Oracle Database/SQL Cheatsheet

This "cheat sheet" covers most of the basic functionality that an Oracle DBA needs to run basic queries and perform basic tasks. It also contains information that a PL/SQL programmer frequently uses to write stored procedures. The resource is useful as a primer for individuals who are new to Oracle, or as a reference for those who are experienced at using Oracle.

A great deal of information about Oracle exists throughout the net. We developed this resource to make it easier for programmers and DBAs to find most of the basics in one place. Topics beyond the scope of a "cheatsheet" generally provide a link to further research.

Other Oracle References

- Oracle XML Reference—the XML reference is still in its infancy, but is coming along nicely.

Contents

SELECT

The SELECT statement is used to retrieve rows selected from one or more tables, object tables, views, object views, or materialized views.

```
SELECT *
FROM beverages
WHERE field1 = 'Kona'
AND field2 = 'coffee'
AND field3 = 122;
```

SELECT INTO

Select into takes the values *name*, *address* and *phone number* out of the table *employee*, and places them into the variables *v_employee_name*, *v_employee_address*, and *v_employee_phone_number*.

This *only* works if the query matches a single item. If the query returns no rows it raises the `NO_DATA_FOUND` built-in exception. If your query returns more than one row, Oracle raises the exception `TOO_MANY_ROWS`.

```
SELECT name,address,phone_number
INTO v_employee_name ,v_employee_address ,v_employee_phone_number
FROM employee
WHERE employee_id = 6;
```

INSERT

The INSERT statement adds one or more new rows of data to a database table.

insert using the VALUES keyword

```
INSERT INTO table_name VALUES ('Value1', 'Value2', ... );
INSERT INTO table_name( Column1, Column2, ... ) VALUES ( 'Value1', 'Value2', ... );
```

insert using a SELECT statement

```
INSERT INTO table_name( SELECT Value1, Value2, ... from table_name );
INSERT INTO table_name( Column1, Column2, ... ) ( SELECT Value1, Value2, ... from table_name );
```

DELETE

The DELETE statement is used to delete rows in a table.

deletes rows that match the criteria

```
DELETE FROM table_name WHERE some_column=some_value
DELETE FROM customer WHERE sold = 0;
```

UPDATE

The UPDATE statement is used to udpate rows in a table.

updates the entire column of that table

```
UPDATE customer SET state='CA';
```

updates the specific record of the table eg:

```
UPDATE customer SET name='Joe' WHERE customer_id=10;
```

updates the column invoice as paid when paid column has more than zero.

```
UPDATE movies SET invoice='paid' WHERE paid > 0;
```

SEQUENCES

Sequences are database objects that multiple users can use to generate unique integers. The sequence generator generates sequential numbers, which can help automatically generate unique primary keys, and coordinate keys across multiple rows or tables.

CREATE SEQUENCE

The syntax for a sequence is:

```
CREATE SEQUENCE sequence_name
    MINVALUE value
    MAXVALUE value
    START WITH value
    INCREMENT BY value
    CACHE value;
```

For example:

```
CREATE SEQUENCE supplier_seq
    MINVALUE 1
    MAXVALUE 9999999999999999999999999999
    START WITH 1
    INCREMENT BY 1
    CACHE 20;
```

ALTER SEQUENCE

Increment a sequence by a certain amount:

```
ALTER SEQUENCE <sequence_name> INCREMENT BY <integer>;
ALTER SEQUENCE seq_inc_by_ten  INCREMENT BY 10;
```

Change the maximum value of a sequence:

```
ALTER SEQUENCE <sequence_name> MAXVALUE <integer>;
ALTER SEQUENCE seq_maxval  MAXVALUE  10;
```

Set the sequence to cycle or not cycle:

```
ALTER SEQUENCE <sequence_name> <CYCLE | NOCYCLE>;
ALTER SEQUENCE seq_cycle NOCYCLE;
```

Configure the sequence to cache a value:

```
ALTER SEQUENCE <sequence_name> CACHE <integer> | NOCACHE;
ALTER SEQUENCE seq_cache NOCACHE;
```

Set whether or not to return the values in order

```
ALTER SEQUENCE <sequence_name> <ORDER | NOORDER>;
ALTER SEQUENCE seq_order NOORDER;
ALTER SEQUENCE seq_order;
```

Generate query from a string

It is sometimes necessary to create a query from a string. That is, if the programmer wants to create a query at run time (generate an Oracle query on the fly), based on a particular set of circumstances, etc.

Care should be taken not to insert user-supplied data directly into a dynamic query string, without first vetting the data very strictly for SQL escape characters; otherwise you run a significant risk of enabling data-injection hacks on your code.

Here is a very simple example of how a dynamic query is done. There are, of course, many different ways to do this; this is just an example of the functionality.

```
PROCEDURE oracle_runtime_query_pcd  IS
    TYPE ref_cursor IS REF CURSOR;
    l_cursor        ref_cursor;

    v_query         varchar2(5000);
    v_name          varchar2(64);
BEGIN
    v_query := 'SELECT name FROM employee WHERE employee_id=5'  ;
    OPEN l_cursor FOR v_query;
    LOOP
       FETCH l_cursor INTO v_name;
       EXIT WHEN l_cursor%NOTFOUND;
    END LOOP;
    CLOSE l_cursor;
END;
```

String operations

Length

Length returns an integer representing the length of a given string. It can be referred to as: **lengthb**, **lengthc**, **length2**, and **length4**.

```
length( string1 );
```

```
SELECT length('hello world') FROM dual;
this returns 11, since the argument is made up of 11 characters including the space
```

```
SELECT lengthb('hello world') FROM dual;
SELECT lengthc('hello world') FROM dual;
SELECT length2('hello world') FROM dual;
SELECT length4('hello world') FROM dual;
these also return 11, since the functions called are equivalent
```

Instr

Instr (in string) returns an integer that specifies the location of a sub-string within a string. The programmer can specify which appearance of the string they want to detect, as well as a starting position. An unsuccessful search returns 0.

```
instr( string1, string2, [ start_position ], [ nth_appearance ] )

instr( 'oracle pl/sql cheatsheet', '/');
this returns 10, since the first occurrence of "/" is the tenth character

instr( 'oracle pl/sql cheatsheet', 'e', 1, 2);
this returns 17, since the second occurrence of "e" is the seventeenth character

instr( 'oracle pl/sql cheatsheet', '/', 12, 1);
this returns 0, since the first occurrence of "/" is before the starting point, which is the 12th character
```

Replace

Replace looks through a string, replacing one string with another. If no other string is specified, it removes the string specified in the replacement string parameter.

```
replace( string1, string_to_replace, [ replacement_string ] );
replace('i am here','am','am not');
this returns "i am not here"
```

Substr

Substr (substring) returns a portion of the given string. The "start_position" is 1-based, not 0-based. If "start_position" is negative, substr counts from the end of the string. If "length" is not given, substr defaults to the remaining length of the string.

substr(*string*, start_position [, length])

```
SELECT substr( 'oracle pl/sql cheatsheet' , 8, 6) FROM dual;
```

returns "`pl/sql`" since the "p" in "pl/sql" is in the 8th position in the string (counting from 1 at the "o" in "oracle")

```
SELECT substr( 'oracle pl/sql cheatsheet' , 15) FROM dual;
```

returns "`cheatsheet`" since "c" is in the 15th position in the string and "t" is the last character in the string.

```
SELECT substr('oracle pl/sql cheatsheet' , -10, 5) FROM dual;
```

returns "`cheat`" since "c" is the 10th character in the string, counting from the *end* of the string with "t" as position 1.

Trim

These functions can be used to filter unwanted characters from strings. By default they remove spaces, but a character set can be specified for removal as well.

```
trim ( [ leading | trailing | both ] [ trim-char ] from string-to-be-trimmed );
trim ('   removing spaces at both sides     ');
this returns "removing spaces at both sides"

ltrim ( string-to-be-trimmed [, trimming-char-set ] );
ltrim ('   removing spaces at the left side     ');
this returns "removing spaces at the left side     "

rtrim ( string-to-be-trimmed [, trimming-char-set ] );
rtrim ('   removing spaces at the right side     ');
this returns "   removing spaces at the right side"
```

DDL SQL

Tables

Create table

The syntax to create a table is:

```
CREATE TABLE [table name]
      ( [column name] [datatype], ... );
```

For example:

```
CREATE TABLE employee
      (id int, name varchar(20));
```

Add column

The syntax to add a column is:

```
ALTER TABLE [table name]
      ADD ( [column name] [datatype], ... );
```

For example:

```
ALTER TABLE employee
      ADD (id int)
```

Modify column

The syntax to modify a column is:

```
ALTER TABLE [table name]
      MODIFY ( [column name] [new datatype] );
```

ALTER table syntax and examples:

For example:

```
ALTER TABLE employee
      MODIFY( sickHours s float );
```

Drop column

The syntax to drop a column is:

```
ALTER TABLE [table name]
      DROP COLUMN [column name];
```

For example:

```
ALTER TABLE employee
      DROP COLUMN vacationPay;
```

Constraints

Constraint types and codes

Type Code	Type Description	Acts On Level
C	Check on a table	Column
O	Read Only on a view	Object
P	Primary Key	Object
R	Referential AKA Foreign Key	Column
U	Unique Key	Column
V	Check Option on a view	Object

Displaying constraints

The following statement shows all constraints in the system:

```
SELECT
       table_name,
       constraint_name,
       constraint_type
FROM user_constraints;
```

Selecting referential constraints

The following statement shows all referential constraints (foreign keys) with both source and destination table/column couples:

```
SELECT
       c_list.CONSTRAINT_NAME as NAME,
       c_src.TABLE_NAME as SRC_TABLE,
       c_src.COLUMN_NAME as SRC_COLUMN,
       c_dest.TABLE_NAME as DEST_TABLE,
       c_dest.COLUMN_NAME as DEST_COLUMN
FROM ALL_CONSTRAINTS c_list,
     ALL_CONS_COLUMNS c_src,
```

```
    ALL_CONS_COLUMNS c_dest
WHERE c_list.CONSTRAINT_NAME = c_src.CONSTRAINT_NAME
  AND c_list.R_CONSTRAINT_NAME = c_dest.CONSTRAINT_NAME
  AND c_list.CONSTRAINT_TYPE = 'R'
GROUP BY c_list.CONSTRAINT_NAME,
         c_src.TABLE_NAME,
        c_src.COLUMN_NAME,
        c_dest.TABLE_NAME,
        c_dest.COLUMN_NAME;
```

Setting constraints on a table

The syntax for creating a check constraint using a CREATE TABLE statement is:

```
CREATE TABLE table_name
(
    column1 datatype null/not null,
    column2 datatype null/not null,
    ...
    CONSTRAINT constraint_name CHECK (column_name condition) [DISABLE]
);
```

For example:

```
CREATE TABLE suppliers
(
    supplier_id  numeric(4),
    supplier_name  varchar2(50),
    CONSTRAINT check_supplier_id
    CHECK (supplier_id BETWEEN 100 and 9999)
);
```

Unique Index on a table

The syntax for creating a unique constraint using a CREATE TABLE statement is:

```
CREATE TABLE table_name
(
    column1 datatype null/not null,
    column2 datatype null/not null,
    ...
    CONSTRAINT constraint_name UNIQUE (column1, column2, column_n)
);
```

For example:

```
CREATE TABLE customer
(
    id   integer not null,
    name varchar2(20),
    CONSTRAINT customer_id_constraint UNIQUE (id)
);
```

Adding unique constraints

The syntax for a unique constraint is:

```
ALTER TABLE [table name]
      ADD CONSTRAINT [constraint name] UNIQUE( [column name] ) USING INDEX [index name];
```

For example:

```
ALTER TABLE employee
      ADD CONSTRAINT uniqueEmployeeId UNIQUE(employeeId) USING INDEX ourcompanyIndx_tbs;
```

Deleting constraints

The syntax for dropping (removing) a constraint is:[1]

```
ALTER TABLE [table name]
      DROP CONSTRAINT [constraint name];
```

For example:

```
ALTER TABLE employee
      DROP CONSTRAINT uniqueEmployeeId;
```

INDEXES

An index is a method that retrieves records with greater efficiency. An index creates an entry for each value that appears in the indexed columns. By default, Oracle creates B-tree indexes.

Create an index

The syntax for creating an index is:

```
CREATE [UNIQUE] INDEX index_name
    ON table_name (column1, column2, . column_n)
    [ COMPUTE STATISTICS ];
```

UNIQUE indicates that the combination of values in the indexed columns must be unique.

COMPUTE STATISTICS tells Oracle to collect statistics during the creation of the index. The statistics are then used by the optimizer to choose an optimal execution plan when the statements are executed.

For example:

```
CREATE INDEX customer_idx
    ON customer (customer_name);
```

In this example, an index has been created on the customer table called customer_idx. It consists of only of the customer_name field.

The following creates an index with more than one field:

```
CREATE INDEX customer_idx
    ON supplier (customer_name, country);
```

The following collects statistics upon creation of the index:

```
CREATE INDEX customer_idx
    ON supplier (customer_name, country)
    COMPUTE STATISTICS;
```

Create a function-based index

In Oracle, you are not restricted to creating indexes on only columns. You can create function-based indexes.

The syntax that creates a function-based index is:

```
CREATE [UNIQUE] INDEX index_name
    ON table_name (function1, function2, . function_n)
    [ COMPUTE STATISTICS ];
```

For example:

```
CREATE INDEX customer_idx
    ON customer (UPPER(customer_name));
```

An index, based on the uppercase evaluation of the customer_name field, has been created.

To assure that the Oracle optimizer uses this index when executing your SQL statements, be sure that UPPER(customer_name) does not evaluate to a NULL value. To ensure this, add UPPER(customer_name) IS NOT NULL to your **WHERE** clause as follows:

```
SELECT customer_id, customer_name, UPPER(customer_name)
FROM customer
WHERE UPPER(customer_name) IS NOT NULL
ORDER BY UPPER(customer_name);
```

Rename an Index

The syntax for renaming an index is:

```
ALTER INDEX index_name
    RENAME TO new_index_name;
```

For example:

```
ALTER INDEX customer_id
    RENAME TO new_customer_id;
```

In this example, **customer_id** is renamed to **new_customer_id**.

Collect statistics on an index

If you need to collect statistics on the index after it is first created or you want to update the statistics, you can always use the **ALTER INDEX** command to collect statistics. You collect statistics so that oracle can use the indexes in an effective manner. This recalcultes the table size, number of rows, blocks, segments and update the dictionary tables so that oracle can use the data effectively while choosing the execution plan.

The syntax for collecting statistics on an index is:

```
ALTER INDEX index_name
    REBUILD COMPUTE STATISTICS;
```

For example:

```
ALTER INDEX customer_idx
    REBUILD COMPUTE STATISTICS;
```

In this example, statistics are collected for the index called **customer_idx**.

Drop an index

The syntax for dropping an index is:

```
  DROP INDEX index_name;
```

For example:

```
  DROP INDEX customer_idx;
```

In this example, the **customer_idx** is dropped.

DBA Related

User Management

Creating a user

The syntax for creating a user is:

```
  CREATE USER username IDENTIFIED BY password;
```

For example:

```
  CREATE USER brian IDENTIFIED BY brianpass;
```

Granting privileges

The syntax for granting privileges is:

```
GRANT privilege TO user;
```

For example:

```
GRANT dba TO brian;
```

Change password

The syntax for changing user password is:

```
ALTER USER username IDENTIFIED BY password;
```

For example:

```
ALTER USER brian IDENTIFIED BY brianpassword;
```

Importing and exporting

There are two methods of backing up and restoring database tables and data. The 'exp' and 'imp' tools are simpler tools geared towards smaller databases. If database structures become more complex or are very large (> 50 GB for example) then using the RMAN tool is more appropriate.

Import a dump file using IMP

This command is used to import Oracle tables and table data from a *.dmp file created by the 'exp' tool. Remember that this a command that is executed from the command line through $ORACLE_HOME/bin and not within SQL*Plus.

The syntax for importing a dump file is:

```
imp KEYWORD=value
```

There are number of parameters you can use for keywords.

To view all the keywords:

```
imp HELP=yes
```

An example:

```
imp brian/brianpassword FILE=mydump.dmp FULL=yes
```

PL/SQL

Operators

Arithmetic operators

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Power (PL/SQL only): **

Examples

gives all employees from customer id 5 a 5% raise

```
UPDATE employee SET salary = salary * 1.05
                  WHERE customer_id = 5;
```

determines the after tax wage for all employees

```
SELECT wage - tax FROM employee;
```

Comparison operators

- Greater Than: >
- Greater Than or Equal To: >=
- Less Than: <
- Less Than or Equal to: <=
- Equivalence: =
- Inequality: != ^= <> ¬= (depends on platform)

Examples

```
SELECT name, salary, email FROM employees WHERE salary > 40000;

SELECT name FROM customers WHERE customer_id < 6;
```

String operators

- Concatenate: ||

create or replace procedure addtest(a in varchar2(100), b in varchar2(100), c out varchar2(200)) IS begin C:=concat(a,'-',b);

Date operators

- Addition: +
- Subtraction: -

Types

Basic PL/SQL Types

Scalar type (defined in package STANDARD): NUMBER, CHAR, VARCHAR2, BOOLEAN, BINARY_INTEGER, LONG\LONG RAW, DATE, TIMESTAMP and its family including intervals)

Composite types (user-defined types): TABLE, RECORD, NESTED TABLE and VARRAY

LOB datatypes : used to store an unstructured large amount of data

%TYPE - anchored type variable declaration

The syntax for anchored type declarations is

```
<var_name> <obj>%type [not null][:= <init-val>];
```

For example

```
name Books.title%type;   /*  name is defined as the same type as column 'title' of table  Books */

commission number(5,2) := 12.5;

x commission%type;   /*  x is defined as the same type as variable 'commission' */
```

Note:

1. Anchored variables allow for the automatic synchronization of the type of anchored variable with the type of <obj> when there is a change to the <obj> type.
2. Anchored types are evaluated at compile time, so recompile the program to reflect the change of <obj> type in the anchored variable.

Collections

A collection is an ordered group of elements, all of the same type. It is a general concept that encompasses lists, arrays, and other familiar datatypes. Each element has a unique subscript that determines its position in the collection.

```
--Define a PL/SQL record type representing a book:
TYPE book_rec IS RECORD
   (title                   book.title%TYPE,
    author                  book.author_last_name%TYPE,
    year_published          book.published_date%TYPE);

--define a PL/SQL table containing entries of type book_rec:
Type book_rec_tab IS TABLE OF book_rec
     INDEX BY BINARY_INTEGER;

my_book_rec  book_rec%TYPE;
my_book_rec_tab book_rec_tab%TYPE;
...
my_book_rec := my_book_rec_tab(5);
find_authors_books(my_book_rec.author);
...
```

There are many good reasons to use collections.

- Dramatically faster execution speed, thanks to transparent performance boosts including a new optimizing compiler, better integrated native compilation, and new datatypes that help out with number-crunching applications.

- The FORALL statement, made even more flexible and useful. For example, FORALL now supports nonconsecutive indexes.
- Regular expressions are available in PL/SQL in the form of three new functions (REGEXP_INSTR, REGEXP_REPLACE, and REGEXP_SUBSTR) and the REGEXP_LIKE operator for comparisons[2].
- Collections, improved to include such things as collection comparison for equality and support for set operations on nested tables.

References

1. http://www.psoug.org/reference/constraints.html
2. "First Expressions" by Jonathan Gennick for more information in this issue

- Taking Up Collections (http://www.oracle.com/technology/oramag/oracle/03-sep/o53plsql.html)
- Oracle Programming with PL/SQL Collections (http://www.developer.com/db/article.php/3379271)

Stored logic

Functions

A function must return a value to the caller.

The syntax for a function is

```
CREATE [OR REPLACE] FUNCTION function_name [ (parameter [,parameter]) ]
RETURN [return_datatype]
IS
    [declaration_section]
BEGIN
    executable_section
    return [return_value]

    [EXCEPTION
        exception_section]
END [function_name];
```

For example:

```
CREATE OR REPLACE  FUNCTION to_date_check_null(dateString IN VARCHAR2, dateFormat IN VARCHAR2)
RETURN DATE IS
BEGIN
    IF dateString IS NULL THEN
        return NULL;
    ELSE
        return to_date(dateString, dateFormat);
    END IF;
END;
```

Procedures

A procedure differs from a function in that it must not return a value to the caller.

The syntax for a procedure is:

```
CREATE [OR REPLACE] PROCEDURE procedure_name [ (parameter [,parameter]) ]
IS
    [declaration_section]
BEGIN
    executable_section
    [EXCEPTION
        exception_section]
END [procedure_name];
```

When you create a procedure or function, you may define parameters. There are three types of parameters that can be declared:

1. **IN** - The parameter can be referenced by the procedure or function. The value of the parameter can not be overwritten by the procedure or function.
2. **OUT** - The parameter can not be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **IN OUT** - The parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

Also you can declare a **DEFAULT** value;

```
CREATE [OR REPLACE] PROCEDURE procedure_name [ (parameter [IN|OUT|IN OUT] [DEFAULT   value] [,parameter]) ]
```

The following is a simple example of a procedure:

```
  /* purpose: shows the students in the course specified by courseId */

  CREATE OR REPLACE Procedure GetNumberOfStudents
     ( courseId IN number, numberOfStudents OUT number )
  IS

      /* although there are better ways to compute the number of students,
         this is a good opportunity to show a cursor in action            */

      cursor student_cur is
      select studentId, studentName
          from course
          where course.courseId = courseId;
      student_rec    student_cur%ROWTYPE;

  BEGIN
      OPEN student_cur;
      LOOP
          FETCH student_cur INTO student_rec;
          EXIT WHEN student_cur%NOTFOUND;
          numberOfStudents := numberOfStudents + 1;
      END LOOP;
      CLOSE student_cur;

  EXCEPTION
  WHEN OTHERS THEN
        raise_application_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);
  END GetNumberOfStudents;
```

anonymous block

```
DECLARE
 x NUMBER(4) := 0;
BEGIN
  x := 1000;
  BEGIN
```

```
    x := x + 100;
  EXCEPTION
    WHEN OTHERS THEN
      x := x + 2;
  END;
  x := x + 10;
  dbms_output.put_line(x);
EXCEPTION
  WHEN OTHERS THEN
    x := x + 3;
END;
```

Passing parameters to stored logic

There are three basic syntaxes for passing parameters to a stored procedure: positional notation, named notation and mixed notation.

The following examples call this procedure for each of the basic syntaxes for parameter passing:

```
CREATE OR REPLACE PROCEDURE create_customer( p_name IN varchar2,
                                             p_id IN number,
                                             p_address IN varchar2,
                                             p_phone IN varchar2 ) IS
BEGIN
    INSERT INTO customer ( name, id, address, phone )
    VALUES ( p_name, p_id, p_address, p_phone );
END create_customer;
```

Positional notation

Specify the same parameters in the same order as they are declared in the procedure. This notation is compact, but if you specify the parameters (especially literals) in the wrong order, the bug can be hard to detect. You must change your code if the procedure's parameter list changes.

```
create_customer('James Whitfield', 33, '301 Anystreet', '251-222-3154');
```

Named notation

Specify the name of each parameter along with its value. An arrow (=>) serves as the association operator. The order of the parameters is not significant. This notation is more verbose, but makes your code easier to read and maintain. You can sometimes avoid changing code if the procedure's parameter list changes, for example if the parameters are reordered or a new optional parameter is added. Named notation is a good practice to use for any code that calls someone else's API, or defines an API for someone else to use.

```
create_customer(p_address => '301 Anystreet', p_id => 33, p_name => 'James Whitfield', p_phone => '251-222-3154');
```

Mixed notation

Specify the first parameters with positional notation, then switch to named notation for the last parameters. You can use this notation to call procedures that have some required parameters, followed by some optional parameters.

```
create_customer(v_name, v_id, p_address=> '301 Anystreet', p_phone => '251-222-3154');
```

Table functions

```
CREATE TYPE object_row_type as OBJECT (
  object_type VARCHAR(18),
  object_name VARCHAR(30)
);

CREATE TYPE object_table_type as TABLE OF object_row_type;

CREATE OR REPLACE FUNCTION get_all_objects
  RETURN object_table_type PIPELINED AS
BEGIN
    FOR cur IN (SELECT * FROM all_objects)
    LOOP
      PIPE ROW(object_row_type(cur.object_type, cur.object_name));
    END LOOP;
    RETURN;
END;

SELECT * FROM TABLE(get_all_objects);
```

Flow control

Conditional Operators

- and: AND
- or: OR
- not: NOT

Example

IF salary > 40000 AND salary <= 70000 THEN() ELSE IF salary>70000 AND salary<=100000 THEN() ELSE()

If/then/else

```
IF [condition] THEN
    [statements]
ELSEIF [condition] THEN
    [statements}
ELSEIF [condition] THEN
    [statements}
ELSEIF [condition] THEN
    [statements}
ELSEIF [condition] THEN
    [statements}
ELSEIF [condition] THEN
    [statements}
ELSEIF [condition] THEN
    [statements}
ELSE
    [statements}
END IF;
```

Arrays

Associative arrays

- Strongly typed arrays, useful as in-memory tables

Example

- Very simple example, the index is the key to accessing the array so there is no need to loop through the whole table unless you intend to use data from every line of the array.
- The index can also be a numeric value.

```
DECLARE
    -- Associative array indexed by string:

    -- Associative array type
    TYPE population IS TABLE OF NUMBER
        INDEX BY VARCHAR2(64);
    -- Associative array variable
    city_population  population;
    i                VARCHAR2(64);
BEGIN
    -- Add new elements to associative array:
    city_population('Smallville')  := 2000;
    city_population('Midland')     := 750000;
    city_population('Megalopolis') := 1000000;

    -- Change value associated with key 'Smallville':
    city_population('Smallville') := 2001;

    -- Print associative array by looping through it:
    i := city_population.FIRST;

    WHILE i IS NOT NULL LOOP
        DBMS_OUTPUT.PUT_LINE
            ('Population of ' || i || ' is ' || TO_CHAR(city_population(i)));
        i := city_population.NEXT(i);
    END LOOP;

    -- Print selected value from a associative array:
    DBMS_OUTPUT.PUT_LINE('Selected value');
    DBMS_OUTPUT.PUT_LINE('Population of');
END;
/

-- Printed results:
Population of Megalopolis is 1000000
Population of Midland is 750000
Population of Smallville is 2001
```

- More complex example, using a record

```
DECLARE
    -- Record type
    TYPE apollo_rec IS RECORD
    (
        commander   VARCHAR2(100),
        launch      DATE
    );
    -- Associative array type
    TYPE apollo_type_arr IS TABLE OF apollo_rec INDEX BY VARCHAR2(100);
    -- Associative array variable
    apollo_arr apollo_type_arr;
BEGIN
    apollo_arr('Apollo 11').commander := 'Neil Armstrong';
    apollo_arr('Apollo 11').launch :=   TO_DATE('July 16, 1969','Month dd, yyyy');
    apollo_arr('Apollo 12').commander := 'Pete Conrad';
    apollo_arr('Apollo 12').launch :=   TO_DATE('November 14, 1969','Month dd, yyyy');
    apollo_arr('Apollo 13').commander := 'James Lovell';
    apollo_arr('Apollo 13').launch :=   TO_DATE('April 11, 1970','Month dd, yyyy');
    apollo_arr('Apollo 14').commander := 'Alan Shepard';
    apollo_arr('Apollo 14').launch :=   TO_DATE('January 31, 1971','Month dd, yyyy');
```

```
    DBMS_OUTPUT.PUT_LINE(apollo_arr('Apollo 11').commander);
    DBMS_OUTPUT.PUT_LINE(apollo_arr('Apollo 11').launch);
end;
/

-- Printed results:
Neil Armstrong
16-JUL-69
```

APEX

Oracle Application Express aka APEX, is a web-based software development environment that runs on an Oracle database.

String substitution

- In SQL: :VARIABLE
- In PL/SQL: V('VARIABLE') or NV('VARIABLE')
- In text: &VARIABLE.

References

- http://www.psoug.org/reference/