http://www.oracle.com/webfolder/technetwork/tutorials/obe/db/oow10/php_db/php_db.htm

Using PHP with Oracle Database 11*g*

[Modidfied to work for Lab05 in SE12 labs with Windows 10 and PHP 7.1.x]

Purpose

This tutorial shows you how to use PHP with Oracle Database 11g.

Time to Complete

Approximately 2 hours

Overview

PHP is a popular web scripting language, and is often used to create database-driven web sites. This tutorial helps you get started with PHP and Oracle Database by showing how to build a web application and by giving techniques for using PHP with Oracle. If you are new to PHP, review the Appendix: PHP Primer to gain an understanding of the PHP language.

Prerequisites

Before starting this Oracle By Example, please have the following prerequisites completed:

1. Install Oracle Database 11.2
2. Start DRCP connection pooling:

```
sqlplus sys/oracle1 as sysdba
execute dbms_connection_pool.start_pool();
execute dbms_connection_pool.restore_defaults();
```

3. Use user hr with password of hr of the Oracle's sample HR schema and make the following changes:

```
create sequence emp_id_seq start with 400;
  create trigger my_emp_id_trigger
  before insert on employees for each row
  begin
    select emp_id_seq.nextval into :new.employee_id from dual;
  end;
  /
  --
  -- Also to simplify the example we remove this trigger otherwise
  -- records can only be updated once without violating the
  -- PYTHONHOL.JHIST_EMP_ID_ST_DATE_PK constraint
  --

  alter trigger update_job_history disable;

  --
  -- Allow employees to be changed when testing the lab after hours.
  --
  alter trigger secure_employees disable;
```

4. Use PHP 7.1.7 with the OCI8 dll extension as describe in Setup.pdf. In php.ini, locate and set:

```
oci8.connection_class = MYPHPAPP
```

Using PHP OCI8 with Oracle Database 11g

This section of the tutorial shows how to use the PHP OCI8 extension directly with Oracle Database. Using the OCI8 extension directly gives programmers maximum control over application performance.

Creating a Standard Connection

To create a connection to Oracle that can be used for the lifetime of the PHP script, perform the following steps.

1. Create **`connect.php [Create all files in D:\work\2714 folder]`**

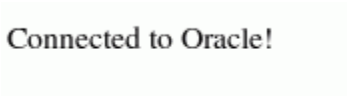
```
<?php
// Create connection to Oracle
$conn = oci_connect("hr", "hr", "//localhost/xe");
if (!$conn) {
   $m = oci_error();
   echo $m['message'], "\n";
   exit;
}
else {
   print "Connected to Oracle!";
}
// Close the Oracle connection
oci_close($conn);
?>
```

The `oci_connect()` function contains the username, the password and the connection string. In this case, Oracle's Easy Connect connection string syntax is used. It consists of the hostname and the DB service name.

The `oci_close()` function closes the connection. Any standard connections not explicitly closed will be automatically released when the script ends.

2. Open a Web browser and enter the following URL to display the output:

`http://localhost:8888/connect.php`



"Connected to Oracle!" is displayed if the connection succeeds.

3. Create **usersess.sql**

```
column username format a30
column logon_time format a18
set pagesize 1000 feedback off echo on

select username, to_char(logon_time, 'DD-MON-YY HH:MI:SS') logon_time
from v$session
where username is not null;

exit
```

This is a SQL script file that you run in SQL*Plus (Oracle's command-line SQL scripting tool). This SQL*Plus script shows the current database sessions, and what time they logged into the database.

4. Open a terminal window and enter the following commands to run the SQL script. Note that you could also execute the script in SQL Developer.

```
D:
cd \work\2714
sqlplus -l system/oracle1 @usersess.sql
```

```
[phphol@localhost public_html]$ sqlplus -l phphol/welcome @usersess.sql

SQL*Plus: Release 11.2.0.1.0 Production on Fri Aug 6 12:31:20 2010

Copyright (c) 1982, 2009, Oracle.  All rights reserved.


Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL>
SQL> select username, to_char(logon_time, 'DD-MON-YY HH:MI:SS') logon_time
  2  from v$session
  3  where username is not null;

USERNAME                       LOGON_TIME
------------------------------ ------------------
PHPHOL                         06-AUG-10 12:31:20
SQL>
SQL> exit
```

The SQL*Plus script lists the current database sessions. The only session shown is for SQL*Plus. The PHP connections from the `oci_connect()` function has been closed.

5. Edit **connect.php** and change **oci_connect()** to use a persistent connection **oci_pconnect().**

```
$conn = oci_pconnect("hr", "hr", "//localhost/xe");
```

Reload the **connect.php** script in the browser. Now rerurn **usersess.sql** in SQL*Plus.

```
D:
cd work\2714
sqlplus -l system/oracle1 @usersess.sql
```

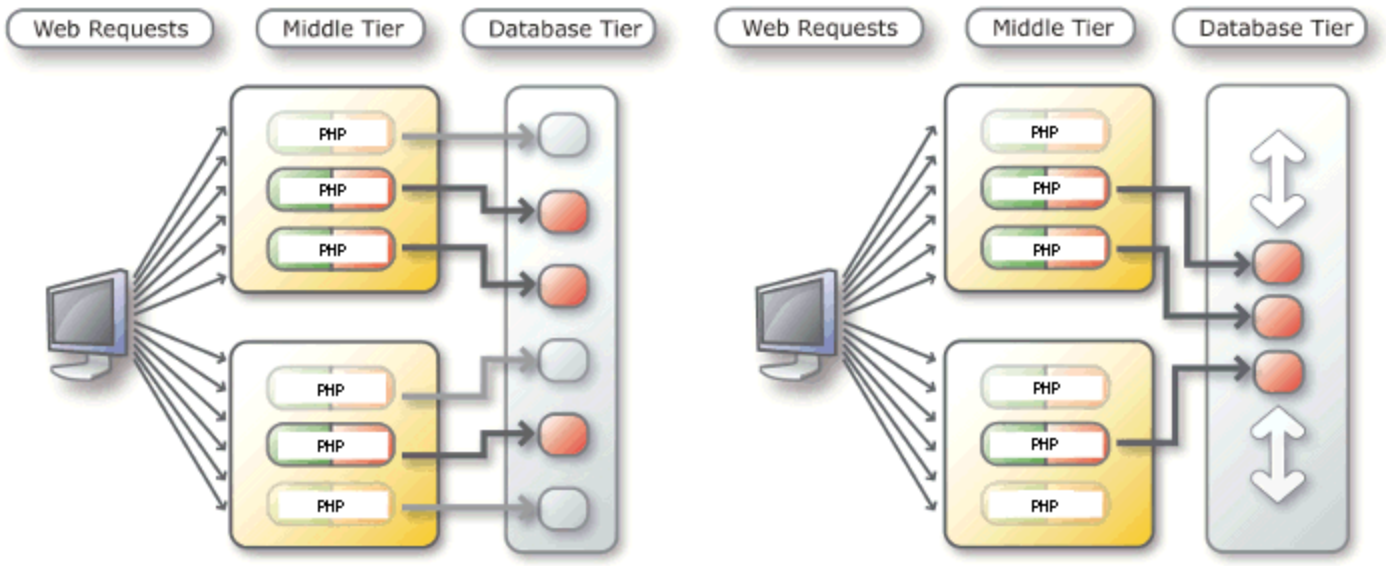
```
SQL> select username, to_char(logon_time, 'DD-MON-YY HH:MI:SS') logon_time
  2  from v$session
  3  where username is not null;

USERNAME                       LOGON_TIME
------------------------------ ------------------
PHPHOL                         06-AUG-10 12:46:21
PHPHOL                         06-AUG-10 12:46:12
SQL>
```

There should be two connected HR users, one from the SQLPlus session, and one from the web browser running connect.php. By default, persistent connections stay open until the process terminates. Subsequent PHP scripts can reuse the already opened connection, making them run faster.

Using Database Resident Connection Pooling

Database Resident Connection Pooling is a new feature of Oracle Database 11g. It is useful for short lived scripts such as typically used by web applications. It allows the number of connections to be scaled as web site usage grows. It allows multiple Apache processes on multiple machines to share a small pool of database server processes. Without DRCP, a non-persistent PHP connection must start and terminate a server process, and a persistent PHP connection keeps hold of database resources even when PHP is idle.

Below left is diagram of nonpooling. Every script has its own database server proces. Scripts not doing any database work still hold onto a connection until the connection is closed and the server is terminated. Below right is a diagram with DRCP. Scripts can use database servers from a pool of servers and return them when no longer needed.



Batch scripts doing long running jobs should generally use non-pooled connections.

This section of the tutorial shows how DRCP can be used by new or existing applications without writing or changing any application logic. Perform the following steps:

1. Check that php has `oci8.connection_class` set. Open a terminal window and execute the following command:

   `php -r "echo ini_get('oci8.connection_class');"`

```
[phphol@localhost public_html]$ php -r 'echo ini_get("oci8.connection_class"),"
\n";'
MYPHPAPP
[phphol@localhost public_html]$
```

   The connection class tells the database server pool that connections are related. Session information (such as the default date format) might be retained between connection calls, giving performance benefits. Session information will be discarded if a pooled server is later reused by a different application with its own connection class name.

2. Create **`query_pooled.php`**

```
<?php

$c = oci_pconnect("hr", "hr", "//localhost/xe:pooled");

$s = oci_parse($c, 'select * from employees');
oci_execute($s);
oci_fetch_all($s, $res);
echo "<pre>\n";
var_dump($res);
echo "</pre>\n";

?>
```

   Create also **`query_nonpooled.php`**

```
<?php

$c = oci_pconnect("hr", "hr", "//localhost/xe");
$s = oci_parse($c, 'select * from employees');
oci_execute($s);
oci_fetch_all($s, $res);
echo "<pre>\n";
var_dump($res);
echo "</pre>\n";

?>
```

   The only difference is the "`:pooled`" in the Easy Connect connection string in `query_pooled.php`.

3. To run the scripts, the Apache Benchmark tool is used. This command repeatedly loads a web page, measuring its performance. From a terminal window, execute the following:

   `ab -c 150 -t 30 http://localhost:8888/query_pooled.php`

```
[phphol@localhost public_html]$ ab -c 150 -t 30 http://localhost/~phphol/qu
ooled.php
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $> apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Finished 353 requests


Server Software:        Apache/2.2.3
Server Hostname:        localhost
Server Port:            80

Document Path:          /~phphol/query_pooled.php
Document Length:        39481 bytes

Concurrency Level:      150
Time taken for tests:   30.29572 seconds
Complete requests:      353
Failed requests:        0
Write errors:           0
Total transferred:      14083305 bytes
HTML transferred:       14017089 bytes
Requests per second:    11.76 [#/sec] (mean)
Time per request:       12760.441 [ms] (mean)
Time per request:       85.070 [ms] (mean, across all concurrent requests)
Transfer rate:          457.98 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  192 805.7      5    9069
Processing:  1833 9306 3335.8  10295   28359
Waiting:     1831 9293 3329.2  10259   28356
Total:       1837 9498 3556.8  10319   28364

Percentage of the requests served within a certain time (ms)
  50%  10314
  66%  10876
  75%  11054
  80%  11173
  90%  12165
  95%  14082
  98%  18177
  99%  20549
 100%  28364 (longest request)
```

The above command sends the web server 150 concurrent requests for the script, repeatedly for 30 seconds.

4. Now look at the number of database connections open. Open another terminal window, execute the following:

```
sqlplus system/oracle1
column username format A10
select username, program from v$session where username = 'HR';
```

The default DRCP pool MAXSIZE is 40. You see up to 40 connections with HR username, depending on which web server you are using, and how many web server processes handled the 'ab' requests. You may also need to execute the query while 'ab' is running to see the pooled servers working.

Oracle manages the DRCP pool, shrinking it after a specified timeout.

[Note: the number of HR connections showing in the above will vary depending on the web server used.
With the built-in PHP web server, how many sessions are there?_____

]

5. Now, you will run the same command except run the non-pooled script to compare the difference. From a terminal window, execute the following:

```
ab -c 150 -t 30 http://localhost:8888/query_nonpooled.php
```

6. Now look at the number of database connections open. Open another terminal window, execute the following:

```
sqlplus system/oracle1
column username format A10
select username, program from v$session where username = 'HR';
```

How many more rows than previously are returned? Each row corresponds to a running web server process holding a database connection open. For PHP, some web servers like Apache runs in a multi-process mode, spawning child processes each of which can handle one PHP script. Depending how the web server allocated these processes to handle the "ab" requests, you may see a varying number of rows in `V$SESSION`.

Compare the number of requests completed in each run. You might want to run each script a few times to warm up the caches.

Performance of the scripts is roughly similar. For the small works loads used in these two files, the tiny overhead of the handoff of pooled servers might make `query_pooled.php` a little slower than `query_nonpooled.php`. But the non-pooled script causes every single web server process to open a separate connection to the database. For larger sites, or where memory is limited, the overall benefits of DRCP are significant.

Fetching Data

A common task when developing Web applications is to query a database and display the results in a Web browser. There are a number of functions you can use to query an Oracle database, but the basics of querying are always the same:

1. **Parse** the statement for execution.
2. **Bind** data values (optional).
3. **Execute** the statement.
4. **Fetch** the results from the database.

To create a simple query, and display the results in an HTML table, perform the following steps.

1. Create **`query.php`**

```
<?php

// Create connection to Oracle
$conn = oci_connect("hr", "hr", "//localhost/xe");

$query = 'select * from departments';
$stid = oci_parse($conn, $query);
$r = oci_execute($stid);

// Fetch each row in an associative array
print '<table border="1">';
while ($row = oci_fetch_array($stid, OCI_RETURN_NULLS+OCI_ASSOC)) {
   print '<tr>';
   foreach ($row as $item) {
       print '<td>'.($item !== null ? htmlentities($item, ENT_QUOTES) :
' ').'</td>';
   }
   print '</tr>';
}
print '</table>';

?>
```

   The `oci_parse()` function parses the statement.

   The `oci_execute()` function executes the parsed statement.

   The `oci_fetch_array()` function retrieves a row of results of the query as an associative array, and includes nulls.

   The `htmlentities()` function escapes any text resembling HTML tags so it displays correctly in the browser.

2. From your Web browser, enter the following URL to display the output:

   `http://localhost:8888/query.php`

10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
150	Shareholder Services		1700
160	Benefits		1700
170	Manufacturing		1700

The results of the query are displayed in the Web browser.

The `OCI_ASSOC` parameter fetches the row as an associative arrary of column names and column data.

Alternatively, the `OCI_NUM` parameter can be passed to `oci_fetch_array()` to fetch the row as a numeric array.

Using Bind Variables

Bind variables enable you to re-execute statements with new values, without the overhead of reparsing the statement. Bind variables improve code reusability, and can reduce the risk of SQL Injection attacks.

To use bind variables in this example, perform the following steps.

1. Create **`bind.php`**

```
<?php

function do_fetch($myeid, $s)
{
  // Fetch the results in an associative array
  print '<p>$myeid is ' . $myeid . '</p>';
  print '<table border="1">';
  while ($row = oci_fetch_array($s, OCI_RETURN_NULLS+OCI_ASSOC)) {
    print '<tr>';
    foreach ($row as $item) {
      print '<td>'.($item?htmlentities($item):' ').'</td>';
    }
    print '</tr>';
  }
  print '</table>';
}

// Create connection to Oracle
$c = oci_connect("hr", "hr", "//localhost/xe");

// Use bind variable to improve resuability,
// and to remove SQL Injection attacks.
$query = 'select * from employees where employee_id = :eidbv';
$s = oci_parse($c, $query);

$myeid = 101;
oci_bind_by_name($s, ":EIDBV", $myeid);
oci_execute($s);
do_fetch($myeid, $s);

// Redo query without reparsing SQL statement
$myeid = 104;
oci_execute($s);
do_fetch($myeid, $s);

// Close the Oracle connection
oci_close($c);

?>
```

2. From your Web browser, enter the following URL to display the output:

`http://localhost:8888/bind.php`

$myeid is 101

101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000		100	90

$myeid is 104

104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000		103	60

The `$myeid` variable is bound to the `:eidbv` bind variable so when the query is re-executed the new value of `$myeid` is passed to the query. This allows you to execute the statement again, without reparsing it with the new value, and can improve performance of your code.

If you don't see the returned rows, you may have deleted these employees in the web application part of the tutorial. Use SQL*Plus to query the `EMPLOYEE_ID` column of the `EMPLOYEES` table, and edit `bind.php` to use IDs that exist in the table.

Creating Transactions

When you manipulate data in an Oracle Database (insert, update, or delete data), the changed or new data is only available within your database session until it is committed to the database. When the changed data is committed to the database, it is then available to other users and sessions. This is a database transaction.

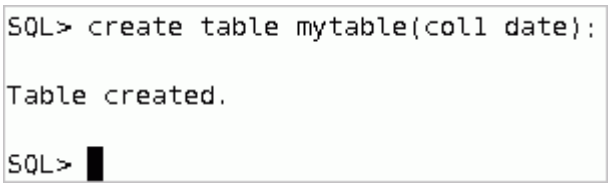
By default, when PHP executes a SQL statement it automatically commits. This can be over-ridden, and the `oci_commit()` and `oci_rollback()` functions used to control transactions. At the end of a PHP script, any uncommitted data is rolled back.

Committing each change individually causes extra load on the server. In general you want all or none of your data committed. Doing your own transaction control has performance and data-integrity benefits.

To learn about transaction management in PHP with an Oracle database, perform the following steps.

1. Start SQL*Plus and create a new table:

```
sqlplus hr/hr
create table mytable (col1 date);
```



2. Create **`trans_rollback.php`**

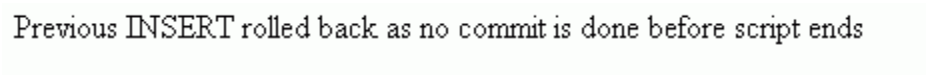
```
<?php

$conn = oci_connect("hr", "hr", "//localhost/xe");

// PHP function to get a formatted date
$d = date('j:M:y H:i:s');

// Insert the date into mytable
$s = oci_parse($conn,
              "insert into mytable values (to_date('" . $d . "',
         'DD:MON:YY HH24:MI:SS'))");

// Use OCI_DEFAULT to insert without committing
$r = oci_execute($s, OCI_DEFAULT);

echo "Previous INSERT rolled back as no commit is done before script ends";

?>
```

The `OCI_DEFAULT` parameter overrides the basic behavior of `oci_execute()`.

3. From your Web browser, enter the following URL to display the output:

`http://localhost:8888/trans_rollback.php`

This script inserts a row into the table.



4. Because there is no automatic or explicit commit, the data is rolled back by PHP when the script finishes. To see that the data has not been committed, query the table to see if there are any inserted rows. From your SQL*Plus session, enter the following commands to select any rows from the mytable table:

```
select to_char(col1, 'YYYY-MM-DD HH:MI:SS') time from mytable;
```

You will see:
no rows selected

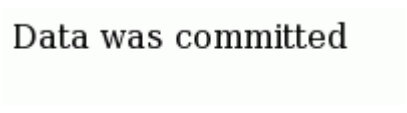
5. Create **trans_autocommit.php**

```
<?php

$conn = oci_connect("hr", "hr", "//localhost/xe");

// PHP function to get a formatted date
$d = date('j:M:y H:i:s');

// Insert the date into mytable
$s = oci_parse($conn,
              "insert into mytable values (to_date('" . $d . "',
          'DD:MON:YY HH24:MI:SS'))");

// Insert & commits
$r = oci_execute($s);

// The rollback does nothing: the data has already been committed
oci_rollback($conn);

echo "Data was committed\n";

?>
```

This script differs from `trans_rollback.php` in that there is no `OCI_DEFAULT` when the data is inserted. This means the new data is auto committed by the `oci_execute()` call.

6. From your Web browser, enter the following URL to display the output:

http://localhost:8888/trans_autocommit.php



The data is now committed.

7. From your SQL*Plus session, enter the following command to select any rows from the mytable table:

```
select to_char(col1, 'YYYY-MM-DD HH:MI:SS') time from mytable;
```

You will see:

```
TIME
-------------------
2008-11-10 03:39:20
```

If you reloaded the PHP script more than once, a row from each execution is inserted.

8. You can compare the performance difference between committing each row individually versus at the end of the transaction.

To test the difference, create **trans_time_autocommit.php**

This code commits on each insert.

```
<?php

function do_insert($conn)
{
  $stmt = "insert into mytable values (to_date('2008-01-01 10:20:35',
       'YYYY-MM-DD HH24:MI:SS'))";
  $s = oci_parse($conn, $stmt);
  $r = oci_execute($s);  // automatically commit
}
function do_row_check($conn)
{
  $stid = oci_parse($conn, "select count(*) c from mytable");
  oci_execute($stid);
  oci_fetch_all($stid, $res);
  echo "Number of rows: ", $res['C'][0], "<br>";
}
function do_delete($conn)
{
  $stmt = "delete from mytable";
  $s = oci_parse($conn, $stmt);
  $r = oci_execute($s);
}

// Program starts here
$c = oci_connect("hr", "hr", "//localhost/xe");

$starttime = microtime(TRUE);
for ($i = 0; $i < 10000; $i++) {
  do_insert($c);
}
$endtime = microtime(TRUE) - $starttime;
echo "Time was ".round($endtime,3)." seconds<br>";

do_row_check($c);  // Check insert done
do_delete($c);     // cleanup committed rows

?>
```

Load the URL http://localhost:8888/trans_time_autocommit.php several times and see how long it takes to insert the 10,000 rows.

```
Time was 18.341 seconds
Number of rows: 10000
```

Note: Your time values may differ depending on the hardware resources you are using.

9. Now create the **`trans_time_explicit.php`** script. The only difference in this script is that in the `do_insert()` function `OCI_DEFAULT` has been added so it doesn't automatically commit, and an explicit commit has been added after the insertion loop:

```
...

function do_insert($conn) {
  $stmt = "insert into mytable values (to_date('2008-01-01 10:20:35',
      'YYYY-MM-DD HH24:MI:SS'))";
  $s = oci_parse($conn, $stmt);
  $r = oci_execute($s, OCI_DEFAULT);  // Don't commit
}

...

$starttime = microtime(TRUE);
for ($i = 0; $i < 10000; $i++) {
  do_insert($c);
}
oci_commit($c);
$endtime = microtime(TRUE) - $starttime;

...
```

Load the URL http://localhost:8888/trans_time_explicit.php. The insertion time is less.

```
Time was 6.224 seconds
Number of rows: 10000
```

In general you want all or none of your data committed. Doing your own transaction control has performance and data-integrity benefits.

Note: Your time values may differ depending on the hardware resources you are using.

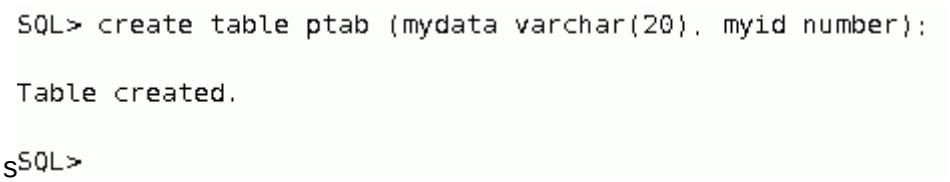
Using Stored Procedures

PL/SQL is Oracle's procedural language extension to SQL. PL/SQL procedures and functions are stored in the database. Using PL/SQL lets all database applications reuse logic, no matter how the application accesses the database. Many data-related operations can be performed in PL/SQL faster than extracting the data into a program (for example, PHP) and then processing it. Oracle also supports Java stored procedures.

In this tutorial, you will create a PL/SQL stored procedure and call it in a PHP script. Perform the following steps:

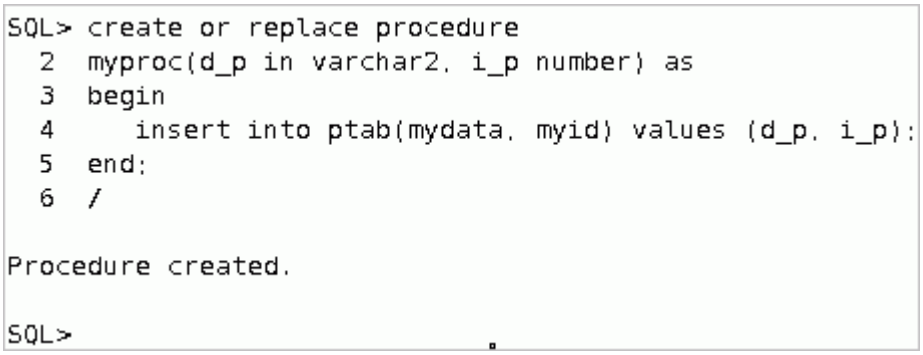
1. Start SQL*Plus and create a new table, `ptab` with the following command:

```
sqlplus hr/hr
create table ptab (mydata varchar(20), myid number);
```



2. In SQL*Plus, create a stored procedure, `myproc`, to insert data into the `ptab` table, with the following commands:

```
create or replace procedure
myproc(d_p in varchar2, i_p in number) as
begin
  insert into ptab (mydata, myid) values (d_p, i_p);
end;
/
```



3. Create **`proc.php`**

```
<?php

$c = oci_connect('hr', 'hr', '//localhost/xe');
$s = oci_parse($c, "call myproc('mydata', 123)");
oci_execute($s);
echo "Completed";

?>
```

4. From a Web browser, enter the following URL to display the output:

http://localhost:8888/proc.php

Completed

The PHP script has created a new row in the `ptab` table by calling the stored procedure `myproc`. The table `ptab`has a new row with the values "`mydata`" and `123`.

Switch to your SQL*Plus session and query the table to show the new row:

select * from ptab;

```
SQL> select * from ptab;

MYDATA                    MYID
-------------------- ----------
mydata                     123
```

5. Extend `proc.php` to use a bind variable. Change **proc.php** to the following (changes are in bold):

```
<?php

$c = oci_connect('hr', 'hr', '//localhost/xe');
$s = oci_parse($c, "call myproc('mydata', :bv)");
$v = 456;
oci_bind_by_name($s, ":bv", $v);
oci_execute($s);
echo "Completed";

?>
```

The `oci_bind_by_name()` function binds the PHP variable `$v` to "`:bv`" and experiment changing the value inserted by changing the value in `$v`.

Rerun the following URL:

http://localhost:8888/proc.php

Completed

Switch to your SQL*Plus session and query the table again to show the new row:

select * from ptab;

```
MYDATA                    MYID
-------------------- ----------
mydata                     123
mydata                     456
```

6. PL/SQL stored functions are also commonly used in Oracle. In SQL*Plus, create a PL/SQL stored function `myfunc()` to insert a row into the `ptab` table, and return double the inserted value:

```
create or replace function
myfunc(d_p in varchar2, i_p in number) return number as
begin
  insert into ptab (mydata, myid) values (d_p, i_p);
  return (i_p * 2);
end;
/
```

7. Create **`func.php`**

```
<?php

$c = oci_connect('hr', 'hr', '//localhost/xe');
$s = oci_parse($c, "begin :bv := myfunc('mydata', 123); end;");
oci_bind_by_name($s, ":bv", $v, 10);
oci_execute($s);
echo $v, "<br>\n";
echo "Completed";

?>
```

Because a value is being returned, the optional length parameter to `oci_bind_by_name()` is set to 10 so PHP can allocate the correct amount of memory to hold up to 10 digits

Run the following URL:

`http://localhost:8888/func.php`

Switch to your SQL*Plus session and query the table again to show the new row:

```
select * from ptab;
```

Improve Query Performance

This section demonstrates some ways to improve query performance. Perform the following steps:

1. Create **`fetch_prefetch.sql`**

```
set echo on
drop table bigtab;
create table bigtab (mycol varchar2(20));
begin
  for i in 1..20000
  loop
   insert into bigtab (mycol) values (dbms_random.string('A',20));
  end loop;
end;
/
commit;
exit
```

This script creates a table with a large number of rows. From your sqlplus session, run the following:

`connect hr/hr`

`@fetch_prefetch`

```
SQL> @fetch_prefetch
SQL>
SQL> drop table bigtab;
drop table bigtab
           *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> create table bigtab (mycol varchar2(20));

Table created.

SQL>
SQL> begin
  2    for i in 1..20000
  3    loop
  4     insert into bigtab (mycol) values (dbms_random.string('A',20));
  5    end loop;
  6  end;
  7  /

PL/SQL procedure successfully completed.

SQL>
SQL> commit;

Commit complete.

SQL>
SQL> exit
Disconnected from Oracle Database 11g Enterprise Edition Release 11.2.0.1.0
oduction
With the Partitioning, OLAP, Data Mining and Real Application Testing optic
```

2. Create **fetch_prefetch.php**

```
<?php

require('helper.php');

function do_prefetch($c, $pf)
{
  $stid = oci_parse($c, "select mycol from bigtab");
  oci_execute($stid);
  oci_set_prefetch($stid, $pf);
  oci_fetch_all($stid, $res);
  return $res;
}

$c = oci_connect("hr", "hr", "//localhost/xe");
$pf_a = array(1, 10, 500, 2000); // Prefetch values to test
foreach ($pf_a as $pf_num)
{
  $start = currTime();
  $r = do_prefetch($c, $pf_num);
  $t = elapsedTime($start);
  print "Prefetch $pf_num - Elapsed time is: " . round($t, 3) .
        " seconds<br>\n";
}

?>
```

This performs the same query with different prefetch sizes. Prefetching is a form of internal row buffering. The number of rows in the buffer is the prefetch value. The larger the prefetch value, the fewer the number of physical database access are needed to return all data to PHP, because each underlying physical request to the database returns more than one row. This can help improve performance. PHP code does not need to change to handle different prefetch sizes. The buffering is handled by Oracle code.

Create helper.php:

```
<?php

function currTime()
{
  return microtime(TRUE);
}

function elapsedTime($start)
{
  return (currTime() - $start);
}

function do_buildarray($num)
{
  for ($i = 0; $i < $num; $i++) {
     $a[] = 'value '.$i;
  }
  return $a;
}
```

```
function do_delete($c)
{
  $stmt = "delete from ptab";
  $s = oci_parse($c, $stmt);
  $r = oci_execute($s);
}

?>
```

The require script `helper.php` contains the simple `currTime()` and `elapsedTime()` timing functions, as well as some additional setup functions used later.

3. Load the following URL to display the output:

`http://localhost:8888/fetch_prefetch.php`

```
Prefetch 1 - Elapsed time is: 2.428 seconds
Prefetch 10 - Elapsed time is: 0.354 seconds
Prefetch 500 - Elapsed time is: 0.083 seconds
Prefetch 2000 - Elapsed time is: 0.117 seconds
```

Reload a few times to see the average times. Your time values may differ depending on your hardware resources, and so on.

The default prefetch size can be set in PHP's initialization file, `php.ini`. Prior to PHP 5.3, the default prefetch size was 10 rows. In 5.3, it is 100 rows. You should choose a suitable default value for your application, and use `oci_set_prefetch()` for specific queries that need a different value.

When using Oracle Database 11g Release 2 client libraries, row prefetching also benefits fetching from REF CURSORS.

4. This section shows the `oci_bind_array_by_name()` function that allows a PHP array to be retrieved from, or passed to, a PL/SQL procedure.

Create **`fetch_bulk.sql`**

```
set echo on
create or replace package fetchperfpkg as
  type arrtype is table of varchar2(20) index by pls_integer;
  procedure selbulk(p1 out arrtype);
end fetchperfpkg;
/
create or replace package body fetchperfpkg as
  procedure selbulk(p1 out arrtype) is
  begin
   select  mycol bulk collect
     into  p1
     from  bigtab;
  end selbulk;
end fetchperfpkg;
/
show errors
exit
```

This script creates a PL/SQL package that fetches from `BIGTAB` using a PL/SQL `BULK COLLECT` statement, and returns the results in a PL/SQL array. From your sqlplus session, run the following:

`sqlplus hr/hr`
`@fetch_bulk`

```
SQL> @fetch_bulk
SQL>
SQL> create or replace package fetchperfpkg as
  2    type arrtype is table of varchar2(20) index by pls_integer;
  3    procedure selbulk(p1 out arrtype);
  4  end fetchperfpkg;
  5  /

Package created.

SQL>
SQL> create or replace package body fetchperfpkg as
  2    procedure selbulk(p1 out arrtype) is
  3    begin
  4     select  mycol bulk collect
  5       into  p1
  6       from  bigtab;
  7    end selbulk;
  8  end fetchperfpkg;
  9  /

Package body created.

SQL>
SQL> show errors
No errors.
SQL>
SQL> exit
```

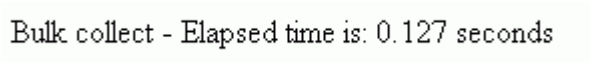
5. Create **fetch_bulk.php**

```
<?php

require('helper.php');

function do_sel_bulk($c)
{
  $s = oci_parse($c, "begin fetchperfpkg.selbulk(:a1); end;");
  oci_bind_array_by_name($s, ":a1", $res, 20000, 20, SQLT_CHR);
  oci_execute($s);
  return($res);
}

$c = oci_connect("hr", "hr", "//localhost/xe");

$start = currTime();
$r = do_sel_bulk($c);
$t = elapsedTime($start);
print "Bulk collect - Elapsed time is: " . round($t, 3) . " seconds\n<br>";

?>
```

This code calls the PL/SQL package and binds a PHP variable to hold the returned data array. No OCI8 fetch call is needed.

6. Load the following URL to display the output:

http://localhost:8888/fetch_bulk.php



Reload a few times to see the average times.

Array binding is a useful technique to reduce database overhead when inserting or retrieving data.

This example doesn't print the returned results. If you want to see them, add "`var_dump($res);`" before the function return statement. The output shows the random 20-character data strings created by `fetch_prefetch.sql`, which you ran earlier.

Using LOBs: Uploading and Querying Images

Oracle Character Large Object (CLOB) and Binary Large Object (BLOB) columns (and PL/SQL variables) can contain very large amounts of data. There are various ways of creating them to optimize Oracle storage. There is also a pre-supplied package `DBMS_LOB` that makes manipulating them in PL/SQL easy.

To create a small application to load and display images to the database, perform the following steps.

1. Before doing this section create a table to store a BLOB. In SQL*Plus logged in as `hr/hr`, execute the following commands:

```
sqlplus hr/hr
create table btab (blobid number, blobdata blob);
```

2. Create **`blobins.php`**

```
<?php
if (!isset($_FILES['lob_upload'])) {
// If nothing uploaded, display the upload form
?>

<form action="<?php echo $_SERVER['PHP_SELF']; ?>"
      method="POST" enctype="multipart/form-data">
Image filename: <input type="file" name="lob_upload">
<input type="submit" value="Upload">
</form>


<?php
} // closing brace from 'if' in earlier PHP code
else {
  // else script was called with data to upload


  $myblobid = 1; // should really be a unique id e.g. a sequence number


  $conn = oci_connect("hr", "hr", "//localhost/xe");

  // Delete any existing BLOB
  $query = 'delete from btab where blobid = :myblobid';
  $stmt = oci_parse ($conn, $query);
  oci_bind_by_name($stmt, ':myblobid', $myblobid);
  $e = oci_execute($stmt);

  // Insert the BLOB from PHP's temporary upload area
  $lob = oci_new_descriptor($conn, OCI_D_LOB);
  $stmt = oci_parse($conn, 'insert into btab (blobid, blobdata) '
        .'values(:myblobid, empty_blob()) returning blobdata into :blobdata');
  oci_bind_by_name($stmt, ':myblobid', $myblobid);
  oci_bind_by_name($stmt, ':blobdata', $lob, -1, OCI_B_BLOB);
  oci_execute($stmt, OCI_DEFAULT);  // Note OCI_DEFAULT
  if ($lob->savefile($_FILES['lob_upload']['tmp_name'])) {
    oci_commit($conn);
    echo "BLOB uploaded";
```

```
  }
  else {
    echo "Couldn't upload BLOB\n";
  }
  $lob->free();
}

?>
```

This shows HTML code embedded in multiple PHP blocks. In particular, a PHP 'if' statement encloses the HTML code.The first time the script is loaded, the HTML upload form is shown. PHP has populated the form action name to call the same script again.

There is a direct relationship between the HTML form name `name="lob_upload"` and the special PHP variable `$_FILES['lob_upload']`. When the form is called with data, the script deletes any existing image from the table, and inserts the new picture.

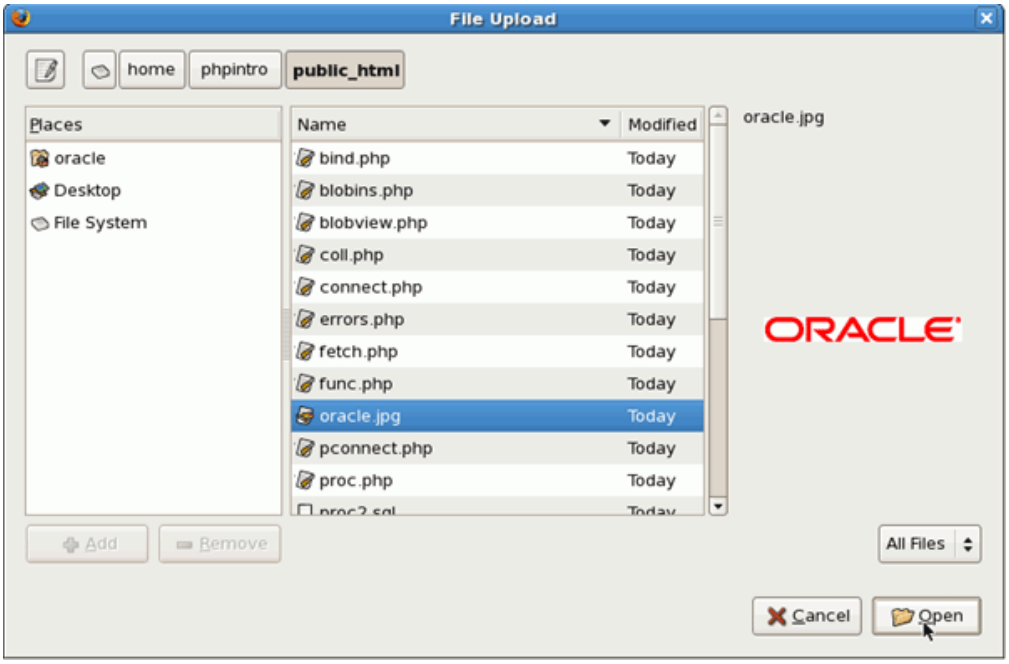
The script shows the use of `oci_new_descriptor()` which is bound to the `empty_blob()` location. The `LOB->savefile()` method inserts the picture to the newly created row. Note the `OCI_DEFAULT` option to `oci_execute()` is necessary for the subsequent LOB method to work.

3. From your Web browser, enter the following URL to display the output:

`http://localhost:8888/blobins.php`

It shows a Web form with Browse and Upload buttons. Click **Browse.**

4. Select **`oracle.jpg`** from the `current D:\work\2714` directory and click **Open**.



5 . Click **Upload**.

The form action calls the script a second time, but now the special variable `$_FILES['lob_upload']` is set and picture is uploaded. The successful `echo` message is displayed.

The image has been uploaded to the Web server.

6 . To show the image, create **`blobview.php`**

```
<?php

$conn = oci_connect("hr", "hr", "//localhost/xe");

$query = 'SELECT BLOBDATA FROM BTAB WHERE BLOBID = :MYBLOBID';
$stmt = oci_parse ($conn, $query);
$myblobid = 1;
oci_bind_by_name($stmt, ':MYBLOBID', $myblobid);
oci_execute($stmt);
$arr = oci_fetch_array($stmt, OCI_ASSOC);
$result = $arr['BLOBDATA']->load();

header("Content-type: image/JPEG");
echo $result;

oci_close($conn);

?>
```

7 . From your Web browser, enter the following URL to display the output:

`http://localhost:8888/blobview.php`

Make sure there is no whitespace before `<?php` and no echo statements in the script, because otherwise the wrong HTTP header will be sent and the browser won't display the image properly. If you have problems, comment out the `header()` function call and see what is displayed.

Incorporating AJAX into Your Page

This section shows the basic technique of updating a section of a page without reloading the whole content.

You can use a XmlHttpRequest to update a section of a page without reloading the whole page content. Perform the following steps:

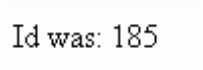
1. Create **`ajax_id.php`**

This file simply echoes the parameter passed in.

```
<?php

if (!isset($_GET['id'])) {
  $id = 'No id passed';
}
else {
  $id = $_GET['id'];
}

echo "Id was: ", htmlentities($id);

?>
```

2. From your browser, enter the following URL to display the output:

`http://localhost:8888/ajax_id.php?id=185`



3. Create **`ajax_id.html`**

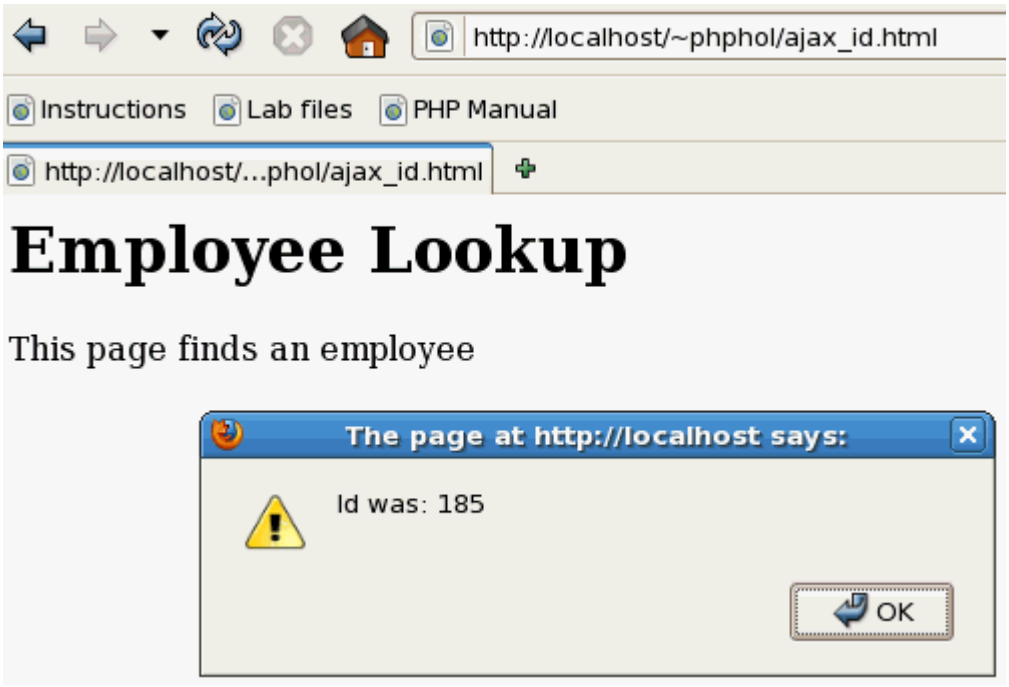
This file contains a JavaScript function, `makeRequest()`.

```
<html>
<head>
  <script type="text/javascript">
  function makeRequest(id)
  {
     httpRequest = new XMLHttpRequest();
     httpRequest.open('GET', 'http://localhost:8888/ajax_id.php?id=' + id);
     httpRequest.onreadystatechange = function()
     {
        if (httpRequest.readyState == 4) {    // The request is complete
          alert(httpRequest.responseText);    // Display the result
        }
     }
     httpRequest.send(null);
  }
  </script>
</head>
```

```
<body onload="makeRequest(185)">
<h1>Employee Lookup</h2>
<div id="descriptionNode">This page finds an employee</div>
</body>
</html>
```

4. From your browser, enter the following URL to display the output:

`http://localhost:8888/ajax_id.html`



Click **OK** to dismiss the alert window.

Note: if you use Internet Explorer, you will need to edit `ajax_id.html` and change the `XMLHttpRequest()` call to `ActiveXObject("Msxml2.XMLHTTP")` or `ActiveXObject("Microsoft.XMLHTTP")`.

When the HTML page is loaded, the `makeRequest()` javascript function is called. It prepares an XMLHttpRequest request to call `ajax_id.php`. A callback function onreadystatechange is set. Finally the request is sent to the webserver asynchronously.

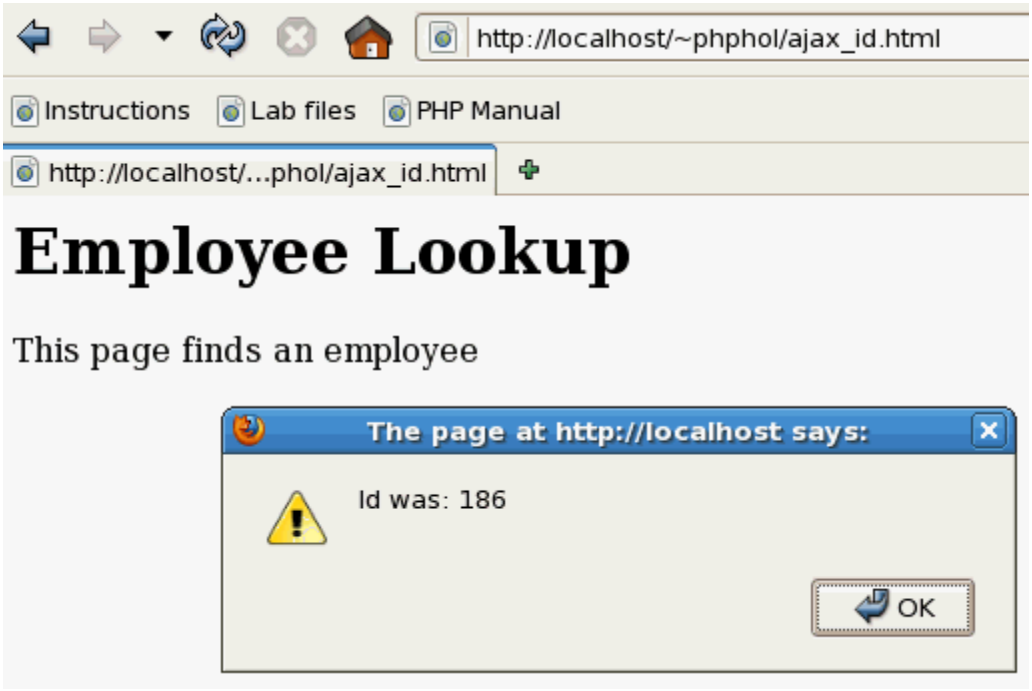
When the callback function is notified that the web server request has returned, the output from `ajax_id.php` is displayed by the `alert()` function. In web application, the Java script could be invoked by various events and could be made to alter the content of the current page.

5. Edit **`ajax_id.html`** and change 185 to **186**.

*ajax_id.html

```
<html>
<head>
  <script type="text/javascript">
  function makeRequest(id)
  {
     httpRequest = new XMLHttpRequest();
     httpRequest.open('GET', 'http://localhost/~phpweb/ajax_id.php?id=' + id);
     httpRequest.onreadystatechange = function()
     {
        if (httpRequest.readyState == 4) {    // The request is complete
          alert(httpRequest.responseText);    // Display the result
        }
     }
     httpRequest.send(null);
  }
  </script>
</head>

<body onload="makeRequest(186)">
<h1>Employee Lookup</h2>
<div id="descriptionNode">This page finds an employee</div>
</body>
</html>
```

6 . Reload it in the browser. The new value is displayed. Click **OK** to dismiss the alert window.



Note: You may also need to flush the browser cache to see the changed value.

Summary

In this tutorial, you have learned how to:

- Create a connection
- Use Database Resident Connection Pooling
- Fetch data
- Use bind variables
- Use transactions
- Call PL/SQL
- Improve query performance
- Use LOBs to upload and query images
- Incorporate AJAX into your page

Appendix: PHP Primer

PHP is a dynamically typed scripting language. It is most often seen in Web applications but can be used to run command-line scripts. Basic PHP syntax is simple to learn. It has familiar loops, tests, and assignment constructs. Lines are terminated with a semi-colon.

Strings can be enclosed in single or double quotes:

```
'A string constant'
"another constant"
```

Variable names are prefixed with a dollar sign. Things that look like variables inside a double-quoted string will be expanded:

```
"A value appears here: $v1"
```

Strings and variables can also be concatenated using a period.

```
'Employee ' . $ename . ' is in department ' . $dept
```

Variables do not need types declared:

```
$count = 1;
$ename = 'Arnie';
```

Arrays can have numeric or associative indexes:

```
$a1[1] = 3.1415;
$a2['PI'] = 3.1415;
```

Strings and variables can be displayed with an echo or print statement. Formatted output with printf() is also possible.

```
echo 'Hello, World!';
echo $v, $x;
print 'Hello, World!';
printf("There is %d %s", $v1, $v2);
```

The var_dump() function is useful for debugging.

```
var_dump($a2);
```

Given the value of $a2 assigned above, this would output:

```
array(1) {
  ["PI"]=>
  float(3.1415)
}
```

Code flow can be controlled with tests and loops. PHP also has a switch statement.
The if/elseif/else statements look like:

```
if ($sal > 900000) {
  echo 'Salary is way too big';
}
elseif ($sal > 500000) {
  echo 'Salary is huge';
}
  else {
  echo 'Salary might be OK';
}
```

This also shows how blocks of code are enclosed in braces.

A traditional loop is:

```
for ($i = 0; $i < 10; $i++) {
  echo $i;
}
```

This prints the numbers 0 to 9. The value of $i is incremented in each iteration. The loop stops when the test condition evaluates to false. You can also loop with while or do while constructs.

The foreach command is useful to iterate over arrays:

```
$a3 = array('Aa', 'Bb', 'Cc');
foreach ($a3 as $v) {
  echo $v;
}
```

This sets $v to each element of the array in turn.

A function may be defined:

```
function myfunc($p1, $p2) {
  echo $p1, $p2;
  return $p1 + $p2;
}
```

Functions may have variable numbers of arguments, and may or may not return values. This function could be called using:

```
$v3 = myfunc(1, 3);
```

Function calls may appear earlier than the function definition.

Sub-files can be included in PHP scripts with an include() or require() statement.

```
include("foo.php");
require("bar.php");
```

A require() will generate a fatal error if the script is not found.

Comments are either single line:

```
// a short comment
```

or multi-line:

```
/*
  A longer comment
*/
```

PHP scripts are enclosed in *<?php* and *?>* tags.

```
<?php
  echo 'Hello, World!';
?>
```

When a Web server is configured to run PHP files through the PHP interpreter, loading the script in a browser will cause the PHP code to be executed and all output to be streamed to the browser.

Blocks of PHP code and HTML code may be interleaved. The PHP code can also explicitly print HTML tags.

```
<?php
  require('foo.php');
  echo '<h3>';
  echo 'Full Results';
  echo '</h3>';
  $output = bar(123);
?>
<table border="1">
  <tr>
    <td>
     <?php echo $output ?>
    </td>
  </tr>
</table>
```

Many aspects of PHP are controlled by settings in the php.ini configuration file. The location of the file is system specific. Its
location, the list of extensions loaded, and the value of all the initialization settings can be found using the phpinfo() function:

```
<?php
  phpinfo();
?>
```

Values can be changed by editing phpl.ini and restarting the Web server. Some values can also be changed within scripts by using the ini_set() function.

A list of the various oci functions include the following:

oci_bind_array_by_name	Binds PHP array to Oracle PL/SQL array by name
oci_bind_by_name	Binds the PHP variable to the Oracle placeholder
oci_cancel	Cancels reading from cursor
oci_close	Closes Oracle connection
oci_commit	Commits outstanding statements
oci_connect	Establishes a connection to the Oracle server
oci_define_by_name	Uses a PHP variable for the define-step during a SELECT
oci_error	Returns the last error found
oci_execute	Executes a statement
oci_fetch_all	Fetches all rows of result data into an array
oci_fetch_array	Returns the next row from the result data as an associative or numeric array, or both
oci_fetch_assoc	Returns the next row from the result data as an associative array
oci_fetch_object	Returns the next row from the result data as an object
oci_fetch_row	Returns the next row from the result data as a numeric array
oci_fetch	Fetches the next row into result-buffer
oci_field_is_null	Checks if the field is NULL
oci_field_name	Returns the name of a field from the statement
oci_field_precision	Tell the precision of a field
oci_field_scale	Tell the scale of the field
oci_field_size	Returns the size of the field
oci_field_type_raw	Tell the raw Oracle data type of the field
oci_field_type	Returns data type of the field
oci_free_statement	Frees all resources associated with statement or cursor
oci_internal_debug	Enables or disables internal debug output
oci_new_collection	Allocates new collection object
oci_new_connect	Establishes a new connection to the Oracle server
oci_new_cursor	Allocates and returns a new cursor (statement handle)

oci_new_descriptor	Initializes a new empty LOB or FILE descriptor
oci_num_fields	Returns the number of result columns in a statement
oci_num_rows	Returns number of rows affected during statement execution
oci_parse	Prepares Oracle statement for execution
oci_password_change	Changes password of Oracle's user
oci_pconnect	Connect to an Oracle database using a persistent connection
oci_result	Returns a field's value from a fetched row
oci_rollback	Rolls back outstanding transaction
oci_server_version	Returns server version
oci_set_prefetch	Sets number of rows to be prefetched
oci_statement_type	Returns the type of an OCI statement