IT 4153 Advanced Database

Module 2 Introduction to SQL Procedures and Functions

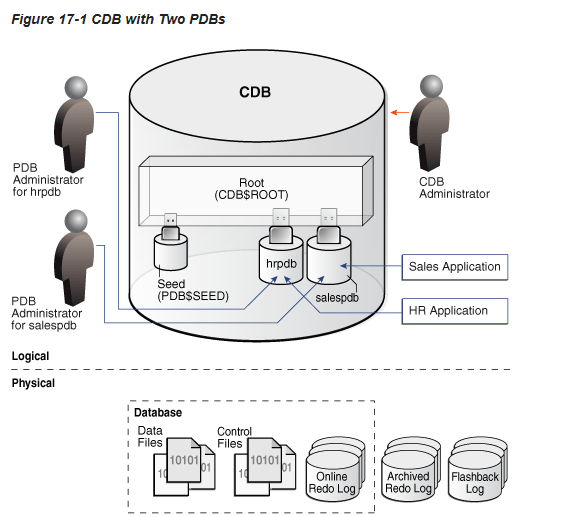
|  |
| --- |
| **Introduction and Module Summary** |
| In this module, you will learn benefits of using procedural SQL, write, execute and test SQL procedures and functions. |
| **Objectives and Outcomes** |
| This module directly supports **highlighted** course outcome(s)  Students who complete this course successfully will be able to   1. Describe current and emerging database models and technologies; 2. **Develop functions and procedures for data manipulation and database access auditing;** 3. Describe database monitoring and performance tuning; 4. Describe database security and administration issues, including backup and recovery; 5. Explain the concepts of data warehousing and data mining   **Module outcomes and activities:**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | After completing this module, students will be able: | to list benefits of procedural SQL | differentiate when to use function and when to use procedures | declare variables  develop procedural SQL code | test and execute procedural SQL code | | Readings | introduced | introduced | introduced | introduced | | Practice exercises | reinforced | reinforced | reinforced | reinforced | | Lab | reinforced | mastered | reinforced | mastered | |
| **Assigned Reading** |
| 1. PL/SQL Language Fundamentals <http://docs.oracle.com/database/121/LNPLS/fundamentals.htm#LNPLS99920PL/> 2. SQL Subprograms <http://docs.oracle.com/database/121/LNPLS/subprograms.htm#LNPLS008> |
| **Optional Reading** |
| 1. Introduction to PL/SQL part I and II (except cursors) <http://w2.syronex.com/jmr/edu/db/introduction-to-plsql/> 2. Variables and Types <http://infolab.stanford.edu/~ullman/fcdb/oracle/or-plsql.html#variables%20and%20types> 3. Oracle Procedures <http://www.psoug.org/reference/procedures.html> 4. Oracle Functions <http://www.psoug.org/reference/functions.html> 5. PL/SQL FAQ <http://www.orafaq.com/wiki/PL/SQL_FAQ> |
| **Assessments and Assignments** |
| 1. Lab (10 points) |
| **Topics** |
| Open the navigation pane |
| **Module Checklist** |
| This is the suggested order of the completion of this module.  Save a copy of this file on your computer and make notes in this document while you are completing your assignments. Use the table below to keep track of your progress.   |  |  | | --- | --- | | **Activity** | **Completion** | | Read this module and assigned materials (2 hour) | NO | | Complete all exercises from the module (2 hours) | NO | | Complete lab (2 hours) | NO | | Complete Module feedback at the end of the module | NO | | Read feedback provided for your discussion and lab. | NO | |
|  |

# Oracle Database, Schema, Tablespace, and Owner

## Oracle pluggable database

The multitenant architecture enables an Oracle database to function as a multitenant container database (CDB) that includes zero, one, or many customer-created pluggable databases (PDBs). A PDB is a portable collection of schemas, schema objects, and nonschema objects that appears to an Oracle Net client as a non-CDB. All Oracle databases before Oracle Database 12c were non-CDBs.

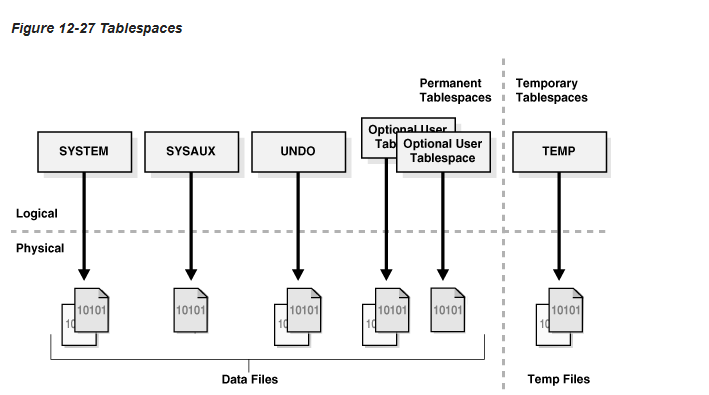
Each container has a unique ID and name in a CDB. You will learn about CDB in the later modules.

 <https://docs.oracle.com/database/121/CNCPT/cdbovrvw.htm#CNCPT89237>

Every CDB has the following containers:

* Exactly one root CDB$ROOT that keeps metadata and common users (users known in every container).
* Exactly one seed PDB PDB$SEED to create new PDBs (think about it as a blueprint).
* Zero or more user-created PDBs (e.g. PDBORCL in CCSE Vlab installation)

## Tablespace



<https://docs.oracle.com/database/121/CNCPT/logical.htm#CNCPT402> is a container for tables, indexes, etc. A tablespace is made up of one or more data files. A database consists of one or more tablespaces.

* Default tablespaces in Oracle: SYSTEM, SYSAUX, UNDO, TEMP, etc

## Schema

<http://docs.oracle.com/database/121/CNCPT/tablecls.htm#CNCPT111>

collection of logical structures of data, or schema objects

In Oracle a schema is owned by a database user and has the same name as that user, each user owns a single schema.

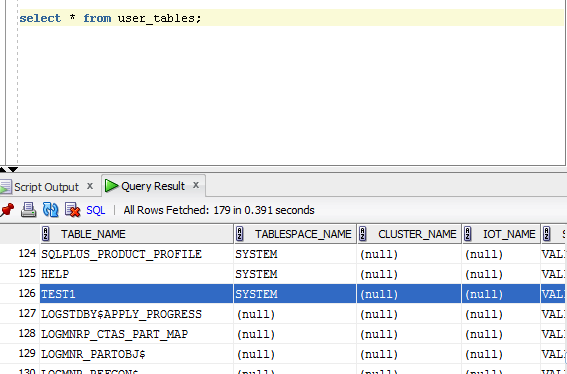
Examples:

**SYSTEM** isa default generic database administrator account for Oracle databases. When you logon using username system, you have a default schema SYSTEM and all objects you create e.g. CREATE TABLE test1 (id NUMBER(2,0)); are created in that schema, unless you change it.

A data dictionary view USER\_TABLES holds information about all user tables. If you run

select \* from user\_tables;

you will find all tables in the user schema. The result of this query for system user shows newly created ***test1*** table.



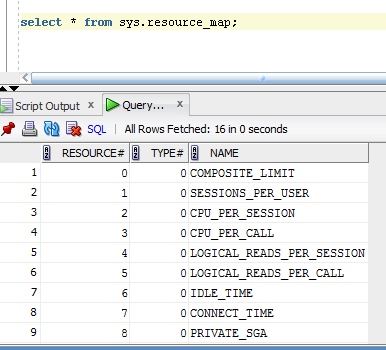
The fully qualified name for this table is SYSTEM.TEST1. Note that Oracle keeps all values in system tables in all CAPs.

Creating table test1 or Test1 or TEST1 will create the table with the same name and Oracle will keep it as ***TEST1***. If the account have permisssion to create tables in other schemas, you should be able to do so by explicitly stating the schema name e.g. CREATE TABLE OTHERACCOUNT.test1 (id NUMBER(2,0));

To select data from tables owned by others, you have to provide schema name.

select \* from sys.resource\_map;

Returns 16 rows



Oracle has a sample schema we will use later in the class. One of the users in that schema is OE. OE.CUSTOMERS is CUSTOMERS table owned by user OE.

OE.CUSTOMERS – 319 rows

There is another user SH and this user owns SH.CUSTOMERS table – 55500 rows. A data dictionary view DBA\_TABLES holds information about all user tables.

select \* from dba\_tables where table\_name='CUSTOMERS';

returns all tables with name CUSTOMERS (one form OE and another one from SH

select \* from dba\_tables where owner='OE';

returns all tables owned by OE

# Oracle Procedural SQL

SQL shortcomings

* Doesn’t support execution of stored procedures based on logical condition
* Fails to support looping operations

Solutions

* Embedded SQL can be called from within procedural programming languages
* Shared Code is isolated and used by all application programs.

Oracle uses Procedural SQL or PL/SQL

SQL Server uses Transact SQL or T-SQL

Procedural SQL allows the use of procedural code and SQL statements that are stored within the database.

The procedural code is executed by the DBMS when it is invoked by the end user.

End users can use PL/SQL to create:

* Triggers
* Stored procedures
* Functions

## Structure of PL/SQL

Block Header

* optional
* block name
* block type (procedure, function, etc)
* if omitted, then "anonymous block"

Declaration section

* optional
* defines and initializes variables and cursors

Execution section

* PL/SQL or SQL statements

Exception section

* handling errors during execution

Type of structures

* Anonymous block
* Named block
* Function
  + accepts in/out parameters
  + returns a single value
* Procedure
  + accepts in/out parameters
* Package
  + named collection of PL/SQL functions and/or procedures

## PL/SQL: Declaring Variables

**Scalar variable**

variable\_name type [CONSTANT] [NOT NULL] [:=initial\_value]

Examples:

salary NUMBER := 0;

hiredate DATE;

**Explicit declaration**

variable\_name [schema.]table\_name.column\_name%TYPE

Examples:

emp\_name emp.name%TYPE;

***emp\_name*** will have the same data type as ***NAME*** column in the table ***EMP***

cust\_id OE.CUSTOMERS.CUSTOMERID%TYPE

***cust\_id*** will have the same type as ***CUSTOMERID*** column in the table ***CUSTOMERS*** in the schema ***OE***

**Declaration by reference**

variable\_name pl/sql\_variable%TYPE

Example

firedate hiredate%TYPE;

***firedate*** variable will have the same data type as ***hiredate*** variable

**Declaration a record type**

TYPE type\_name IS RECORD ( variable\_name type [NOT NULL] [:=initial\_value] [ , variable\_name type [NOT NULL] [:=initial\_value] ...])

Example:

TYPE DeptRec IS RECORD (

dept\_id dept.deptno%TYPE,

dept\_name VARCHAR2(14),

dept\_loc VARCHAR2(13)

);

department DeptRec;

***DeptRecord*** is a record that consists of department ID the same type as ***deptno*** from the ***DEPT*** table, and two strings: one for the department name and another for the location. ***DeptRecord*** will be used as a blue print to create variables of this type.

***department*** is declared as a ***DeptRecord*** type, so it can hold department ID, department name and department location.

To reference part of a record use a dot after the variable name: record\_name.field\_name

department.dept\_name := 'Information Technology';

**Declaring a record based on a table row**

record\_name [schema.] table\_name%ROWTYPE

customer\_type OE.CUSTOMERS%ROWTYPE

***customer\_type*** will have the same type as a in the table ***CUSTOMERS*** in the schema ***OE***

## PL/SQL Block

Turn server output on (see module 1 for details).

The basic unit of a PL/SQL source program is the block, which groups related declarations and statements.

Syntax:

*DECLARE -- Declarative part (optional)*

*-- Declarations of local types, variables, & subprograms*

*BEGIN -- Executable part (required)*

*-- Statements (which can use items declared in declarative part)*

*[EXCEPTION -- Exception-handling part (optional)*

*-- Exception handlers for exceptions (errors) raised in executable part]*

*END;*

Example without DECLARE

begin

dbms\_output.put\_line('Hello');

end;

/

Example with DECLARE

DECLARE

x NUMBER := 0;

BEGIN

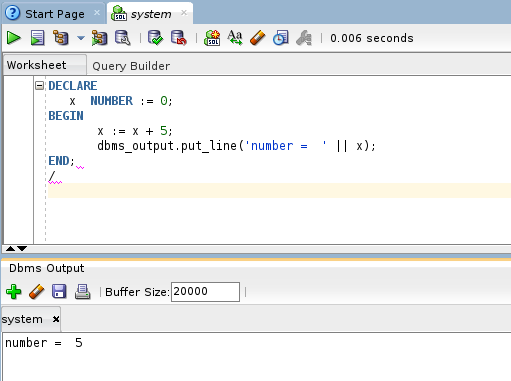
x := x + 5;

dbms\_output.put\_line('number = ' || x);

END;

/

We first declare a variable ***x*** type number and assign zero to it. Inside the block ***x*** is increased by five and the result of addition is displayed. Two pipe characters || are used to concatenate (append) the results of addition to the string "number ="



Example with DECLARE and EXCEPTION

DECLARE

x NUMBER := 0;

BEGIN

x := 5 / x;

dbms\_output.put\_line('number = ' || x);

EXCEPTION

WHEN ZERO\_DIVIDE THEN

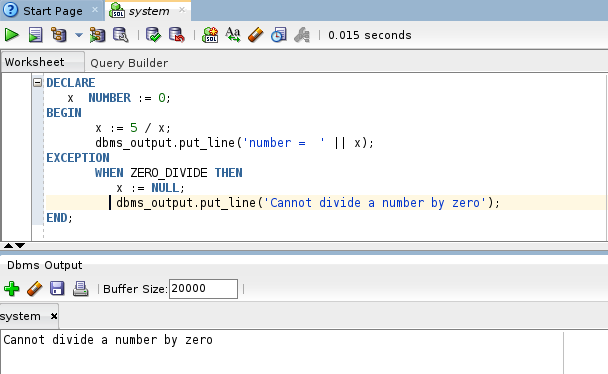
x := NULL;

dbms\_output.put\_line('Cannot divide a number by zero');

END;

/

The same example, but instead of addition we will use division. If the eception handling is enabled, then " Cannot divide a number by zero" will be displayed and the new value of ***x*** will be null. The list of predefined exceptions <http://docs.oracle.com/database/121/LNPLS/errors.htm#LNPLS00703>



If the exception is not handled, then error message will be displayed

*Error report -*

*ORA-01476: divisor is equal to zero*

*ORA-06512: at line 4*

*01476. 00000 - "divisor is equal to zero"*

*\*Cause:*

*\*Action:*

As developers you must catch and handle all exceptions in order to create reliable and secure applications.

## PL/SQL subprogram

A PL/SQL subprogram is a named PL/SQL block that can be invoked repeatedly. If the subprogram has parameters, their values can differ for each invocation. PL/SQL has two types of subprograms, procedures and functions. A function returns a result.

### SQL Built in Functions

SQL functions are built into Oracle Database and are available for use in SQL statements.

DUAL is a table in ORACLE that contains ONE column and ONE record. It is used to makes some calculations more convenient.

**String Functions**

* upper(s), lower(s) convert string s to upper/lower-case
* initcap(s) capitalize first letter of each word
* trim(s), rtrim(s) remove blank char. from left/right
* substr(s,start,len) sub-string of length len from position start
* length(s) length of s

**Date Functions**

* sysdate current date (on Oracle server)
* to\_date(date, format) date formatting
* Number Functions
* round(x) round real number x to integer
* mod(n,p) n modulus p abs(x) absolute value of x
* dbms\_random.random() generate a random integer
* Read more about date and time at <http://infolab.stanford.edu/~ullman/fcdb/oracle/or-time.html>

**Type Conversion Functions**

* to\_char() convert to string
* to\_date() convert to date
* to\_number() convert to number

Other Useful Functions

* user returns name of user who runs the session
* sysdate returns system date

Examples

SELECT length('asdfgh') from DUAL; --the length of the string asdfgh

SELECT user from DUAL; --user who runs this code

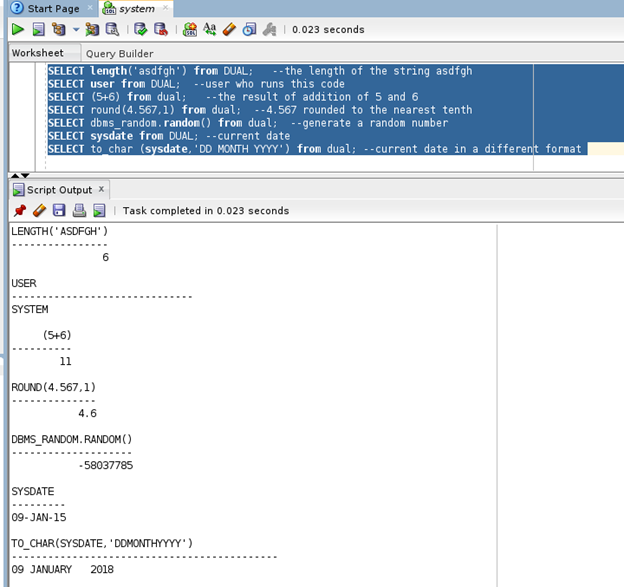
SELECT (5+6) from dual; --the result of addition of 5 and 6

SELECT round(4.567,1) from dual; --4.567 rounded to the nearest tenth

SELECT dbms\_random.random() from dual; --generate a random number

SELECT sysdate from DUAL; --current date

SELECT to\_char (sysdate,'DD MONTH YYYY') from dual; --current date in a different format



**Time and Date functions**

Current date

select sysdate from dual;

Yesterday

select sysdate - 1 from dual;

Tomorrow

select sysdate + 1 from dual;

The date format element D returns the number of the day of the week (1-7).

select to\_char(sysdate,'D') from dual;

MM returns two-digit numeric abbreviation of month (01-12; JAN = 01)

select to\_char(sysdate,'MM') from dual;

Examples of the output

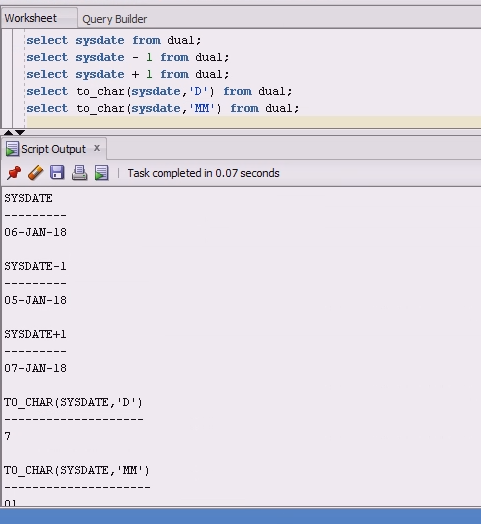
SYSDATE: 06-JAN-18

SYSDATE-1: 05-JAN-18

SYSDATE+1: 07-JAN-18

to\_char(sysdate,'D') : 7 (Saturday)

to\_char(sysdate,'MM') 01 (January)



### PL/SQL Stored Procedures

Named collection of procedural and SQL statements stored in database and can be invoked by name

Syntax

*CREATE OR REPLACE PROCEDURE procedure\_name (argument IN/OUT data-type, etc)   
IS/AS BEGIN  
 DECLARE variable name and data type  
 PL/SQL or SQL statements;  
END;*

Invoked with EXEC

*EXEC store\_procedure\_name (parameter, parameter, …)*

Example 1:

CREATE OR REPLACE PROCEDURE display as

x NUMBER := 0;

BEGIN

x := x + 5;

dbms\_output.put\_line('number = ' || x);

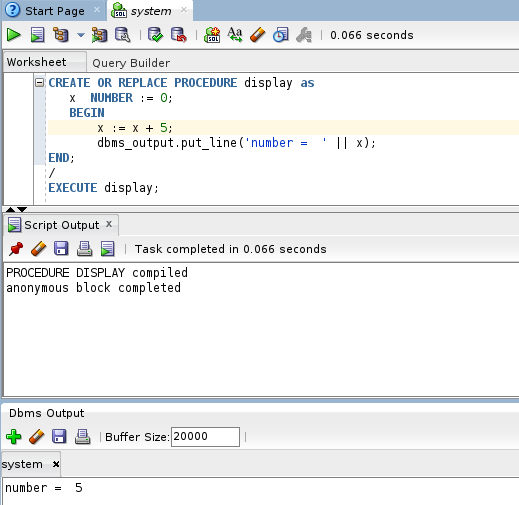
END;

/

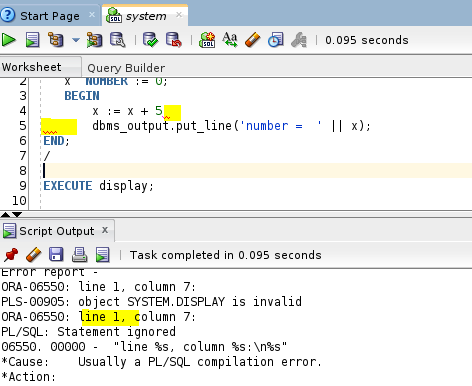
EXECUTE display;

This is the same code we used before, but the block is named now and can be invoked by using execute.

You have to use / between the procedure declaration and the procedure invocation. The slash tells the interpreter to execute the loaded script.

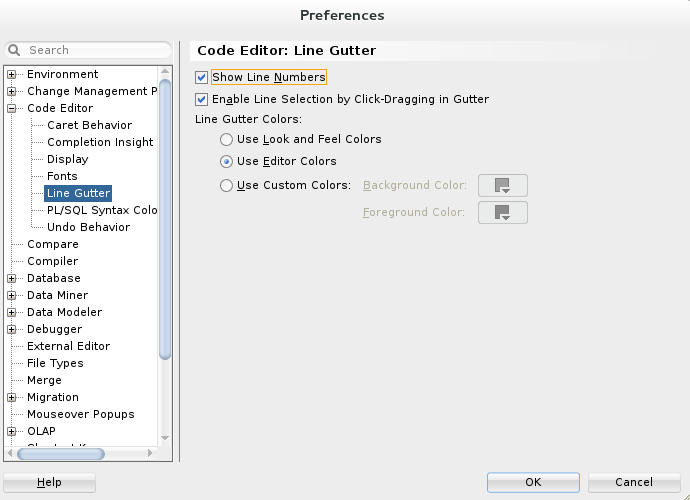


NOTE: If you received an error or warning while compiling or executing your procedure

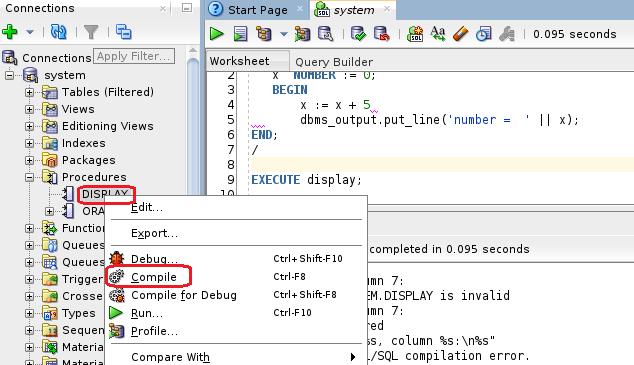


It is useful to show line numbers. On the SQL Developer menu choose View - Preferences and under Code Editor/Line Gutter select ***Show*** ***Line Numbers***

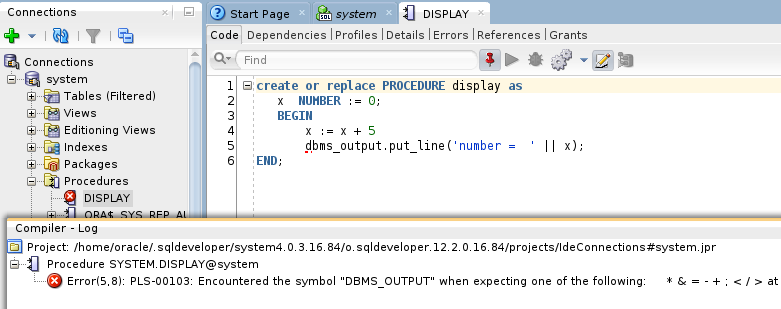
Notice that SQL developer shows you location of the error, while Oracle Server reports that the error ocured on the line 1.



To find a better explanation of your error, expand connection tree and find the procedure with name display, right-click and choose compile



Double-click the error and line of the code that contains an error message will be highlighted. In this case the error was using missing semicolon.



To delete procedure:

DROP PROCEDURE display;

Note: entire code unit is terminated with a slash (/). The slash is required when running the PL/SQL in a SQL script or at the SQL Command Line prompt.

**PL/SQL Procedures with Parameters**

You can display the output in the Script Output window if you add SET SERVEROUTPUT ON; statement.

CREATE OR REPLACE PROCEDURE display( x NUMBER)

AS

y NUMBER;

BEGIN

y := x + 5;

dbms\_output.put\_line('number = ' || y);

END;

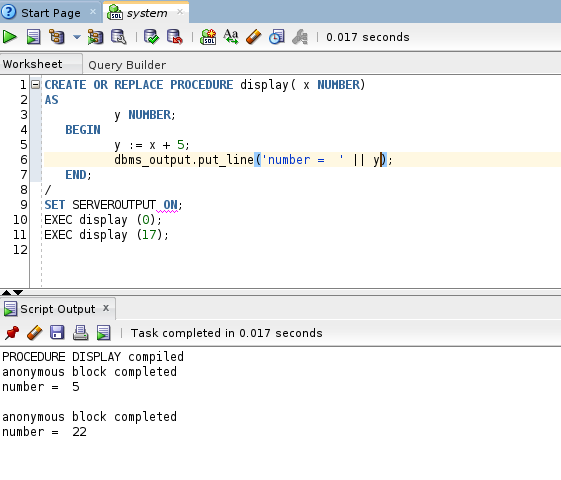
/

SET SERVEROUTPUT ON;

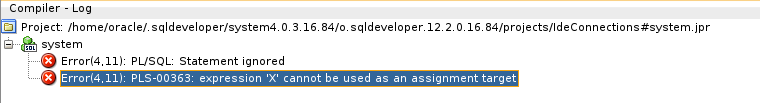
EXEC display (0);

EXEC display (17);

This is the same code as above, but by providing parameter, we can execute this code for different values of ***x***. We have to declare a variable ***y***, Oracle does not allow changing the value of input parameters.



If we try to add 5 to x, an error message



Error(4,11): PLS-00363: expression 'X' cannot be used as an assignment target

You can compare input parameters with system variables or any values from stored tables.

CREATE OR REPLACE PROCEDURE guess\_current\_month( str VARCHAR2)

AS

BEGIN

dbms\_output.put\_line('Your guess: ' || str);

IF (str = trim(to\_char(SYSDATE, 'Month'))) THEN

dbms\_output.put\_line('Correct');

ELSE

dbms\_output.put\_line('Try again.');

END IF;

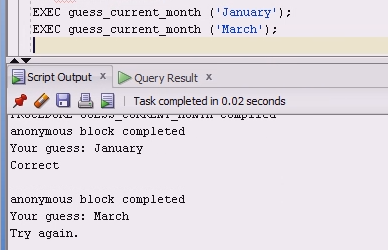
END;

/

SET SERVEROUTPUT ON;

EXEC guess\_current\_month ('January');

EXEC guess\_current\_month ('March');



Pay attention that comparison operator is = not == as in most programming languages. .

### PL/SQL Stored Functions

Only functions can return a value to the caller, the procedures cannot do that.

|  |  |  |
| --- | --- | --- |
|  | **Function** | **Procedure** |
| **Parameters** | **input, output** | **input, output** |
|  |  |  |
| **Returns value** | **yes** | **no** |
|  |  |  |
| **Can be called within SQL** | **yes** | **no** |

The function is a named group of procedural and SQL statements that returns a value.

Synatx:

*CREATE FUNCTION function\_name (argument IN data-type, etc)  
RETURN data-type  
AS BEGIN  
 PL/SQL statements;  
 RETURN (value); ……  
END;*

Invoked as

*SELECT function\_name FROM DUAL;*

Can be invoked from within stored procedures, triggers or SQL statements.

Example:

CREATE OR REPLACE FUNCTION display

RETURN NUMBER

AS

x NUMBER := 0;

BEGIN

x := x + 5;

return x;

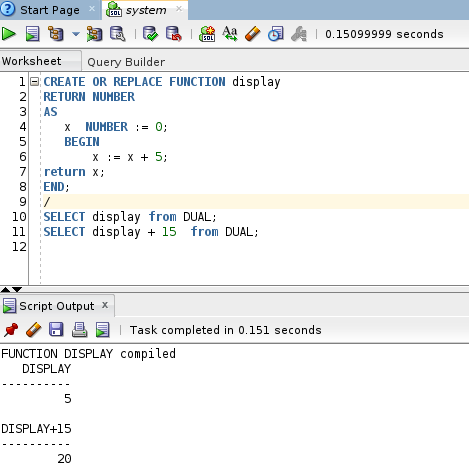
END;

/

SELECT display from DUAL;

SELECT display + 15 from DUAL;

NOTE: You have to drop procedure display before you can create function display. You cannot have two PL/SQL blocks with the same name.



To add date to the display

1. Use to\_char(SYSDATE) to convert date format to a string
2. Convert X to a string <http://docs.oracle.com/cd/B19306_01/server.102/b14200/sql_elements004.htm>
3. Change return type to VARCHAR2

CREATE OR REPLACE FUNCTION display

RETURN VARCHAR2

AS

x NUMBER := 0;

BEGIN

x := x + 5;

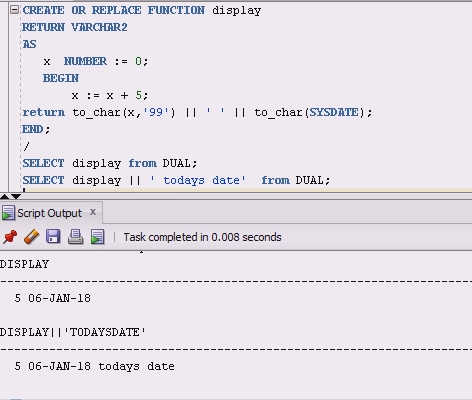
return to\_char(x,'99') || ' ' || to\_char(SYSDATE);

END;

/

SELECT display from DUAL;

SELECT display || ' todays date' from DUAL;



### PL/SQL Statements

**IF-THEN ELSE Statement**

Syntax

*IF condition THEN*

*sequence\_of\_statements*

*ELSE*

*sequence\_of\_statements*

*END IF;*

Examples

CREATE OR REPLACE FUNCTION display\_if (x NUMBER)

RETURN NUMBER

AS

BEGIN

IF ( x > 0) Then

return x + 10;

ELSE

return 0;

END IF;

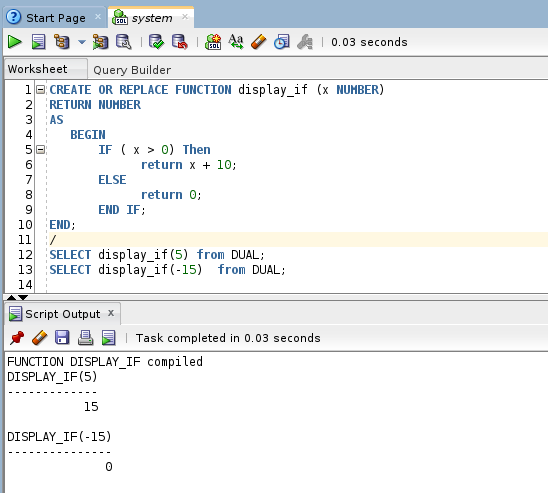
END;

/

SELECT display\_if(5) from DUAL;

SELECT display\_if(-15) from DUAL;

This function accepts a number as input parameter and then if the number is positive, then 10 is added. Otherwise the function returns zero.



Based on the condition, you can run SQL staments on tables.

IF sales > 1000 THEN

UPDATE payroll SET salary = salary \* 1.10 ;

END IF;

**IF-THEN-ELSE Statement**

IF condition THEN

sequence\_of\_statements

[ELSEIF expression THEN

sequence\_of\_statements]

[ELSE expression THEN

sequence\_of\_statements]

END IF;

IF sales > 1000 THEN

UPDATE payroll SET salary = salary \* 1.10 ;

ELSEIF sales BETWEEN 500 AND 1000 THEN

UPDATE payroll SET salary = salary \* 1.05 ;

ELSE

UPDATE payroll SET salary = salary \* 1.01 ;

END IF;

**CASE Statement**

CASE grade

WHEN 'A' THEN dbms\_output.put\_line('Excellent');

WHEN 'B' THEN dbms\_output.put\_line('Very Good');

WHEN 'C' THEN dbms\_output.put\_line('Good');

WHEN 'D' THEN dbms\_output.put\_line('Fair');

WHEN 'F' THEN dbms\_output.put\_line('Poor');

ELSE dbms\_output.put\_line('No such grade');

END CASE;

**Loops**

Basic loop

Syntax:

*LOOP*

*sequence\_of\_statements*

*END LOOP;*

Example:

CREATE OR REPLACE PROCEDURE display\_all( x NUMBER)

AS

loop\_counter NUMBER := x;

BEGIN

LOOP

IF (loop\_counter > 0) THEN

dbms\_output.put(loop\_counter || ' ');

loop\_counter := loop\_counter - 1;

ELSE

dbms\_output.put\_line(' ');

RETURN;

END IF;

END LOOP;

END;

/

SET SERVEROUTPUT ON;

EXEC display\_all (5);

EXEC display\_all (17);

EXEC display\_all (1);

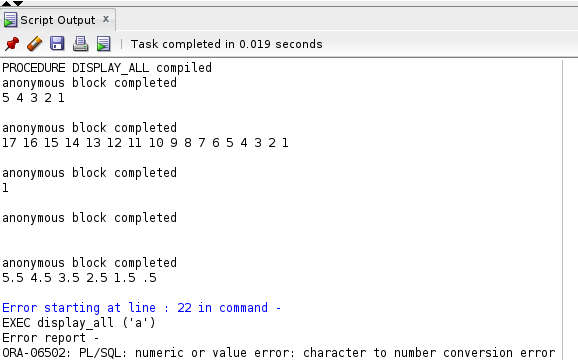
EXEC display\_all (-5);

EXEC display\_all (5.5);

EXEC display\_all ('a');

This code will accept a number, assign it to the variable ***loop\_counter***, display the value of the ***loop\_counter*** and repeatedly subtract one from the value until the value is not less than zero.

Note that to display the value of the ***loop\_counter*** variable we used ***put***, not ***put\_line***. The method put is used to display outputs on the same line. You have to remember that Oracle output is buffered, so to see the values, you have to push a new line character or use ***dbms\_output.put\_line(' ');*** before the RETURN.



When the procedure reseives negative number, there is no output. If the parameter is not a number, then error message is returned. We should add exception handling, we will do it later.

While loop

Syntax:

*WHILE condition LOOP*

*sequence\_of\_statements*

*END LOOP;*

The same code, but using the WHILE loop

CREATE OR REPLACE PROCEDURE display\_all( x NUMBER)

AS

loop\_counter NUMBER := x;

BEGIN

WHILE loop\_counter > 0 LOOP

dbms\_output.put(loop\_counter || ' ');

loop\_counter := loop\_counter - 1;

END LOOP;

dbms\_output.put\_line(' ');

RETURN;

END;

/

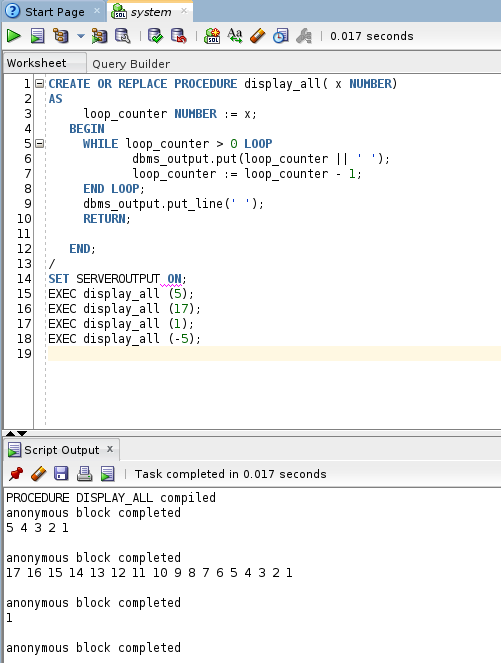
SET SERVEROUTPUT ON;

EXEC display\_all (5);

EXEC display\_all (17);

EXEC display\_all (1);

EXEC display\_all (-5);



FOR loop

Note: do not need explicitly declare the loop counter

Syntax:

*FOR counter IN [REVERSE] lower\_bound..higher\_bound LOOP*

*sequence\_of\_statements*

*END LOOP;*

The same code, but using the WHILE loop will produce the same output.

CREATE OR REPLACE PROCEDURE display\_all( x NUMBER)

AS

loop\_counter NUMBER;

BEGIN

FOR loop\_counter IN REVERSE 1..x LOOP

dbms\_output.put(loop\_counter || ' ');

END LOOP;

dbms\_output.put\_line(' ');

RETURN;

END;

/

SET SERVEROUTPUT ON;

EXEC display\_all (5);

EXEC display\_all (17);

EXEC display\_all (1);

EXEC display\_all (-5);

## SQL in PL/SQL

Create a table, insert data in the table and then read that data using PL/SQL stored procedure

Create table *ProductTable*

|  |  |  |  |
| --- | --- | --- | --- |
| **ProductID** | **ProductName** | **ListPrice** | **Category** |
| int, not null | Not null, length 50 | $ | int, not null |

Add values to table ProductTable

|  |  |  |  |
| --- | --- | --- | --- |
| **ProductID** | **Name** | **ListPrice** | **Category** |
| 299 | Chest | $99.99 | 10 |
| 300 | Wave Cruiser | $49.99 | 11 |
| 301 | Megaland Play Tent | $59.99 | 11 |
| 302 | Wind-Up Water Swimmers | $2.00 | 11 |
| 303 | Garmin Pocket or Vehicle GPS Navigator | $609.99 | 12 |

CREATE TABLE ProductTable(

ProductID INTEGER NOT NULL primary key,

ProductName VARCHAR(50) NOT NULL,

ListPrice NUMBER(10,2),

Category INTEGER NOT NULL

);

/

INSERT INTO ProductTable VALUES(299,'Chest',99.99,10);

INSERT INTO ProductTable VALUES(300,'Wave Cruiser',49.99,11);

INSERT INTO ProductTable VALUES(301,'Megaland Play Tent',59.99,11);

INSERT INTO ProductTable VALUES(302,'Wind-Up Water Swimmers',2.00,11);

INSERT INTO ProductTable VALUES(303,'Garmin Pocket or Vehicle GPS Navigator',609.99,12);

CREATE OR REPLACE PROCEDURE sqltest AS

V\_Category NUMBER;

BEGIN

-- comments

SELECT Category INTO v\_Category FROM ProductTable

WHERE ProductId = 300;

dbms\_output.put\_line('Product category for product #300 is ' || V\_Category);

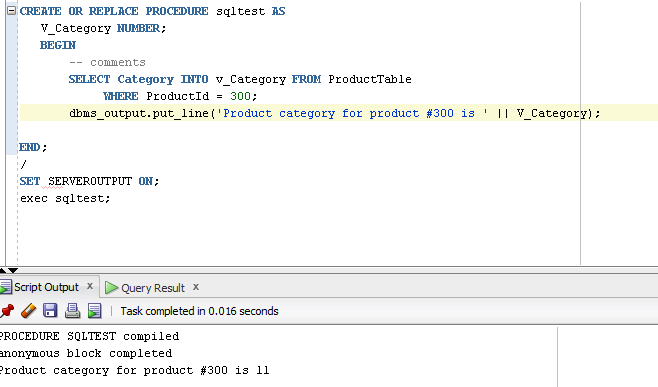
END;

/

set serveroutput on;

exec sqltest;

The following code will find resource type from the table ProductTable. To check the result, run SELECT Category FROM ProductTable WHERE ProductId = 300; The result will be 11.



## Oracle sequence

Oracle SEQUENCE is a database object from which users may generate unique integers.

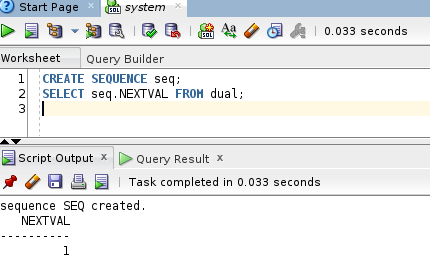
Synatx

*CREATE SEQUENCE seqname [ INCREMENT increment ] [ MINVALUE minvalue ] [ MAXVALUE maxvalue ] [ START start ] [ CACHE cache ] [ CYCLE ]*

Example

CREATE SEQUENCE seq;

SELECT seq.NEXTVAL FROM dual;



## Managing Errors with Exception Handler

DECLARE

emp\_column VARCHAR2(30) := 'last\_name';

temp\_var VARCHAR2(30);

BEGIN

temp\_var := emp\_column;

SELECT COLUMN\_NAME INTO temp\_var FROM USER\_TAB\_COLS WHERE TABLE\_NAME = 'EMPLOYEES' AND COLUMN\_NAME = UPPER(emp\_column);

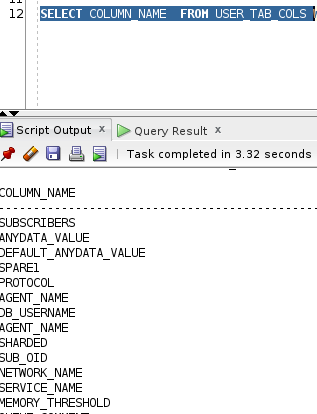
EXCEPTION

WHEN NO\_DATA\_FOUND THEN

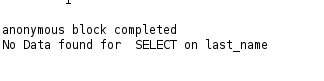
DBMS\_OUTPUT.PUT\_LINE ('No Data found for SELECT on ' || temp\_var); END;

/

This code checks data dictionary view USER\_TAB\_COLS that describes the columns of the tables, views, and clusters owned by the current user. If



It checks that tbale EMPLOYEES has a column LAST\_NAME.



**Lab PL/SQL**

**Before you start this lab assignment, you have to run all examples from this module and understand the purpose of each line of code.**

**You have to include the text of your code, not only screenshots. To test your answers the code will be copied from your report and run in SQL Developer.**

**Objectives** Learn how to write PL/SQL procedures, and functions

1. (3 points for code and screenshots of testing your code)

The following code will concatenate two strings **str1** and **str2**

CREATE OR REPLACE PROCEDURE concatenate\_strings as

str1 VARCHAR2(25) := 'bit';

str2 VARCHAR2(25) := 'coin';

result VARCHAR2(50);

BEGIN

result := str1 || str2;

dbms\_output.put\_line('The result is ' || result);

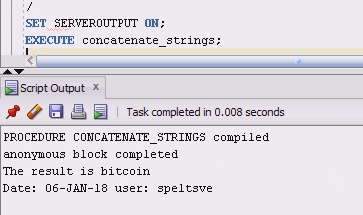
dbms\_output.put\_line('Date: ' || SYSDATE || ' user: ' || SYS\_CONTEXT('USERENV','OS\_USER'));

END;

/

SET SERVEROUTPUT ON;

EXECUTE concatenate\_strings;

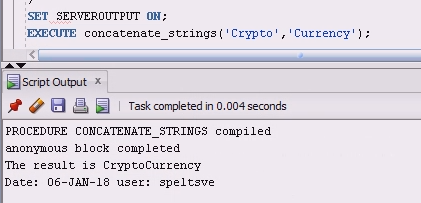


Modify this procedure to accept two string parameters. Example of the output for

EXECUTE concatenate\_strings('Crypto','Currency');

The result is CryptoCurrency

26-Jan-18

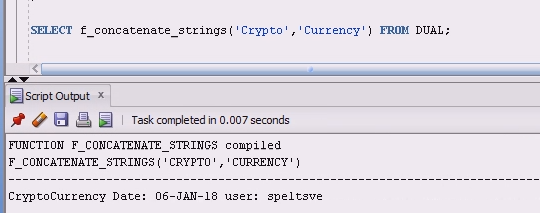


*Provide code and screenshot showing successful execution and results*

1. (3 points for code and screenshots of testing your code) Write a function f\_concatenate\_strings that will do the same as concatenate\_strings procedure from #1, but can be called from a select statement. Examples of output:

SELECT f\_concatenate\_strings('Crypto','Currency') FROM DUAL;

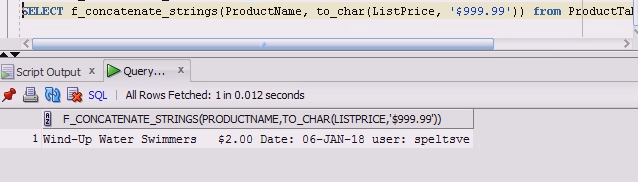
Will return CryptoCurrency Date: 06-JAN-18 user: speltsve



SELECT f\_concatenate\_strings(ProductName, to\_char(ListPrice, '$999.99')) from ProductTable WHERE ProductID=302;

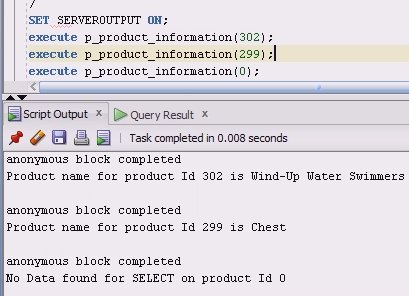
First, the table ProductTable, then product with ID 302 will be located. Product name of that product will be concatenated with the price and current date.

Will return Wind-Up Water Swimmers $2.00 06-JAN-18 user: speltsve



*Provide code and screenshot showing successful execution and results. You have to provide at least two test cases.*

1. (4 points for code and screenshots) Create a procedure that accepts product ID as a parameter and returns the name of the product from ***ProductTable*** table. Add exception handling to catch if product ID is not in the table.



Use ***ProductTable*** created earlier in the module.

*Provide code and screenshot showing successful execution and results. You have to provide at least two test cases.*

**Feedback:**

Difficulty (-2 - too easy ... 0 - just right ... 2 - too hard)

Interest level (-2 - low interest ... 0 - just right ... 2 - high interest)

Time to complete (min)

Make a suggestion to improve