototype of the wrapper method will be the same as the PL/SQL procedure or function it wraps. This provides an “object” view of PL/SQL packages and allows PL/SQL stored procedures be called seamlessly.

For example, consider the following Employee PL/SQL package that contains a function and a procedure:

```
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;
```

Run **pl2java** to generate the wrapper class:

```
pl2java scott/tiger@db Employee
```

The wrapper class `Employee` will be generated which has an interface like this:

```
class Employee {
    public Employee(Session dbSession) { ... }
    public PDouble count_employees(PStringBuffer dept_name) { ... }
    public void list_employees(
        PStringBuffer dept_name,
        PStringBuffer employee_name[],
        PDouble employee_no[]
    ) { ... }
}
```

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 data member of the wrapper class 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. For information about overloading of functions in PL/SQL, see “Overloading” in “Using the PL/SQL Cartridge”. You can find 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:

```
package Employee as
    function employee_name (
        employee_number    in    number
    ) return varchar2;
end;
```

The wrapper class `Employee` contains the data member `employee_name_0_return_length` which can be overridden:

```
class Employee {
```

```

...
public PStringBuffer employee_name(PDouble employee_number);
public int employee_name_0_return_length = 255;
}

```

Similarly, if the function returns a PL/SQL table (PL/SQL array), you can specify the length of the array with the data member of the wrapper class:

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

Making Connection to the Database

Database connection is encapsulated by the `Session` class in the `oracle.rdbms` package. The `Session` class provides methods to perform common database tasks, such as logon, logoff, commit, rollback and so on. Before you connect to an Oracle database, you need to define environment properties, such as the `ORACLE_HOME` environment variable. Retrieve `ORACLE_HOME` of Web Application Server in the Java Cartridge's system properties "oracleHome" with the `System.getProperty` method. After defining Oracle environment properties, instantiate a `Session` object and logon to the database. The following sample code illustrates how this is done:

```

// Define ORACLE_HOME
Session.setProperty("ORACLE_HOME",
System.getProperty("oracleHome"));

// Creates a new database session and logon
Session session = new Session();
session.logon("scott", "tiger", "sales_db");

```

Invoking PL/SQL Stored Procedures and Passing PL/SQL Values with PValue Classes

To invoke a PL/SQL stored procedure, you need to instantiate the wrapper class for the PL/SQL package (or for anonymous PL/SQL procedures) with a `Session` object. This prepares the object for the execution of the PL/SQL package in the session. If you want to invoke the PL/SQL package in multiple database sessions, you need to instantiate the wrapper class for each one of them. For example, assume the `Employee` package. You would instantiate the `Employee` class like this:

```

// Instantiate Employee wrapper class:
Employee emp = new Employee(session);

```

When you invoke a PL/SQL procedure that takes parameters, you need to pass the values to the parameters by storing them in PL/SQL data type wrapper objects. Instantiate these wrapper objects and set the values with `setValue` method and pass these objects to the wrapper method as parameters. To pass a PL/SQL null value, use `setNull` method on the wrapper object to set the value to null.

When a PL/SQL function returns, it may return some values in its out parameters or return value. To retrieve the return values, use the get-value methods of the wrapper classes. For example, use the `intValue` method of the `PInteger` class to retrieve an int value. Note that when a PL/SQL return value is null, the get-value method throws a `NullPointerException`. This `NullPointerException` signifies a null value, and you should handle this exception properly with a try-catch block. Alternatively, you can first use the `isNull` method to determine if the value is null, and invoke the get-value method when it is not null.

The following sample code illustrates how to invoke a PL/SQL procedure and pass parameters in and out:

```

// Instantiate a PStringBuffer to pass a string to the PL/SQL
procedure
PStringBuffer pDeptName = new PStringBuffer(30);
pDeptName.setValue("Sales");

// Invokes the PL/SQL procedure
PDouble pEmployeeCount = employee.count_employees(pDeptName);

// Retrieves the return value
if (!pEmployeeCount.isNull())
    int employeeCount = pEmployeeCount.intValue();

```

Note that when a PL/SQL parameter is a PL/SQL table, either an in or an out parameter, you need to create the Java array as well as the elements in the array. For example:

```

//Creates a PL/SQL table parameter
PStringBuffer pEmployeeNames[] = new PStringBuffer[30];
for(int i = 0; i < pEmployeeNames.length; i++)
    pEmployeeNames[i] = new PStringBuffer(80);

// Invokes a PL/SQL procedure
employee.list_employees(pEmployeeNames);

```

Note that all wrapper classes that encapsulate PL/SQL values have a `toString` method and therefore can be concatenated with Java Strings. For example, you can use the `pEmployeeCount` object from above directly in a string concatenation:

```

// Output the employee count
System.out.print("There are " + pEmployeeCount + " employees.");

```

Handling Database Errors with `ServerException`

Java-PL/SQL provides tight integration between Java and PL/SQL not only in the way PL/SQL procedures are invoked, but also in the way database exception is handled. When an error occurs during a database operation, an exception is thrown. The exception is returned to Java and is thrown as a `ServerException`. For example, when a no-data-found exception occurs in a SQL select statement, a `ServerException` is thrown. You can use the `getSqlcode` and `getSqlerrm` methods to retrieve the SQL code and error message of the exception.

Most methods of the `Session` class as well as the wrapper methods in PL/SQL wrapper classes throws `ServerException`. You should catch the exception by putting the calls in a try-catch block.

Freeing Database Sessions

Java provides a garbage collector which frees up objects when they are no longer needed. When a database session is no longer needed and becomes garbage, the session will be disconnected before it is garbage-collected. However, the garbage collector does not guarantee that any garbage objects will be collected immediately. In fact, Java's garbage collector waits until the program is idle, which for a busy Web site could be infrequently, or until resources are low, before it collects garbage objects. Therefore, you should try to logoff from the database when the session is no longer needed to free up database resources explicitly.

For documentation on JDBC, see [\\$ORACLE_HOME/jdbcoci73/doc/contents.htm](#).

Maintaining Persistent State

The Java Cartridge offers a number of benefits over running Java applications with Sun's Java VM as a CGI program. Besides the performance improvement through the reduction of process start-up cost, the Java Cartridge allows Java applications to retain their application state. Values stored in static data member variables are preserved across multiple HTTP requests. The following example illustrates how to write a Java application that counts the number of requests a Java application has handled since it was started:

```
import oracle.html.*;
class HelloWorld
{
    // The value stored in static data member is preserved across
    // multiple requests.
    private static int count = 0;

    public static void main (String args[])
    {
        HtmlStream out = HtmlStream.theStream();

        out.println("Content-type: text/html\n");

        count++;
        out.println("This application has been accessed " + count +
            " times since it was started.");
    }
}
```

By preserving values in static data members, you may be able to improve the performance of your Java application in some situations. For example, you can open a database connection and maintain the connection open in a static data member. By doing so, you can avoid opening the database connection every time a request is handled, which is a time-consuming operation. The following example shows how this can be done:

```
import oracle.html.*;
import oracle.rdbms.*;
import oracle.plsql.*;

class EmployeeReport
{
    // The database connection can be maintained open across multiple
    // requests by storing it in a static data member.
    private static Session session = null;

    public static void main (String args[])
    {
        HtmlStream out = HtmlStream.theStream();
        out.println("Content-type: text/html\n");

        // If the database session has never been opened, open it
```

```

// (once and for all). Otherwise, we just use the connection
// we opened in the previous request.
if (session == null)
    session = new Session("scott", "tiger", "db");

// Performance database operation
...
...

// *DO NOT* logoff from the section by the end of your application.
// We will reuse the connection in the next request.
}
}

```

Database sessions that are left opened by the time the Java Cartridge shuts down will be logged off by the cartridge. Notice that each Java Cartridge instance maintains these data members within itself. They are not shared among the cartridge instances. In the first example, each Java Cartridge instance has its own `count` data member. In the second example, each Java Cartridge instance maintains one open database connection.

Java Runtime Configuration Flags

In addition to the usual configuration attributes, the Java Cartridge supports the runtime flags of Sun's Java Interpreter via similarly named attributes in the [JAVA] section of Web Request Broker configuration. The following table lists the names and valid values of these flags:

Config flag in [JAVA]	Sun's Java Interpreter flag	Legal value
VERBOSE	-verbose	true or false
NOASYNCGC	-noasyncgc	true or false
VERBOSEGC	-verbosegc	true or false
TRACEGC	-tracegc	true or false
CHECKSOURCE	-checksource	true or false
C_STACK	-ss	A number followed by M, m, K, or k, where M or m indicates megabytes, and K or k indicates kilobytes.
JAVA_STACK	-oss	A number followed by M, m, K, or k, where M or m indicates megabytes, and K or k indicates kilobytes.
INITIAL_HEAP	-ms	A number followed by M, m, K, or k, where M or m indicates megabytes, and K or k indicates kilobytes.

Config flag in [JAVA]	Sun's Java Interpreter flag	Legal value
MAX_HEAP	-mx	A number followed by M, m, K, or k, where M or m indicates megabytes, and K or k indicates kilobytes.
PROF	-prof	true or false
VERIFY	-verify	true or false
VERIFYREMOTE	-verifyremote	true or false
NOVERIFY	-noverify	true or false
SYSTEM_PROPERTY	-D=	<i>prop=value</i>

Note the following:

- Sun's "-debug" flag is not supported.
- Multiple "SYSTEM_PROPERTY" flags can be used.
- "true/false" values are case-sensitive.

For example, if you want to turn off the asynchronous garbage collector, set maximum heap size to 32 Mbytes and define system properties `user.default` and `password.default`, add these flags to the Web Request Broker configuration:

```
[JAVA]
CLASSPATH           = ...
LD_LIBRARY_PATH    = ...
NOASYNCGC          = true
MAX_HEAP           = 32M
SYSTEM_PROPERTY    = user.default=scott
SYSTEM_PROPERTY    = password.default=tiger
```

Security

Java is designed to be a secure execution environment where programs can be executed with a set of security restriction boundaries. When a Java program is executed, every operation is monitored to ensure that the program is behaving properly. If the Java runtime environment discovers that a program is trying to do an operation that can cause a security breach, it stops the program.

Java Applets and Applications

Java programs are divided into two categories: applets and applications. A Java applet is a program that is downloaded from a Web Application Server by a browser and is executed on the client side. Since the applet can be downloaded from an unknown web site across an insecure network, the applet cannot be trusted. To ensure that the applet will not cause a security breach, Java executes the applet while monitoring it with the Security Manager. Whenever the applet tries to perform an operation which can compromise security, such as reading a file in the local machine, it checks that the applet has the appropriate privilege to perform such an operation. If the applet does not have such a privilege, a security exception is thrown and the operation is aborted.

A Java application is a trusted program that usually resides on the local machine. When executed, it is granted access to all privileged operations without the monitor of the Security Manager. It is the responsibility of the user who executes a Java application to ensure that the Java program is trustworthy.

Programs executed by the cartridge are Java applications. The Security Manager is not invoked to ensure proper behavior by the applications. This allows Java applications maximum freedom to execute. However, without security restriction, poorly designed Java applications may open security holes. Therefore, users should pay particular attention when they implement and deploy applications with the Java Cartridge.

Java Application Protection by Partitioning Virtual Paths

As mentioned above, when the Java Cartridge executes Java applications, it does not impose security restriction on them. Therefore, you are strongly advised to consider implementing access control to Java applications. You can utilize the WRB authorization mechanism to protect the Java Cartridge from unauthorized access.

To protect the Java Cartridge with the WRB, simply protect all virtual paths which map to the Java Cartridge. That is, add all virtual paths to the Application Protection section of WRB configuration, and apply appropriate authorization or protection schemes to them.

However, not all Java applications are intended to be protected. If you have Java applications which are open to public access, you should create another Java Cartridge to run those public applications. You can add one more entry for the Java Cartridge in the Application section of WRB configuration.

```
JAVA          ... $ORAWEB_HOME/lib/libjava.so (UNIX)
JAVA          ... %ORAWEB_HOME%\lib\libjava.so (NT)
SECURED_JAVA ... $ORAWEB_HOME/lib/libjava.so (UNIX)
SECURED_JAVA ... %ORAWEB_HOME%\lib\libjava.so (NT)
```

And in the Virtual Path section, add virtual paths for the secured Java Cartridge:

```
/java          JAVA (UNIX)
\java          JAVA (NT)
/secured/java  SECURED_JAVA (UNIX)
\secured\java  SECURED_JAVA (NT)
```

Then, protect all the virtual paths that map to SECURED_JAVA in the Application Protection section of WRB configuration:

```
/secured/java  Basic(Admin Server) (UNIX)
\secured\java  Basic(Admin Server) (NT)
```

And finally, duplicate the configuration of the Java Cartridge named JAVA for the one named SECURED_JAVA. However, these two Java Cartridges must have separate class directories in their CLASSPATH setting. Put the public Java application classes under JAVA's class directory and put the protected ones under the SECURED_JAVA's class directory. For example, Java Cartridge's CLASSPATH may be:

```
CLASSPATH = $ORAWEB_HOME/java/classes.zip;$ORAWEB_HOME/java/oracle.zip;$ORAWEB_HOME/java (UNIX)

CLASSPATH =
%ORAWEB_HOME%\java\classes.zip;%ORAWEB_HOME%\java\oracle.zip;%ORAWEB_HOME%\java (NT)
```

while that for SECURED_JAVA must contain a different class directory:

```
CLASSPATH = $ORAWEB_HOME/java/classes.zip;$ORAWEB_HOME/java/oracle.zip;$ORAWEB_HOME/secured_java (UNIX)
```

```
CLASSPATH = %ORAWEB_HOME%\java/classes.zip;%ORAWEB_HOME%\java\oracle.zip;%ORAWEB_HOME%\secured_java (NT)
```

Note that if the two cartridges share class directories, it is possible to bypass the security check by invoking the protected Java application through a non-protected virtual path.

Similarly, if you want to have two or more different protection schemes for your Java applications, you may create as many different Java Cartridges and protect their corresponding virtual paths with different protection schemes.

NLS Support

The Java Cartridge provides the following NLS features:

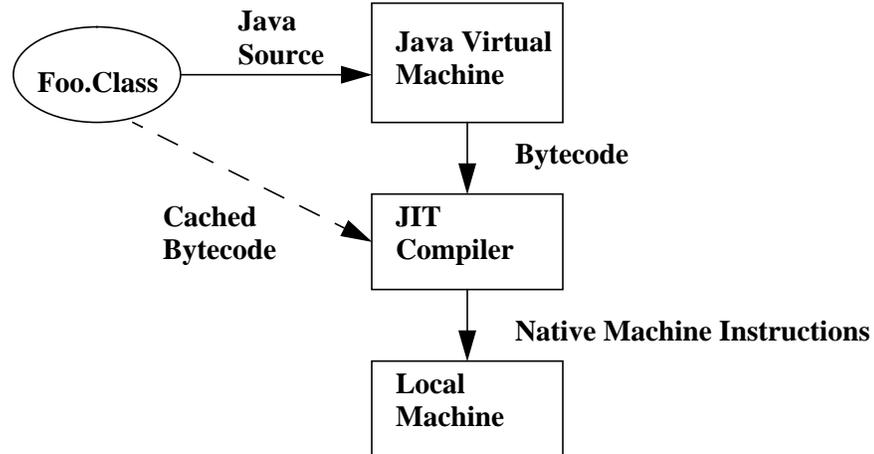
- The **pl2java** command supports databases which use non-ASCII character sets. Characters in different languages can now be exchanged safely with the database and converted to Unicode, the character set used internally by Java.
- A character set conversion library is provided in Oracle's Java Web Toolkit. In the package `oracle.owas.nls`, the class `CharacterSet` is provided to convert messages from one character set to another (or to Unicode). Such character sets can be retrieved from the `CharacterSetManager` class.
- In the `HTTP` class, a set of methods is provided to access language and the character set preference of the browser. This preference is passed by using the new HTTP 1.1 headers, which are supported by the latest browsers like Netscape Navigator 3.0 and Microsoft Internet Explorer 3.0. You can use the `getAcceptLanguage` and `getAcceptCharset` methods to retrieve the language and character set acceptable to the client.
- Oracle's NLS environment is controlled by the environment variable `NLS_LANG`. In the package `oracle.owas.nls`, the class `NLS_LANG` is provided to utilize the language and character set preference of the client and define the `NLS_LANG` environment to control the NLS behavior of the database.

For details on developing a Java Application for NLS environments, please check the `NLSSample` Java example which illustrates how NLS can be handled in the Java Web Application's interaction with the client browser and the database. This sample is available on the Oracle Web Application Server's home page.

Just-In-Time Compiler

To provide portability on many hardware architecture platforms, Java uses an interpreter that compiles Java programs into bytecodes that are instructions of a virtual machine (the Java Virtual Machine). Each platform implements the virtual machine by software emulation, which interprets the virtual machine instructions when a Java program is executed. Therefore, the bytecodes compiled from a Java program do not depend on any particular platform and they can be executed on any machine.

Just-in-Time Compiler



A Java program is usually executed more slowly than the same program compiled to native machine instructions. This is the performance penalty paid to achieve portability.

With new compilation technology, it is possible to take a series of virtual machine instruction codes and recompile them into native machine instructions of the hardware platform. By doing so, the same set of codes can be executed much faster. The cost of this speed-up is the time spent to recompile the instructions.

Just-In-Time (JIT) compilation is the lazy recompilation of the virtual machine instructions into native machine instructions at the moment when the Java program is about to be executed. Recompilation needs to occur once, and the recompiled instructions can be executed every time. Many JIT compilers recompile virtual machine instructions only once after the corresponding Java method is invoked for the first time, and cache the recompiled instructions until they exit. The Java interpreter defines a standard interface so that any JIT compiler can be plugged into it.

The JIT compiler is usually provided as a dynamic-linked library and you should put the library in one of the `LD_LIBRARY_PATH` directories. To enable the JIT compiler, add a `JAVA_COMPILER` environment variable which should contain the name of the library. For example, the library of SunSoft's JIT compiler is `libsunwjit.so` and you should have `JAVA_COMPILER` as:

```
LD_LIBRARY_PATH = ...
...
JAVA_COMPILER = sunwjit
```

(Note that only the Sun JIT is supported.)

Performance

JIT compilation should provide the single most significant performance improvement. A few-fold speedup is common. However, it may cause a degradation in performance in some situations when it takes longer to recompile byte codes than to interpret the

byte codes directly. Therefore, you should compare the performance of your application executed with and without JIT compiler to determine the best option.

In addition to JIT compilers, consider re-implementing some heavily used methods in native code, which execute more efficiently than interpreted instructions.

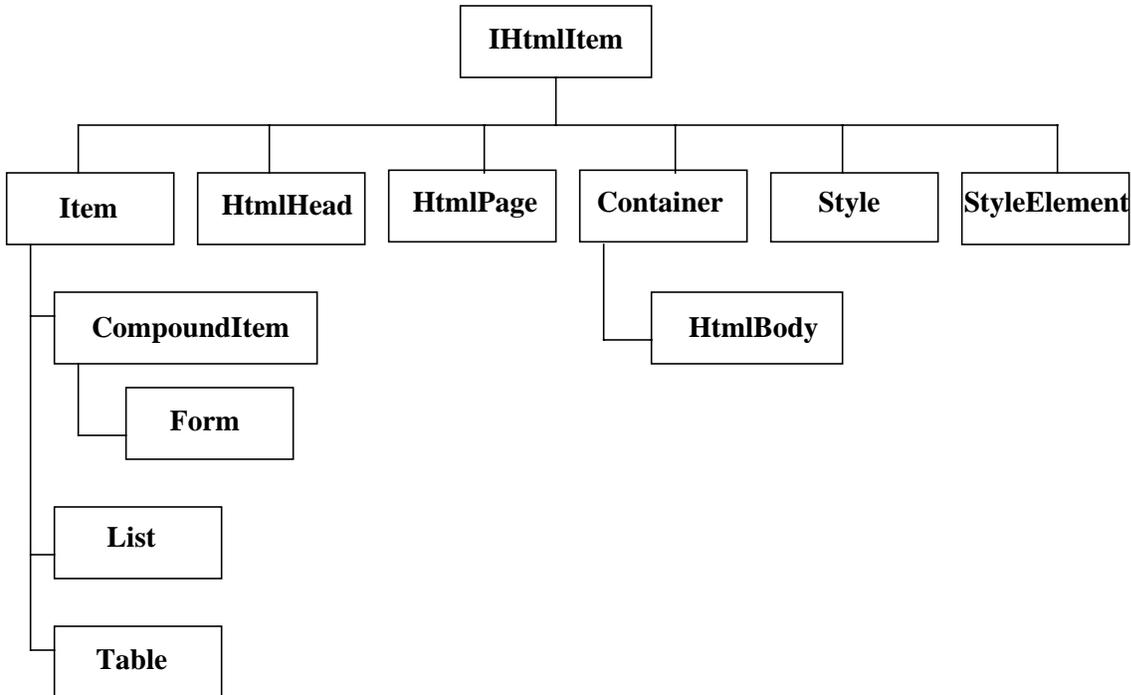
Dynamic HTML Generation

The **oracle.html** package allows you to generate HTML pages *dynamically*. It hides the details of HTML so that you can focus on the actual layout and format of your HTML pages. There are numerous benefits provided by this package:

- **Ease of use** - This package enables Java programmers to dynamically compose HTML 3.2-compliant pages with CSS1 (Cascading Style Sheet, Level 1) support without having to learn either HTML or CSS1. Every markup tag has been encapsulated within Java classes in an object-oriented manner. To the Java programmer, an HTML page is simply an object that contains various kinds of markup objects. A markup component/object can be simple (a piece of text) or complex (a dynamically sized table with pictures/links within each table cell).
- **Extensibility** - Due to the rapid pace of evolution in the HTML space there are bound to be new markup elements that are not supported by even the latest release of this package. To accommodate this, our package can be extended. Java programmers can write their own Java components that subclass from one of the many classes in this package to provide for the new markup tags, and these new components will be treated just like any built-in components/objects via *polymorphism*. Furthermore, any Java class that implements the interface **IHtmlItem** will be treated as a first-class member of the HTML object family.
- **Efficiency** - The package deals with overheads and memory consumption in a sensible manner.
- **Intelligence** - One of the major headaches faced by the HTML content programmer/designer today is to deal with the vast number of browsers available and the various incompatibilities these browsers introduce in their handling of HTML. Our package is designed to deal with these issues in an intelligent manner. Part of the package is a mini-database that stores useful information about various existing browsers. Our components are designed to take advantage of this information so that the generated HTML is optimized for the browser that is requesting this page *now*. Since this capability is built-in, this means one less detail to which the Java programmers need to attend.
- **Up-to-date support for W3C markup standards** - Oracle is committed to providing up-to-date and comprehensive support for standard HTML and CSS1 in the industry.

Class Hierarchy

The following diagram illustrates the hierarchy of the major classes in the package.



The most important piece in this package is the interface **IHtmlItem**. This interface defines the basic requirements and behaviors for any HTML component. This enables polymorphic treatment of any HTML components inside this package. This interface defines methods for converting content of any component into a String object (**toString**) and printing/writing content of the object as HTML to any arbitrary Output stream object (**print**). Any custom object should implement this interface so that it can benefit the most from this package.

To make creating custom HTML components easy, Oracle provides a default implementation of the interface in class **IHtmlItemImpl**. This class also serves as the base class of virtually all other classes in this package.

Another basic building block in the `oracle.html` package is the **Item** class. This *abstract* class provides useful markup capabilities for any component that can itself be regarded as a markup component (a **SimpleItem** object — an object that deals with Strings — is also an **Item**, which means that it immediately possesses capabilities usually associated with a markup component such as color and font size). This class serves as a base class only and cannot be used to instantiate objects.

Structurally, the two major container classes — **CompoundItem** and **Container** — provide the basic infrastructure for any complex HTML component that may relate to other HTML components *via containment*. Class **CompoundItem** derives from class **Item**, which means an object of this type also inherits the many useful markup capabilities provided by class **Item**. Many classes in this package use (**List**) or inherit from (**Form**) this class to take advantage of this. In contrast, the **Container** class is a light-weight version of **CompoundItem**, except that it does not derive from class **Item**.

Thus objects of this type have less overhead, but do not possess markup capabilities. Notable classes that derive from **Container** class include **HtmlBody** and **ImageMap**.

In order to construct a HTML page dynamically, you need to (in most cases) instantiate objects from the following classes: **HtmlHead**, **HtmlBody**, and **HtmlPage**. To add Style Sheet support, you may want to use classes **Style** and **StyleElement**. Oracle provides hooks into various components (e.g. classes **Item** and **HtmlHead**) to make it easy for you to implement CSS1.

There are many classes in this package that make creating various HTML objects as simple as possible. Those who are not familiar with HTML should use the following classes: **List**, **Form**, and **Table** (or **DynamicTable**).

In some cases, interfaces are used as a grouping mechanism for some attributes of existing HTML tags. The purpose of these interfaces is to make it easier for Java programmers to specify values of various attributes. Examples of such interfaces are **IVAlign** and **ITableRules**.

HTML Features: A Summary

This section presents a quick summary of major HTML features supported in this package:

The following HTML elements can be generated by the corresponding Java classes:

HTML Element	Corresponding Java Class	Description
<HTML>	HtmlPage	Page section
<HEAD>	HtmlHead	Head section
<BODY>	HtmlBody	Body section
<A>	Anchor - for link target; Link - for hypertext link	Anchor
<APPLET>	Applet	Client-side Java applet
<! ... >	Comment	Comment
<DL>	DefinitionList	Definition list
<DIR>	DirectoryList	Directory list
<TABLE>	DynamicTable	Dynamic table
	Font	Font
<FORM>	Form	Form
<FRAME>, <FRAMESET>	Frame; Frameset	Frames
<H[1-6]>	Heading	Heading level
<HR>	HorizontalRule	Horizontal rule
	Image	Image
 	LineBreak	Line break
<INPUT>	FormElement	Form input element

HTML Element	Corresponding Java Class	Description
<PRE>	PreFormat	Preformat
<STYLE>	Style	Style

Examples

Example: “Hello World”

```
// Import dynamic HTML package into current namespace
import oracle.html.*;

// Class HelloWorld
class HelloWorld {
    // Main Method: entry point for any server-side Java App
    public static void main(String args[]) {
        // Creates an HtmlPage object
        HtmlPage hp = new HtmlPage("Hello World");

        // Adds a string object ("Hello World!") to the page
        hp.getBody().addItem("Hello World!");

        // Prints header info to output stream
        hp.printHeader();
        // Prints out content of this page
        hp.print();
    }
}
```

When this program is compiled and executed on the command-line (outside of the WRB environment), the following output is generated (assume “>” is an interactive shell prompt):

```
>javac HelloWorld.java // compile class defined as above
>java HelloWorld // invoke method main in class HelloWorld
Content-type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<!-- Generated by Oracle's Dynamic HTML Generation Package -->
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
</HEAD>
<BODY>
Hello World!</BODY>
</HTML>
```

In HTTP parlance, the first two lines are known as the HTTP Response headers of the document that is to be returned to the user-agent (usually a browser).

Following that are two lines (enclosed by “<!” and “>”). The first line declares the DOCTYPE as compliant with HTML 3.2 (as specified and recommended by W3C). The second line is a comment stating this document is generated using Oracle Corp’s Dynamic HTML generation package.

The rest is standard HTML; the HTML tags “<HTML>” and “</HTML>” denote the beginning and ending of this HTML document. Each HTML document is composed of

HEAD and BODY sections (denoted by tags “<HEAD>...</HEAD>” and “<BODY>...</BODY>” respectively). The content of this document is the simple string: “Hello World!” and this will be the only text that appears on the browser.

As you might have noticed, when this package is used outside of the WRB environment, all output is redirected to standard output. This doesn't mean, however, that Oracle uses the same OutputStream as System.out — Oracle actually provides its own stream for redirecting output back to the user-agent when this class is executed inside WRB environment.

Example: Simple Markup

It is essential for a package that encapsulates a markup language to support adding markup properties to text and other objects. As mentioned earlier, most markup support is defined in the *abstract* class **Item**. It defines methods like `setItalic()` (set the Italic/I markup attribute of the target object) and `setEmphasis()` (set the Emphasis/EM markup attribute of the target object) which enable the Java programmer to deal with each HTML component in an object-oriented manner: an object with markup attributes that can be individually set and cleared. What follows is a simple example that uses various methods from class **Item** to accomplish the desired result:

```
import oracle.html.*;
//
// This simple demo demonstrates various markup capabilities
// of this package
//
class SimpleMarkup {

    // the entry point of this program
    public static void main(String args[])
    {
        // Creates an HtmlPage object
        HtmlPage hp = new HtmlPage();

        // Gets default HtmlBody object from HtmlPage object
        HtmlBody bd = hp.getBody();

        // Adds a banner-like heading to this page
        bd.addItem(new SimpleItem("Welcome! Java
programmers!").setHeading(1));
        // Adds a "LineBreak" to indicate end-of-line to the browser
        bd.addItem(SimpleItem.LineBreak);

        // Prints the text string "Hello World!" in different Heading
formats
        for (int i=2; i<5; i++) {
            bd.addItem(new SimpleItem("Hello World!").setHeading(i));
            bd.addItem(SimpleItem.LineBreak);
        }

        // Prints the text string "Java is cool!" in BOLD-faced font
        bd.addItem(new SimpleItem("Java is cool!").setBold());
        bd.addItem(SimpleItem.LineBreak);

        // Print out header information
    }
}
```

```

        hp.printHeader();
        // Print out content of this page
        hp.print();
    }
}

```

When the above program is compiled and executed outside of the WRB environment the following output is generated:

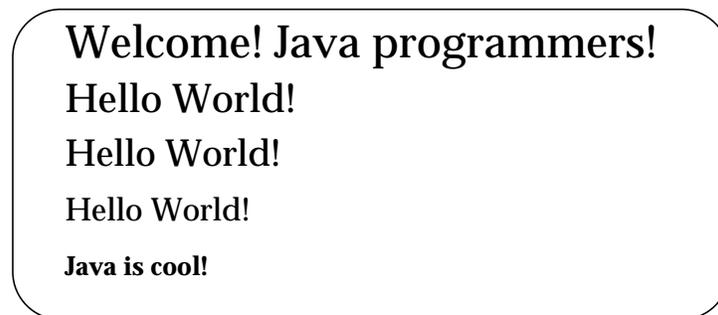
```

Content-type:  text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Draft//EN">
<!-- Generated by Oracle's Dynamic HTML Generation Package -->
<HTML>
<HEAD>
</HEAD>
<BODY>
<H1>Welcome! Java programmers!</H1><BR>
<H2>Hello World!</H2><BR>
<H3>Hello World!</H3><BR>
<H4>Hello World!</H4><BR>
<B>Java is cool!</B><BR>
</BODY>
</HTML>

```

This is what you will see:



Example: HTML Form

Once you have begun using Java to write Web-based server-side applications, you will soon find the need to generate results based on the input of users (an example would be to present a catalog of sale items to the user according to the user's choice of categories). There are many ways to accomplish this, but by far the most common is by using an *HTML Form* that has the capability to present the user with several types of input elements such as text fields, checkboxes and radio buttons, and when a "submit" element is invoked, the browser sends the user input to a URL specified by the Form itself.

What follows is a code fragment (not a complete program):

```

// Creates a Form object
Form form = new Form("GET", "http://www.myhome.com/wrb/doit");
// Creates a TextField object and adds to the form
form.addItem(new TextField("Name"));
// Creates a Submit object and adds to form

```

```

form.addItem(new Submit("foo", "Submit!"));
// Adds the form object to the HtmlBody object
body.addItem(form);

```

What this code fragment does is to generate the appropriate HTML so that a text field will be presented to the user, and when the user presses the “Submit!” button, the content of the text field will be sent to the specified URL. For more information, please refer to the W3C HTML 3.2 reference specification at

<http://www.w3.org/pub/WWW/MarkUp/Wilbur>

Example: Lists

In HTML, there are two principle ways to present information to the user in a tabular manner: List and Table. The following code fragment illustrates how to construct an ordered list of text items:

```

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

```

Example: Table

The following code fragment illustrates how to create a table using the DynamicTable class:

```

// 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]);
}

```

Extending the `oracle.html` Package

You can easily create your own HTML components by deriving them from the `CompoundItem` or `Container` classes. You can create high-level HTML classes that define particular layout styles and use them as templates. For example, you can create a `CompanyBanner` class that has a company logo, a hyperlink to its home page, and another hyperlink to its copyright notice. Each time when you want to include a company banner in any page, you create a `CompanyBanner` object, and fill in the company logo GIF file and the two URLs for the hyperlinks. Here is a sample `CompanyBanner` class:

```

class CompanyBanner extends CompoundItem {
    // Constructor (takes a logoGIF and 2 links as arguments)
    public CompanyBanner (String logoGIF, String homepageLink,
        String copyrightLink) {
        addItem(new Link(homepageLink,
            new Image(logoGIF, "Company Logo", IAlign.TOP, true)));
        addItem(new Link(copyrightLink, "Copyright Notice"));
    }
}

```

Then, you can simply add a `CompanyBanner` to your source code to generate HTML:

```

// Adds a company banner
bd.addItem(new CompanyBanner("img/oracle.gif",
    "http://www.oracle.com",
    "http://www.oracle.com/copyright.html"));

```

You can create HTML classes that encompass computation logic. For example, you can create a `BalanceSheet` class that performs a query of a customer's purchase information from a database, and formats the results in HTML. To create a balance sheet for a customer, simply instantiate a `BalanceSheet` object and specify the customer's identifier. Then add the `BalanceSheet` item to your `HtmlPage`.

Using Dynamic Content

To many WWW content designer (including some Java programmers), the thought of *programmatically* generating HTML seems absurd: there already are great tools tailored to generate HTML in a WYSIWYG ("what-you-see-is-what-you-get") environment — why add an extra level of complexity to the process by forcing them to use *yet another* language to generate HTML? To others, the concern is more pragmatic in nature: I already have lots of nicely designed HTML pages throughout my web site — how do I add dynamic content to my web pages without re-creating what is already there?

To address these concerns, Oracle provides a class in this package that enables the Web content designers to use their favorite tools to create HTML pages with the *right* look-and-feel *and then add dynamic content to these pages*.

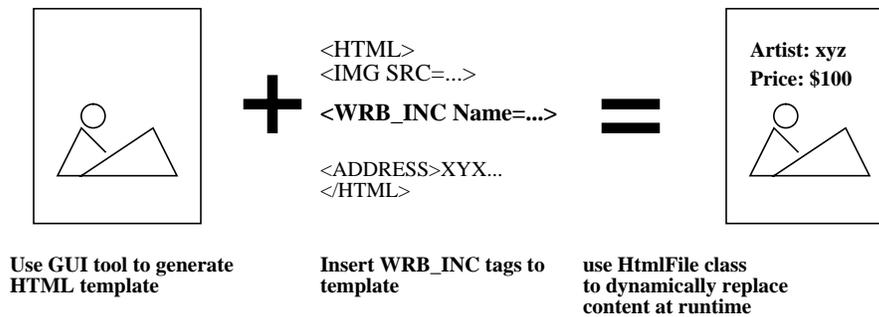
The process is easy:

1. Use your favorite tool to create the HTML with the right look and feel.
2. In places where you would like to have dynamic content added to the page, insert the following tag into the tool-generated HTML:

```
<WRB_INC NAME="dynItem1" VALUE="defaultValue">
```
3. Import the HTML template (i.e. the tool-generated HTML plus tags added in step 2) into Java by using class **HtmlFile**.
4. Dynamically insert content into HTML template (or "legacy HTML pages") by using `HtmlFile`'s **setItemAt** method.

The process allows you to have the flexibility to use your favorite Web design tool to generate the *static* part of the content while benefiting from the *dynamic* nature of server-side Java applications.

Here is a simple diagram that illustrates the process:



Extending the Java Cartridge

One strength of Java is the ability to extend its functionality by adding classes and packages to it. You can implement sophisticated, Java-based Web applications by purchasing Java packages from third-parties and by writing classes yourself. Well-designed Java packages allow you to snap them into positions easily and minimize your programming effort.

Adding Classes and Packages

To add classes, either your own or third-party, put them in the class directory of your Java Cartridge. A class directory is one of the directories in the *CLASSPATH* setting of your Java Cartridge, set in its configuration section. *CLASSPATH* lists the directories that the Java Cartridge searches for Java classes to invoke. The classes include those that are invoked by the URL and those that are used by other classes. Since the physical paths of the Java Cartridge are automatically added to the *CLASSPATH*, you may also put the classes under the physical path directories.

If you have several classes, you may consider packing them up as a ZIP file. The ZIP file must *not* be compressed. Add the full path of your ZIP file to *CLASSPATH*. For example, if you pack your classes in a file called **myclasses.zip** under **/usr/local/oracle/java/classes** directory, your *CLASSPATH* should contain:

```
CLASSPATH = ...:/usr/local/oracle/java/classes/myclasses.zip:...
```

Packing classes as a ZIP file gives you slightly better performance because the Java Cartridge can load all classes together from the file system. In fact, Oracle's Java packages are bundled in the ZIP file **oracle.zip**. Be aware that including files that are not needed can slow down class invocation.

Note that if your classes belong to packages, you must create the sub-directories that correspond to those packages. For example, if your class belongs to **a.b.c** package, the class file must be under sub-directory **a/b/c** under the class directory.

Adding Classes with Native Libraries

To add classes or packages with native libraries, the native libraries should be put in one of the directories listed in the *LD_LIBRARY_PATH* configuration setting of the Java Cartridge in its configuration section. For example, if you put your native libraries

under the **/usr/local/oracle/java/lib** directory, your `LD_LIBRARY_PATH` should contain:

```
LD_LIBRARY_PATH = ...:/usr/local/oracle/java/lib:...
```

Your classes should be added as shown in the previous section.

Troubleshooting and Debugging

- Web Request Broker
- The Java Cartridge
- Java Web Applications
- No Java Debugger
- Shutting Down the Java VM
- Logging Information
- Exception Handling

Web Request Broker

When your Java Web application cannot be invoked properly, first make sure that the Web Listener and the WRB are started and functioning properly. Also, make sure that the Java Cartridge is registered with the WRB and the virtual path for your application maps to the Java Cartridge.

The Java Cartridge

The Java Cartridge requires two minimum configuration settings, *CLASSPATH* and *LD_LIBRARY_PATH*.

CLASSPATH must contain:

- **%ORAWEB_HOME%/java/classes.zip** - the JavaSoft JDK class library
- **%ORAWEB_HOME%/java/oracle.zip** - the Java Web Toolkit
- **%ORAWEB_HOME%/java** - the directory for storing your Java class files. This is the default directory, you can change this directory if you like.

LD_LIBRARY_PATH must contain:

- **%ORAWEB_HOME%/java/lib** - the JDK native libraries
- **%ORAWEB_HOME%/lib** - the Java Web Toolkit native libraries

If the Java Cartridge is configured, you should be able to invoke the HelloWorld sample by the URL `http://<host>.<domain>:<port>/sample/Java/run/HelloWorld`.

Make sure that your Java application's classes are stored under one of the directories in *CLASSPATH* or the physical paths of the Java Cartridge. If your classes use native libraries, make sure that they are stored under one of the directories in *LD_LIBRARY_PATH*.

Java Web Applications

Make sure that the `main` entry-point method has the correct prototype: `public, static, no return type`; and that `main` takes an array of strings as its only argument.

When an error occurs in your Java application, an exception is thrown. If you do not catch the exception and handle it by surrounding your statements with a try-catch block, the Java Cartridge will report the uncaught exception. If your Java classes are compiled with the debug flag, the cartridge will also report the source where the exception originates. In some situations, an error may be thrown instead, and an error cannot be caught by a try-catch block.

There are many common exceptions and errors, such as:

- **NullPointerException** - a method or a data member is accessed with a null object reference. Check your program logic and fix the error.
- **ClassNotFoundException** - access to a class that cannot be found. Make sure that the class is under one of the directories in `CLASSPATH` or the physical paths of the Java Cartridge.
- **NoSuchFieldError, NoSuchMethodError** - access to a field or a method that cannot be found in the class. This may be caused by a change in a class that other classes use and they are not updated. Update all those classes that use it.
- **UnsatisfiedLinkError** - fail to load a native library. This is usually caused by a missing native library, or an unresolved symbol in the library. Make sure that the native library is in one of the directories in `LD_LIBRARY_PATH` and all referenced symbols are defined.

No Java Debugger

Since the Java Cartridge does not include a debug environment to develop your Java Web Application, you should use your JDK or Java IDE to write and debug your application in the same way as you do for your other Java applications.

When you debug your applications in these tools, some of the functionalities in the Java Web Toolkit are disabled because your Web Application is not executed in the WRB environment. Therefore, you should write and debug as much of your application which does not use the Java Web Toolkit with JDK or Java IDE. In some situations, you may need to insert dummy code in your application to substitute your Java Web Toolkit calls so that your application will function as if it is running in the Java Cartridge.

Once you complete the non-Java Web Toolkit portions of your application, put in your Java Web Toolkit calls and compile your application. To compile your application in your JDK or Java IDE with Java Web Toolkit, you need to include the Toolkit's library in your environment. Specifically, you need to include the Toolkit's class library, namely the library ZIP file `$_ORAWEB_HOME/java/oracle.zip`, to the `CLASSPATH` setting, and the directory which contains the supplementary native libraries, namely `$_ORAWEB_HOME/lib`, to your native library path

Shutting Down the Java VM

Once a Java Web Application is invoked, the classes that implement the application are cached by the Java Cartridge. This is done to speed up the performance of the application so that Java Cartridge does not have to reload the classes in subsequent

invocations. If you modify your Java Application, you need to restart the Java Cartridge to reload the classes. Furthermore, reloading the Java Cartridge does not cause the cartridge to reload the classes.

Logging Information

The WRB provides the Logger Service for cartridges to write error, warning, or other useful messages to a central log repository (a file system or a database). Web listener and WRB messages will also be logged in the same repository. The Web Application Server suite includes a Log Analyzer tool that can be used to analyze the log messages and generate reports for auditing, performance-tuning, or other purposes.

The Java Web Toolkit provides an `OutputLogStream` class in the `oracle.owas.wrb.services.logger` package for your Java applications to write log messages with the Logger Service. To write a log message, instantiate an `OutputLogStream` object, specify the component name and the log destination, and use the `print` or `println` methods to write a message.

You may include the severity level of the message. The Logger Service defines the semantics of each severity level. It is a good idea to follow its semantics when you write your log messages so that the Log Analyzer can produce consistent analysis of the log. See the WRB Logger APIs section for a description of the severity levels.

Here are examples that illustrate how to log messages with `OutputLogStream`:

```
// Instantiates an OutputLogStream
OutputLogStream log = new OutputLogStream("my component");

// Log an error message of severity 1
log.println(1, "An error occurred");
```

Exception Handling

When a Java class encounters an error, the Java Cartridge throws an exception. Catch the exception by surrounding your statements with a try-catch block and print the error message. When you are building and debugging the application, you may want to print the error message to the system's standard error stream. For example:

```
try{
    ....
} catch(Exception e) {
    System.err.println("Exception: " + e.getMessage());
}
```

After you have debugged the application, you may want to change the destination of the error message to go to the client. The above code then becomes:

```
HTMLStream out = HTMLStream.theStream();
try{
    ....
} catch(Exception e) {
    out.println("An error occurred in this class: " + e.getMessage());
}
```

Examples

This section contains an expanded “HelloWorld” example. The example illustrates:

- Oracle’s HTML-generation package
- Access to HTTP request information
- Database access through PL/SQL stored procedures

The example defines the following methods:

- `main()` - the main entry method
- `initialize()` - initializes the environment before serving a request
- `saveClientInfo()` - stores client information into a database
- `retrieveClientInfo()` - retrieves client information from a database
- `getHTTPInfo()` - retrieves User-Agent and client’s host name information from an HTTP request
- `printHTML()` - constructs and returns results in HTML back to the client
- `generateBrowserReport()` - formats the database-query results in HTML table format, illustrates the use of `DynamicTable` to generate an HTML table

`main()`

This is the main entry function to HelloWorldX:

```
import oracle.html.*;
import oracle.rdbms.*;
import oracle.plsql.*;
import oracle.owas.wrb.services.http.*;
import java.util.*;

class HelloWorldX {

    public static void main(String args[]) {

        /* Initializes environment */
        initialize();

        /* Retrieve User-Agent and client's host name information from the
         * HTTP request. */
        getHTTPInfo();

        /* Stores client information into a database */
        saveClientInfo();

        /* Retrieves client information from a database */
        retrieveClientInfo();

        /* Constructs and returns results in HTML back to the client */
        printHTML();

    }
}
```

```
initialize()
```

This method initializes the environment before serving a request.

```
private static void initialize()
{
    /* Creates a new HTML item to store any error message. */
    errorMesg = new CompoundItem();
}
```

```
saveClientInfo()
```

This method stores client information into a database. It illustrates the invocation of PL/SQL stored procedures in Java to store data into a database.

```
private static void saveClientInfo()
{
    /* Defines Oracle Session ORACLE_HOME property. ORACLE_HOME value
    * can be retrieved from Java's system properties if this class is
    * executed in the Java Cartridge.
    */
    Session.setProperty("ORACLE_HOME",
System.getProperty("oracleHome"));

    try {

        /* Creates a new database session and passes username, password
        * and connect-string to logon.
        */
        dbSession = new Session("rpang", "rpang", "wdk7322");

        /* Prepares PL/SQL Browser_Stat package for access. BrowserStat
        * is a Java class-wrapper for the PL/SQL package "browser_stat".
        */
        browserStat = new BrowserStat(dbSession);

        /* Creates parameters to invoke a PL/SQL stored procedure. For
        * each type of PL/SQL data type, there is a corresponding Java
class-
        * wrapper. To invoke a PL/SQL stored procedure, we need to
construct
        * the parameters which we will pass to a PL/SQL stored procedure.
        */
        PStringBuffer pBrowser = new PStringBuffer(userAgent);
        PStringBuffer pVisitHost = new PStringBuffer(remoteHost);
        PDate pVisitDate = new PDate(new Date());

        /* Invokes "browser_stat.update_browser" PL/SQL stored procedure.
        * This is done by invoking the corresponding method in the
        * corresponding Java class-wrapper.
        */
        browserStat.update_browser(pBrowser, pVisitHost, pVisitDate);

        /* Commit changes to browser statistics in the database */
        dbSession.commit();
    }
}
```

```

    } catch (ServerException e) {

        /* All database operation may throw ServerException when a
        * database exception occurs. This exception should be caught and
        * properly handled.
        */
        errorMsg.addItem(new SimpleItem("Database error while storing
        browser information to database: " + e.getMessage()));
    }
}

```

retrieveClientInfo()

This method retrieves client information from a database. It illustrates the invocation of PL/SQL stored procedures in Java to access data in a database.

```

private static void retrieveClientInfo()
{
    try {

        /* Invokes "browser_stat.count_browsers" PL/SQL stored procedure.
        * This is done by invoking the corresponding method in the
        * corresponding Java class-wrapper.
        */
        PDouble pCount = browserStat.count_browsers();

        /* Retrieves value from a PL/SQL value. Notice that all value-
        * retrieval methods of PL/SQL data-type wrappers will throw
        * NullPointerException if the PL/SQL value is null.
        */
        int count = pCount.intValue();

        browsers    = new String[count];
        visitCounts = new int[count];
        visitHosts  = new String[count];
        visitDates  = new Date[count];

        if (count > 0)
        {
            /* Creates parameters to invoke PL/SQL stored procedure */
            PStringBuffer pBrowsers[] = new PStringBuffer[count];
            PDouble       pVisitCounts[] = new PDouble[count];
            PStringBuffer pVisitHosts[] = new PStringBuffer[count];
            PDate         pVisitDates[] = new PDate[count];
            PDouble       pMax          = new PDouble(count);

            for(int i = 0; i < pBrowsers.length; i++)
            {
                /* For array data-types, we need to create each array element. */
                pBrowsers[i] = new PStringBuffer(80);
                pVisitCounts[i] = new PDouble();
                pVisitHosts[i] = new PStringBuffer(80);
                pVisitDates[i] = new PDate();
            }
        }
    }
}

```

```

/* Invokes "browser_stat.get_browser_stat" PL/SQL stored procedure.
 * This is done by invoking the corresponding method in the
 * corresponding Java class-wrapper.
 */
pCount = browserStat.get_browser_stat(pBrowsers,
    pVisitCounts,
    pVisitHosts,
    pVisitDates,
    pMax);

count = pCount.intValue();

for(int i = 0; i < count; i++)
{
    /* Retrieves the PL/SQL values and store them in Java variables. */
    browsers[i] = pBrowsers[i].stringValue();
    visitCounts[i] = pVisitCounts[i].intValue();
    visitHosts[i] = pVisitHosts[i].stringValue();
    visitDates[i] = pVisitDates[i].dateValue();
}

} catch (ServerException e) {

    /* All database operation may throw ServerException when a
    * database exception occurs. This exception should be caught and
    * properly handled.
    */
    errorMsg.addItem(new SimpleItem("Database error while retrieving
    browser statistics from database: " + e.getMessage()));
} finally {

    /* No matter we have error or not, we should disconnect the database
    * session.
    */
    try {
    dbSession.logoff();
    } catch (ServerException e) {
    /* We will ignore any database error when we logoff from database. */
    }
}
}

```

getHTTPInfo()

This method retrieves User-Agent and client's host name information from the HTTP request. It illustrates how to retrieve information from an HTTP request.

```

public static void getHTTPInfo()
{
    /* Retrieves the HTTP request. getRequest returns the current
    request. */
    HTTP request = HTTP.getRequest();

    /* Retrieves User-Agent information from the current request. */
    userAgent = request.getHeader("User-Agent");
}

```

```

    /* Retrieves client's host name from the current request. */
    remoteHost = request.getCGIEnvironment("REMOTE_HOST");
}

```

printHTML()

This method constructs and returns results in HTML back to the client.

```

private static void printHTML()
{
    /* Constructs the HTML page. An HTML page consists of a head and
    * a body.
    */
    HtmlHead hd = new HtmlHead();
    HtmlBody bd = new HtmlBody();
    HtmlPage pg = new HtmlPage(hd, bd);

    /* Puts results in the page body. The basic building-block of an
    * HTML page is an HTML item (class Item), which can be added to
    * an HTML page body. SimpleItem is a basic Item.
    */
    bd.addItem(new SimpleItem("Hello, user at "))
    .addItem(new SimpleItem(remoteHost).setItal()) // Sets IP addr
    italic
    .addItem(new SimpleItem(":"))
    .addItem(SimpleItem.Paragraph)
    .addItem(SimpleItem.Paragraph);

    /* Group together a set of HTML items so that they can be treated as
    * a single item.
    */
    CompoundItem infoBody = new CompoundItem();
    infoBody.addItem(new SimpleItem("I found that you are using "))
    .addItem(new SimpleItem(userAgent).setBold()) // Sets userAgent
    bold
    .addItem(new SimpleItem(" browser. How do you like it?"))
    .addItem(SimpleItem.Paragraph)
    .addItem(new SimpleItem("I am conducting a poll to decide which " +
    "browser is the most popular, and your " +
    "browser information has been saved.))
    .addItem(SimpleItem.Paragraph);

    /* Use HTML definition list to indent paragraphs. */
    DefinitionList clientInfo = new DefinitionList();
    SimpleItem emptyCell = new SimpleItem();
    clientInfo.addDef(emptyCell, infoBody)
    .addDef(emptyCell, generateBrowserReport());

    bd.addItem(clientInfo);

    /* Include error message in the bottom if there is any. */
    if (errorMesg.size() > 0)
    {
        bd.addItem(SimpleItem.Paragraph)
    .addItem(new SimpleItem("Error: "))
    .addItem(errorMesg);
    }
}

```

```

    }

    /* Returns the HTTP header of the HTML page. */
    pg.printHeader();

    /* Returns the HTML page to the client. */
    pg.print();
}

```

generateBrowserReport()

This method formats the database-query results in HTML table format. It illustrates the use of DynamicTable to generate an HTML table. The result is returned as an HTML Item.

```

private static Item generateBrowserReport()
{
    CompoundItem report = new CompoundItem();

    report.addItem(new SimpleItem("Here is the result of the poll as of
" +
new Date() + ":")
        .addItem(SimpleItem.Paragraph);

    /* Constructs a HTML table. An HTML table is represented as a
    * DynamicTable.
    */
    DynamicTable tab = new DynamicTable(2);

    /* Added the table headers. */
    TableRow row = new TableRow();
    row.addCell(new TableHeaderCell("Browsers"))
        .addCell(new TableHeaderCell("Visit counts"))
        .addCell(new TableHeaderCell("Last visit hosts"))
        .addCell(new TableHeaderCell("Last visit time"));
    tab.addRow(row);

    /* Added each browser as a row to the HTML table */
    for (int i = 0; i < browsers.length; i++) {
        row = new TableRow();
        row.addCell(new TableDataCell(browsers[i]))
            .addCell(new TableDataCell(Integer.toString(visitCounts[i])))
            .addCell(new TableDataCell(visitHosts[i]))
            .addCell(new TableDataCell(visitDates[i].toString()));
        tab.addRow(row);
    }

    report.addItem(tab);

    return report;
}

/* User agent (browser) information */
private static String userAgent;

/* Client's host name */

```

```

private static String remoteHost;

/* Known browsers */
private static String browsers[];

/* Times they visited */
private static int visitCounts[];

/* Last visit host names */
private static String visitHosts[];

/* Last visit time */
private static Date visitDates[];

/* Database session */
private static Session dbSession;

/* PL/SQL Browser_Stat package wrapper class */
private static BrowserStat browserStat;

/* HTML item to store error messages */
private static CompoundItem errorMesg;
}

```

PL/SQL Tables

The following PL/SQL table and package are used by the HelloWorldX Java example.

```

CREATE TABLE browser_stat_table
(
  browser_name          VARCHAR2(80) NOT NULL,
  browser_visitcount    NUMBER(10)   NOT NULL,
  browser_visithost     VARCHAR2(80) NOT NULL,
  browser_visittimeDATE NOT NULL,

  CONSTRAINT unq_browser_name UNIQUE(browser_name)
);

CREATE OR REPLACE PACKAGE browser_stat AS

  type string_table is table of varchar2(80) index by binary_integer;
  type number_table is table of number(10) index by binary_integer;
  type date_table   is table of date index by binary_integer;

  procedure update_browser (
    v_browser_name      in   varchar2,
    v_browser_visithost in   varchar2,
    v_browser_visittime in   date
  );

  function count_browsers return number;

  function get_browser_stat (
    v_browser_name      out   string_table,
    v_browser_visitcount out   number_table,

```

```
v_browser_visithost out string_table,  
v_browser_visittime out date_table,  
v_max in number  
) return number;
```

```
END;
```

3

Using the LiveHTML Cartridge

This chapter describes how to use the LiveHTML Cartridge, Oracle Web Application Server's implementation of Server Side Includes. The following topics are covered:

- Overview
- LiveHTML File Structure
- Limiting Use of LiveHTML ("Crippled" Includes)
- Database Access via LiveHTML
- LiveHTML and Intercartridge Exchange Service
- LiveHTML Commands
 - config
 - include
 - echo
 - fsize
 - flastmod
 - exec
 - request
- LiveHTML Examples

Overview

The LiveHTML Cartridge is Oracle's implementation and extension of the standard Server Side Includes (SSI) functionality defined by the National Center for Supercomputing Applications (NCSA). The cartridge 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 command 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 be CGI.
- A system variable, for example, FMODDATE.
- A URL.

Before returning a LiveHTML request, the LiveHTML Cartridge processes the above commands and inserts the results in the returned HTML data.

LiveHTML File Structure

A Web page that uses LiveHTML must be parsed by the Web Application Server. For this reason, it differs slightly from ordinary Web pages written in HTML, which the Web Application Server simply delivers to the browser. To have the LiveHTML commands executed on the server, you use the Web Application Server 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.

Configure the Web Application Server to direct requests to the LiveHTML Cartridge by virtual paths or by extension. Any extension is valid including html.

For performance reasons, do not parse large binary files through the LiveHTML Cartridge. That is, specify a virtual path to large binary files at the Listener level which is different from the virtual paths specified at the LiveHTML Cartridge level. The virtual paths can target the same physical paths.

The Web Application Server administrator normally creates file extensions for this type, and these are what you use in your application code. The default file extension is SHTML. The administrator can also specify HTML as a file extension, 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.

Limiting Use of LiveHTML (“Crippled” Includes)

Enabling users to execute scripts on the server can create security problems and other risks. For this reason, you can specify, as part of the server configuration, the Cartridge allows only “crippled” includes. This means that LiveHTML can retrieve static HTML environment variables only, or other server parsable files, not executable scripts.

Database Access via LiveHTML

You can use LiveHTML to run the PL/SQL Agent via InterCartridge Exchange (ICX), and thereby incorporate dynamic Oracle data in hardcoded Web pages. You can also use ODBC.

LiveHTML and Intercartridge Exchange Service

The Intercartridge Exchange Service (ICX) feature of the Web Application Server allows cartridges to communicate with each other by making HTTP requests. Cartridges can communicate to retrieve documents from another cartridge or to

perform some computations on another cartridges. For more information about ICX, see the Introduction to the Web Request Broker.

From your LiveHTML document, you can issue ICX requests. For example, you can invoke the PL/SQL Cartridge from the LiveHTML document, and the results from the PL/SQL Cartridge would be included in the LiveHTML document.

LiveHTML Commands

Command Format

LiveHTML commands are formatted as HTML comments, so that they are not seen by the user if the server fails to execute the commands for any reason. LiveHTML commands have the following format:

```
<!--#command [arg1="value1" ...] -->
```

command is the name of the LiveHTML command, *arg1* is the name of the parameter to pass to the command, and *value1* is the value for the parameter.

Note: Only one LiveHTML command can be present per line. For example:

```
http://<!--#echo var="SERVER_NAME" -->  
<!--#echo var="DOCUMENT_URI" -->
```

must be broken into two separate lines.

Also, for NT, each line in your LiveHTML files cannot exceed 268 characters.

LiveHTML Commands

The following table summarizes the LiveHTML commands:

Command	Description
config	This command sets parameters for how the included files or scripts are to be parsed. It is normally the first LiveHTML command in a file.
include	This command specifies that a file is to be included in the generated HTML page at this point.
echo	This command gives the value of an environment variable.
fsize	This command produces the size of the file.
flastmod	This command produces the last modification date of the file.
exec	This command executes a script.
request	This command allows you to include the results of another HTTP request including, but not limited by, calling another cartridge via ICX.

config

This command sets parameters for how a file or script is to be parsed. It is normally the first LiveHTML command in a file. The possible arguments are:

Argument	Description
<code>errmsg</code>	Specifies the error message that is sent to the client if an error occurs while parsing the document.
<code>timefmt</code>	Specifies the format to use when displaying dates. The conventions follow the <code>strftime</code> library call. Ordinary characters in the format are copied to the document without conversion, so you can insert “on” or “at” or other useful strings.
<code>sizefmt</code>	Specifies the format to use when displaying file size. Possible values are: <ul style="list-style-type: none">• <code>bytes</code> - the file size is given in bytes• <code>abbrev</code> - the file size is given in kilobytes or megabytes
<code>cmdecho</code>	Specifies whether non-CGI scripts subsequently executed have their output incorporated into this HTML page. The possible values are <code>ON</code> and <code>OFF</code> . <code>ON</code> specifies that the output is included. The default is <code>OFF</code> .
<code>cmdprefix</code>	Specifies a string to prepend to each line of the script output.
<code>cmdpostfix</code>	Specifies a string to append to each line of the script output.

For example:

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

include

This command specifies that a file is to be included in the generated HTML page at this point. The file can be any of the following:

- another LiveHTML file
- a regular HTML file
- an ASCII file

The Web Application Server determines the type of the included file by its extension.

The command can take the following arguments:

Argument	Description
<code>virtual</code>	This gives a virtual path to the file. The directory mappings for virtual paths are set by the Web Application Server administrator using Web Application Server Manager.
<code>file</code>	This gives a pathname relative to the current directory. References to parent directories or uses of absolute pathnames are forbidden.

For example, if your file contains the following `include` command:

```
<!--#include file="inc.html"-->
```

and the `inc.html` file contains:

```
<p><a href="#Date/TimeFormatting">Date/Time Formats</a> |  
<a href="#DocumentNames/Paths">Doc Names/Paths</a> |  
<a href="#FileSizeAndDate">File Size/Date</a> <p>
```

this would result in the following links being inserted into your Web page:

```
Date/Time Formats | Doc Names/Paths | FileSize/Date
```

echo

This command gives the value of a standard CGI environment variable or a Server Side Includes environment variable.

This command requires the `var` argument, which specifies the name of the variable. The Server Side Includes environment variables are:

Variable	Description
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 <code>\</code> .
DATE_LOCAL	The current date and local time zone, given in the format specified in the most recent config <code>timefmt</code> command.
DATE_GMT	The current date and time zone in Greenwich Mean Time, given in the format specified in the most recent config <code>timefmt</code> command.
LAST_MODIFIED	The last modification date of the file, given in the format specified in the most recent config <code>timefmt</code> command.

filesize

This command produces the size of the file in the format specified in the most recent config `filesize` command. Valid arguments are the same as for the `include` command.

flastmod

This command produces the last modification date of the file in the format specified in the most recent config `timefmt` command. Valid arguments are the same as for the `include` command.

exec

This command causes execution of a script. The argument specifies whether or not the script is CGI.

Argument	Description
cmd	<p>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.</p> <p>The non-CGI environment variables specified under echo above can be referenced. For the output of the script to be included in the HTML page, you have to set the following line in the page:</p> <pre><!--#config cmdecho="ON"--></pre>
cgi	<p>Specifies a CGI script. The value is the virtual path of the CGI script. URL locations are automatically converted into HTML anchors.</p>

Note: Before you can use the exec command, you need to enable it in the LiveHTML Cartridge configuration.

request

This command allows you to include resources specified in an extended URL.

```
<!--#request URL=<URL>>
```

For example:

```
<!--#request URL="http://whatever/hr/$user/  
work?SQLString='select$property from employee'&name=$user">
```

The command requires the URL argument, which identifies the extended URL format. This format relieves you from encoding the URL and allows you to use variable substitution.

You embed a request command in an HTML document. This document itself can also take a query string. Only the form style query string is relevant and usually refers to name-value pairs. The names in the name-value pairs of the query string (consisting of alphanumeric characters and beginning with an alphabetic character) are defined as the ARGS of this document.

The syntax of the HTTP URL is the following:

```
http://user:password@host:port/url-path?QS
```

- *url-path* extends the semantics of the common URL because variables are subject to substitution. The representation of the variable substitution is \$ARG. The variable must be in between '/' or between '/' and the end of the <url-path> is recognized as a variable substitution. A variable must be one of the ARGS for this document.
- QS represents the query string.

- LiveHTML expects content within single quotes to be a non-encoded URL since Server Side Includes performs the encoding. However, you must correctly encode the rest of the HTTP URL.
 - LiveHTML ICX support contains some characters that have a special meaning for *PString*: '\$', "'", '\'. You need to escape these characters to preserve their literal meaning.
 - Note that the request URL might point to a complete HTML document. However, LiveHTML ICX support includes only the content inside the <BODY> and strips any other tags such as <HTML> or <HEAD>.

For an example of LiveHTML ICX support, see LiveHTML Examples.

LiveHTML Examples

This section provides examples of LiveHTML commands.

Displaying Date and Time

The following config command defines a date and time format, which is used by the echo command.

```
<!--#config timefmt="%A, %B %d, %Y, at %I:%M %p"-->
<p>GMT date/time is
<!--#echo var="DATE_GMT"-->
<p>LOCAL date/time is
<!--#echo var="DATE_LOCAL"-->
<p>Updated on
<!--#echo var="LAST_MODIFIED"-->
```

This generates the following:

```
GMT date/time is Friday, August 23, 1996, at 03:14 AM
LOCAL date/time is Thursday, August 22, 1996, at 08:14 PM
Updated on Tuesday, August 13, 1996, at 03:42 AM
```

Getting Information About the Current File

The following lines:

```
This document is <!--#echo var="PATH_TRANSLATED"-->
Its virtual path is <!--#echo var="DOCUMENT_URI"-->
```

generate

```
This document is /private1/oracle/ows21/sample/ssi/sstest.html
Its virtual path is /sample/ssi/sstest.html
```

Getting Information on Other Files

The `fsize` and `flastmod` commands allow you to get the file size and last modification date of any file on the server rather than just the current document.

For example, the following lines:

```
<!--#config sizefmt="bytes"-->
<p>This gives the file size of 'sstest.html' in bytes:
<!--#fsize file="sstest.html"-->
```

generate

```
This gives the file size of 'sstest.html' in bytes: 6405 bytes
```

The following lines:

```
<!--#config sizefmt="abbrev"-->
<p>This gives the file size of 'sstest.html' in bytes:
<!--#fsize file="sstest.html"-->
```

generate

```
This gives the file size of 'sstest.html' in kilobytes: 6 Kbytes
```

One of the best uses of `fsize` is to provide the user with the sizes of graphic files to be downloaded. This is a tremendous timesaver if you add and change downloadable images frequently, since you never have to look up the file sizes and enter them manually.

Getting Information on the Client's Browser

You can display to the user the browser and version that he or she is using to read your pages.

The line:

```
<p>You are using <!--#echo var="HTTP_USER_AGENT"-->
```

generate

```
You are using Mozilla/2.01 (compatible) Oracle(tm) PowerBrowser(tm)/
1.0a
```

Providing Host and Server Information

The following lines:

```
Host: <!--#echo var="REMOTE_HOST"-->
(<!--#echo var="REMOTE_ADDR"-->) - Server:
<!--#echo var="SERVER_NAME"-->
(<!--#echo var="SERVER_SOFTWARE"-->)
```

generate

```
Host: scuba.us.oracle.com (144.33.288.777) - Server:
spider.us.oracle.com (Oracle_Web_listener2.1/1.20in2)
```

Using LiveHTML to Send an ICX Request

Assume you want to compose a LiveHTML document, *status.html*, which takes two parameters: a user name and a property of interest. Also, assume the following URL is real:

```
status.html?user=foo+bar&property=job+title
```

Then, you can include in the *status.html* document a command to retrieve this information from some URL, which LiveHTML composes dynamically.

```
<!--#request URL=http://whatever/hr/$user/work?SQLString='select
$property from employee'&name=$user">
```

LiveHTML expands the above request to the following:

```
http://whatever/hr/
foo%20bar?SQLString=select+job%20title+from+employee
&name=foo+bar
```

The returned document (must have MIME type *text/html*) now replaces this line of the *status.html* document with the appropriate information.



4

Using the Perl Cartridge

Contents

- [Overview](#)
- [Tutorial](#)
- [Configuration](#)
- [Invocation](#)
- [Writing Perl Scripts for the Perl Cartridge](#)
- [Developing Perl Extension Modules](#)
- [Troubleshooting](#)

Overview

Perl is an interpreted language that is commonly used to write CGI scripts. Perl has powerful text processing capabilities, which makes it ideal for parsing requests from clients and generating dynamic HTML. You can download Perl from many sites on the Internet; you can find a list of pointers at <http://www.perl.com>.

The Perl Cartridge is essentially the Perl interpreter that has been modified to run under the Web Request Broker. Although you can run Perl scripts under Web Application Server without using the Perl Cartridge (that is, they are run as regular CGI scripts), you can get better performance if you run the Perl scripts under the Perl Cartridge. In addition, the Perl Cartridge has the Perl interpreter in it, so you do not need to have it on your system.

Perl scripts written for the Perl Cartridge are slightly different from Perl scripts for a CGI environment because of how the cartridge runs the interpreter. If you already have Perl scripts on your system that you run in a CGI environment, you may need to modify them to make them run correctly under the cartridge.

How the Perl Cartridge Improves Performance

Perl scripts run faster under the Perl Cartridge than under a regular CGI environment. The cartridge achieves this by:

- Maintaining a persistent Perl interpreter
This avoids the overhead of allocating and constructing a new interpreter each time the server receives a request to run a Perl CGI script. The interpreter is loaded once in memory and it keeps running after handling each request.
- Compiling Perl scripts beforehand, so that the interpreter only has to run the compiled script when a request is received

The Perl Cartridge caches the compiled script. If it receives subsequent requests for the same script, it does not recompile the script. Before using the cached version of the script, the Perl Cartridge checks whether it has been modified. If it has, the cartridge removes the compiled script from the cache and compiles the modified script and stores it in the cache.

Versions of Perl Supported

The Perl Cartridge supports Perl version 5.003.

Tutorial

This section provides a step-by-step guide on creating and invoking a simple Perl script that displays the values of some standard CGI environment variables.

This tutorial steps you through the following tasks:

1. [Writing the Perl Script](#)
2. [Adding a Virtual Path for the Perl Cartridge](#)
3. [Stopping and Restarting the Listener](#)
4. [Creating an HTML Page to Invoke the Perl Script](#)

This tutorial assumes you can log in as the “admin” user for the Web Application Server. This is required because you will be adding new settings to the configuration of the server.

1. Writing the Perl Script

Type the following script into a file and save it as **showEnv.pl**. Place the file in the **%ORAWEB_HOME%\test** directory. If you do not have permissions to create this directory, you can put the file in another directory, but remember your directory name when you are specifying the virtual path mapping.

```
print "Content-type: text/html\n\n";

print "<html>\n";
print "<head>\n";
print "<title>Some CGI environment variables</title>\n";
print "</head>\n";
print "<body bgcolor=white>\n";

print "<h1>Some CGI environment variables</h1>\n";

@varsToDisplay = (
    'HTTP_USER_AGENT',
    'REQUEST_METHOD',
    'PATH_INFO',
    'PATH_TRANSLATED');

print "<dl>\n";

while (@varsToDisplay) {
    $envVar = shift @varsToDisplay;
    print "<dt>$envVar\n";
    print "<dd>$ENV{$envVar}\n";
}

print "</dl>\n";
print "</body></html>\n";
```

2. Adding a Virtual Path for the Perl Cartridge

You need to be the “admin” user to add the virtual path.

1. Run your browser and display the home page for the Web Application Server.
2. Click the Web Application Server Manager icon to display the Administration home page.
3. Click Oracle Web Application Server to display the Administration page.
4. Click Cartridge Administration to display the Cartridge Administration page.
5. Click Perl Cartridge to display the Perl Cartridge Configuration page.
6. Click Web Request Broker Parameters for Perl to display the Update Cartridge Configuration page.
7. In the Virtual Paths section, add a new virtual path. Name the virtual path “/perl/test” and the physical path “%ORAWEB_HOME%\test”.
8. Scroll to the bottom of the page and click Modify Cartridge.

You should see a success message. The Virtual Paths section should display the new virtual path.

3. Stopping and Restarting the Listener

After reconfiguring the Web Application Server, you have to stop and restart the Listener for the new configuration to take effect.

1. Click the Listener button on the bottom of the page to go to the Oracle Web Listener Administration page.
2. Click Stop to stop the Listener process.
3. Click Start to restart the Listener process.

4. Creating an HTML Page to Invoke the Perl Script

To run the `current_users` procedure, type in the following URL in your browser:

```
http://your_machine_name/perl/test/showEnv.pl
```

It is more common, however, to invoke the procedure from an HTML page. For example, the following HTML page has a link that calls the URL.

```
<HTML>
<HEAD>
<title>CGI Environment Variables</title>
</HEAD>
<BODY>
<H1>CGI Environment Variables</H1>
<p><a href="http://hal.us.oracle.com:9999/perl/test/
showEnv.pl">Show CGI environment variables</a>
</BODY>
</HTML>
```

The following figures show the source page (the page containing the link that invokes the **showEnv.pl** script), and the page that is generated by the script.

Figure 4-1: The source page and the dynamically generated page in the tutorial

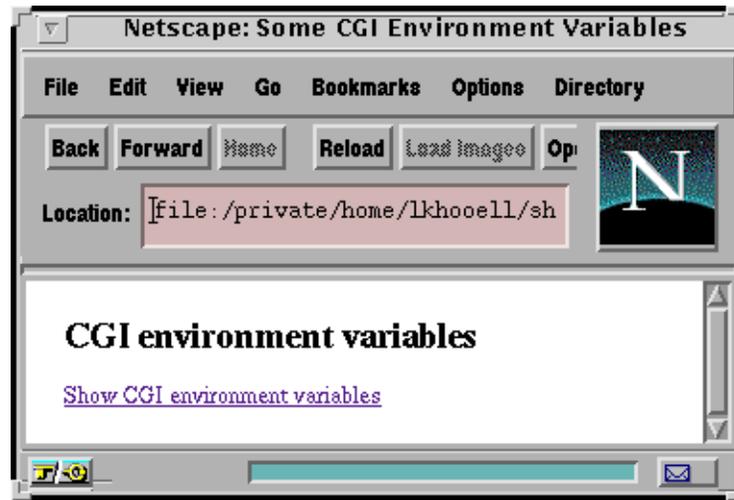
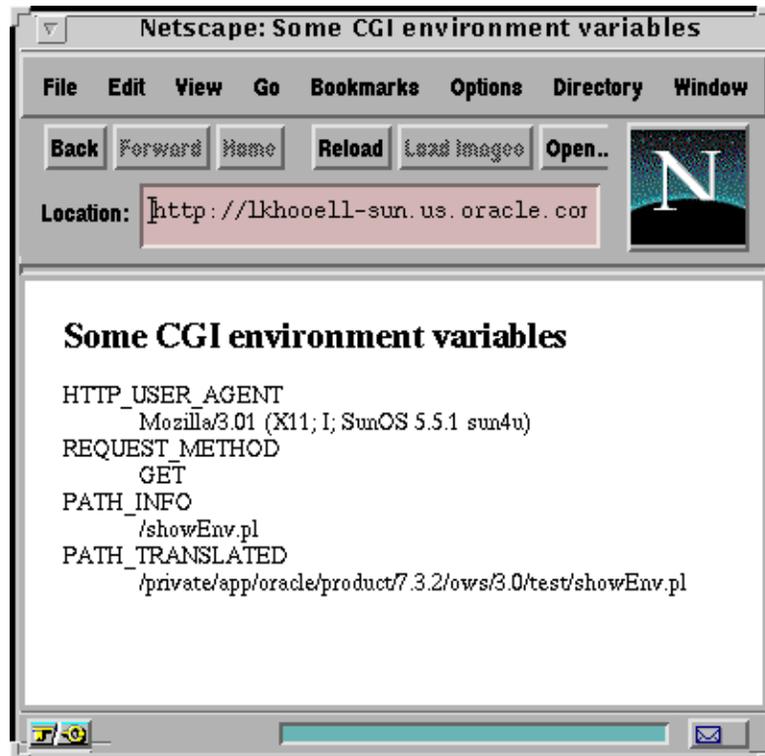


Figure 4-2: The page generated by the Perl script



Configuration

To configure the Perl Cartridge, you use the Web Application Server Manager, which is a collection of HTML forms. On these forms you enter information such as virtual paths for the Perl Cartridge, the minimum and maximum number of instances of the cartridge, and protection for the virtual paths.

To display the Perl Cartridge Configuration page:

1. Run your browser and display the home page for the Web Application Server.
2. Click the Web Application Server Manager icon to display the Administration home page.
3. Click Oracle Web Application Server to display the Administration page.
4. Click Cartridge Administration to display the Cartridge Administration page.
5. Click Perl Cartridge to display the Perl Cartridge Configuration page, which contains links to pages that let you configure the cartridge.
 - Web Request Broker Parameters for the Perl Cartridge displays the Update Cartridge Configuration page.
 - Logger
 - Transaction
 - Perl Cartridge-specific parameters

Virtual Paths

The following table shows the default virtual paths for the Perl Cartridge:

Table 4-1: Default virtual paths for the Perl Cartridge

Virtual Path	Physical Path
/sample/perl	%ORAWEB_HOME%\sample\perl
/perl	%ORAWEB_HOME%\perl

The /sample/perl virtual path is used by the Perl samples. You should not modify these virtual path mappings.

To add a virtual path (you need to be the “admin” user to do this):

1. Display the Perl Cartridge Configuration page.
2. Click Web Request Broker Parameters For Perl to display the Update Cartridge Configuration page.
3. In the Virtual Paths section, enter the name of the new virtual path and the physical path to which it maps.
4. Scroll to the bottom of the page and click Modify Cartridge.

You should see a success message. You must stop and restart the Listener for the new configuration to take effect.

Number of Perl Cartridge Instances

You can configure the minimum and maximum number of Perl Cartridge instances, which handle requests from the Web Application Server. When the WRB receives the first request for a Perl Cartridge, it starts up the minimum number of Perl Cartridge instances. If the WRB receives a request for the cartridge and all the running cartridge instances are busy, it starts a new instance of the Perl Cartridge, provided the number of running instances does not exceed the maximum value. The default values are 0 and 30 for minimum and maximum.

Number of Requests Processed by a Cartridge Instance

The Perl Cartridge process grows for each Perl script execution. Some AUTOLOADED subroutines can cause the Perl Cartridge's symbol table to grow. To limit the growth, enter "MaxRequests" as a new parameter on the Cartridge Configuration page and set its value to a small number. MaxRequests specifies the number of requests to which a cartridge instance can service before it terminates.

Logging

The Perl Cartridge uses the Logger Service to write information and error messages to a file in the file system (the default) or to an Oracle database. The Logger Administration page provides you with an option for choosing between these two modes of logging.

To enable logging for the cartridge:

1. Go to the "Update Cartridge Configuration" page and check that `LOGGER` is selected in the Services field.
2. Click "Go Back to Perl Cartridge Configuration Page" and click Logger on that page to display the "Logger Configuration For Cartridge Perl" page.
3. On this page, enter the name of the log file.

Cartridge Configuration Parameters

The Perl Cartridge uses a set of configuration parameters to get information about the path for Perl libraries and Perl modules. You can check these values, which are set during installation, using the Perl Cartridge Configuration page.

Table 4-2: Configuration parameters for the Perl Cartridge

Variable	Description
PRIVLIB	The path for private libraries. Default value: %ORAWEB_HOME%\perl\lib

Files in the Distribution

In addition to distributing the Perl Cartridge, the Web Application Server also distributes the Perl binaries, sources, and man pages. The binaries and man pages are installed when you install the Web Application Server, but the Perl sources are

not installed. You can access the source files, which are in compressed form, from the CD.

Table 4-3: Perl-related files in \$ORAWEB_HOME

Directory	Description
\$ORAWEB_HOME/perl	Supporting files
\$ORAWEB_HOME/perl/lib	Perl runtime libraries
\$ORAWEB_HOME/perl/bin	Perl binaries

Using %ORAWEB_HOME%\perl as Your Main Perl Installation

You can use the Perl distribution that is provided with the Web Application Server as your main Perl installation and use it to run Perl scripts outside the context of the Web Application Server (for example, you can use it to run Perl scripts from a shell).

To use the Perl executables:

1. Add the %ORAWEB_HOME%\perl\bin to your search path.

In a DOS shell, type:

```
set path=%path%;%ORAWEB_HOME%\perl\bin
```

2. Set the PERL5LIB environment variable.

In a DOS shell, type:

```
set PERL5LIB=%ORAWEB_HOME%\perl\lib
```

Invocation

To invoke a Perl script under the Perl Cartridge, the URL must be in the following format:

```
http://host_and_domain_name[:port]/virtual_path/  
script_name[?query_string]
```

where:

- *host_and_domain_name* specifies the domain and machine where the Web server is running
- *port* specifies the port at which the Web server is listening. If omitted, port 80 is assumed
- *virtual_path* specifies a virtual path mapped to the Perl Cartridge
- *script_name* specifies the file containing the Perl script. By convention, Perl scripts have a “.pl” extension.
- *query_string* specifies parameters for the script

For example, if a browser sends the following URL:

```
http://www.acme.com:9000/perl/myScript.pl
```

the web server running on `www.acme.com` and listening at port 9000 would handle the request. When the Listener receives the request, it passes the request to the WRB because the `/perl` virtual directory is configured to call the Perl Cartridge. The Perl Cartridge then executes `myScript.pl`.

Writing Perl Scripts for the Perl Cartridge

For Perl scripts to run correctly under the Perl Cartridge, they need to follow the following rules. Note that you may need to modify existing CGI Perl scripts so that they comply with these rules.

Variable Scoping

Be careful with namespace and variable scoping when running Perl scripts under the Perl Cartridge. In conventional CGI scripts, you declare a variable and use it. You do not have to worry about undefining the variable, because the script is restarted for each request and is not reentrant.

In the case of the Perl Cartridge, global variables persist across multiple calls. The value acquired by a variable at the end of one execution of the script is the initial value for the variable when the script is executed next time. This might cause inconsistent outputs, as seen in the following example:

```
1 print "Content-type: text/plain\n\n";
2 @question = (where, are, you, staying);
3 print "@question\n";
4 $" = "\n";
5 @answer = (all, on, separate, lines);
6 print "@answer\n";
```

When run by the cartridge for the first time, it outputs:

```
where are you staying
all
on
separate
lines
```

When run the second time and thereafter, it outputs:

```
where
are
you
staying
all
on
separate
lines
```

This is because the script changes the value of the Perl special variable `$"` to `"\n"` at line 4 and does not reset it to its original value before exiting the script. The Perl Cartridge has a initial value of `" "` (blank) for the special variable.

One way of fixing the problem is to add this line to the end of the script to reset the value of the variable:

```
$" = " " ;
```

If the execution of your script depends on values of global variables, make sure that these variables are reset to the original values.

To avoid the above problem:

- Limit the scope of your variable to the required extent only. Make sure that the variables expire after their use (by scoping the variable with `my ()`) or that the script resets them to their original values.
- Reduce global variables and localize them where possible. When you need to modify Perl's global variables, localize them so that the modification affects the local instance of the variable only. This causes the modified value of the variable to be applicable only for that run of the script.

Another way to fix the example above is to localize the `$"` variable:

```
1 print "Content-type: text/plain\n\n";
2 @question = (where, are, you, staying);
3 print "@question\n";
4 local("$");
5 $" = "\n";
6 @answer = (all, on, separate, lines);
7 print "@answer\n";
```

Line 4 localizes the `$"` Perl special variable. When the script runs as a subroutine in a package, the localized variable `$"` has life only for that run of the script.

Namespace Collisions

The Perl Cartridge caches compiled Perl scripts to speed up the response time. If not properly handled, the caching of multiple Perl scripts can lead to namespace collisions. To avoid this, the Perl Cartridge translates the Perl script file name into a guaranteed-unique package name, and then compiles the code into the package using `eval`. The script is now available to the Perl Cartridge in compiled form as a subroutine in the unique package name. When a request for the script is received, the cartridge translates the filename to the package name and runs the subroutine handler.

Although the above mechanism avoids the namespace collisions, you need to remember that the default package name for the script is no longer "main". The default package name for the script is something that is generated by the cartridge.

The following example shows how the different package name affects your Perl scripts.

Perl comes with library files, which are Perl scripts that provide utility functions. For example, **bigint.pl** provides Perl with arbitrary size integer mathematics subroutines. This library defines many of the subroutines in the namespace of the package "main". Here is an example:

```
# normalize string form of number. Strip leading zeros. Strip any
# white space and add a sign, if missing.
```

```

# Strings that are not numbers result the value 'NaN'.
sub main'bnorm { #(num_str) return num_str
    local($_) = @_;
    s/\s+//g;                # strip white space
    if (s/^( [+ - ]? )0*(\d+)$/$1$2/) {    # test if number
        substr($_,$[,0) = '+' unless $1; # Add missing sign
        s/^-0/+0/;
        $_;
    } else {
        'NaN';
    }
}
}

```

A conventional CGI Perl script can use this library as follows:

```

1 # namespace.pl
2 require "bigint.pl";
3 print "Content-type: text/plain\n\n";
4 $one = &bnorm( 456);
5 print "one = $one\n";

```

Line 4 can call the `bnorm` subroutine without specifying the package name, because the default package name for conventional scripts is “main” and the `bnorm` subroutine is available in the “main” package’s namespace. But under the Perl Cartridge, a script is compiled into a package whose name depends on the filename. Any `eval` statement (note that `require` calls `eval`) is evaluated in this package’s namespace. In the example, although **bigint.pl** is compiled and stored in the package of your script, the subroutines are explicitly stored in the “main” package by fully qualified subroutine name (for example, `main'bnorm`).

To run the example under the Perl Cartridge, you need to modify the script to call `bnorm` as `main'bnorm`.

```

1 # namespace.pl
2 require "bigint.pl";
3 print "Content-type: text/plain\n\n";
4 $one = &main'bnorm( 456);
5 print "one = $one\n";

```

You should not assume that the default package name is “main”. To see the name of the package that you are in currently, you can invoke the Perl function `caller()`, which returns a list containing the package name as generated by the Perl Cartridge, the file name, and the current line number.

No Need for the #! Line

Typically, the first line of Perl CGI scripts tells the system the location of the Perl interpreter. The line begins with “#!” and looks something like:

```
#!/usr/bin/perl
```

Perl scripts that are executed under the Perl Cartridge do not need to have that line because the cartridge uses its own built-in Perl interpreter.

System Resources

System resources acquired by your Perl script should be freed before the script exits; otherwise, the persistent Perl interpreter in the Perl Cartridge will reach system limits for the resources.

In conventional CGI Perl scripts, you can open a file and do file operations without closing it before the script exits. It does not matter in this case because the resources are returned when the Perl interpreter exits, but in the Perl Cartridge environment, the file remains open even after the script execution is finished. You have to explicitly close the file in your script.

Developing Perl Extension Modules

The Perl Cartridge installation comes with the Perl interpreter runtime environment. Perl extension modules that you develop using this Perl interpreter runtime environment can be accessed by the Perl Cartridge.

The Perl interpreter installation is in the `%ORAWEB_HOME%\perl` directory. The `%ORAWEB_HOME%\perl\bin` directory contains the “perl” executable. You can develop and install your extension modules under the `%ORAWEB_HOME%\perl\lib` directory.

For this you need to set the following environment variables:

- Set your path variable to use the “perl” executable from the `%ORAWEB_HOME%\perl\bin` directory.
- Set the PERL5LIB environment variable to:
`%ORAWEB_HOME%\perl\lib`

Troubleshooting

Problems with Invoking Your Perl Script

If your Perl script cannot be invoked:

- Make sure that the Perl Cartridge is registered with the WRB, and the virtual path for the directory containing your script maps to the Perl Cartridge.
- Make sure that the Web Listener and the WRB are functioning properly. For example, check that you can invoke other Perl scripts and other cartridges. You can try invoking the sample Perl scripts.

Log Files

If you have enabled logging for the Perl Cartridge, error messages are logged to the file you specified in the “Logger Configuration For Cartridge Perl” page.

In addition, the Perl Cartridge also writes any messages sent to stderr from within the Perl scripts (for example, output from `warn` and `die`) to the log file.

Unhandled Errors

The Perl Cartridge runs your Perl scripts as subroutines that are `eval`'ed. If an error occurs in the script, `eval` returns the error to the Perl Cartridge, which writes the error to the log file and sends an error message to the browser.

A

The http and htf Packages

This chapter describes the functions, procedures, and data types in the **http** and **htf** (hypertext procedures and hypertext functions) packages in the PL/SQL Web Toolkit. The contents of these packages generate HTML tags that you can use to create dynamic web pages.

For every **http** procedure that generates HTML tags, there is a corresponding **htf** function with identical parameters. The difference is that the function passes its output to its caller and is typically used for nesting within procedures or other functions.

Parameters that have default values are optional.

Note: To look up **htf** functions, see the entry for the corresponding HTTP procedures. The string listed under “Generates” is the return value of the function.

The following list provides pointers to locations where you can get more information:

- For information on the Web in general:
 - <http://www.boutell.com/faq>
- For information on HTML 3.2:
 - <http://www.w3.org/pub/WWW/MarkUp/Wilbur/features.html>

Summary

HTML, HEAD, and BODY Tags

[http.htmlOpen](#), [http.htmlClose](#) - generate <HTML> and </HTML>

[http.headOpen](#), [http.headClose](#) - generate <HEAD> and </HEAD>

[http.bodyOpen](#), [http.bodyClose](#) - generate <BODY> and </BODY>

Comment Tag

[http.comment](#) - generates <!-- and -->

Tags in the <HEAD> Area

[http.base](#) - generates <BASE>

[http.linkRel](#) - generates <LINK> with the REL attribute

[http.linkRev](#) - generates <LINK> with the REV attribute

[http.title](#) - generates <TITLE>

[http.meta](#) - generates <META>

[http.script](#) - generates <SCRIPT>

[http.style](#) - generates <STYLE>

[http.isindex](#) - generates <ISINDEX>

Applet Tags

[http.appletopen](#), [http.appletclose](#) - generate <APPLET> and </APPLET>

[http.param](#) - generates <PARAM>

List Tags

[http.olistOpen](#), [http.olistClose](#) - generate and

[http.ulistOpen](#), [http.ulistClose](#) - generate and

[http.dlistOpen](#), [http.dlistClose](#) - generate <DL> and </DL>

[http.dlistTerm](#) - generates <DT>

[http.dlistDef](#) - generates <DD>

[http.dirlistOpen](#), [http.dirlistClose](#) - generate <DIR> and </DIR>

[http.listHeader](#) - generates <LH>

[http.listingOpen](#), [http.listingClose](#) - generate <LISTING> and </LISTING>

[http.menulistOpen](#), [http.menulistClose](#) - generate <MENU> and </MENU>

[http.listItem](#) - generates

Form Tags

[http.formOpen](#), [http.formClose](#) - generate <FORM> and </FORM>

[http.formCheckbox](#) - generates <INPUT TYPE="CHECKBOX">

[http.formHidden](#) - generates <INPUT TYPE="HIDDEN">

[http.formImage](#) - generates <INPUT TYPE="IMAGE">

[http.formPassword](#) - generates <INPUT TYPE="PASSWORD">

[http.formRadio](#) - generates <INPUT TYPE="RADIO">

[http.formSelectOpen](#), [http.formSelectClose](#) - generate <SELECT> and </SELECT>

[http.formSelectOption](#) - generates <OPTION>

[http.formText](#) - generates <INPUT TYPE="TEXT">

[http.formTextarea](#), [http.formTextarea2](#) - generate <TEXTAREA>

[http.formTextareaOpen](#), [http.formTextareaOpen2](#), [http.formTextareaClose](#) - generate <TEXTAREA> and </TEXTAREA>

[http.formReset](#) - generates <INPUT TYPE="RESET">

[http.formSubmit](#) - generates <INPUT TYPE="SUBMIT">

Table Tags

[http.tableOpen](#), [http.tableClose](#) - generate <TABLE> and </TABLE>

[http.tableCaption](#) - generates <CAPTION>

[http.tableRowOpen](#), [http.tableRowClose](#) - generate <TR> and </TR>

[http.tableHeader](#) - generates <TH>

[http.tableData](#) - generates <TD>

IMG, HR, and A Tags

[http.line](#), [http.hr](#) - generate <HR>

[http.img](#), [http.img2](#) - generate

[http.anchor](#), [http.anchor2](#) - generates <A>

[http.mapOpen](#), [http.mapClose](#) - generate <MAP> and </MAP>

Paragraph Formatting Tags

[http.header](#) - generates heading tags (<H1> to <H6>)

[http.para](#), [http.paragraph](#) - generate <P>

[http.print](#), [http.prn](#) - generate any text that is passed in

[http.prints](#), [http.ps](#) - generate any text that is passed in; special characters in HTML are escaped

[http.preOpen](#), [http.preClose](#) - generate <PRE> and </PRE>

[http.blockquoteOpen](#), [http.blockquoteClose](#) - generate <BLOCKQUOTE> and </BLOCKQUOTE>

[http.div](#) - generates <DIV>

[http.nl](#), [http.br](#) - generate

[http.nobr](#) - generates <NOBR>

[http.wbr](#) - generates <WBR>

[http.plaintext](#) - generates <PLAINTEXT>

[http.address](#) - generates <ADDRESS>

[http.mailto](#) - generates <A> with the MAILTO attribute

[http.area](#) - generates <AREA>

[http.bgsound](#) - generates <BGSOUND>

Character Formatting Tags

[http.basefont](#) - generates <BASEFONT>

[http.big](#) - generates <BIG>

[http.bold](#) - generates

[http.center](#) - generates <CENTER> and </CENTER>

[http.centerOpen](#), [http.centerClose](#) - generate <CENTER> and </CENTER>

[http.cite](#) - generates <CITE>

[http.code](#) - generates <CODE>

[http.dfn](#) - generates <DFN>

[http.emphasis](#), [http.em](#) - generate

[http.fontOpen](#), [http.fontClose](#) - generate and

[http.italic](#) - generates <I>

[http.keyboard](#), [http.kbd](#) - generate <KBD> and </KBD>

[http.s](#) - generates <S>

[http.sample](#) - generates <SAMP>

[http.small](#) - generates <SMALL>

[http.strike](#) - generates <STRIKE>

[http.strong](#) - generates

[http.sub](#) - generates <SUB>

[http.sup](#) - generates <SUP>

[http.teletype](#) - generates <TT>

[http.underline](#) - generates <U>

[http.variable](#) - generates <VAR>

Frame Tags

[http.frame](#) - generates <FRAME>

[http.framesetOpen](#), [http.framesetClose](#) - generate <FRAMESET> and </FRAMESET>

[http.noframesOpen](#), [http.noframesClose](#) - generate <NOFRAMES> and </NOFRAMES>

htp.address

Syntax

```
htp.address (
    cvalue          in   varchar2
    cnowrap         in   varchar2   DEFAULT NULL
    cclear          in   varchar2   DEFAULT NULL
    cattributes     in   varchar2   DEFAULT NULL);

htf.address (cvalue, cnowrap, cclear, cattributes) return varchar2;
```

Purpose

Generates the <ADDRESS> and </ADDRESS> tags, which specify the address, author and signature of a document.

Parameters

cvalue - the string that goes between the <ADDRESS> and </ADDRESS> tags

cnowrap - if the value for this parameter is not NULL, the NOWRAP attribute is included in the tag

cclear - the value for the CLEAR attribute

cattributes - other attributes to be included as-is in the tag

Generates/Returns

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

http.anchor, http.anchor2

Syntax

```
http.anchor (  
    curl          in    varchar2  
    ctext         in    varchar2  
    cname         in    varchar2    DEFAULT NULL  
    cattributes  in    varchar2    DEFAULT NULL);  
  
http.anchor (curl, ctext, cname, cattributes) return varchar2;
```

```
http.anchor2 (  
    curl          in    varchar2  
    ctext         in    varchar2  
    cname         in    varchar2    DEFAULT NULL  
    ctarget       in    varchar2    DEFAULT NULL  
    cattributes  in    varchar2    DEFAULT NULL);  
  
http.anchor2 (curl, ctext, cname, ctarget, cattributes) return  
varchar2;
```

Purpose

Generates the <A> and HTML tags, which specify the source or destination of a hypertext link. This tag accepts several attributes, but either HREF or NAME is required. HREF specifies to where to link. NAME allows this tag to be a target of a hypertext link.

The difference between these subprograms is that http.anchor2 provides a target and therefore can be used for a frame.

Parameters

curl - the value for the HREF attribute

ctext - the string that goes between the <A> and tags

cname - the value for the NAME attribute

ctarget - the value for the TARGET attribute

cattributes - other attributes to be included as-is in the tag

Generates

http.anchor generates:

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

http.anchor2 generates:

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

http.appletopen, http.appletclose

Syntax

```
http.appletopen(  
    ccode          in    varchar2  
    cheight        in    number  
    cwidth         in    number  
    cattributes    in    varchar2    DEFAULT NULL);  
htf.appletopen(ccode, cheight, cwidth, cattributes) return varchar2;  
  
http.appletclose;  
htf.appletclose return varchar2;
```

Purpose

http.appletopen generates the <APPLET> HTML tag, which begins the invocation of a Java applet. You close the applet invocation with http.appletclose, which generates the </APPLET> HTML tag.

You can specify parameters to the Java applet using the [http.param](#) procedure.

Note: You have to use the cattributes parameter to specify the CODEBASE attribute because the PL/SQL Cartridge does not know where to find the class files. The CODEBASE attribute specifies the virtual path containing the class files.

Parameters

ccode - the value for the CODE attribute, which specifies the name of the applet class

cheight - the value for the HEIGHT attribute

cwidth - the value for the WIDTH attribute

cattributes - other attributes to be included as-is in the tag

Generates

http.appletopen generates:

```
<APPLET CODE=ccode HEIGHT=cheight WIDTH=cwidth>
```

http.appletclose generates:

```
</APPLET>
```

Example

```
http.appletopen('testclass.class', 100, 200, 'CODEBASE="/ows-applets"')
```

generates

```
<APPLET CODE="testclass.class" height=100 width=200 CODEBASE="/ows-applets">
```

htp.area

Syntax

```
htp.area(  
    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);  
  
htf.area(ccoords, cshape, chref, cnohref, ctarget, cattributes)  
return varchar2;
```

Purpose

Generates the <AREA> HTML tag, which defines a client-side image map. This tag defines areas within the image and destinations for the areas.

Parameters

ccoords - the value for the COORDS attribute

cshape - the value for the SHAPE attribute

chref - the value for the HREF attribute

cnohref - if the value for this parameter is not NULL, the NOHREF attribute is added to the tag

ctarget - the value for the TARGET attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.base

Syntax

```
htp.base(  
    ctarget      in   varchar2   DEFAULT NULL  
    cattributes  in   varchar2   DEFAULT NULL);  
htf.base(ctarget, cattributes) return varchar2;
```

Purpose

Generates the <BASE> HTML tag, which records the URL of the document.

Parameters

ctarget - the value for the TARGET attribute, which establishes a window name to which all links in this document are targeted

cattributes - other attributes to be included as-is in the tag

Generates

```
<BASE HREF="<current URL>" TARGET="ctarget" cattributes>
```

htp.basefont

Syntax

```
htp.basefont(nsize in integer);  
htf.basefont(nsize) return varchar2;
```

Purpose

Generates the <BASEFONT> HTML tag, which specifies the base font size for a web page.

Parameters

nsize - the value for the SIZE attribute

Generates

```
<BASEFONT SIZE="nsize">
```

htp.bgsound

Syntax

```
htp.bgsound(  
  csrc          in   varchar2  
  cloop        in   varchar2   DEFAULT NULL  
  cattributes  in   varchar2   DEFAULT NULL);  
  
htf.bgsound(csrc, cloop, cattributes) return varchar2;
```

Purpose

Generates the <BGSOUND> HTML tag, which includes audio for a web page.

Parameters

csrc - the value for the SRC attribute

cloop - the value for the LOOP attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.big

Syntax

```
htp.big(  
    ctext          in   varchar2  
    cattributes   in   varchar2   DEFAULT NULL);  
htf.big(ctext, cattributes) return varchar2;
```

Purpose

Generates the <BIG> and </BIG> tags, which direct the browser to render the text in a bigger font.

Parameters

ctext - the text that goes between the tags

cattributes - other attributes to be included as-is in the tag

Generates

```
<BIG cattributes>ctext</BIG>
```

http.blockquoteOpen, http.blockquoteClose

Syntax

```
http.blockquoteOpen (
  cnowrap      in   varchar2      DEFAULT NULL
  cclear       in   varchar2      DEFAULT NULL
  cattributes  in   varchar2      DEFAULT NULL);

htf.blockquoteOpen (cnowrap, cclear, cattributes) return varchar2;

http.blockquoteClose;

htf.blockquoteClose return varchar2;
```

Purpose

Generates the <BLOCKQUOTE> and </BLOCKQUOTE> tag, which mark a section of quoted text.

Parameters

cnowrap - if the value for this parameter is not NULL, the NOWRAP attribute is added to the tag

cclear - the value for the CLEAR attribute

cattributes - other attributes to be included as-is in the tag

Generates

http.blockquoteOpen generates:

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

http.blockquoteClose generates:

```
</BLOCKQUOTE>
```

http.bodyOpen, http.bodyClose

Syntax

```
http.bodyOpen(  
    cbackground    in    varchar2 DEFAULT NULL  
    cattributes    in    varchar2 DEFAULT NULL);  
htf.bodyOpen(cbackground, cattributes) return varchar2;  
  
http.bodyClose;  
htf.bodyClose return varchar2;
```

Purpose

Generates the <BODY> and </BODY> tags, which mark the body section of an HTML document.

Parameters

cbackground - the value for the BACKGROUND attribute, which specifies a graphic file to use for the background of the document

cattributes - other attributes to be included as-is in the tag

Generates

http.bodyOpen generates:

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

http.bodyClose generates:

```
</BODY>
```

Example

```
http.bodyOpen('/img/background.gif');
```

generates:

```
<BODY background="/img/background.gif">
```

htp.bold

Syntax

```
htp.bold (  
    ctext          in   varchar2  
    cattributes    in   varchar2   DEFAULT NULL);  
htf.bold (ctext, cattributes) return varchar2;
```

Purpose

Generates the and tags, which direct the browser to display the text in boldface.

Parameters

ctext - the text that goes between the tags

cattributes - other attributes to be included as-is in the tag

Generates

<B *cattributes*>*ctext*

htp.center

Syntax

```
htp.center(ctext in varchar2);  
htf.center(ctext in varchar2) return varchar2;
```

Purpose

Generates the <CENTER> and </CENTER> tags, which center a section of text within a web page.

Parameters

ctext - the text to center

Generates

```
<CENTER>ctext</CENTER>
```

htp.centerOpen, htp.centerClose

Syntax

```
htp.centerOpen;  
htf.centerOpen return varchar2;  
  
htp.centerClose;  
htf.centerClose return varchar2;
```

Purpose

Generates the <CENTER> and </CENTER> tags, which mark the section of text to center.

Parameters

none

Generates

htp.centerOpen generates:

```
<CENTER>
```

htp.centerClose generates:

```
</CENTER>
```

htp.cite

Syntax

```
htp.cite (  
    ctext          in    varchar2  
    cattributes    in    varchar2    DEFAULT NULL);  
  
htf.cite (ctext, cattributes) return varchar2;
```

Purpose

Generates the <CITE> and </CITE> tags, which direct the browser to render the text as citation.

Parameters

ctext - the text to render as citation

cattributes - other attributes to be included as-is in the tag

Generates

```
<CITE cattributes>ctext</CITE>
```

htp.code

Syntax

```
htp.code (  
    ctext          in          varchar2  
    cattributes   in          varchar2   DEFAULT NULL);  
htf.code (ctext, cattributes) return varchar2;
```

Purpose

Generates the <CODE> and </CODE> tags, which direct the browser to render the text in monospace font.

Parameters

ctext - the text to render as code

cattributes - other attributes to be included as-is in the tag

Generates

```
<CODE cattributes>ctext</CODE>
```

htp.comment

Syntax

```
htp.comment (c $text$  in varchar2);  
htf.comment (c $text$  in varchar2) return varchar2;
```

Purpose

Generates the comment tags.

Parameters

c $text$ - the comment

Generates

```
<!-- c $text$  -->
```

htp.dfn

Syntax

```
htp.dfn(ctext in varchar2);  
htf.dfn(ctext in varchar2) return varchar2;
```

Purpose

Generates the <DFN> and </DFN> tags, which direct the browser to render the text in italics

Parameters

ctext - the text to render in italics

Generates

```
<DFN>ctext</DFN>
```

http.dirlistOpen, http.dirlistClose

Syntax

```
http.dirlistOpen;  
htf.dirlistOpen return varchar2;  
  
http.dirlistClose;  
htf.dirlistClose return varchar2;
```

Purpose

Generates the <DIR> and </DIR> tags, which create a directory list section. A directory list presents a list of items that contains up to 20 characters. Items in this list are typically arranged in columns, typically 24 characters wide. The tag or [http.listItem](#) must appear directly after you use this tag to define the items in the list.

Parameters

none

Generates

http.dirlistOpen generates:

```
<DIR>
```

http.dirlistClose generates:

```
</DIR>
```

htp.div

Syntax

```
htp.div(  
    calign          in   varchar2    DEFAULT NULL  
    cattributes    in   varchar2    DEFAULT NULL);  
htf.div(calign, cattributes) return varchar2;
```

Purpose

Generates the <DIV> tag, which creates document divisions.

Parameters

calign - the value for the ALIGN attribute

cattributes - other attributes to be included as-is in the tag

Generates

```
<DIV ALIGN="calign" cattributes>
```

http.dlistOpen, http.dlistClose

Syntax

```
http.dlistOpen (
    cclear          in    varchar2    DEFAULT NULL
    cattributes     in    varchar2    DEFAULT NULL);
http.dlistOpen (cclear, cattributes) return varchar2;

http.dlistClose;
http.dlistClose return varchar2;
```

Purpose

Generates the <DL> and </DL> tags, which create a definition list. A definition list looks like a glossary: it contains terms and definitions. Terms are inserted using [http.dlistTerm](#), and definitions are inserted using [http.dlistDef](#).

Parameters

cclear - the value for the CLEAR attribute

cattributes - other attributes to be included as-is in the tag

Generates

http.dlistOpen generates:

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

http.dlistClose generates:

```
</DL>
```

htp.dlistDef

Syntax

```
htp.dlistDef(  
  ctext      in   varchar2  DEFAULT NULL  
  cclear     in   varchar2  DEFAULT NULL  
  cattributes in   varchar2  DEFAULT NULL);  
  
htf.dlistDef(ctext, cclear, cattributes) return varchar2;
```

Purpose

Generates the <DD> tag, which is used to insert definitions of terms. This tag is used in the context of the definition list <DL>, where terms are tagged with <DT> and definitions are tagged with <DD>.

Parameters

ctext - the definition for the term

cclear - the value for the CLEAR attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.dlistTerm

Syntax

```
htp.dlistTerm (  
  ctext      in   varchar2  DEFAULT NULL  
  cclear     in   varchar2  DEFAULT NULL  
  cattributes in   varchar2  DEFAULT NULL);  
  
htf.dlistTerm (ctext, cclear, cattributes) return varchar2;
```

Purpose

Generates the <DT> tag, which defines a term in a definition list <DL>.

Parameters

ctext - the term

cclear - the value for the CLEAR attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.emphasis, htp.em

Syntax

```
htp.em (
    ctext          in    varchar2
    cattributes    in    varchar2    DEFAULT NULL);
htf.em (ctext, cattributes) return varchar2;

htp.emphasis (
    ctext          in    varchar2
    cattributes    in    varchar2    DEFAULT NULL);
htf.emphasis (ctext, cattributes) return varchar2;
```

Purpose

Generates the and tags, which define text to be emphasized.

Parameters

ctext - the text to emphasize

cattributes - other attributes to be included as-is in the tag

Generates

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

htf.escape_sc

Syntax

```
htf.escape_sc(ctext IN VARCHAR2) return VARCHAR2;  
htp.escape_sc(ctext IN VARCHAR2);
```

Purpose

Replaces characters that have special meaning in HTML with their escape sequences. The following characters are converted:

From	To
&	&
"	"
<	<
>	>

Note that the procedure version of this subprogram does the same thing as [htp.prints](#) and [htp.ps](#).

Parameters

ctext - the string to convert

Returns

The converted string.

htf.escape_url

Syntax

```
htf.escape_url(p_url IN VARCHAR2) return VARCHAR2;
```

Purpose

Replaces characters that have special meaning in HTML and HTTP with their escape sequences. The following characters are converted:

From	To
&	&
"	"
<	<
>	>
%	%25

Parameters

p_url - the string to convert

Returns

The converted string.

http.fontOpen, http.fontClose

Syntax

```
http.fontOpen(  
  ccolor      in   varchar2  DEFAULT NULL  
  cface       in   varchar2  DEFAULT NULL  
  csize       in   varchar2  DEFAULT NULL  
  cattributes in   varchar2  DEFAULT NULL);  
http.fontOpen(ccolor, cface, csize, cattributes) return varchar2;  
  
http.fontClose;  
http.fontClose return varchar2;
```

Purpose

Generates the and tags, which mark a section of text with the specified font characteristics.

Parameters

ccolor - the value for the COLOR attribute

cface - the value for the FACE attribute

csize - the value for the SIZE attribute

cattributes - other attributes to be included as-is in the tag

Generates

http.fontOpen generates:

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

http.fontClose generates:

```
</FONT>
```

htp.formCheckbox

Syntax

```
htp.formCheckbox (  
  cname      in   varchar2  
  cvalue     in   varchar2   DEFAULT 'on'  
  cchecked   in   varchar2   DEFAULT NULL  
  cattributes in   varchar2   DEFAULT NULL);  
  
htf.formCheckbox (cname, cvalue, cchecked, cattributes) return  
varchar2;
```

Purpose

Generates the <INPUT> tag with TYPE="checkbox", which inserts a checkbox element in a form. A checkbox element is a button that the user can toggle on or off.

Parameters

cname - the value for the NAME attribute

cvalue - the value for the VALUE attribute

cchecked - if the value for this parameter is not NULL, the CHECKED attribute is added to the tag

cattributes - other attributes to be included as-is in the tag

Generates

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

http.formOpen, http.formClose

Syntax

```
http.formOpen (  
    curl          in    varchar2  
    cmethod       in    varchar2    DEFAULT 'POST'  
    ctarget       in    varchar2  
    cenctype      in    varchar2    DEFAULT NULL  
    cattributes   in    varchar2    DEFAULT NULL);  
  
http.formOpen (curl, cmethod, ctarget, cenctype, cattributes) return  
varchar2;  
  
http.formClose;  
  
http.formClose return varchar2;
```

Purpose

Generates the <FORM> and </FORM> tags, which create a form section in an HTML document.

Parameters

curl - the URL of the WRB cartridge or CGI script to which the contents of the form is sent. This parameter is required.

cmethod - the value for the METHOD attribute. The value can be "GET" or "POST".

ctarget - the value for the TARGET attribute

cenctype - the value for the ENCTYPE attribute

cattributes - other attributes to be included as-is in the tag

Generates

http.formOpen generates:

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

http.formClose generates:

```
</FORM>
```

htp.formHidden

Syntax

```
htp.formHidden (  
  cname      in   varchar2  
  cvalue     in   varchar2   DEFAULT NULL  
  cattributes in   varchar2   DEFAULT NULL);  
  
htf.formHidden (cname, cvalue, cattributes) return varchar2;
```

Purpose

Generates the <INPUT> tag with TYPE="hidden", which inserts a hidden form element. This element is not seen by the user and is used to submit additional values to the script.

Parameters

cname - the value for the NAME attribute

cvalue - the value for the VALUE attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.formImage

Syntax

```
htp.formImage (  
  cname      in   varchar2  
  csrc       in   varchar2  
  calign     in   varchar2   DEFAULT NULL  
  cattributes in   varchar2   DEFAULT NULL);  
  
htf.formImage (cname, csrc, calign, cattributes) return varchar2;
```

Purpose

Generates the <INPUT> tag with TYPE="image", which creates an image field on which the user can click and cause the form to be submitted immediately. The coordinates of the selected point are measured in pixels, and returned (along with other contents of the form) in two nam-/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.

Parameters

cname - the VALUE for the NAME attribute
csrc - the value for the SRC attribute, which specifies the image file
calign - the value for the ALIGN attribute
cattributes - other attributes to be included as-is in the tag

Generates

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

htp.formPassword

Syntax

```
htp.formPassword (  
    cname      in   varchar2  
    csize      in   varchar2  
    cmaxlength in   varchar2   DEFAULT NULL  
    cvalue     in   varchar2   DEFAULT NULL  
    cattributes in   varchar2   DEFAULT NULL);  
  
htf.formPassword (cname, csize, cmaxlength, cvalue, cattributes)  
return varchar2;
```

Purpose

Generates the <INPUT> tag with TYPE="password", which creates a single-line text entry field. When the user enters text in the field, each character is represented by one asterisk. This is usually used for entering passwords.

Parameters

cname - the value for the NAME attribute

csize - the value for the SIZE attribute

cmaxlength - the value for the MAXLENGTH attribute

cvalue - the value for the VALUE attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.formRadio

Syntax

```
htp.formRadio (  
  cname          in   varchar2  
  cvalue         in   varchar2  
  cchecked       in   varchar2      DEFAULT NULL  
  cattributes    in   varchar2      DEFAULT NULL);  
  
htf.formRadio (cname, cvalue, cchecked, cattributes) return varchar2;
```

Purpose

Generates the <INPUT> tag with TYPE="radio", which creates a radio button on the HTML form.

Within a set of radio buttons, the user can select only one button. Each radio button in the same set should have the same name, but different value. The selected radio button generates a name/value pair.

Parameters

cname - the value for the NAME attribute

cvalue - the value for the VALUE attribute

cchecked - if the value for this parameter is not NULL, the CHECKED attribute is added to the tag

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.formReset

Syntax

```
htp.formReset (
    cvalue      in   varchar2      DEFAULT 'Reset'
    cattributes in   varchar2      DEFAULT NULL);

htf.formReset (cvalue, cattributes) return varchar2;
```

Purpose

Generates the <INPUT> tag with TYPE="reset", which creates a button that, when clicked, resets all the form fields to their initial values.

Parameters

cvalue - the value for the VALUE attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

http.formSelectOpen, http.formSelectClose

Syntax

```
http.formSelectOpen (
    cname          in   varchar2
    cprompt        in   varchar2   DEFAULT NULL
    nsize          in   integer     DEFAULT NULL
    cattributes    in   varchar2   DEFAULT NULL);

http.formSelectOpen (cname, cprompt, nsize, cattributes) return
varchar2;

http.formSelectClose;

http.formSelectClose return varchar2;
```

Purpose

Generates the <SELECT> and </SELECT> tags, which create a Select form element. A Select form element is a listbox, from which the user can select one or more values. The values are inserted using [http.formSelectOption](#).

Parameters

cname - the value for the NAME attribute
cprompt - the string preceding the list box
nsize - the value for the SIZE attribute
cattributes - other attributes to be included as-is in the tag

Generates

http.formSelectOpen generates:

```
cprompt <SELECT NAME="cname" SIZE="nsize" cattributes>
```

http.formSelectClose generates:

```
</SELECT>
```

Example

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

Generates:

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

htp.formSelectOption

Syntax

```
htp.formSelectOption (  
    cvalue          in    varchar2  
    cselected       in    varchar2    DEFAULT NULL  
    cattributes     in    varchar2    DEFAULT NULL);  
  
htf.formSelectOption (cvalue, cselected, cattributes) return  
varchar2;
```

Purpose

Generates the <OPTION> tag, which represents one choice in a Select element.

Parameters

cvalue - the text for the option

cselected - if the value for this parameter is not NULL, the SELECTED attribute is added to the tag

cattributes - other attributes to be included as-is in the tag

Generates

```
<OPTION SELECTED cattributes>cvalue
```

Example

See [htp.formSelectOpen](#), [htp.formSelectClose](#)

htp.formSubmit

Syntax

```
htp.formSubmit (
  cname      in   varchar2      DEFAULT NULL
  cvalue     in   varchar2      DEFAULT 'Submit'
  cattributes in   varchar2      DEFAULT NULL);

htf.formSubmit (cname, cvalue, cattributes) return varchar2;
```

Purpose

Generates the <INPUT> tag with TYPE="submit", which creates a button that, when clicked, submits the form.

If the button has a NAME attribute, the button contributes a name/value pair to the submitted data.

Parameters

cname - the value for the NAME attribute

cvalue - the value for the VALUE attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.formText

Syntax

```
htp.formText (  
  cname          in   varchar2  
  csize          in   varchar2   DEFAULT NULL  
  cmaxlength     in   varchar2   DEFAULT NULL  
  cvalue         in   varchar2   DEFAULT NULL  
  cattributes    in   varchar2   DEFAULT NULL);  
  
htf.formText (cname, csize, cmaxlength, cvalue, cattributes) return  
varchar2;
```

Purpose

Generates the <INPUT> tag with TYPE="text", which creates a field for a single line of text.

Parameters

cname - the value for the NAME attribute

csize - the value for the SIZE attribute

cmaxlength - the value for the MAXLENGTH attribute

cvalue - the value for the VALUE attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

http.formTextarea, http.formTextarea2

Syntax

```
http.formTextarea (
  cname      in   varchar2
  nrows      in   integer
  ncolumns   in   integer
  calign     in   varchar2   DEFAULT NULL
  cattributes in   varchar2   DEFAULT NULL);

http.formTextarea (cname, nrows, ncolumns, calign, cattributes) return
varchar2;
```

```
http.formTextarea2 (
  cname      in   varchar2
  nrows      in   integer
  ncolumns   in   integer
  calign     in   varchar2   DEFAULT NULL
  cwrap      in   varchar2   DEFAULT NULL
  cattributes in   varchar2   DEFAULT NULL);

http.formTextarea2 (cname, nrows, ncolumns, calign, cwrap,
cattributes) return varchar2;
```

Purpose

Generates the <TEXTAREA> tag, which creates a text field that has no predefined text in the text area. This field is used to enable the user to enter several lines of text.

The difference between these subprograms is that `http.formTextarea2` has the *cwrap* parameter, which specifies a wrap style.

Parameters

cname - the value for the NAME attribute

nrows - the value for the ROWS attribute. This is an integer.

ncolumns - the value for the COLS attribute. This is an integer.

calign - the value for the ALIGN attribute

cwrap - the value for the WRAP attribute

cattributes - other attributes to be included as-is in the tag

Generates

`http.formTextarea` generates:

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

`http.formTextarea2` generates:

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

http.formTextareaOpen, http.formTextareaOpen2, http.formTextareaClose

Syntax

```
http.formTextareaOpen (  
    cname          in   varchar2  
    nrows          in   integer  
    ncolumns       in   integer  
    calign         in   varchar2   DEFAULT NULL  
    cattributes    in   varchar2   DEFAULT NULL);  
  
http.formTextareaOpen (cname, nrows, ncolumns, calign, cattributes)  
return varchar2;  
  
http.formTextareaOpen2(  
    cname          in   varchar2  
    nrows          in   integer  
    ncolumns       in   integer  
    calign         in   varchar2   DEFAULT NULL  
    cwrap          in   varchar2   DEFAULT NULL  
    cattributes    in   varchar2   DEFAULT NULL);  
  
http.formTextareaOpen2(cname, nrows, ncolumns, calign, cwrap,  
    cattributes) return varchar2;  
  
http.formTextareaClose;  
  
http.formTextareaClose return varchar2;
```

Purpose

Generates the <TEXTAREA> and </TEXTAREA> tags, which creates a text area form element.

The difference between the two open subprograms is that http.formTextareaOpen2 has the *cwrap* parameter, which specifies a wrap style.

Parameters

cname - the value for the NAME attribute

nrows - the value for the ROWS attribute. This is an integer.

ncolumns - the value for the COLS attribute. This is an integer.

calign - the value for the ALIGN attribute

cwrap - the value for the WRAP attribute

cattributes - other attributes to be included as-is in the tag

Generates

http.formTextareaOpen generates:

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

http.formTextareaOpen2 generates:

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

http.formTextareaClose generates:

```
</TEXTAREA>
```

htp.frame

Syntax

```
htp.frame(  
    csrc          in varchar2  
    cname         in varchar2  DEFAULT NULL  
    cmarginwidth  in varchar2  DEFAULT NULL  
    cmarginheight in varchar2  DEFAULT NULL  
    cscrolling    in varchar2  DEFAULT NULL  
    cnoresize     in varchar2  DEFAULT NULL  
    cattributes   in varchar2  DEFAULT NULL);  
  
htf.frame(csrc, cname, cmarginwidth, cmarginheight, cscrolling,  
cnoresize, cattributes) return varchar2;
```

Purpose

Generates the <FRAME> tag, which defines the characteristics of a frame created by a <FRAMESET> tag.

Parameters

csrc - the URL to display in the frame

cname - the value for the NAME attribute

cmarginwidth - the value for the MARGINWIDTH attribute

cmarginheight - the value for the MARGINHEIGHT attribute

cscrolling - the value for the SCROLLING attribute

noresize - if the value for this parameter is not NULL, the NORESIZE attribute is added to the tag

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.framesetOpen, htp.framesetClose

Syntax

```
htp.framesetOpen(  
  crows      in   varchar2  DEFAULT NULL  
  ccols      in   varchar2  DEFAULT NULL  
  cattributes in   varchar2  DEFAULT NULL);  
  
htf.framesetOpen(crows, ccols, cattributes) return varchar2;  
  
htp.framesetClose;  
htf.framesetClose return varchar2;
```

Purpose

Generates the <FRAMESET> and </FRAMESET> tags, which define a frameset section.

Parameters

crows - the value for the ROWS attribute

ccols - the value for the COLS attribute

cattributes - other attributes to be included as-is in the tag

Generates

htp.framesetOpen generates:

```
<FRAMESET ROWS="nrows" COLS="ccols">
```

htp.framesetClose generates:

```
</FRAMESET>
```

http.headOpen, http.headClose

Syntax

```
http.headOpen;  
htf.headOpen return varchar2;  
  
http.headClose;  
htf.headClose return varchar2;
```

Purpose

Generates the <HEAD> and </HEAD> tags, which mark the HTML document head section.

Generates

http.headOpen generates:

```
<HEAD>
```

http.headClose generates:

```
</HEAD>
```

htp.header

Syntax

```
htp.header (  
    nsize      in    integer  
    cheader    in    varchar2  
    calign     in    varchar2    DEFAULT NULL  
    cnowrap    in    varchar2    DEFAULT NULL  
    cclear     in    varchar2    DEFAULT NULL  
    cattributes in    varchar2    DEFAULT NULL);  
  
htf.header (nsize, cheader, calign, cnowrap, cclear, cattributes)  
return varchar2;
```

Purpose

Generates opening heading tags (<H1> to <H6>) and their corresponding closing tags (</H1> to </H6>).

Parameters

nsize - the heading level. This is an integer between 1 and 6.

cheader - the text to display in the heading

calign - the value for the ALIGN attribute

cnowrap - the value for the NOWRAP attribute

cclear - the value for the CLEAR attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

Example

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

produces

```
<H1>Overview</H1>
```

http.htmlOpen, http.htmlClose

Syntax

```
http.htmlOpen;  
htf.htmlOpen return varchar2;  
  
http.htmlClose;  
htf.htmlClose return varchar2;
```

Purpose

Generates the <HTML> and </HTML> tags, which mark the beginning and the end of an HTML document.

Parameters

none

Generates

http.htmlOpen generates:

```
<HTML>
```

http.htmlClose generates:

```
</HTML>
```

htp.img, htp.img2

Syntax

```
htp.img (
    curl          in    varchar2    DEFAULT NULL
    calign        in    varchar2    DEFAULT NULL
    calt          in    varchar2    DEFAULT NULL
    cismap        in    varchar2    DEFAULT NULL
    cattributes   in    varchar2    DEFAULT NULL);

htf.img (curl, calign, calt, cismap, cattributes) return varchar2;

htp.img2(
    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   in    varchar2    DEFAULT NULL);

htf.img2(curl, calign, calt, cismap, cusemap, cattributes) return
varchar2;
```

Purpose

Generates the tag, which directs the browser to load an image onto the HTML page.

The difference between these subprograms is that `htp.img2` takes the *cusemap* parameter.

Parameters

curl - the value for the SRC attribute

calign - the value for the ALIGN attribute

calt - the value for the ALT attribute, which specifies alternative text to display if the browser does not support images

cismap - if the value for this parameter is not NULL, the ISMAP attribute is added to the tag. The attribute indicates that the image is an imagemap.

cusemap - the value for the USEMAP attribute, which specifies a client-side image map.

cattributes - other attributes to be included as-is in the tag

Generates

`htp.img` generates:

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

`htp.img2` generates:

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

htp.isindex

Syntax

```
htp.isindex (  
    cprompt      in   varchar2   DEFAULT NULL  
    curl         in   varchar2   DEFAULT NULL);  
  
htf.isindex (cprompt, curl) return varchar2;
```

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

cprompt - the value for the PROMPT attribute

curl - the value for the HREF attribute

Generates

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

htp.italic

Syntax

```
htp.italic (  
    ctext      in   varchar2  
    cattributes in   varchar2   DEFAULT NULL);  
htf.italic (ctext, cattributes) return varchar2;
```

Purpose

Generates the <I> and </I> tags, which direct the browser to render the text in italics.

Parameters

ctext - the text to be rendered in italics

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.keyboard, htp.kbd

Syntax

```
htp.keyboard (  
    ctext          in    varchar2  
    cattributes    in    varchar2    DEFAULT NULL  
htf.keyboard (ctext, cattributes) return varchar2;  
htp.kbd (  
    ctext          in    varchar2  
    cattributes    in    varchar2    DEFAULT NULL  
htf.kbd (ctext, cattributes) return varchar2;
```

Purpose

Generates the <KBD> and </KBD> tags, which direct the browser to render the text in monospace. These subprograms do the same thing.

Parameters

ctext - the text to render in monospace

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.line, htp.hr

Syntax

```
htp.line (
    cclear      in   varchar2   DEFAULT NULL
    csrc        in   varchar2   DEFAULT NULL
    cattributes in   varchar2   DEFAULT NULL);
htf.line (cclear, csrc, cattributes) return varchar2;

htp.hr (
    cclear      in   varchar2   DEFAULT NULL
    csrc        in   varchar2   DEFAULT NULL
    cattributes in   varchar2   DEFAULT NULL);
htf.hr (cclear, csrc, cattributes) return varchar2;
```

Purpose

Generates the
 tag, which generates a line in the HTML document.

Parameters

cclear - the value for the CLEAR attribute

csrc - the value for the SRC attribute, which specifies a custom image as the source of the line

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.linkRel

Syntax

```
htp.linkRel (  
    crel    in    varchar2  
    curl    in    varchar2  
    ctitle  in    varchar2    DEFAULT NULL);  
  
htf.linkRel (crel, curl, ctitle) return varchar2;
```

Purpose

Generates the <LINK> tag with the REL attribute, which 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 create a link, use [htp.anchor](#), [htp.anchor2](#).

Parameters

crel - the value for the REL attribute

curl - the value for the HREF attribute

ctitle - the value for the TITLE attribute

Generates

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

htp.linkRev

Syntax

```
htp.linkRev (  
    crev    in    varchar2  
    curl    in    varchar2  
    ctitle  in    varchar2    DEFAULT NULL);  
  
htf.linkRev (crev, curl, ctitle) return varchar2;
```

Purpose

Generates the <LINK> tag with the REV attribute, which 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 create a link, use [htp.anchor](#), [htp.anchor2](#).

Parameters

crev - the value for the REV attribute

curl - the value for the HREF attribute

ctitle - the value for the TITLE attribute

Generates

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

htp.listHeader

Syntax

```
htp.listHeader (
    ctext          in    varchar2
    cattributes    in    varchar2    DEFAULT NULL);
htf.listHeader (ctext, cattributes) return varchar2;
```

Purpose

Generates the <LH> and </LH> tags, which print an HTML tag at the beginning of the list.

Parameters

ctext - the text to place between <LH> and </LH>

cattributes - other attributes to be included as-is in the tag

Generates

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

http.listingOpen, http.listingClose

Syntax

```
http.listingOpen;  
http.listingOpen return varchar2;  
  
http.listingClose;  
http.listingClose return varchar2;
```

Purpose

Generates the <LISTING> and </LISTING> tags, which mark a section of fixed-width text in the body of an HTML page.

Parameters

none

Generates

http.listingOpen generates:

```
<LISTING>
```

http.listingClose generates:

```
</LISTING>
```

htp.listItem

Syntax

```
htp.listItem (  
    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);  
  
htf.listItem (ctext, cclear, cdingbat, csrc, cattributes) return  
varchar2;
```

Purpose

Generates the tag, which indicates a list item.

Parameters

ctext - the text for the list item

cclear - the value for the CLEAR attribute

cdingbat - the value for the DINGBAT attribute

csrc - the value for the SRC attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.mailto

Syntax

```
htp.mailto (  
    caddress      in   varchar2  
    ctext         in   varchar2  
    cname         in   varchar2  
    cattributes  in   varchar2    DEFAULT NULL);  
  
htf.mailto (caddress, ctext, cname, cattributes) return varchar2;
```

Purpose

Generates the <A> tag with the HREF set to 'mailto' prepended to the mail address argument.

Parameters

caddress - the email address of the recipient
ctext - the clickable portion of the link
cname - the value for the NAME attribute
cattributes - other attributes to be included as-is in the tag

Generates

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

Example

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

generates

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

http.mapOpen, http.mapClose

Syntax

```
http.mapOpen(  
    cname          in   varchar2  
    cattributes    in   varchar2    DEFAULT NULL);  
http.mapOpen(cname, cattributes) return varchar2;  
  
http.mapClose;  
http.mapClose return varchar2;
```

Purpose

Generates the <MAP> and </MAP> tags, which mark a set of regions in a client-side image map.

Parameters

cname - the value for the NAME attribute

cattributes - other attributes to be included as-is in the tag

Generates

http.mapOpen generates:

```
<MAP NAME="cname" cattributes>
```

http.mapClose generates:

```
</MAP>
```

http.menuListOpen, http.menuListClose

Syntax

```
http.menuListOpen;  
http.menuListOpen return varchar2;  
  
http.menuListClose;  
http.menuListClose return varchar2;
```

Purpose

Generates the <MENU> and </MENU> tags, which create a list that presents one line per item. The items in the list appear more compact than an unordered list. The [http.listItem](#) defines the list items in a menu list.

Parameters

none

Generates

http.menuListOpen generates:

```
<MENU>
```

http.menuListClose generates:

```
</MENU>
```

htp.meta

Syntax

```
htp.meta (  
  chttp_equiv in varchar2  
  cname      in varchar2  
  ccontent   in varchar2);  
  
htf.meta (chttp_equiv, cname, ccontent) return varchar2;
```

Purpose

Generates the <META> tag, which enables you to embed meta-information about the document and also specify values for HTTP headers. For example, you can specify the expiration date, keywords, and author name.

Parameters

chttp_equiv - the value for the HTTP-EQUIV attribute

cname - the value for the NAME attribute

ccontent - the value for the CONTENT attribute

Generates

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

Example

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

generates

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

On some web browsers, this causes the current URL to be reloaded automatically every 120 seconds.

htp.nl, htp.br

Syntax

```
htp.nl (
    cclear      in   varchar2   DEFAULT NULL
    cattributes in   varchar2   DEFAULT NULL);
htf.nl (cclear, cattributes) return varchar2;
htp.br (
    cclear      in   varchar2   DEFAULT NULL
    cattributes in   varchar2   DEFAULT NULL);
htf.br (cclear, cattributes) return varchar2;
```

Purpose

Generates the
 tag, which begins a new line of text.

Parameters

cclear - the value for the CLEAR attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.nobr

Syntax

```
htp.nobr(ctext in varchar2);  
htf.nobr(ctext) return varchar2;
```

Purpose

Generates the <NOBR> and </NOBR> tags, which turn off line-breaking in a section of text.

Parameters

ctext - the text that is to be rendered on one line

Generates

```
<NOBR>ctext</NOBR>
```

http.noframesOpen, http.noframesClose

Syntax

```
http.noframesOpen  
htf.noframesOpen return varchar2;  
  
http.noframesClose  
htf.noframesClose return varchar2;
```

Purpose

Generates the <NOFRAMES> and </NOFRAMES> tags, which mark a no-frames section.

Parameters

none

Generates

http.noframesOpen generates:

```
<NOFRAMES>
```

http.noframesClose generates:

```
</NOFRAMES>
```

See Also

[http.frame](#), [http.framesetOpen](#), [http.framesetClose](#)

htp.olistOpen, htp.olistClose

Syntax

```
htp.olistOpen (  
  cclear      in   varchar2      DEFAULT NULL  
  cwrap       in   varchar2      DEFAULT NULL  
  cattributes in   varchar2      DEFAULT NULL);  
  
htf.olistOpen (cclear, cwrap, cattributes) return varchar2;  
  
htp.olistClose;  
htf.olistClose return varchar2;
```

Purpose

Generates the and tags, which define an ordered list. An ordered list presents a list of numbered items. The numbered items are added using [htp.listItem](#).

Parameters

cclear - the value for the CLEAR attribute

cwrap - the value for the WRAP attribute

cattributes - other attributes to be included as-is in the tag

Generates

htp.olistOpen generates:

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

htp.olistClose generates:

```
</OL>
```

htp.para, htp.paragraph

Syntax

```
htp.para;  
htf.para return varcahr2;  
  
htp.paragraph (  
    calign          in    varchar2    DEFAULT NULL  
    cnowrap        in    varchar2    DEFAULT NULL  
    cclear         in    varchar2    DEFAULT NULL  
    cattributes    in    varchar2    DEFAULT NULL);  
htf.paragraph (calign, cnowrap, cclear, cattributes) return varchar2;
```

Purpose

Generates the <P> tag, which indicates that the text that comes after the tag is to be formatted as a paragraph.

htp.paragraph enables you add attributes to the tag.

Parameters

calign - the value for the ALIGN attribute

cnowrap - if the value for this parameter is not NULL, the NOWRAP attribute is added to the tag

cclear - the value for the CLEAR attribute

cattributes - other attributes to be included as-is in the tag

Generates

htp.para generates:

```
<P>
```

htp.paragraph generates:

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

http.param

Syntax

```
http.param(  
    cname          in   varchar2  
    cvalue         in   varchar2);  
  
htf.param(cname, cvalue) return varchar2;
```

Purpose

Generates the <PARAM> tag, which specifies parameters values for Java applets. The values can reference HTML variables.

To invoke a Java applet from a web page, use `http.appletopen` to begin the invocation, use one [http.param](#) for each desired name-value pair, and use `http.appletclose` to end the applet invocation.

Parameters

`cname` - the value for the NAME attribute

`cvalue` - the value for the VALUE attribute

Generates

```
<PARAM NAME=cname VALUE=cvalue>
```

htp.plaintext

Syntax

```
htp.plaintext(  
    ctext      in      varchar2  
    cattributes in      varchar2  DEFAULT NULL);  
htf.plaintext(ctext, cattributes) return varchar2;
```

Purpose

Generates the <PLAINTEXT> and </PLAINTEXT> tags, which direct the browser to render the text they surround in fixed-width type.

Parameters

ctext - the text to be rendered in fixed-width font

cattributes - other attributes to be included as-is in the tag

Generates

```
<PLAINTEXT cattributes>ctext</PLAINTEXT>
```

http.preOpen, http.preClose

Syntax

```
http.preOpen (
  cclear      in   varchar2   DEFAULT NULL
  cwidth      in   varchar2   DEFAULT NULL
  cattributes in   varchar2   DEFAULT NULL);

htf.preOpen (cclear, cwidth, cattributes) return varchar2;

http.preClose;

htf.preClose return varchar2;
```

Purpose

Generates the <PRE> and </PRE> tags, which mark a section of preformatted text in the body of the HTML page.

Parameters

cclear - the value for the CLEAR attribute

cwidth - the value for the WIDTH attribute

cattributes - other attributes to be included as-is in the tag

Generates

http.preOpen generates:

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

http.preClose generates:

```
</PRE>
```

htp.print, htp.prn

Syntax

```
htp.print (cbuf in varchar2 | dbuf in date | nbuf in number );
```

```
htp.prn(cbuf in varchar2 | dbuf in date | nbuf in number );
```

Purpose

htp.print generates the specified parameter as a string terminated with the \n newline character.

Note that the \n character is not the same as
. \n is used to format the HTML source; it does not affect how the browser renders the HTML source. Use
 to control how the browser renders the HTML source.

htp.prn generates the specified parameter as a string. Unlike htp.print, the string is not terminated with the \n newline character.

These subprograms are procedures only, they do not come as functions.

Parameters

cbuf, dbuf, nbuf - the string to generate

Generates

htp.print generates a string terminated with a newline.

htp.prn generates the specified string, not terminated with a newline.

htp.prints, htp.ps

Syntax

```
htp.prints (ctext in varchar2);  
htp.ps (ctext in varchar2);
```

Purpose

Both these subprograms generate a string and replaces all occurrences of the following characters with the corresponding escape sequence.

Replaces this character	with this escape sequence
<	<
>	>
"	"
&	&

If not replaced, the special characters would be interpreted as HTML control characters and would produce garbled output. This procedure is the same as `htp.prn` but with the character substitution.

These subprograms are procedures only, they are not available as functions. If you need a string conversion function, use [htf.escape_sc](#).

Parameters

`ctext` - the string in which to perform character substitution

Generates

A string.

htp.s

Syntax

```
htp.s(  
    ctext          in   varchar2  
    cattributes   in   varchar2   DEFAULT NULL);  
htf.s(ctext, cattributes) return varchar2;
```

Purpose

Generates the <S> and </S> tags, which direct the browser to render the text they surround in strikethrough type.

Parameters

ctext - the text to render in strikethrough type

cattributes - other attributes to be included as-is in the tag

Generates

```
<S cattributes>ctext</S>
```

htp.sample

Syntax

```
htp.sample (  
    ctext      in      varchar2  
    cattributes in      varchar2  DEFAULT NULL);  
htf.sample (ctext, cattributes) return varchar2;
```

Purpose

Generates the <SAMP> and </SAMP> tags, which direct the browser to render the text they surround in monospace font.

Parameters

ctext - the text to render in monospace font

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.script

Syntax

```
htp.script(  
    cscript          in   varchar2  
    clanguage       in   varchar2   DEFAULT NULL);  
htf.script(cscript, clanguage) return varchar2;
```

Purpose

Generates the <SCRIPT> and </SCRIPT> tags, which contain a script written in languages such as JavaScript and VBscript.

Parameters

cscript - the text of the script. This is the text that makes up the script itself, not the name of a file containing the script.

clanguage - the language in which the script is written. If this parameter is omitted, the user's browser determines the scripting language.

Generates

```
<SCRIPT LANGUAGE=clanguage>cscript</SCRIPT>
```

Example

```
htp.script ('Erupting_Volcano', 'Javascript');
```

Generates:

```
<SCRIPT LANGUAGE=Javascript>"script text here"  
</SCRIPT>
```

This would cause the browser to run the script enclosed in the tags.

htp.small

Syntax

```
htp.small(  
    ctext          in   varchar2  
    cattributes   in   varchar2   DEFAULT NULL);  
htf.small(ctext, cattributes) return varchar2;
```

Purpose

Generates the <SMALL> and </SMALL> tags, which direct the browser to render the text they surround using a small font.

Parameters

ctext - the text to render in a small font

cattributes - other attributes to be included as-is in the tag

Generates

```
<SMALL cattributes>ctext</SMALL>
```

htp.strike

Syntax

```
htp.strike(  
    ctext          in   varchar2  
    cattributes   in   varchar2   DEFAULT NULL);  
htf.strike(ctext, cattributes) return varchar2;
```

Purpose

Generates the <STRIKE> and </STRIKE> tags, which direct the browser to render the text they surround in strikethrough type.

Parameters

ctext - the text to be rendered in strikethrough type

cattributes - other attributes to be included as-is in the tag

Generates

```
<STRIKE cattributes>ctext</STRIKE>
```

htp.strong

Syntax

```
htp.strong (  
    ctext          in    varchar2  
    cattributes    in    varchar2    DEFAULT NULL);  
htf.strong (ctext, cattributes) return varchar2;
```

Purpose

Generates the and tags, which direct the browser to render the text they surround in bold.

Parameters

ctext - the text to be emphasized

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.style

Syntax

```
htp.style(cstyle in varchar2);  
htf.style(cstyle) return varchar2;
```

Purpose

Generates the <STYLE> and </STYLE> tags, which include a style sheet in your web page. Style sheets are a feature of HTML 3.2. You can get more information about style sheets at <http://www.w3.org>.

This feature is generally not compatible with browsers that support only HTML versions 2.0 or earlier. Such browsers will ignore this tag.

Parameters

cstyle - the style information to include

Generates

```
<STYLE>cstyle</STYLE>
```

htp.sub

Syntax

```
htp.sub(  
    ctext          in   varchar2  
    calign         in   varchar2   DEFAULT NULL  
    cattributes   in   varchar2   DEFAULT NULL);  
  
htf.sub(ctext, calign, cattributes) return varchar2;
```

Purpose

Generates the _{and} tags, which direct the browser to render the text they surround as subscript.

Parameters

ctext - the text to render in subscript

calign - the value for the ALIGN attribute

cattributes - other attributes to be included as-is in the tag

Generates

```
<SUB ALIGN="calign" cattributes>ctext</SUB>
```

htp.sup

Syntax

```
htp.sup(  
    ctext          in   varchar2  
    calign         in   varchar2   DEFAULT NULL  
    cattributes   in   varchar2   DEFAULT NULL);  
  
htf.sup(ctext, calign, cattributes) return varchar2;
```

Purpose

Generates the ^{and} tags, which direct the browser to render the text they surround as superscript.

Parameters

ctext - the text to render in subscript

calign - the value for the ALIGN attribute

cattributes - other attributes to be included as-is in the tag

Generates

```
<SUP ALIGN="calign" cattributes>ctext</SUP>
```

htp.tableCaption

Syntax

```
htp.tableCaption (
    ccaption      in   varchar2
    calign        in   varchar2      DEFAULT NULL
    cattributes   in   varchar2      DEFAULT NULL);

htf.tableCaption (ccaption, calign, cattributes) return varchar2;
```

Purpose

Generates the <CAPTION> and </CAPTION> tags, which place a caption in an HTML table.

Parameters

ccaption - the text for the caption

calign - the value for the ALIGN attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.tableData

Syntax

```
htp.tableData (  
    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);  
  
htf.tableData (cvalue, calign, cdp, cnowrap, crowspan, ccolspan,  
    cattributes) return varchar2;
```

Purpose

Generates the <TD> and </TD> tags, which insert data into a cell of an HTML table.

Parameters

cvalue - the data for the cell in the table

calign - the value for the ALIGN attribute

cdp - the value for the DP attribute

cnowrap - if the value of this parameter is not NULL, the NOWRAP attribute is added to the tag

crowspan - the value for the ROWSPAN attribute

ccolspan - the value for the COLSPAN attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.tableHeader

Syntax

```
htp.tableHeader (
  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);

htf.tableHeader (cvalue, calign, cdp, cnowrap, crowspan, ccolspan,
cattributes) return varchar2;
```

Purpose

Generates the <TH> and </TH> tags, which insert a header cell in an HTML table. <TH>s are similar to <TD>s, except that the text in the rows are usually rendered in bold type.

Parameters

cvalue - the data for the cell in the table

calign - the value for the ALIGN attribute

cdp - the value for the DP attribute

cnowrap - if the value of this parameter is not NULL, the NOWRAP attribute is added to the tag

crowspan - the value for the ROWSPAN attribute

ccolspan - the value for the COLSPAN attribute

cattributes - other attributes to be included as-is in the tag

Generates

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

http.tableOpen, http.tableClose

Syntax

```
http.tableOpen (  
  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);  
  
http.tableOpen (cborder, calign, cnowrap, cclear, cattributes) return  
varchar2;  
  
http.tableClose;  
http.tableClose return varchar2;
```

Purpose

Generates the <TABLE> and </TABLE> tags, which define an HTML table.

Parameters

cborder - the value for the BORDER attribute

calign - the value for the ALIGN attribute

cnowrap - if the value of this parameter is not NULL, the NOWRAP attribute is added to the tag

cclear - the value for the CLEAR attribute

cattributes - other attributes to be included as-is in the tag

Generates

http.tableOpen generates:

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

http.tableClose generates:

```
</TABLE>
```

http.tableRowOpen, http.tableRowClose

Syntax

```
http.tableRowOpen (
  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);

http.tableRowOpen (calign, cvalign, cdp, cnowrap, cattributes) return
varchar2;

http.tableRowClose;

http.tableRowClose return varchar2;
```

Purpose

Generates the <TR> and </TR> tags, which inserts a new row in an HTML table.

Parameters

calign - the value for the ALIGN attribute

cvalign - the value for the VALIGN attribute

cdp - the value for the DP attribute

cnowrap - if the value of this parameter is not NULL, the NOWRAP attribute is added to the tag

cattributes - other attributes to be included as-is in the tag

Generates

http.tableRowOpen generates:

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

http.tableRowClose generates:

```
</TR>
```

htp.teletype

Syntax

```
htp.teletype (  
    ctext      in      varchar2  
    cattributes in      varchar2  DEFAULT NULL);  
htf.teletype (ctext, cattributes) return varchar2;
```

Purpose

Generates the <TT> and </TT> tags, which direct the browser to render the text they surround in a fixed width typewriter font, for example, the Courier font.

Parameters

ctext - the text to render in a fixed width typewriter font

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.title

Syntax

```
htp.title (ctitle in varchar2);  
htf.title (ctitle) return varchar2;
```

Purpose

Generates the <TITLE> and </TITLE> tags, which specify the text to display in the titlebar of the browser window

Parameters

ctitle - the text to display in the titlebar of the browser window

Generates

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

http.ulistOpen, http.ulistClose

Syntax

```
http.ulistOpen (  
  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  
  
http.ulistOpen (cclear, cwrap, cdingbat, csrc, cattributes) return  
varchar2;  
  
http.ulistClose;  
http.ulistClose return varchar2;
```

Purpose

Generates the and tags, which define an unordered list. An unordered list presents listed items marked off by bullets. You add list items with [http.listItem](#).

Parameters

cclear - the value for the CLEAR attribute

cwrap - the value for the WRAP attribute

cdingbat - the value for the DINGBAT attribute

csrc - the value for the SRC attribute

cattributes - other attributes to be included as-is in the tag

Generates

http.ulistOpen generates:

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

http.ulistClose generates:

```
</UL>
```

htp.underline

Syntax

```
htp.underline(  
    ctext          in    varchar2  
    cattributes   in    varchar2    DEFAULT NULL);  
htf.underline(ctext, cattributes) return varchar2;
```

Purpose

Generates the <U> and </U> tags, which direct the browser to render the text they surround with an underline.

Parameters

ctext - the text to render with an underline

cattributes - other attributes to be included as-is in the tag

Generates

```
<U cattributes>ctext</U>
```

htp.variable

Syntax

```
htp.variable (
    ctext      in      varchar2
    cattributes in      varchar2  DEFAULT NULL);
htf.variable (ctext, cattributes) return varchar2;
```

Purpose

Generates the <VAR> and </VAR> tags, which direct the browser to render the text they surround in italics.

Parameters

ctext - the text to render in italics

cattributes - other attributes to be included as-is in the tag

Generates

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

htp.wbr

Syntax

```
htp.wbr ;  
htf.wbr return wbr ;
```

Purpose

Generates the <WBR> tag, which inserts a soft linebreak within a section of NOBR text.

Parameters

none

Generates

<WBR>

B

The owa_cookie Package

This chapter describes the functions, procedures, and data types in the **owa_cookie** package in the PL/SQL Web Toolkit.

Parameters that have default values are optional.

These sites have more information about cookies:

- http://home.netscape.com/newsref/std/cookie_spec.html
- <http://www.virtual.net/Projects/Cookies/>

Summary

[owa_cookie.cookie data type](#) - data type to contain cookie name-value pairs

[owa_cookie.get function](#) - gets the value of the specified cookie

[owa_cookie.get_all procedure](#) - gets all cookie name-value pairs

[owa_cookie.remove procedure](#) - removes the specified cookie

[owa_cookie.send procedure](#) - generates a "Set-Cookie" line in the HTTP header

owa_cookie.cookie data type

```
type cookie is RECORD (  
  name      varchar2(4096),  
  vals      vc_arr,  
  num_vals  integer);
```

Since the HTTP standard allow cookie names to 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.

vc_arr is defined as:

```
type vc_arr is table of varchar2(4096) index by binary_integer
```

Note: The largest cookie that can be handled with OCI7 is 2000 bytes. If you use OCI8, 4000 bytes can be handled.

owa_cookie.get function

Syntax

```
owa_cookie.get(name in varchar2) return cookie;
```

Purpose

This function returns the values associated with the specified cookie. The values are returned in a `owa_cookie.cookie` data type.

Parameters

name - the name of the cookie

Return Value

An [owa_cookie.cookie data type](#).

owa_cookie.get_all procedure

Syntax

```
owa_cookie.get_all(  
    names      out   vc_arr,  
    vals       out   vc_arr,  
    num_vals   out   integer);
```

Purpose

This procedure returns all cookie names and their values from the client's browser. The values appear in the order in which they were sent from the browser.

Parameters

names - the names of the cookies

vals - the values of the cookies

num_vals - the number of cookie-value pairs

owa_cookie.remove procedure

Syntax

```
owa_cookie.remove(  
    name in varchar2,  
    val  in varchar2,  
    path in varchar2 DEFAULT NULL);
```

Purpose

This procedure forces a cookie to expire immediately by setting the “expires” field of a Set-Cookie line in the HTTP header to “01-Jan-1990”. This procedure must be called within the context of an HTTP header.

Parameters

name - the name of the cookie to expire

value - the value of the cookie

path - currently unused

Generates

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

owa_cookie.send procedure

Syntax

```
owa_cookie.send(  
  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);
```

Purpose

This procedure generates a Set-Cookie line, which transmits a cookie to the client. This procedure must occur in the context of an HTTP header.

Parameters

name -the name of the cookie

value - the value of the cookie

expires - the date at which the cookie will expire

path - the value for the path field

domain - the value for the domain field

secure - if the value of this parameter is not NULL, the “secure” field is added to the line

Generates

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

C

The owa_image Package

This chapter describes the functions, procedures, and data types in the **owa_image** package in the PL/SQL Web Toolkit.

Parameters that have default values are optional.

Summary

[owa_image.NULL_POINT package variable](#) - variable of type point whose X and Y values are NULL

[owa_image.point data type](#) - data type to contain the X and Y values of a coordinate

[owa_image.get_x function](#) - gets the X value of a point type

[owa_image.get_y function](#) - gets the Y value of a point type

owa_image.NULL_POINT package variable

This package variable of type `point` is used to default point parameters. Both the X and the Y fields of this variable are NULL.

owa_image.point data type

This data type provides the x and y coordinates of a user's click on an imagemap. It is defined as:

```
type point is table of varchar2(32767) index by binary_integer
```

owa_image.get_x function

Syntax

```
owa_image.get_x(p in point) return integer;
```

Purpose

This function returns the X coordinate of the point where the user clicked on an image map.

Parameters

p - the point where the user clicked

Return Value

The X coordinate as an integer.

owa_image.get_y function

Syntax

```
owa_image.get_y(p in point) return integer;
```

Purpose

This function returns the Y coordinate of the point where the user clicked on an image map.

Parameters

p - the point where the user clicked

Return Value

The Y coordinate as an integer.

D

The owa_opt_lock Package

This chapter describes the functions, procedures, and data types in the **owa_opt_lock** package in the PL/SQL Web Toolkit.

Parameters that have default values are optional.

Summary

[owa_opt_lock.vcArray data type](#) - data type to contain ROWIDs

[owa_opt_lock.checksum function](#) - returns the checksum value

[owa_opt_lock.get_rowid function](#) - returns the ROWID value

[owa_opt_lock.store_values procedure](#) - stores unmodified values in hidden fields for later verification

[owa_opt_lock.verify_values function](#) - verifies the stored values against modified values

owa_opt_lock.vcArray data type

This data type is a PL/SQL table intended to hold ROWIDs.

```
type vcArray is table of varchar2(2000) index by binary_integer
```

Note: This is different from the [owa_text.vc_arr data type](#).

owa_opt_lock.checksum function

Syntax

```
owa_opt_lock.checksum(p_buff in varchar2) return number;  
  
owa_opt_lock.checksum(  
    p_owner      in   varchar2  
    p_tname      in   varchar2  
    p_rowid      in   rowid)  
return number;
```

Purpose

This function returns a checksum value for a specified string, or for a row in a table. For a row in a table, the function calculates the checksum value based on the values of the columns in the row. This function comes in two versions.

The first version returns a checksum based on the specified string. This is a “pure” 32-bit checksum executed by the database and based on the Internet 1 protocol.

The second version returns a checksum based on the values of a row in a table. This is a “impure” 32-bit checksum based on the Internet 1 protocol.

Parameters

p_buff - the string for which you want to calculate the checksum

p_owner - the owner of the table

p_tname - the table name

p_rowid - the row in *p_tname* for which you want to calculate the checksum value. You can use the [owa_opt_lock.get_rowid function](#) to convert vcArray values to proper rowids.

Return Value

A checksum value.

owa_opt_lock.get_rowid function

Syntax

```
owa_opt_lock.get_rowid(p_old_values in vcArray) return rowid;
```

Purpose

This function returns the ROWID data type from the specified [owa_opt_lock.vcArray data type](#).

Parameters

p_old_values - this parameter is usually passed in from an HTML form.

Return Value

A ROWID.

owa_opt_lock.store_values procedure

Syntax

```
owa_opt_lock.store_values(  
    p_owner      in   varchar2  
    p_tname      in   varchar2  
    p_rowid      in   rowid);
```

Purpose

This procedure stores the column values of the row that you want to update later. The values are stored in hidden HTML form elements.

Before you update the row, you compare these values with the current row values to ensure that the values in the row have not been changed. If the values have been changed, you can warn the users and let them decide if the update should still take place.

Parameters

p_owner - the owner of the table

p_tname - the name of the table

p_rowid - the row for which you want to store values

Generates

A series of hidden form elements.

- One hidden form element is created for the table owner. The name of the element is "old_p_tname", where *p_tname* is the name of the table. The value of the element is the owner name.
- One hidden form element is created for the table name. The name of the element is "old_p_tname", where *p_tname* is the name of the table. The value of the element is the table name.
- One element is created for each column in the row. The name of the element is "old_p_tname", where *p_tname* is the name of the table. The value of the element is the column value.

See Also

[owa_opt_lock.verify_values function](#)

owa_opt_lock.verify_values function

Syntax

```
owa_opt_lock.verify_values(p_old_values in vcArray) return boolean;
```

Purpose

This function verifies whether or not values in the specified row have been updated since the last query. This function is used with the [owa_opt_lock.store_values procedure](#).

Parameters

p_old_values - a PL/SQL table containing the following information:

- p_old_values(1) specifies the owner of the table
- p_old_values(2) specifies the table
- p_old_values(3) specifies the rowid of the row you want to verify
- The remaining indexes contain values for the columns in the table.

Typically, this parameter is passed in from the HTML form, where you have previously called the [owa_opt_lock.store_values procedure](#) to store the row values on hidden form elements.

Return Value

TRUE if no other update has been performed; FALSE otherwise.

See Also

[owa_opt_lock.store_values procedure](#)

E

The owa_pattern Package

This chapter describes the functions, procedures, and data types in the **owa_pattern** package in the PL/SQL Web Toolkit.

Parameters that have default values are optional.

Summary

[owa_pattern.amatch function](#) - determines if a string contains the specified pattern

[owa_pattern.change function and procedure](#) - replaces a pattern within a string

[owa_pattern.getpat procedure](#) - generates a pattern data type from a VARCHAR2 type

[owa_pattern.match function](#) - determines if a string contains the specified pattern

[owa_pattern.pattern data type](#) - data type used to store regular expressions

owa_pattern.amatch function

Syntax

```
owa_pattern.amatch(
  line      in      VARCHAR2
  from_loc  in      INTEGER
  pat       in      VARCHAR2
  flags     in      VARCHAR2 DEFAULT NULL) return INTEGER;

owa_pattern.amatch(
  line      in      VARCHAR2
  from_loc  in      INTEGER
  pat       in out  pattern
  flags     in      VARCHAR2 DEFAULT NULL) return INTEGER;

owa_pattern.amatch(
  line      in      VARCHAR2
  from_loc  in      INTEGER
  pat       in      VARCHAR2
  backrefs  out    owa_text.vc_arr
  flags     in      VARCHAR2          DEFAULT NULL) return INTEGER;

owa_pattern.amatch(
  line      in      VARCHAR2
  from_loc  in      INTEGER
  pat       in out  pattern
  backrefs  out    owa_text.vc_arr
  flags     in      VARCHAR2          DEFAULT NULL) return INTEGER;
```

Purpose

This function enables you to specify if a pattern occurs in a particular location in a string. There are four versions to this function:

- The first and second versions of the function do not save the matched tokens (these are saved in the *backrefs* parameters in the third and fourth versions). The difference between the first and second versions is the *pat* parameter, which can be a VARCHAR2 or a pattern data type.
- The third and fourth versions of the function save the matched tokens in the *backrefs* parameter. The difference between the third and fourth versions is the *pat* parameter, which can be a VARCHAR2 or a pattern data type.

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

Parameters

line - the text to search in

from_loc - the location (in number of characters) in *line* where the search is to begin

pat - the string to match. It can contain regular expressions. This can be either a VARCHAR2 or a pattern. If it is a pattern, the output value of this parameter is the pattern matched.

backrefs - the text that is matched. Each token that is matched is placed in a cell in the [owa_text.vc_arr data type](#) PL/SQL table.

flags - whether or not the search is case-sensitive. If the value of this parameter is "i", the search is case-insensitive. Otherwise the search is case-sensitive.

Return Value

The index of the character after the end of the match, counting from the beginning of *line*. If there was no match, the function returns 0.

owa_pattern.change function and procedure

Syntax

```
/* function */
owa_pattern.change(
  line      in out  VARCHAR2
  from_str  in      VARCHAR2
  to_str    in      VARCHAR2
  flags     in      VARCHAR2 DEFAULT NULL) return INTEGER;

/* procedure */
owa_pattern.change(
  line      in out  VARCHAR2
  from_str  in      VARCHAR2
  to_str    in      VARCHAR2
  flags     in      VARCHAR2 DEFAULT NULL);

/* function */
owa_pattern.change(
  mline     in out  owa_text.multi_line
  from_str  in      VARCHAR2
  to_str    in      VARCHAR2
  flags     in      VARCHAR2          DEFAULT NULL) return INTEGER;

/* procedure */
owa_pattern.change(
  mline     in out  owa_text.multi_line
  from_str  in      VARCHAR2
  to_str    in      VARCHAR2
  flags     in      VARCHAR2 DEFAULT NULL);
```

Purpose

This function or procedure performs a search and replace on a string or multi_line data type.

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

Parameters

line - the text to search in. The output value of this parameter is the altered string.

mline - the text to search in. This is a [owa_text.multi_line data type](#) data type. The output value of this parameter is the altered string.

from_str - the regular expression to replace

to_str - the substitution pattern

flags - whether or not the search is case-sensitive, and whether or not changes are to be made globally. If “i” is specified, the search is case-insensitive. If “g” is specified, changes are made to all matches. Otherwise, the function stops after the first substitution is made.

Return Value

The number of substitutions made.

Example

In the following example, *num_found* is 1, and *theline* is changed to “what is the idea?”.

```
create or replace procedure test_pattern as
```

```
theline VARCHAR2(256);
num_found integer;
begin
theline := 'what is the goal?';
num_found := owa_pattern.change(theline, 'goal', 'idea', 'g');
htp.print(num_found); -- num_found is 1
htp.print(theline); -- theline is 'what is the idea?'
end;
/
show errors
```

owa_pattern.getpat procedure

Syntax

```
owa_pattern.getpat(arg in VARCHAR2, pat in out pattern);
```

Purpose

This procedure converts a VARCHAR2 string into a [owa_pattern.pattern data type](#).

Parameters

arg - the string to convert

pat - the [owa_pattern.pattern data type](#) initialized with *arg*

owa_pattern.match function

Syntax

```
owa_pattern.match(
  line      in      VARCHAR2
  pat       in      VARCHAR2
  flags     in      VARCHAR2 DEFAULT NULL) return boolean;

owa_pattern.match(
  line      in      VARCHAR2
  pat       in out   pattern
  flags     in      VARCHAR2 DEFAULT NULL) return boolean;

owa_pattern.match(
  line      in      VARCHAR2
  pat       in      VARCHAR2
  backrefs  out     owa_text.vc_arr
  flags     in      VARCHAR2 DEFAULT NULL) return boolean;

owa_pattern.match(
  line      in      VARCHAR2
  pat       in out   pattern
  backrefs  out     owa_text.vc_arr
  flags     in      VARCHAR2 DEFAULT NULL) return boolean;

owa_pattern.match(
  mline     in      owa_text.multi_line
  pat       in      VARCHAR2
  rlist     out     owa_text.row_list
  flags     in      VARCHAR2 DEFAULT NULL) return boolean;

owa_pattern.match(
  mline     in      owa_text.multi_line
  pat       in out   pattern
  rlist     out     owa_text.row_list
  flags     in      VARCHAR2 DEFAULT NULL) return boolean;
```

Purpose

This function determines if a string contains the specified pattern. Pattern can contain regular expressions.

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

Parameters

line - the text to search in

mline - the text to search in. This is a [owa_text.multi_line data type](#) data type.

pat - the pattern to match. This is either a VARCHAR2 or a [owa_pattern.pattern data type](#) data type. If it is a pattern, the output value of this parameter is the pattern matched.

backrefs - the text that is matched. Each token that is matched is placed in a cell in the [owa_text.vc_arr data type](#) PL/SQL table.

rlist - an output parameter containing a list of matches

flags - whether or not the search is case-sensitive. If the value of this parameter is "i", the search is case-insensitive. Otherwise the search is case-sensitive.

Return Value

TRUE if a match was found, FALSE otherwise.

Examples

The following example searches for the string “goal” followed by any number of characters in *sometext*. If found,

```
sometext  VARCHAR2(256);
pat       VARCHAR2(256);

sometext := 'what is the goal?'
pat := 'goal.*';
if owa.pattern.match(sometext, pat) then
    http.print('Match found');
else
    http.print('Match not found');
end if;
```

owa_pattern.pattern data type

This data type is used to store regular expressions. It is defined as:

```
type pattern is TABLE OF VARCHAR2(4) index by BINARY_INTEGER
```

The advantage is that you can use a pattern as both an input and output parameter. Thus, you can pass the same regular expression to OWA_PATTERN function calls, and it only has to be parsed once.

F

The owa_sec Package

This chapter describes the functions, procedures, and data types in the **owa_sec** package in the PL/SQL Web Toolkit.

Parameters that have default values are optional.

Summary

[owa_sec.get_client_hostname function](#) - returns the client's hostname

[owa_sec.get_client_ip function](#) - returns the client's IP address

[owa_sec.get_password function](#) - returns the password that the user entered

[owa_sec.get_user_id function](#) - returns the username that the user entered

[owa_sec.set_authorization procedure](#) - enables the PL/SQL application to use custom authentication

[owa_sec.set_protection_realm procedure](#) - defines the realm that the page is in

owa_sec.get_client_hostname function

Syntax

```
owa_sec.get_client_hostname return varchar2;
```

Purpose

This function returns the hostname of the client.

Parameters

none

Return Value

The hostname.

owa_sec.get_client_ip function

Syntax

```
owa_sec.get_client_ip return owa_util.ip_address;
```

Purpose

This function returns the IP address of the client.

Parameters

none

Return Value

The IP address. The [owa_util.ip_address data type](#) is a PL/SQL table where the first four elements contain the four numbers of the IP address. For example, if the IP address is 123 . 45 . 67 . 89 and the variable *ipaddr* is of the *owa_util.ip_address* data type, the variable would contain the following values:

```
ipaddr(1) = 123  
ipaddr(2) = 45  
ipaddr(3) = 67  
ipaddr(4) = 89
```

owa_sec.get_password function

Syntax

```
owa_sec.get_password return varchar2;
```

Purpose

This function returns the password that the user used to log in.

Parameters

none

Return Value

The password.

owa_sec.get_user_id function

Syntax

```
owa_sec.get_user_id return varchar2;
```

Purpose

This function returns the username that the user used to log in.

Parameters

none

Return Value

The username.

owa_sec.set_authorization procedure

Syntax

```
owa_sec.set_authorization(scheme in integer);
```

Purpose

This procedure sets the authorization scheme for a PL/SQL Agent. Setting the scheme parameter to `GLOBAL` or `PER_PACKAGE` enables you to perform define your own authentication routine. This procedure is called in the initialization portion of the `owa_init` package.

Parameters

scheme - the authorization scheme. It is one of:

- `OWA_SEC.NO_CHECK`
Specifies that the PL/SQL application is not to do any custom authentication. This is the default.
- `OWA_SEC.GLOBAL`
Specifies that the `owa_init.authorize` function is to be used to authorize the user. You have to define this function in the `owa_init` package.
- `OWA_SEC.PER_PACKAGE`
Specifies that the `package.authorize` function or the anonymous `authorize` function is to be used to authorize the user. You have to define this function. If the request is for a procedure defined in a package, the `package.authorize` function is called. If the request is for a procedure that is not in a package, the anonymous `authorize` function is called.

The custom authorize function has the following signature:

```
function authorize return boolean;
```

If the function returns `TRUE`, authentication succeeded. If it returns `FALSE`, authentication failed.

If the authorize function is not defined and `scheme` is set to `GLOBAL` or `PER_PACKAGE`, the cartridge returns an error and fails.

owa_sec.set_protection_realm procedure

Syntax

```
owa_sec.set_protection_realm(realm in varchar2);
```

Purpose

This procedure sets the realm of the page that is returned to the user. The user needs to enter a username and login that already exists in the realm for authorization to succeed.

Parameters

realm - the realm in which the page should belong. This string is displayed to the user.

G

The owa_text Package

This chapter describes the functions, procedures, and data types in the **owa_text** package in the PL/SQL Web Toolkit.

Parameters that have default values are optional.

Summary

[owa_text.add2multi procedure](#) - adds text to an existing multi_line type

[owa_text.multi_line data type](#) - data type for holding large amounts of text

[owa_text.row_list data type](#) - data type for holding data to be processed

[owa_text.new_row_list](#) - creates a new row_list

[owa_text.print_multi procedure](#) - prints out the contents of a multi_list

[owa_text.print_row_list procedure](#) - prints out the contents of a row_list

[owa_text.stream2multi procedure](#) - converts a varchar2 to a multi_line type

[owa_text.vc_arr data type](#) - data type for holding large amounts of text

owa_text.add2multi procedure

Syntax

```
owa_text.add2multi(  
  stream      in      varchar2  
  mline      in out  multi_line  
  continue    in      boolean DEFAULT TRUE);
```

Purpose

This procedure adds content to an existing [owa_text.multi_line data type](#) data type.

Parameters

stream - the text to add

mline - the [owa_text.multi_line data type](#) data type. The output of this parameter contains *stream*.

continue - if TRUE, the procedure appends *stream* within the previous final row (assuming it is less than 32K). If FALSE, the procedure places *stream* in a new row.

owa_text.multi_line data type

This data type is a PL/SQL record that is used to hold large amounts of text. It is defined as:

```
type multi_line is record (  
    rows          vc_arr,  
    num_rows      integer,  
    partial_row    boolean);
```

The rows field, of type [owa_text.vc_arr data type](#), contains the text data in the record.

owa_text.new_row_list

Syntax

```
/* procedure */  
owa_text.new_row_list(rlist out row_list);  
  
/* function */  
owa_text.new_row_list return row_list;
```

Purpose

This function or procedure creates a new [owa_text.row_list data type](#).

The function version takes no parameters and returns a new empty row_list.

The procedure version creates the row_list data type as an output parameter.

Parameters

rlist - this is an output parameter containing the new row_list data type.

Return Value

The function version returns the new row_list data type.

owa_text.print_multi procedure

Syntax

```
owa_text.print_multi(mline in multi_line);
```

Purpose

This procedure uses [http.print.http.prn](#) to print the “rows” field of the [owa_text.multi_line data type](#).

Parameters

mline - the multi_line data type to print out

Generates

The contents of the multi_line.

owa_text.print_row_list procedure

Syntax

```
owa_text.print_row_list(rlist in row_list);
```

Purpose

This procedure uses [http.print.http.prn](#) to print the “rows” field of the [owa_text.row_list data type](#).

Parameters

rlist - the row_list data type to print out

Generates

The contents of the row_list.

owa_text.row_list data type

A PL/SQL record defined as:

```
type row_list is record (  
    rows      int_arr,  
    num_rows  integer);
```

int_arr is defined as:

```
type int_arr is table of integer index by binary_integer
```

owa_text.stream2multi procedure

Syntax

```
owa_text.stream2multi(  
    stream      in   varchar2  
    mline      out   multi_line);
```

Purpose

This procedure converts a string to a multi_line data type.

Parameters

stream - the string to convert

mline - the stream in [owa_text.multi_line data type](#) format

owa_text.vc_arr data type

This data type is defined as:

```
type vc_arr is table of varchar2(32767) index by binary_integer
```

This is a component of the [owa_text.multi_line data type](#).

H The owa_util Package

This chapter describes the functions, procedures, and data types in the **owa_util** package in the PL/SQL Web Toolkit.

Parameters that have default values are optional.

Summary

[owa_util.bind_variables function](#) - prepares a SQL query and binds variables to it

[owa_util.calendarprint procedure](#) - prints a calendar

[owa_util.cellsprint procedure](#) - prints the contents of a query in an HTML table

[owa_util.choose_date procedure](#) - generates HTML form elements that allow the user to select a date

[owa_util.dateType data type](#) - data type to hold date information

[owa_util.get_cgi_env function](#) - returns the value of the specified CGI environment variable

[owa_util.get_owa_service_path function](#) - returns the full virtual path for the cartridge

[owa_util.get_procedure function](#) - returns the name of the procedure that is invoked by the PL/SQL Agent

[owa_util.http_header_close procedure](#) - closes the HTTP header

[owa_util.ident_arr data type](#)

[owa_util.ip_address data type](#)

[owa_util.listprint procedure](#) - generates a HTML form element that contains data from a query.

[owa_util.mime_header procedure](#) - generates the Content-type line in the HTTP header

[owa_util.print_cgi_env procedure](#) - generates a list of all CGI environment variables and their values

[owa_util.redirect_url procedure](#) - generates the Location line in the HTTP header

[owa_util.showpage procedure](#) - prints a page generated by the htp and htf packages in SQL*Plus

[owa_util.showsource procedure](#) - prints the source for the specified subprogram

[owa_util.signature procedure](#) - prints a line that says that the page is generated by the PL/SQL Agent

[owa_util.status_line procedure](#) - generates the Status line in the HTTP header

[owa_util.tablePrint function](#) - prints the data from a table in the database as an HTML table

[owa_util.todate function](#) - converts dateType data to the standard PL/SQL date type

[owa_util.who_called_me procedure](#) - returns information on the caller of the procedure

owa_util.bind_variables function

Syntax

```
owa_util.bind_variables(  
    theQuery      in   varchar2 DEFAULT NULL  
    bv1Name       in   varchar2 DEFAULT NULL  
    bv1Value      in   varchar2 DEFAULT NULL  
    bv2Name       in   varchar2 DEFAULT NULL  
    bv2Value      in   varchar2 DEFAULT NULL  
    bv3Name       in   varchar2 DEFAULT NULL  
    bv3Value      in   varchar2 DEFAULT NULL  
    ...  
    bv25Name      in   varchar2 DEFAULT NULL  
    bv25Value     in   varchar2 DEFAULT NULL) return integer;
```

Purpose

This function prepares a SQL query by binding variables to it, and stores the output in an opened cursor. You normally use this function as a parameter to a procedure to which you desire to send a dynamically generated query. You can specify up to 25 bind variables.

Parameters

theQuery - the SQL query statement. This must be a SELECT statement.

bv1Name - the name of the variable

bv2Value - the value of the variable

Return Value

An integer identifying the opened cursor.

owa_util.calendarprint procedure

Syntax

```
owa_util.calendarprint(  
    p_query      in   varchar2  
    p_mf_only    in   varchar2 DEFAULT 'N');  
  
owa_util.calendarprint(  
    p_cursor     in   integer  
    p_mf_only    in   varchar2 DEFAULT 'N');
```

Purpose

This procedure creates a calendar in HTML. Each date in the calendar can contain any number of hypertext links. To achieve this effect, design your query as follows:

- The first column should be a DATE. This is used to correlate the information produced by the query with the calendar output automatically generated by the procedure. **Note:** the query output must be sorted on this column using ORDER BY.
- The second column contains the text, if any, you want printed for that date.
- The third column contains the destination for automatically generated links. Each item in the second column becomes a hypertext link to the destination given in this column. If this column is omitted, the items in the second column are simple text, not links.

This procedure has 2 versions. Version 1 uses a hard-coded query stored in a varchar2 string. Version 2 uses a dynamic query prepared with the [owa_util.bind_variables function](#).

Parameters

p_query - a PL/SQL query. See the description above on what the query should return.

p_cursor - a PL/SQL cursor containing the same format as *p_query*.

p_mf_only - if "N" (the default), the generated calendar includes Sunday through Saturday. Otherwise, it includes Monday through Friday only.

Generates

A calendar in the form of an HTML table with a visible border.

owa_util.cellsprint procedure

Syntax

```
owa_util.cellsprint(  
  p_theQuery      in   varchar2  
  p_max_rows      in   number      DEFAULT 100  
  p_format_numbers in   varchar2  DEFAULT NULL);  
  
owa_util.cellsprint(  
  p_theCursor     in   integer  
  p_max_rows      in   number      DEFAULT 100  
  p_format_numbers in   varchar2  DEFAULT NULL);  
  
owa_util.cellsprint(  
  p_theQuery      in   varchar2  
  p_max_rows      in   number      DEFAULT 100  
  p_format_numbers in   varchar2  DEFAULT NULL  
  p_skip_rec      in   number default 0  
  p_more_data     out   boolean);  
  
owa_util.cellsprint(  
  p_theCursor     in   integer  
  p_max_rows      in   number      DEFAULT 100  
  p_format_numbers in   varchar2  DEFAULT NULL  
  p_skip_rec      in   number default 0  
  p_more_data     out   boolean);
```

Purpose

This procedure generates an HTML table from the output of a SQL query. SQL atomic data items are mapped to HTML cells and SQL rows to HTML rows. You must write the code to begin and end the HTML table.

There are four versions to this procedure. The first and second versions display rows (up to the specified maximum) returned by the query or cursor. The third and fourth versions allow you to exclude the specified number of rows from the HTML table. You can also use the third and fourth versions to scroll through result sets by saving the last row seen in a hidden form element.

Parameters

p_theQuery - a SQL SELECT statement.

p_theCursor - a cursor ID. This can be the return value from the [owa_util.bind_variables function](#).

p_max_rows - the maximum number of rows to print

p_format_numbers - if the value of this parameter is not NULL, number fields are right-justified and rounded to two decimal places

p_skip_rec - the number of rows to exclude from the HTML table

p_more_data - TRUE if there are more rows in the query or cursor, FALSE otherwise.

Generates

```
<tr><td>QueryResultItem</td><td>QueryResultItem</td></tr>  
<tr><td>QueryResultItem</td><td>QueryResultItem</td></tr>
```

owa_util.choose_date procedure

Syntax

```
owa_util.choose_date(  
    p_name in varchar2,  
    p_date in date DEFAULT SYSDATE);
```

Purpose

This procedure generates three HTML form elements that allow the user to select the day, the month, and the year.

The parameter in the procedure that receives the data from these elements should be a [owa_util.dateType data type](#). You can use the [owa_util.todate function](#) to convert the [owa_util.dateType data type](#) value to the standard Oracle7 DATE data type.

Parameters

p_name - the name of the form elements

p_date - the initial date that is selected when the HTML page is displayed

Generates

```
<SELECT NAME="" SIZE="1">  
<OPTION value="01">1  
<OPTION value="02">2  
<OPTION value="03">3  
<OPTION value="04">4  
<OPTION value="05">5  
<OPTION value="06">6  
<OPTION value="07">7  
<OPTION value="08">8  
<OPTION value="09">9  
<OPTION value="10">10  
<OPTION value="11">11  
<OPTION value="12">12  
<OPTION value="13">13  
<OPTION value="14">14  
<OPTION value="15">15  
<OPTION value="16">16  
<OPTION value="17">17  
<OPTION value="18">18  
<OPTION value="19">19  
<OPTION value="20">20  
<OPTION value="21">21  
<OPTION value="22">22  
<OPTION value="23">23  
<OPTION SELECTED value="24">24  
<OPTION value="25">25
```

```
<OPTION value="26">26
<OPTION value="27">27
<OPTION value="28">28
<OPTION value="29">29
<OPTION value="30">30
<OPTION value="31">31
</SELECT>
-
<SELECT NAME="p_name" SIZE="1">
<OPTION value="01">JAN
<OPTION SELECTED value="02">FEB
<OPTION value="03">MAR
<OPTION value="04">APR
<OPTION value="05">MAY
<OPTION value="06">JUN
<OPTION value="07">JUL
<OPTION value="08">AUG
<OPTION value="09">SEP
<OPTION value="10">OCT
<OPTION value="11">NOV
<OPTION value="12">DEC
</SELECT>
-
<SELECT NAME="p_name" SIZE="1">
<OPTION value="1992">1992
<OPTION value="1993">1993
<OPTION value="1994">1994
<OPTION value="1995">1995
<OPTION value="1996">1996
<OPTION SELECTED value="1997">1997
<OPTION value="1998">1998
<OPTION value="1999">1999
<OPTION value="2000">2000
<OPTION value="2001">2001
<OPTION value="2002">2002
</SELECT>
```

owa_util.dateType data type

This data type holds date information. It is defined as:

```
type dateType is table of varchar2(10) index by binary_integer
```

The [owa_util.todate function](#) converts an item of this type to the type DATE, which is understood and properly handled as data by the database. The procedure [owa_util.choose_date procedure](#) enables the user to select the desired date.

owa_util.get_cgi_env function

Syntax

```
owa_util.get_cgi_env(param_name in varchar2) return varchar2;
```

Purpose

This function returns the value of the specified CGI environment variable. Although the WRB is not operated through CGI, many WRB cartridges, including the PL/SQL Cartridge, can make use of CGI environment variables.

Parameters

param_name - the name of the CGI environment variable. It is case-insensitive.

Return Value

The value of the specified CGI environment variable. If the variable is not defined, the function returns NULL.

owa_util.get_owa_service_path function

Syntax

```
owa_util.get_owa_service_path return varchar2;
```

Purpose

This function returns the full virtual path of the PL/SQL Cartridge that is handling the request.

Parameters

none

Return Value

A virtual path of the PL/SQL Cartridge that is handling the request.

owa_util.get_procedure function

Syntax

```
owa_util.get_procedure return varchar2;
```

Purpose

This function returns the name of the procedure that is being invoked by the PL/SQL Agent.

Parameters

none

Return Value

The name of a procedure, including the package name if the procedure is defined in a package.

owa_util.http_header_close procedure

Syntax

```
owa_util.http_header_close;
```

Purpose

This procedure generates a newline character to close the HTTP header.

Use this procedure if you have not explicitly closed the header by using the *bclose_header* parameter in calls such as [owa_util.mime_header_procedure](#), [owa_util.redirect_url_procedure](#), or [owa_util.status_line_procedure](#). The HTTP header must be closed before any `http.print` or `http.prn` calls.

Parameters

none

Generates

A newline character, which closes the HTTP header.

owa_util.ident_arr data type

This data type is defined as:

```
type ident_arr is table of varchar2(30) index by binary_integer
```

owa_util.ip_address data type

This data type is defined as:

```
type ip_address is table of integer index by binary_integer
```

This data type is used by the [owa_sec.get_client_ip function](#).

owa_util.listprint procedure

Syntax

```
owa_util.listprint(  
    p_theQuery    in    varchar2  
    p_cname       in    varchar2  
    p_nsize       in    number  
    p_multiple    in    boolean    DEFAULT FALSE);  
  
owa_util.listprint(  
    p_theCursor   in    integer  
    p_cname       in    varchar2  
    p_nsize       in    number  
    p_multiple    in    boolean    DEFAULT FALSE);
```

Purpose

This procedure generates an HTML selection list form element from the output of a SQL query. The columns in the output of the query are handled in the following manner:

- The first column specifies the values that are sent back. These values are used for the VALUE attribute of the OPTION tag.
- The second column specifies the values that the user sees.
- The third column specifies whether or not the row is marked as SELECTED in the OPTION tag. If the value is not NULL, the row is selected.

There are two versions of this procedure. The first version contains a hard-coded SQL query, and the second version uses a dynamic query prepared with the [owa_util.bind_variables function](#).

Parameters

p_theQuery - the SQL query

p_theCursor - the cursor ID. This can be the return value from the [owa_util.bind_variables function](#).

p_cname - the name of the HTML form element

p_nsize - the size of the form element (this controls how many items the user can see without scrolling)

p_multiple - whether multiple selection is permitted

Generates

```
<SELECT NAME="p_cname" SIZE="p_nsize">  
<OPTION SELECTED  
value='value_from_the_first_column'>value_from_the_second_column  
<OPTION SELECTED  
value='value_from_the_first_column'>value_from_the_second_column  
...  
</SELECT>
```

owa_util.mime_header procedure

Syntax

```
owa_util.mime_header(  
    ccontent_type    in    varchar2 DEFAULT 'text/html',  
    bclose_header    in    boolean  DEFAULT TRUE);
```

Purpose

This procedure changes the default MIME header that the PL/SQL Agent returns.

This procedure must come before any `http.print` or `http.prn` calls in order to direct the PL/SQL Agent not to use the default.

Parameters

`ccontent_type` - the MIME type to generate

`bclose_header` - whether or not to close the HTTP header. If TRUE, two newlines are sent, which closes the HTTP header. Otherwise, one newline is sent, and the HTTP header is still open.

Generates

```
Content-type: <ccontent_type>\n\n
```

owa_util.print_cgi_env procedure

Syntax

```
owa_util.print_cgi_env;
```

Purpose

This procedure generates all the CGI environment variables and their values made available by the PL/SQL Agent to the PL/SQL procedure.

Parameters

none

Generates

A list in the following format:

```
cgi_env_var_name = value\n
```

owa_util.redirect_url procedure

Syntax

```
owa_util.redirect_url(  
    curl          in    varchar2  
    bclose_header in    boolean DEFAULT TRUE);
```

Purpose

This procedure specifies that the Web Application Server is to visit the specified URL. The URL may specify either a web page to return or a program to execute.

This procedure must come before any `http.print` or `http.prn` calls in order to tell the PL/SQL Agent to do the redirect.

Parameters

`curl` - the URL to visit

`bclose_header` - whether or not to close the HTTP header. If `TRUE`, two newlines are sent, which closes the HTTP header. Otherwise, one newline is sent, and the HTTP header is still open.

Generates

```
Location: <curl>\n\n
```

owa_util.showpage procedure

Syntax

```
owa_util.showpage;
```

Purpose

This procedure prints out the HTML output of a procedure in SQL*Plus, SQL*DBA, or Oracle Server Manager. The procedure must use the **htp** or **htf** packages to generate the HTML page, and this procedure must be issued after the 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.

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

Parameters

none

Generates

The output of htp procedure is displayed in SQL*Plus, SQL*DBA, or Oracle Server Manager. For example:

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

This would generate an HTML page that could be accessed from web browsers.

owa_util.showsource procedure

Syntax

```
owa_util.showsource (cname in varchar2);
```

Purpose

This procedure prints the source of the specified procedure, function, or package. If a procedure or function which belongs to a package is specified, then the entire package is displayed.

Parameters

cname - name of the procedure or function

Generates

The source code of the specified function, procedure, or package.

owa_util.signature procedure

Syntax

```
owa_util.signature;  
owa_util.signature (cname in varchar2);
```

Purpose

This procedure generates an HTML line followed by a signature line on the HTML document. If a parameter is specified, the procedure also generates a hypertext link to view the PL/SQL source for that procedure. The link calls the [owa_util.showsource procedure](#).

Parameters

cname - the function or procedure whose source you want to show

Generates

Without a parameter, the procedure generates a line that looks like the following:

```
This page was produced by the PL/SQL Agent on August 9, 1995 09:30
```

With a parameter, the procedure generates a signature line in the HTML document that might look like the following:

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

owa_util.status_line procedure

Syntax

```
owa_util.status_line(  
    nstatus      in      integer,  
    creason      in      varchar2 DEFAULT NULL  
    bclose_header in      boolean DEFAULT TRUE);
```

Purpose

This procedure sends a standard HTTP status code to the client. This procedure must come before any `http.print` or `http.prn` calls so that the status code is returned as part of the header, rather than as “content data”.

Parameters

`nstatus` - the status code

`creason` - the string for the status code

`bclose_header` - whether or not to close the HTTP header. If `TRUE`, two newlines are sent, which closes the HTTP header. Otherwise, one newline is sent, and the HTTP header is still open.

Generates

```
Status: <nstatus> <creason>\n\n
```

owa_util.tablePrint function

Syntax

```
owa_util.tablePrint(  
    ctable          in   varchar2  
    cattributes     in   varchar2          DEFAULT NULL  
    ntable_type     in   integer           DEFAULT HTML_TABLE  
    ccolumns        in   varchar2          DEFAULT '*'  
    cclauses        in   varchar2          DEFAULT NULL  
    ccol_aliases    in   varchar2          DEFAULT NULL  
    nrow_min        in   number            DEFAULT 0  
    nrow_max        in   number            DEFAULT NULL )  
return boolean;
```

Purpose

This function generates either preformatted or HTML tables (depending on the capabilities of the user's browser) from database tables. Note that RAW columns are supported, but LONG RAW columns are not. References to LONG RAW columns will print the result 'Not Printable'. In this function, *cattributes* is the second, rather than the last, parameter.

Parameters

ctable - the database table

cattributes - other attributes to be included as-is in the tag

ntable_type - how to generate the table. Specify "HTML_TABLE" to generate the table using <TABLE> tags or "PRE_TABLE" to generate the table using the <PRE> tags

ccolumns - a comma-delimited list of columns from *ctable* to include in the generated table

cclauses - WHERE or ORDER BY clauses, which let you specify which rows to retrieve from the database table, and how to order them

ccol_aliases - a comma-delimited list of headings for the generated table

nrow_min - the first row, of those retrieved, to display

nrow_max - the last row, of those retrieved, to display

Generates

A preformatted or HTML table.

Returns

TRUE if there are more rows beyond the *nrow_max* requested, FALSE otherwise.

Example

For browsers that do not 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/plsql/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/plsql/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 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/plsql/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

owa_util.toDate function

Syntax

```
owa_util.toDate(p_dateArray in dateType) return date;
```

Purpose

This function converts the [owa_util.dateType data type](#) to the standard Oracle database DATE type.

Parameters

p_dateArray - the value to convert

Generates

A standard DATE.

owa_util.who_called_me procedure

Syntax

```
owa_util.who_called_me(  
    owner          out   varchar2  
    name           out   varchar2  
    lineno        out   number  
    caller_t      out   varchar2);
```

Purpose

This procedure returns information (in the form of output parameters) about the PL/SQL code unit that invoked it.

Parameters

owner - the owner of the program unit

name - the name of the program unit. This is the name of the package, if the calling program unit is wrapped in a package, and the name of the procedure or function if the calling program unit is a standalone procedure or function. If the calling program unit is part of an anonymous block, this is NULL.

lineno - the line number within the program unit where the call was made.

caller_t - the type of program unit that made the call. The possibilities are: package body, anonymous block, procedure, and function. Procedure and function are used only for standalone procedures and functions.

Index

A

- <A> A-6
 - with MAILTO A-59
- <ADDRESS> A-5
- <APPLET> A-7
- applet tags (PL/SQL Cartridge) A-2
- <AREA> A-8
- authentication (PL/SQL Cartridge) 1-29
 - authorize function 1-31
 - custom authentication 1-31
- authorize function (PL/SQL Cartridge) 1-31

B

- <BASE> A-9
- <BASEFONT> A-10
- <BGSOUND> A-11
- <BIG> A-12
- <BLOCKQUOTE> A-13
- <BODY> A-14
- <BOLD> A-15
-
 A-53, A-63

C

- <CAPTION> A-82
- cartridges
 - Java Cartridge 2-1
 - PL/SQL Cartridge 1-1
- cattributes parameter
 - use in passing exact text 1-13
- <CENTER> A-16, A-17
- CGI environment variables H-9, H-17
- character formatting tags (PL/SQL Cartridge) A-4
- <CITE> A-18

- <CODE> A-19
- comments in HTML A-20
- Content Service
 - Java Cartridge 2-12
- cookies B-1
 - PL/SQL Cartridge 1-13, 1-24
- crippled includes 3-2

D

- data types supported in PL/SQL Cartridge 1-16
- database access
 - Java Cartridge 2-12
 - LiveHTML Cartridge 3-2
- Database Access Descriptor (DAD) 1-3
 - creating 1-5
- Database Connection Descriptor (DCD) 1-4
- <DD> A-25
- debugging Java Cartridge 2-34
- developing Java applications 2-4
- <DFN> A-21
- <DIR> A-22
- <DIV> A-23
- <DL> A-24
- <DT> A-26
- Dynamic HTML 2-25

E

- A-27
- environment variables
 - retrieving in the PL/SQL Cartridge H-9
- error-reporting levels in the PL/SQL Cartridge 1-29

F

 A-30
<FORM> A-32
form tags 1-11
form tags (PL/SQL Cartridge) A-2
<FRAME> A-44
frame tags (PL/SQL Cartridge) A-4
<FRAMESET> A-45

H

<H1> A-47
<HEAD> A-46
HTML
 extending 1-24
 form tags 1-11
 version 3.2 A-1
<HTML> A-48
HTML comment A-20
HtmlStream 2-10
http.address A-5
http.anchor A-6
http.anchor2 A-6
http.appletclose A-7
http.appletopen A-7
http.area A-8
http.base A-9
http.basefont A-10
http.bgsound A-11
http.big A-12
http.blockquoteClose A-13
http.blockquoteOpen A-13
http.bodyClose A-14
http.bold A-15
http.br A-63
http.center A-16
http.centerClose A-17
http.centerOpen A-17
http.cite A-18
http.code A-19
http.comment A-20
http.dfn A-21
http.dirlistClose A-22
http.dirlistOpen A-22
http.div A-23
http.dlistClose A-24
http.dlistDef A-25
http.dlistOpen A-24
http.dlistTerm A-26
http.em A-27
http.emphasis A-27
http.fontClose A-30
http.fontOpen A-30
http.formCheckbox A-31
http.formClose A-32

http.formHidden A-33
http.formImage A-34
http.formOpen A-32
http.formPassword A-35
http.formRadio A-36
http.formReset A-37
http.formSelectClose A-38
http.formSelectOpen A-38
http.formSelectOption A-39
http.formSubmit A-40
http.formText A-41
http.formTextarea A-42
http.formTextareaClose A-43
http.frame A-44
http.framesetClose A-45
http.framesetOpen A-45
http.headClose A-46
http.header A-47
http.headOpen A-46
http.hr A-53
http.htmlClose A-48
http.htmlOpen A-48
http.img A-49
http.img2 A-49
http.isindex A-50
http.italic A-51
http.kbd A-52
http.keyboard A-52
http.line A-53
http.linkRel A-54
http.linkRev A-55
http.listHeader A-56
http.listingClose A-57
http.listingOpen A-57
http.listItem A-58
http.mailClose A-60
http.mailto A-59
http.mapOpen A-60
http.menulistClose A-61
http.menulistOpen A-61
http.meta A-62
http.nl A-63
http.nobr A-64
http.noframesClose A-65
http.noframesOpen A-65
http.olistClose A-66
http.olistOpen A-66
http.para A-67
http.paragraph A-67
http.param A-68
http.plaintext A-69
http.preClose A-70
http.preOpen A-70
http.print A-71
http.prints A-72
http.prn A-71

- http.ps A-72
- http.s A-73
- http.sample A-74
- http.script A-75
- http.small A-76
- http.strike A-77
- http.strong A-78
- http.style A-79
- http.sub A-80
- http.sup A-81
- http.tableCaption A-82
- http.tableClose A-85
- http.tableData A-83
- http.tableHeader A-84
- http.tableOpen A-85
- http.tableRowClose A-86
- http.tableRowOpen A-86
- http.teletype A-87
- http.textareaOpen A-43
- http.textareaOpen2 A-43
- http.title A-88
- http.ulistClose A-89
- http.ulistOpen A-89
- http.underline A-90
- http.variable A-91
- http.wbr A-92
- HTTP request information 2-9
- HTTP response information 2-10

I

- <I> A-51
- ICX Service
 - Java Cartridge 2-11
 - LiveHTML Cartridge 3-2
 - PL/SQL Cartridge 1-28
- A-49
- <INPUT> (checkbox) A-31
- <INPUT> (hidden) A-33
- <INPUT> (image) A-34
- <INPUT> (password) A-35
- <INPUT> (radio) A-36
- <INPUT> (reset) A-37
- <INPUT (submit)> A-40
- <INPUT (text)> A-41
- IP address
 - retrieving in the PL/SQL Cartridge F-3
- <ISINDEX> A-50

J

- Java applets
 - referencing A-7
- Java Cartridge 2-1
 - applet 2-2

- application 2-2
- application invocation 2-3
- application structure 2-4
- architecture 2-2
- building an application 2-5
- cartridge version 2-3
- class hierarchy 2-26
- Content Service 2-12
- database access 2-12
- database errors 2-18
- debugging 2-34
- developing applications 2-4
- Dynamic HTML 2-25
- example code 2-37
- extending 2-33
- Hello World example 2-7
- HTTP request information 2-9
- HTTP response information 2-10
- ICX Service 2-11
- JIT Compiler 2-23
- Logger Service 2-12
- oracle.html package 2-31
- PL/SQL data type mapping 2-14
- PL/SQL procedure mapping 2-15
- PL/SQL stored procedures 2-17
- security 2-21
- Session Service 2-12
- static HTML 2-5
- supported Java versions 2-4
- Transaction Service 2-11
- troubleshooting 2-34
- WRB services 2-10

- Just-In-Time Compiler 2-23

K

- <KBD> A-52

L

- <LH> A-56
- A-58
- <LINK> A-54, A-55
- list tags (PL/SQL Cartridge) A-2
- <LISTING> A-57
- LiveHTML Cartridge
 - commands 3-3
 - crippled includes 3-2
 - database access 3-2
 - examples 3-7
 - file structure 3-2
 - Intercartridge Exchange 3-2
 - overview 3-1
- Logger Service
 - Java Cartridge 2-12

M

<MAP> A-60
<MENU> A-61
<META> A-62

N

NLS extensions in PL/SQL Cartridge 1-20
<NOBR> A-64
<NOFRAMES> A-65

O

 A-66
<OPTION> A-39
oracle.html package 2-31
overloading (PL/SQL Cartridge) 1-16
owa_cookie.cookie data type B-2
owa_cookie.get function B-3
owa_cookie.get_all procedure B-4
owa_cookie.remove procedure B-5
owa_cookie.send procedure B-6
owa_cookie.vc_arr data type B-2
owa_image.get_x function C-4
owa_image.get_y function C-5
owa_image.NULL_POINT package variable C-2
owa_image.point data type C-3
owa_opt_lock.checksum function D-3
owa_opt_lock.get_rowid function D-4
owa_opt_lock.store_values procedure D-5
owa_opt_lock.vcArray data type D-2
owa_opt_lock.verify_values function D-6
owa_pattern.amatch function E-2
owa_pattern.change function and procedure E-4
owa_pattern.change function or procedure 1-26
owa_pattern.getpat procedure E-6
owa_pattern.match function 1-26, E-7
owa_pattern.pattern data type E-9
owa_sec.get_client_hostname function F-2
owa_sec.get_client_ip function F-3
owa_sec.get_password function F-4
owa_sec.get_user_id function F-5
owa_sec.set_authorization procedure 1-31, F-6
owa_sec.set_protection_realm procedure F-7
owa_text.add2multi procedure G-2
owa_text.multi_line data type G-3
owa_text.new_row_list G-4
owa_text.print_multi procedure G-5
owa_text.print_row_list procedure G-6
owa_text.row_list data type G-7
owa_text.stream2multi procedure G-8
owa_text.vc_arr data type G-9
owa_util.bind_variables function H-3
owa_util.calendarprint procedure H-4

owa_util.cellsprint procedure H-5
owa_util.choose_date procedure H-6
owa_util.dateType data type H-8
owa_util.get_cgi_env function H-9
owa_util.get_owa_service_path function H-10
owa_util.get_procedure function H-11
owa_util.http_header_close procedure H-12
owa_util.ident_arr data type H-13
owa_util.ip_address data type H-14
owa_util.listprint procedure H-15
owa_util.mime_header procedure H-16
owa_util.print_cgi_env procedure H-17
owa_util.redirect_url procedure H-18
owa_util.showpage procedure 1-35, H-19
owa_util.showsource procedure H-20
owa_util.signature procedure H-21
owa_util.status_line procedure H-22
owa_util.tablePrint function H-23
owa_util.todate function H-26
owa_util.who_called_me procedure H-27
owains.sql 1-10

P

<P> A-67
packages (PL/SQL Cartridge)
 extending the htp and htf packages 1-24
 htf package 1-10
 htp package 1-10
 installing 1-5
 overview 1-9
 owa package 1-11
 owa_cookie package 1-13
 owa_image package 1-12
 owa_init package 1-11
 owa_opt_lock package 1-13
 owa_pattern package 1-12
 owa_sec package 1-12
 owa_text package 1-12
 owa_util package 1-12
paragraph formatting tags (PL/SQL Cartridge) A-3
<PARAM> A-68
performance
 PL/SQL Cartridge 1-33
PL/SQL Agent 1-3
 configuring 1-6
PL/SQL Cartridge 1-1
 applet tags A-2
 authentication and security 1-29
 character formatting tags A-4
 cookies 1-24
 creating a DAD 1-5
 custom authentication 1-31
 DAD 1-3
 data types supported 1-16

- dynamic username/password authentication 1-30
- error-reporting levels 1-29
- extending the htp and htf packages 1-24
- form tags A-2
- frame tags A-4
- htf package 1-10
- htp package 1-10
- ICX and 1-28
- installing packages 1-5
- invoking 1-14, 4-8
- life cycle 1-15
- list tags A-2
- NLS extensions 1-20
- overloading 1-16
- overview 1-3
- owa package 1-11
- owa_cookie package 1-13
- owa_image package 1-12
- owa_init package 1-11
- owa_opt_lock package 1-13
- owa_pattern package 1-12
- owa_sec package 1-12
- owa_text package 1-12
- owa_util package 1-12
- packages overview 1-9
- paragraph formatting tags A-3
- parameters passed to subprograms 1-13
- performance 1-33
- PL/SQL Agent 1-3
- PL/SQL Agent configuration 1-6
- request processing 1-3
- string matching and manipulation 1-25
- subprograms summary A-1, B-1, C-1, D-1, E-1, F-1, G-1, H-1
- table tags A-3
- tracing levels 1-36
- transactions 1-21
- troubleshooting 1-34
- tutorial 1-5
- URL details 1-14, 4-8
- URL to invoke the PL/SQL Cartridge 1-3
- variables with multiple values 1-17
- virtual paths 1-7
- PL/SQL table in PL/SQL Cartridge 1-17
- PL/SQL Web Toolkit
 - customizing 1-24
 - htf package A-1
 - htp package A-1
 - installation 1-10
- pl2java 2-13
- <PLAINTEXT> A-69
- <PRE> A-70

R

- regular expressions 1-27

S

- <S> A-73
- <SAMP> A-74
- <SCRIPT> A-75
- security
 - in the Java Cartridge 2-21
 - in the PL/SQL Cartridge 1-29
- <SELECT> A-38
- Session Service
 - Java Cartridge 2-12
- <SMALL> A-76
- static HTML files in Java Cartridge 2-5
- <STRIKE> A-77
- strings
 - matching and manipulating in the PL/SQL Cartridge 1-25
 - regular expressions 1-27
- A-78
- <STYLE> A-79
- <SUB> A-80
- <SUP> A-81
- System.out stream 2-10

T

- <TABLE> A-85
- table tags (PL/SQL Cartridge) A-3
- <TD> A-83
- <TEXTAREA> A-42, A-43
- <TH> A-84
- <TITLE> A-88
- <TR> A-86
- tracing levels
 - PL/SQL Cartridge 1-36
- Transaction Service
 - Java Cartridge 2-11
 - PL/SQL Cartridge 1-21
- troubleshooting Java Cartridge 2-34
- <TT> A-87

U

- <U> A-90
- A-89

V

- <VAR> A-91

vc_arr data type B-2

W

<WBR> A-92