C.F.G.S. DESARROLLO DE APLICACIONES WEB

MÓDULO:

BASES DE DATOS

Unidad 7:

Programación de Bases de datos, PL/SQL.

Parte I

ÍNDICE DE CONTENIDOS

OBJETIVOS

Con esta unidad se pretende:

- Adquirir una visión general del PL/SQL
- Saber cuáles son las posibilidades y limitaciones
- Manejar la estructura básica del lenguaje
- Conocer los tipos de datos que soporta el lenguaje, y como declararlos
- Conocer las estructuras de control y diseñar programas
- Utilizar cursores explícitos e implícitos para procesar la información contenida en la base de datos
- Diseñar programas capaces de recuperarse ante los posibles errores que puedan aparecer utilizando las excepciones
- Realizar procedimientos y funciones para desarrollar programas

1.- Introducción.

SQL ofrece una gran potencia para interrogar y administrar la base de datos, sin embargo hay ciertos tipos de preguntas o acciones que no es posible realizar, se necesita un lenguaje más potente, para ello **ORACLE** desarrolla un lenguaje procedimental el **PL/SQL** (Procedural Language/Structured Query Language) que va a extender la potencia que ofrece SQL.

PL/SQL soporta el lenguaje de consultas, es decir el (LMD) de SQL, pero no soporta órdenes de definición de datos (DDL) ni de control de datos (DCL).

PL/SQL incluye las características propias de un lenguaje procedimental:

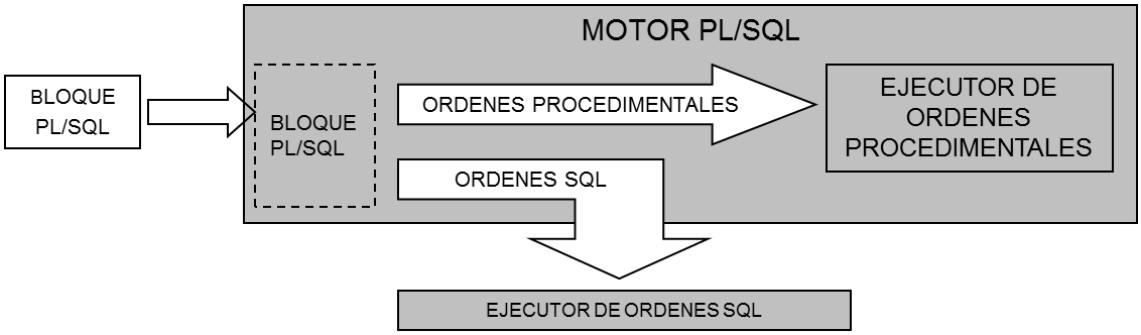
- El uso de variables.
- Estructuras de control.
- Control de excepciones.
- Reutilización de código a través de paquetes, procedimientos y funciones.

El código desarrollado en **PL/SQL** se puede almacenar como objetos de la base de datos, creando paquetes, procedimientos, y funciones pudiendo ser utilizado este código por los usuarios que estén autorizados.
El código **PL/SQL** se ejecuta en el servidor, lo que supone un ahorro de recursos a los clientes.

> PL/SQL es un lenguaje procedimental estructurado en **bloques** que amplía la funcionalidad de SQL.
> Con PL/SQL podemos usar sentencias SQL para manipular datos y sentencias de control de flujo para procesar los datos. Por tanto, PL/SQL combina la potencia de SQL para la manipulación de datos, con la potencia de los lenguajes procedimentales para procesar los datos.

Aunque **PL/SQL** fue creado por **Oracle**, hoy día todos los gestores de bases de datos utilizan un lenguaje procedimental muy parecido al ideado por **Oracle** para poder programar las bases de datos.

El motor de **PL/SQL** acepta como entrada bloques PL/SQL o subprogramas, ejecuta sentencias procedimentales y envía sentencias SQL al servidor de bases de datos. En el esquema adjunto puedes ver su funcionamiento.



Las características del lenguaje PL/SQL más relevantes son:

- Es un lenguaje procedimental diseñado por Oracle para trabajar con la BD
- Incluido en el servidor y en algunas herramientas del cliente
- Soporta todos los comandos de consulta y manipulación de datos, añadiendo al SQL: estructuras de control y otros elementos propios de lenguajes procedimentales
- La unidad de trabajo es el BLOQUE.
  BLOQUE: conjunto de declaraciones, instrucciones y mecanismos de gestión de errores y excepciones.

Como para cualquier otro lenguaje de programación, debemos conocer las reglas de sintaxis que podemos utilizar, los diferentes elementos de que consta, los tipos de datos de los que disponemos, las estructuras de control que nos ofrece (tanto iterativas como condicionales) y cómo se realiza el manejo de los errores.

2.- Conceptos básicos.

2.1.- Bloques PL/SQL.

Los programas PL/SQL están estructurados en **bloques**. Cualquier programa estará formado por al menos un bloque lógico. Cada bloque lógico puede tener anidados cualquier número de subbloques.

Un bloque PL/SQL consta de tres zonas:

- **Zona de Declaraciones:** donde se declaran los objetos (variables, constantes, cursores…). Va precedido por **DECLARE**. Es opcional.
- **Zona de Proceso o ejecución**: Zona donde se realizará el proceso en sí, conteniendo las sentencias ejecutables. Va precedido por **BEGIN**. Es obligatoria.
- **Zona de Excepciones**: Zona de manejo de errores en tiempo de ejecución. Precedido por **EXCEPTION**. Es opcional.

El bloque más sencillo es el bloque anónimo y la sintaxis es la siguiente:

```
DECLARE
        [Declaración de variables, constantes, cursores y excepciones]
BEGIN [nombre del bloque]
        [Sentencias ejecutables]
[EXCEPTION
        [Gestión de excepciones]
END;
```

Ejemplo-1:

```
DECLARE
        v_numemp NUMBER(2);
BEGIN
        INSERT INTO departamento VALUES (99,'PROVISIONAL',NULL);
        UPDATE empleado SET depnume= 99 where depnume = 20;
        v_numemp :=SQL%ROWCOUNT;  --num filas afectadas
        DELETE FROM departamento WHERE numedep= 20;
        DBMS_OUTPUT.PUT_LINE (v_numemp || 'Empleados ubicados en PROVISIONAL');
EXCEPTION
        WHEN OTHERS THEN  ROLLBACK;
        RAISE_APPLICATION_ERROR (-2000,'Error en la aplicación');
END;
```

Los bloques PL/SQL pueden anidarse a cualquier nivel.

```
DECLARE
        [Declaración de variables, constantes, cursores y excepciones]
BEGIN [nombre del bloque]
        DECLARE
                [Declaración de variables, constantes, cursores y excepciones]
        BEGIN [nombre del bloque]
                [Sentencias ejecutables]
        [EXCEPTION
                [Gestión de excepciones]
        END;
[EXCEPTION
        [Gestión de excepciones]
END;
```

2.2.- Unidades Léxicas.

Al igual que en nuestra lengua podemos distinguir diferentes unidades léxicas como palabras, signos de puntuación, etc. En los lenguajes de programación también existen diferentes unidades léxicas que definen los elementos más pequeños que tienen sentido propio y que al combinarlos de manera adecuada, siguiendo las reglas de sintaxis, dan lugar a sentencias válidas sintácticamente.

PL/SQL es un lenguaje no sensible a las mayúsculas, por lo que será equivalente escribir en mayúsculas o minúsculas, excepto cuando hablemos de literales de tipo cadena o de tipo carácter.

Las unidades léxicas se pueden clasificar en:

- Delimitadores.
- Identificadores.
- Literales.
- Comentarios.

Delimitadores.

PL/SQL tiene un conjunto de símbolos denominados delimitadores utilizados para representar operaciones entre tipos de datos, delimitar comentarios, etc.
En la siguiente tabla puedes ver un resumen de los mismos.

Delimitadores en PL/SQL.			
Delimitadores Simples.		**Delimitadores Compuestos.**	
Símbolo.	**Significado.**	**Símbolo.**	**Significado.**
+	Suma.	**	Exponenciación.
%	Indicador de atributo.	<>	Distinto.
.	Selector.	¡=	Distinto.
/	División.	<=	Menor o igual.
(	Delimitador de lista.	>=	Mayor o igual.
)	Delimitador de lista.	..	Rango.
:	Variable host.	||	Concatenación.
,	Separador de elementos.	<<	Delimitador de etiquetas.
*	Producto.	>>	Delimitador de etiquetas.
"	Delimitador de identificador acotado.	--	Comentario de una línea.
=	Igual relacional.	/*	Comentario de varias líneas.
<	Menor.	*/	Comentario de varias líneas.
>	Mayor.	:=	Asignación.
@	Indicador de acceso remoto.	=>	Selector de nombre de parámetro.
;	Terminador de sentencias.		
-	Resta/negación		

Identificadores.

Los identificadores en PL/SQL, como en cualquier otro lenguaje de programación, son utilizados para nombrar elementos de nuestros programas, como variables, constantes, cursores, excepciones, procedimientos, funciones, etiquetas, etc.

A la hora de utilizar los identificadores debemos tener en cuenta los siguientes aspectos:

- Pueden tener hasta 30 caracteres.
- No existe diferencia entre mayúsculas y minúsculas.
- Un identificador debe comenzar por una letra seguida opcionalmente de letras, números, $, _, #.
- No podemos utilizar como identificador una palabra reservada.
  - ✓ Ejemplos válidos: X, A1, codigo_postal.
  - ✓ Ejemplos no válidos: rock&roll, on/off.
- PL/SQL nos permite además definir los identificadores acotados, en los que podemos usar cualquier carácter con una longitud máxima de 30 y deben estar delimitados por comillas ".
  Ejemplo: "X*Y".
- En PL/SQL existen algunos identificadores predefinidos y que tienen un significado especial ya que nos permitirán darle sentido sintáctico a nuestros programas. Estos identificadores son las palabras reservadas y no las podemos utilizar como identificadores en nuestros programas.
  Ejemplo: IF, THEN, ELSE...
- Algunas palabras reservadas para PL/SQL no lo son para SQL, por lo que podríamos tener una tabla con una columna llamada 'type' por ejemplo, que nos daría un error de compilación al referirnos a ella en PL/SQL.
  La solución sería acotarlos. SELECT "TYPE" …

Literales.

Los literales se utilizan en las comparaciones de valores o para asignar valores concretos a los identificadores que actúan como variables o constantes. Para expresar estos literales tendremos en cuenta que:

- Los literales numéricos se expresarán por medio de notación decimal o de notación exponencial.
  Ejemplos: 234, +341, 2e3, -2E-3, 7.45, 8.1e3
- Los literales tipo carácter y tipo cadena se deben delimitar con unas **comillas simples**.
- Los literales lógicos son TRUE y FALSE.
- El literal NULL expresa que una variable no tiene ningún valor asignado.

- **Comentarios.**

  En los lenguajes de programación es muy conveniente utilizar comentarios en mitad del código. Los comentarios no tienen ningún efecto sobre el código pero sí ayudan mucho al programador o la programadora a recordar qué se está intentando hacer en cada caso (más aún cuando el código es compartido entre varias personas que se dedican a mejorarlo o corregirlo).

  En PL/SQL podemos utilizar dos tipos de comentarios:

  - Los comentarios de una línea se expresaran por medio del delimitador **--**

    Ejemplo-2:

    ```
    a:=b;  --asignación
    ```

  - Los comentarios de varias líneas se acotarán por medio de los delimitadores **/*** y ***/**.

    Ejemplo-3:

    ```
    /* Primera línea de comentarios.
       Segunda línea de comentarios. */
    ```

2.3.- Tipos de datos simples.

En PL/SQL contamos con todos los tipos de datos simples utilizados en SQL y algunos más. En este apartado vamos a enumerar los más utilizados.

- **Numéricos.**

BINARY_INTEGER:	Tipo de dato numérico entero, cuyo rango es: Desde -2147483647 a 2147483647. PL/SQL además define algunos subtipos de éste: NATURAL, NATURALN, POSITIVE, POSITIVEN, SIGNTYPE.
NUMBER[(p, e)]	Tipo de dato numérico para almacenar números racionales. Podemos especificar su escala **e**, que es el número de decimales, y su precisión **p** que es el número total de dígitos. PL/SQL también define algunos subtipos como: DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NUMERIC, REAL, SMALLINT.
PLS_INTEGER:	Tipo de datos numérico cuyo rango es el mismo que el del tipo de dato BINARY_INTEGER, pero que su representación es distinta por lo que las operaciones aritméticas llevadas a cabo con los mismos serán más eficientes que en los dos casos anteriores.

- **Alfanuméricos.**

CHAR (longitud):	Cadena de caracteres de longitud fija., máximo 2000 bytes. Si no especificamos longitud sería 1.
VARCHAR2 (longitud):	Cadenas de longitud variable con un máximo de 32760. bytes
LONG (longitud):	Cadena de caracteres de longitud variable. Similar a VARCHAR2, almacena hasta 3GB.
RAW [longitud]:	Mismas características que CHAR pero para almacenar información en binario.
LONG RAW:	Mismas características que LONG almacenando la información en binario..

- **Grandes objetos.**

BFILE:	Almacena la referencia a un archivo del sistema que contiene datos físicos.
BLOB:	Almacena un objeto binario, hasta 4 GB (Se puede utilizar para guardar imágenes, gráficos, etc)
CLOB:	Almacena cadenas de caracteres de longitud variable, hasta 4GB

- **Otros.**

BOOLEAN:	Almacena valores TRUE, FALSE y NULL.
DATE	Almacena fechas en formato estándar: 'dd-mm-yy'
ROWID	Se utiliza para definir variables que contendrán valores correspondientes a la pseudocolumna ROWID, o sea, los valores binarios que identifican físicamente cada fila de datos de una base.

2.4.- Variables y constantes.

Para poder utilizar variables y constantes es necesario antes declararlas en el apartado DECLARE del bloque anónimo.

PL/SQL permite declarar variables y constantes y utilizarlas en cualquier parte de una sentencia SQL o procedimental en que pueda usarse una expresión.

- **Variables.**

  Las variables pueden ser de cualquiera de los tipos vistos en SQL o de otros tipos específicos de PL/SQL. Al declarar una variable se está reservando el espacio de memoria necesario para almacenarla, y se le puede asignar un valor.

El formato es el siguiente:

```
nombre_variable  tipo  [NOT NULL] [{:= |DEFAULT}  valor];
```

La palabra reservada **DEFAULT** puede usarse para inicializar una variable a la vez que se define, en lugar del operador de asignación.

Si no se inicializa el valor es NULL

Ejemplo-4:

```
DECLARE
      vimporte    NUMBER(8,2);
      vnombre     VARCHAR2(20);
      vdirec      VARCHAR2(20) DEFAULT 'AVILES';
      IVA         NUMBER (4,2): = 21.00....
```

También podemos realizar la asignación de valores a una variable en la parte ejecutable de un bloque mediante el **operador :=**

PL/SQL permite definir tipos complejos basados en estructuras de tablas y registros, ayudándose de los atributos %TYPE y %ROWTYPE.

- Uso de **%TYPE**: Permite declarar una variable del mismo tipo que otra variable o que una **columna** de una tabla.

Ejemplo-5:

```
DECLARE
      total     importe%TYPE; --total es del mismo tipo que la variable importe
      moroso    clientes.nombre%TYPE;  --moroso es del mismo tipo que la variable nombre de la
                                          tabla clientes
```

- Uso de **%ROWTYPE**: Se puede declarar una variable para guardar una **fila** completa de una tabla.

Ejemplo-6:

```
DECLARE
      moroso clientes%ROWTYPE;  --moroso tiene la misma estructura que la tabla clientes
```

Si una columna tiene la restricción **NOT NULL** la variable definida de ese tipo no asume dicha restricción, pudiendo por lo tanto ponerla a NULL.

- **Constantes**

Las constantes también necesitan ser inicializadas en la sección **DECLARE**, y a diferencia de las variables no se puede asignar ningún otro valor en la ejecución del programa. De hacerlo, se produciría un error durante la ejecución del código indicando que la constante no puede ser objeto de asignación.

El formato es el siguiente:

```
nombre_constante CONSTANT tipo:= valor;
```

Ejemplo-7:

```
impuestoIVA CONTANT number(2):= 21;
```

El valor de una constante, una vez determinada, puede utilizarse en cualquier lugar del bloque pero no modificarse. La declaración de constantes facilita la modificación de programas que contienen datos constantes.

- **Subtipos.**

En cualquier lenguaje de programación, las variables y las constantes tienen un tipo de dato asignado (bien sea explícitamente o implícitamente).
PL/SQL nos permite definir subtipos de tipos de datos para darles un nombre diferente y así aumentar la legibilidad de nuestros programas. Los tipos de operaciones aplicables a estos subtipos serán las mismas que los tipos de datos de los que proceden.

La sintaxis será:

```
SUBTYPE subtipo IS tipo_base;
```

Donde **subtipo** será el nombre que le demos a nuestro subtipo y **tipo_base** será cualquier tipo de dato en PL/SQL.

A la hora de especificar el tipo base, podemos utilizar el modificador %TYPE para indicar el tipo de dato de una variable o de una columna de la base de datos y %ROWTYPE para especificar el tipo de un cursor o tabla de una base de datos.

Ejemplo-8:

```
SUBTYPE idfamilia IS familias.identificador%TYPE;

SUBTYPE agente IS agentes%ROWTYPE;
```

Los subtipos son intercambiables con su tipo base. También son intercambiables si tienen el mismo tipo base o si su tipo base pertenece a la misma familia:

Ejemplo-9:

```
DECLARE
    SUBTYPE numero IS NUMBER;
    numeroDigitos NUMBER(3);
    miNumeroSuerte numero;
    SUBTYPE encontrado IS BOOLEAN;
    SUBTYPE resultado IS BOOLEAN;
    loEcontre encontrado;
    resultadoBusqueda resultado;
    SUBTYPE literal IS CHAR;
    SUBTYPE sentencia IS VARCHAR2;
    literalNulo literal;
    sentenciaVacia sentencia;
BEGIN
    numeroDigitos :=  miMumeroSuerte;          --legal
    ……
    loEncontrado := resultadoBusqueda;         --legal
    ……
    sentenciaVacia := literalNulo;             --legal
    ……
END;
```

2.5.- Operadores y expresiones.

Se pueden realizar distintas operaciones con las variables y constantes, para ello se usan los operadores, dando lugar a expresiones cuyos resultados serán el fundamento de la lógica de la programación.

Por tanto **una expresión** estará formada por un conjunto de operandos unidos por operadores. Un operando puede ser una variable, constante, literal o llamada a función que contribuye a la expresión con un valor.

Los valores de los distintos operandos se combinan de acuerdo con los operadores dando lugar a un valor único. Las operaciones en una expresión se hacen según el orden de precedencia de los operadores.

En la siguiente tabla están los operadores disponibles, con su orden de precedencia:

Orden	Operador	Operación
1º	** NOT	Potencia Negación Lógica
2º	+ , -	Signo positivo o negativo
3º	* /	Multiplicación División
4º	+ - ||	Suma Resta Concatenación
5º	=, <, >,<=, >=, <>,!=	Comparaciones: igual, menor, mayor, menor o igual mayor o igual, distinto, distinto
6º 7º	AND OR	Conjunción Inclusión

Las operaciones en una expresión se realizan como en todos los lenguajes siguiendo el orden de precedencia, que se puede modificar con el uso de paréntesis.

2.6.- Etiquetas

Son textos que se ponen para conseguir mayor legibilidad, o permitir acceder a determinados puntos del bloque con la instrucción GOTO.

Su formato es:

```
nombre-etiqueta
```

Ejemplo-10:

```
principal
DECLARE
    ...
    BEGIN
        ...
    END
END principal;
```

2.7.- Ámbito de las variables.

Cuando estamos trabajando en un programa que consta de varios bloques tenemos que tener en cuenta dónde y cómo actúan las variables que tenemos declaradas:

- El ámbito de una variable es el bloque en el que se declara y los bloques hijos de dicho bloque.
- La variable es local para el bloque en el que se declara y global para los bloques hijos.
- Las variables declaradas en los bloques hijos no son accesibles desde el bloque padre.

- Distintos bloques pueden tener identificadores iguales, por defecto el nombre del identificador referencia a la variable local. Para acceder a la variable del mismo nombre de un bloque padre, será necesario especificar la etiqueta del bloque al que pertenece dicha variable. **etiqueta.variable**

Ejemplo-11:

```
DECLARE       -- bloque padre
    v1          CHAR;
BEGIN
    v1:=1;
    DECLARE     --bloque hijo
        v2              CHAR;
    BEGIN
        v2:=2;
        ....
        v1:=v2;
        .....
    END;          -- fin  bloque hijo
    V2:=V1;       -- error, v2 es desconocida en el bloque padre!!!!!
END;              -- fin bloque padre
```

Ejemplo-12:

```
<<padre>>         -- Etiqueta para referenciar variables de este bloque
DECLARE           -- Bloque padre
    nota  NUMBER(2);
BEGIN
    .....
    nota:=5;
    DECLARE           -- Bloque hijo
        nota    NUMBRE(2);
    BEGIN
        nota:= padre.nota;
        /* nota referencia la variable local y padre.nota a la variable nota del bloque padre */
        .....
    END;                -- Fin  bloque hijo
END padre;              -- Fin bloque padre
```

2.8.- Entrada y salida. Interacción con el usuario

PL/SQL está pensado para trabajar con la Base de Datos y no está pensado para interactuar con el usuario final, por lo que carece de instrucciones especializadas de entrada/salida, teclado y pantalla.

Para solucionar esto PL/SQL dispone de algún método que permite mostrar el resultado de lo que se está haciendo en pantalla, así como poder asignar valores a las variables desde teclado.

- **La Salida.**

  Para poder mostrar datos PL/SQL dispone del paquete **DBMS_OUTPUT** que contiene el procedimiento **PUT_LINE** para visualizar textos en pantalla.

  La sintaxis es la siguiente:

  DBMS_OUTPUT.PUT_LINE (expresión)

  Para que funcione tiene que activarse la variable de entorno **SET SERVEROUTPUT**. Solamente es necesario hacerlo una vez en la sesión tecleando **SET SERVEROUTPUT ON**.

  Una vez activada ya podemos utilizar el método **DBMS_OUTPUT.PUT_LINE.**

  Se puede mostrar texto, o el contenido de variables y constantes. Para mostrar texto hay que ponerlo entre comillas simples, y para mostrar el contenido de una variable o constante, basta poner el nombre.

  Ejemplo-13:

```
DECLARE
    nota NUMBER(2):= 5;
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Hola que tal');
    DBMS_OUTPUT.PUT_LINE (nota);
END;
```

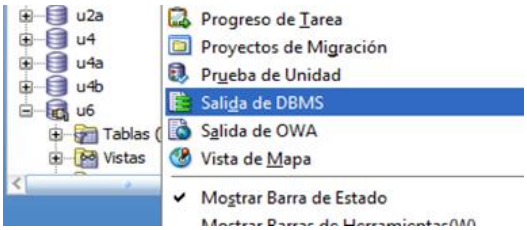
  En la mayoría de los casos puede interesar mostrar información de distintos tipos, por ejemplo cadenas de testo para aclarar lo que se va a mostrar seguido de valores de las variables, para ello es muy útil el operador que concatena cadenas, el **operador '||'**.

No importa de qué tipo sea la variable o constante.

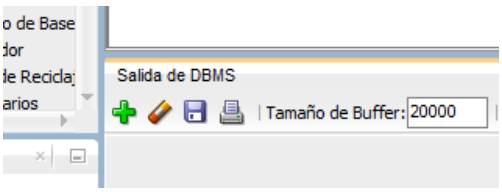
Ejemplo-14:

```
DECLARE
    nombre VARCHAR2(20):= 'Maria';
    nota NUMBER(2):= 5;
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Nota del módulo: ' || nota);
    DBMS_OUTPUT.PUT_LINE ('Hola ' || nombre || '. Hoy es  '|| sysdate);
END;
```

Si estamos trabajando en el **SQL-DEVELOPER** otra forma de ver los resultados será activar la salida **DBMS**, para ellos en la **Barra de Menús** seleccionar **Ver →Salida de DBMS.**



Se abre una ventana en la parte inferior derecha. En esta ventana pulsamos el + para seleccionar la conexión sobre la que queremos activar la salida.



- **La entrada.**

Para leer valores de teclado necesitamos el uso de **ACCEPT** y **PROMPT** el formato es el siguiente:

```
ACCEPT variable PROMPT 'mensaje'
```

Esta instrucción no puede estar dentro del cuerpo del bloque, es decir tiene que estar antes de **DECLARE**. Una vez introducido la variable, para poder utilizarla en el bloque tenemos que referenciarla utilizando el símbolo **&.**

Si la variable es numérica se referencia → **&variable**

Si la variable es alfanumérica se referencia entre comillas simple. → **'&variable'**

Ejemplo-15:

```
ACCEPT nombre PROMPT 'Introduce tu nombre'
ACCEPT edad PROMPT 'Introduce tu edad'
BEGIN
        DBMS_OUTPUT.PUT_LINE ('Hola me llamo '||'&nombre');
        DBMS_OUTPUT.PUT_LINE ('Tengo '||&edad||' años');
END;
```

Ejemplo-16:

```
ACCEPT salAnual PROMPT 'Introducir el salario anual'
DECLARE
        vSalario NUMBER(9,2) :=  &salAnual;
BEGIN
        vSalario:= vSalario/12;
        DBMS_OUTPUT.PUT_LINE ('Salario Mensual :'|| vSalario);
END;
```

2.9.- Estructuras de Control.

El teorema del programa estructurado establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Estas estructuras son llamadas estructuras de control son:

- **Secuencia:** ejecución de una instrucción tras otra.
- **Selección:** ejecución de una serie de dos instrucciones o conjunto de instrucciones, según el valor de una variable o expresión booleana.
- **Iteración:** ejecución de una instrucción o conjunto de instrucciones mientras una variable o instrucción booleana sea verdadera. Esta estructura lógica también se conoce como ciclo o bucle.

2.9.1.- Secuencia.

Se trata de órdenes del lenguaje, asignaciones, llamadas a funciones o procedimientos, una detrás de otra y separadas por **punto y coma ";"**

2.9.2.- Selección.

También conocida coma alternativa o condicional. Se trata de evaluar una expresión y en función del valor de esta expresión (verdadera o falsa) se hacen unas acciones u otras.

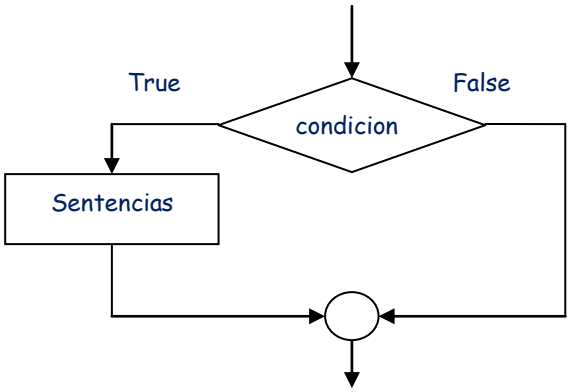
Sentencia IF. (Alternativa simple)

Es la forma más simple de las sentencias de control condicional. Si la evaluación de la **condición** es **TRUE**, entonces se ejecuta la secuencia de sentencias, y si es **FALSE** no se ejecuta nada.

Los operadores utilizados en la condición son los mismos que en SQL.

Su formato:

```
IF condicion THEN
    sentencias;
END IF;
```



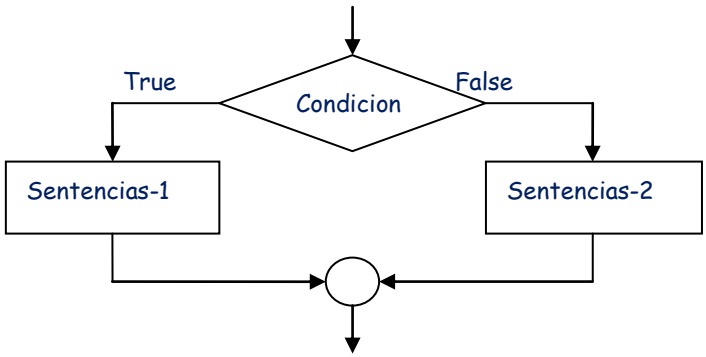
Ejemplo-17:

```
IF cuota < 50000 THEN
    cuota := cuota * 1.10;
END IF
```

Sentencia IF THEN ..ELSE. (Alternativa doble)

La sentencia alternativa **IF..THEN ..ELSE**, permite que se ejecute unas sentencias u otras según sea el resultado de evaluar una condición.

```
IF condición THEN
    Sentencias-1;
ELSE
    Sentencias-2;
END IF;
```



Si la condición es verdadera ejecuta el grupo de Sentencias-1, si es falsa o NULL el grupo de Sentencias-2.

Ejemplo-18:

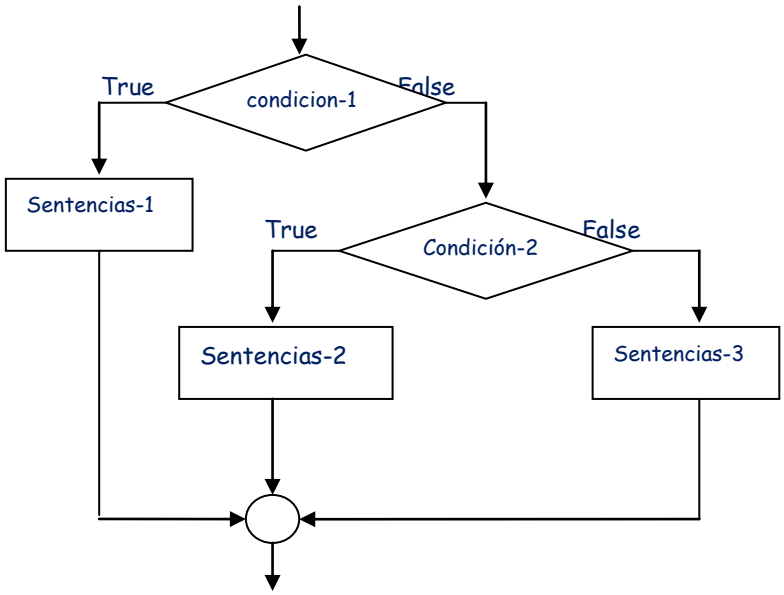
```
IF cuota < 50000 THEN
    cuota := cuota * 1.10;
ELSE
    cuota := cuota * 1.05;
END IF;
```

Sentencia IF-THEN-ELSIF. (Alternativa múltiple)

Cuando se utilizan sentencias de control es muy común desear anidar un **IF** dentro de otro **IF.** Se puede realizar un anidamiento de **IF** a través de la opción **ELSIF.**

Con esta sentencia condicional podemos hacer una selección múltiple, y tener tantos anidamientos como queramos.

```
IF condicion-1 THEN
     Sentencias-1;
ELSIF condicion-2 THEN
          Sentencias-2;
     ELSE
          Sentencias-3;
END IF;
```



Si la evaluación de la condición-1 da **TRUE**, ejecutamos el grupo de sentencias-1, sino evaluamos la condición-2. Si esta condición-2 evalúa a **TRUE** ejecutamos el grupo de sentencias-2 y así para todos los **ELSIF** que haya.

Ejemplo-19:

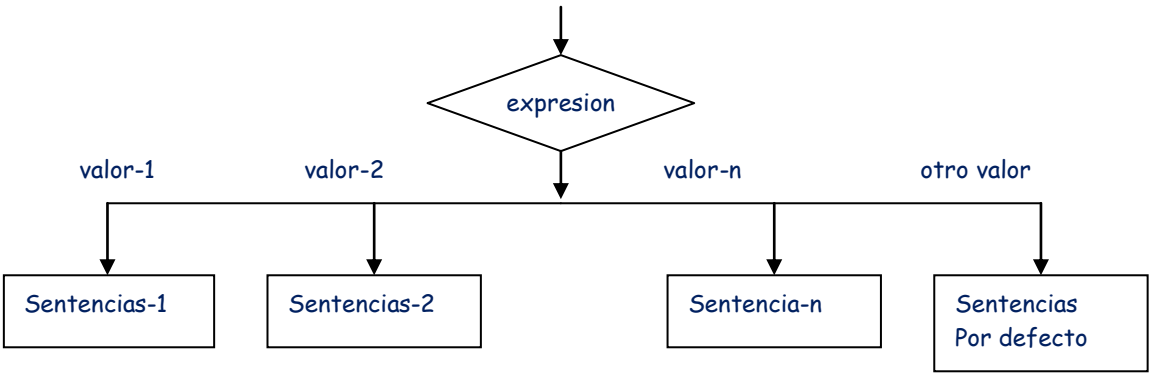
```
IF cuota > 50000 THEN
        cuota := cuota * 1.05;
ELSIF cuota > 30000 THEN
        cuota := cuota * 1.10;
     ELSE
        cuota := cuota * 1.15;
END IF;
```

Sentencia CASE. (Alternativa múltiple)

La sentencia **CASE** permite realizar una ejecución condicional, la diferencia con la sentencia **IF** es que se adapta mejor a las condiciones que implican varias opciones diferentes.

Evalúa cada condición hasta encontrar una que se cumpla.

```
CASE [elemento|expresión]
    WHEN {condición-1 | valor1 }THEN sentencias-1;
    WHEN {condición-2 | valor2 }THEN sentencias-2;
    ……
    [ELSE sentencias por defecto]
END CASE;
```



En este sentencia CASE se evalúa una expresión y en cada clausula WHEN se pone un valor o condición que se pueda dar para la expresión y a continuación del THEN la sentencia que se ejecutara en caso afirmativo. Por último se pone un ELSE que recoge las acciones que se ejecutaran en el caso de que el valor de la expresión no coincida con ninguno de los valores anteriores

Ejemplo-20:

```
DECLARE
    localidad   NUMBER(2):= 10;
    nombre    VARCHAR2(30);
BEGIN
    CASE localidad
        WHEN 10 THEN nombre := 'Avilés';
        WHEN 20 THEN nombre := 'Oviedo';
        WHEN 30 THEN nombre := 'Santoña';
        WHEN 40 THEN nombre := 'Solares';
        ELSE nombre:= 'desconocida';
    END CASE;
    DBMS_OUTPUT.PUT_LINE(nombre);
END;
```

También se puede evaluar la condición en cada clausula WHEN

```
Ejemplo-21:
DECLARE
    localidad    NUMBER(2):= 10;
    comunidad  VARCHAR2(30);
BEGIN
    CASE
        WHEN localidad IN (10,20) THEN comunidad := 'Asturias';
        WHEN localidad IN (30,40) THEN comunidad := 'Cantabria';
    END CASE;
    DBMS_OUTPUT.PUT_LINE(comunidad);
END;
```

2.9.3.- Iteración o bucles.

Los bucles repiten una sentencia o un grupo de sentencias varias veces, es decir permiten ejecutar de forma iterativa un grupo se sentencias.

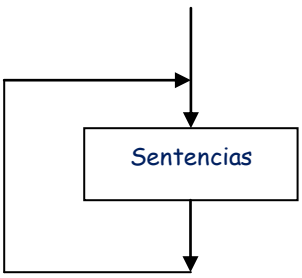
Hay varios tipos de bucles que dependiendo de si se sabe o no el número de veces que se van a repetir las acciones, o si por el contrario conocemos la condición de repetición o de salida del bucle, será más interesante usar un tipo de bucle u otro.

En PL/SQL existen tres tipos de bucles:

- **Sentencia LOOP.**

  Es el bucle más sencillo.

```
LOOP
    Sentencias;
END LOOP;
```



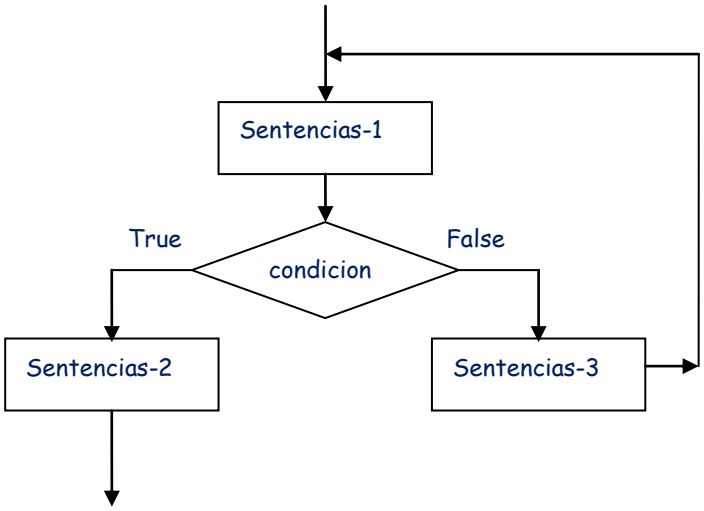
Es un bucle infinito, dado que no hay ninguna sentencia vinculada a **LOOP** que provoque una salida, estaría repitiendo infinitamente las sentencias.

Este bucle no es operativo ya que no es aconsejable provocar un bucle infinito, por ese existe una instrucción llamada **EXIT** que permite abandonar el bucle. Cuando Oracle encuentra la instrucción **EXIT**, el programa continúa desde la siguiente instrucción a **END LOOP**.

Lo normal es colocar el EXIT dentro de una sentencia IF a fin de establecer una condición de salida del bucle. También se puede acompañar la palabra EXIT de la cláusula WHEN seguida de una condición, si esa condición es cierta, se abandona el bucle, sino continuamos dentro.

Sentencia LOOP con IF

```
LOOP
    Sentencias-1;
    IF condición THEN
        Sentencias-2;
        EXIT;
    END IF;
        Sentencias-3;
END LOOP;
```



Ejemplo-22:

```
DECLARE
    contador NUMBER(2);
BEGIN
    contador:= 5;
    LOOP
        DBMS_OUTPUT.PUT_LINE(contador);
        IF contador > 10 THEN
            EXIT;
        END IF;
        Contador:= contador + 1;
    END LOOP;
END;
```

Sentencia LOOP con WHEN.

```
LOOP
    sentencias-1;
    EXIT WHEN condicion;
    sentencias-2
END LOOP;
```

Con este formato se sale del bucle cuando la condición sea verdadera.

Ejemplo-23

```
DECLARE
    contador NUMBER(2);
BEGIN
    contador:= 5;
    LOOP
        DBMS_OUTPUT.PUT_LINE(contador);
        EXIT WHEN contador > 10;
        Contador:= contador + 1;
    END LOOP;
END;
```

- **Sentencia LOOP con etiquetas.**

Los bucles pueden identificarse con una etiqueta, o sea con un identificador encerrado entre ángulos inmediatamente antes del bucle.

```
<<proceso1>>
LOOP
    sentencias
END LOOP proceso1;
```

La sentencia EXIT admite un nombre de bucle

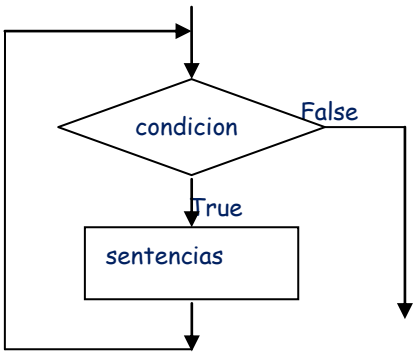
```
<<proceso1>>
LOOP
    Sentencias-1;
    <<proceso2>>
    LOOP
        Sentencias-2;
        EXIT proceso1 WHEN sentencias-3
    END LOOP;
END LOOP;
```

En este caso la sentencia EXIT daría lugar a la finalización tanto del bucle proceso2 (el interno) como del proceso1 (el externo).

- **Sentencia WHILE.**

Esta sentencia constituye un bucle cuyas sentencias se ejecutan mientras la condición asociada a la sentencia sea cierta. Se sale del bucle cuando el resultado de la evaluación sea falso o nulo.

```
WHILE condición LOOP
    Sentencias
END LOOP;
```



Algunas de las sentencias tienen que afectar la condición para que en algún momento pase a falsa, de lo contrario el bucle sería infinito.

Ejemplo-24

```
DECLARE
    contador NUMBER(2);
BEGIN
    contador:= 0;
    WHILE contador < 10 LOOP
        DBMS_OUTPUT.PUT_LINE (contador);
        Contador:= contador +1;
    END LOOP;
END;
```

- **Sentencia FOR.**

Se utilizan para bucles que se recorren un número concreto de veces. El bucle se controla con un **índice o contador** que progresa en un rango establecido.

La variable **índice o contador** no tiene que estar declarada en **DECLARE**, ya que es declarada automáticamente en el bucle **FOR**, es de tipo **NUMBER** y toma valores enteros.

```
FOR índice IN [RESERVE] valor_inicial .. valor_final LOOP
    Sentencias;
END LOOP;
```

Se tiene que indicar un valor inicial del índice y este ira incrementándose de uno en uno hasta llegar al valor final. No se puede modificar su valor pero si consultar y no se puede referenciar fuera del bucle.

Si utilizamos la opción **REVERSE**, en cada iteración el índice decrece desde el valor más alto al más bajo.

Ejemplo-25

```
BEGIN
    FOR conta IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE (conta);
    END LOOP;
END;
```

Con la opción **REVERSE**

Ejemplo-26

```
BEGIN
    FOR conta IN REVERSE  1..5 LOOP
        DBMS_OUTPUT.PUT_LINE (conta);
    END LOOP;
END;
```

Con la sentencia EXIT – WHEN se puede salir del bucle antes de llegar al límite del rango.

Ejemplo-27:

```
BEGIN
    FOR conta IN  1..6 LOOP
        DBMS_OUTPUT.PUT_LINE (conta);
        EXIT WHEN conta= 2;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE ('Hola');
END;
```

Sentencia GOTO

PL/SQL dispone de una sentencia **GOTO**, aunque cualquier programa puede ser escrito utilizando solo las sentencias de control anteriores. Su uso **no es aconsejable** en general por oscurecer la lógica del programa, pero en ciertos casos muy concretos resulta ser una solución adecuada por simplificar el código.

La sentencia **GOTO** provoca una bifurcación incondicional. Hace que el control pase al punto del programa marcado por la etiqueta.

```
BEGIN
Sentencias-1;
GOTO etiquete-1;
Sentencias-2;
<<etiqueta-1>>
Sentencias-3;
END;
```

3.- Cursores.

PL/SQL maneja un área de trabajo, también llamada **área de contexto**, que utiliza para ejecutar las sentencias SQL y almacenar la información que proporciona una consulta, o por manipulación de datos con sentencias de actualización o inserción de datos. Con los cursores podemos acceder a la información contenida en esta área de contexto.

> Un cursor es un área de memoria donde se almacena el conjunto de filas devuelto por una consulta a la base de datos.

Siempre aparece cuando se ejecuta una sentencia SQL. Existen dos tipos de cursores: implícitos y explícitos.

- **Cursores implícitos**: Creado por el servidor Oracle para analizar y ejecutar las sentencias SQL. Se crean automáticamente para todas las sentencias DML y el SELECT ... INTO... del PL/SQL
- **Cursores explícitos**: son los definidos por el programador.

3.1.- Cursores implícitos.

No necesitan ser declarados por el programador, ya que se declara **implícitamente** un cursor en todas las sentencias de DML y SELET de PL/SQL que devuelven una sola fila. En caso de que devuelvan más de una fila se produciría un error que deberíamos tratar en el bloque de excepciones.

- **Atributos:**

  Cada cursor tiene 4 atributos que podemos usar para obtener información sobre la ejecución del mismo o sobre los datos. Estos atributos pueden ser usados en PL/SQL, pero no en SQL. Estos atributos se refieren en general a cursores implícitos y explícitos ya que tienen que ver con las operaciones que hayamos realizado con el cursor.

  Son los Siguientes:

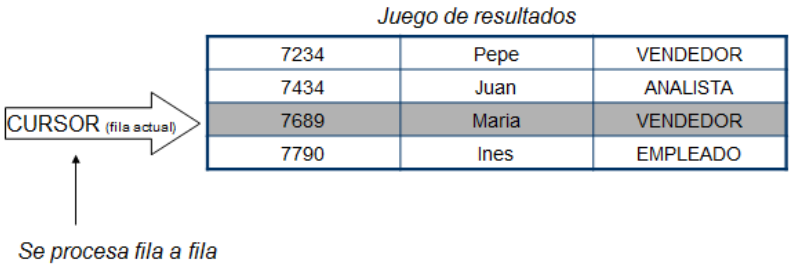
- **SQL%ROWCOUNT**: Devuelve el número de filas afectadas por la sentencia SQL (UPDATE, DELETE, INSERT),o el número de filas devueltas por un SELECT ... INTO ..
- **SQL%FOUND**: Devuelve TRUE si la sentencia SQL (UPDATE, DELETE, INSERT) afecta a una o más filas. O un SELECT ... INTO ..devuelve una o más filas. En otro caso devolverá FALSE.
- **SQL%NOTFOUND**: Devuelve TRUE si la sentencia no afecta a ninguna fila, es decir ha fallado
- **SQL%ISOPEN:** Para los cursores implícito siempre devuelve FALSE, ya que se cierran automáticamente después de ejecutar una SQL

3.2.- Cursores Explícitos.

Los cursores explícitos son declarados y definidos por el programador. Cuando una consulta devuelve múltiples filas (lo que se llama juego de resultados), podemos declarar explícitamente un cursor para procesar las filas devueltas. Cuando declaramos un cursor, lo que hacemos es darle un nombre y asociarle una consulta.
Un programa PL/SQL: abre el cursor, procesa filas devueltas por la consulta, y después cierra el cursor.

El cursor nos permite procesar individualmente las filas devueltas por una sentencia SELECT. El tamaño del cursor explícito es el número de filas que cumplen los criterios de búsqueda de la sentencia SELECT

El cursor marca la posición actual en el juego de resultados



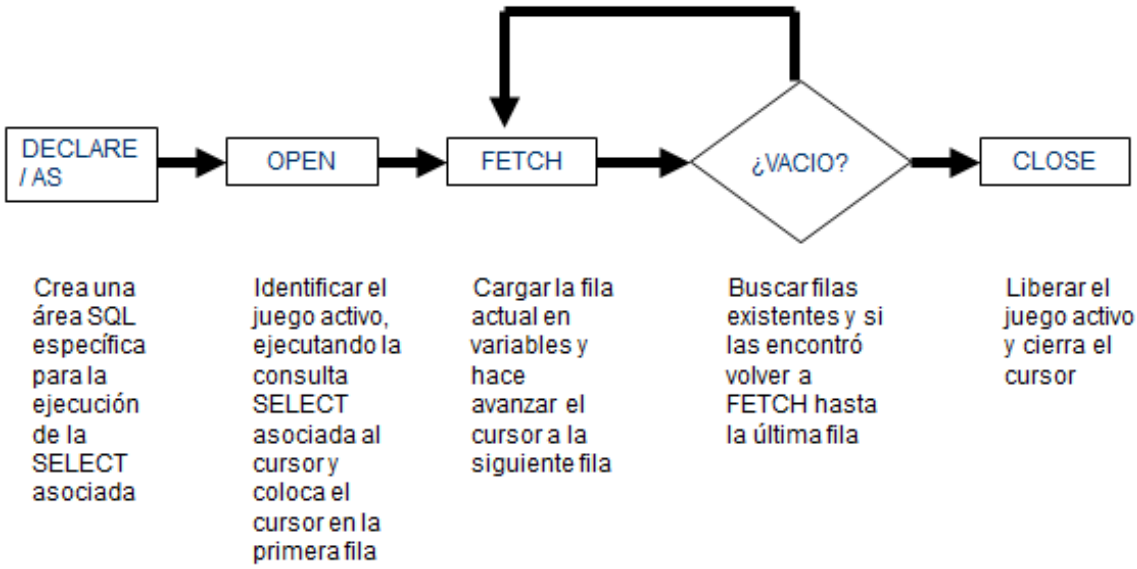
3.2.1.- Funciones de los cursores explícitos:

- Puede procesar más allá de la primera fila devuelta por la consulta
- La manipulación de las filas devueltas (JUEGO DE RESULTADOS) se realiza fila a fila.
- Comprueba la fila que está siendo procesada actualmente
- Permiten al programador controlar la fila manualmente

3.2.2.- Control de cursores.

Para poder utilizar un cursor explicito tendremos que dar los siguientes pasos:

- **Declarar el cursor:** Dándole un nombre y definiendo la estructura de la consulta **SELECT** que se va a ejecutar en él.
- **Abrir el cursor:** Con la sentencia **OPEN**, que ejecuta la consulta y resuelve las variables a las que hace referencia. Las filas devueltas por la consulta se denominan **JUEGO ACTIVO**, y están disponibles para ser recuperadas.
- **Recuperar datos del cursor**: La sentencia **FETCH** toma la fila actual del cursor y la carga en variables. Cada recuperación hace que el cursor se mueva hacia la siguiente fila del juego activo.
- **Cierre del cursor**: Cuando se llega al final la sentencia **CLOSE** libera el juego activo de filas. En este momento es posible volver a abrir el cursor para establecer un juego activo refrescado.



- **Declaración del cursor.**

Sintaxis:

```
DECLARE
    CURSOR nombreCursor  IS SELECT…..
```

Ejemplo-28

```
DECLARE
    CURSOR deptoCursor IS SELECT * FROM departamento WHERE numedep= 10;
    CURSOR empleCursor IS SELECT numemp, apell FROM empleado;
```

El nombre del cursor **nombreCursor** es un identificador, no es una variable, se utiliza para identificar la consulta, no se puede utilizar en expresiones. Las reglas para nombrar los cursores son las mismas que las descritas para nombrar las variables.

- **Apertura del cursor.**

Sintaxis:

```
BEGIN
    ……
    OPEN nombreCursor;
```

Cuando se abre un cursor la sentencia OPEN realiza las siguientes operaciones:

- Asigna memoria dinámicamente.
- Analiza la sentencia SELECT
- Identifica el juego de resultados que cumplen la SELECT, pero no se cargan en las variables de memoria hasta que se ejecute FETCH
- Posiciona el puntero antes de la 1ª fila del juego activo

- **Recuperación de datos.**

Sintaxis:

```
BEGIN
    …..
    FETCH nombreCursor INTO [variable1, variable2,…];
```

La sentencia FETCH:

- Recupera los valores de la fila actual y los almacena en las variables.
- Las variables se definirán en el DECLARE.
- Se deben incluir el mismo número de variables que columnas devuelve la sentencia SELECT (los tipos han de ser compatibles).
- Cada vez que se ejecuta FETCH el puntero se desplaza a la siguiente fila del juego activo.
- Se utiliza en combinación de un bucle, para saber cuándo hemos llegado al final del juego de resultados.

Ejemplo-29

```
DECLARE
      vEmp      empleado.numemp%type;
      vApell    empleado.apell%type;
      CURSOR  empCursor  IS  SELECT numemp, apell FROM empleado;
BEGIN
      ....
      OPEN empCursor;
      LOOP
            FETCH empCursor  INTO vEmp, vApell; --si devuelve filas carga la 1ª
            EXIT  WHEN  empCursor%NOTFOUND; --si no devuelve filas ➔ TRUE y sale
            DBMS_OUTPUT.PUT_LINE (vEmp ||' '||vApell);
      END LOOP;
      CLOSE empCursor;
END;
```

- **Cierre del cursor.**

  Sintaxis:

```
BEGIN
      .....
      CLOSE nombreCursor;
```

  Una vez completado el procesamiento de la sentencia **SELECT** es necesario cerrar el cursor, para liberar el área de contexto en memoria. Una vez cerrado el cursor no se pueden recuperar datos.

  Existe un número máximo de cursores abiertos por usuario determinado por el parámetro OPEN_CURSORS, por defecto son 50.

3.2.3.- Atributos de los cursores explícitos.

Cada cursor tiene asociados cuatro atributos predefinidos. Cuando se abre un cursor estos atributos devuelven información sobre la ejecución de la sentencia SQL asociada al cursor.

- **nombreCursor%FOUND:** Devolverá TRUE si el último FETCH ha devuelto una fila y FALSE en caso contrario.
- **nombreCursor%NOTFOUND:** Es lógicamente lo contrario a %FOUND. Devolverá TRUE si el último FETCH no ha devuelto una fila.
- **nombreCursor%ISOPEN:** Devolverá a TRUE si el cursor está abierto y FALSE en caso contrario.
- **nombreCursor%ROWCOUNT**: Para un cursor abierto devuelve el número de filas que hemos procesado hasta el momento.

Ejemplo-30

```
DECLARE
    vEmp      NUMBER(4);
    vCom      NUMBER(10);
    CURSOR  empCursor1  IS  SELECT numemp, comision FROM empleado;
BEGIN
    OPEN empCursor1;
    LOOP
        FETCH empCursor1  INTO vEmp,vCom;
        IF empCursor1%FOUND THEN
            DBMS_OUTPUT.PUT_LINE ('Numero empleado: '||vEmp ||' Comision '||vCom);
        ELSE
            EXIT;
        END IF;
    END LOOP;
    CLOSE empCursor1;
END;
```

Ejemplo-31

```
DECLARE
    vEmp      NUMBER(4);
    vCom      NUMBER(10);
    CURSOR  empCursor1  IS  SELECT numemp, comision FROM empleado;
BEGIN
    OPEN empCursor1;
    LOOP
        FETCH empCursor1  INTO vEmp,vCom;
        IF empCursor1%ROWCOUNT < 5 THEN
            DBMS_OUTPUT.PUT_LINE ('Numero empleado: '||vEmp ||' Comision '||vCom);
        ELSE
            EXIT;
        END IF;
    END LOOP;
    CLOSE empCursor1;
END;
```

3.2.4.- Sentencia FOR para el manejo de cursores.

El manejo de cursores explícitos se simplifica en la mayoría de los casos mediante el uso de un formato de la sentencia FOR disponible para esta función.

Permite procesar filas en un cursor explícito, es otra alternativa al FETCH, mucho más sencilla de usar

Sintaxis:

```
FOR  nombreRegistro  IN  nombreCursor  LOOP
      sentencias;
END LOOP;
```

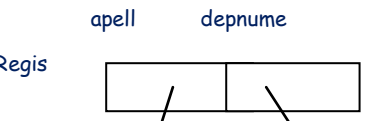
Dónde:

- **nombreCursor** debe estar declarada anteriormente en DECLARE
- Facilita el procesamiento de cursores explícitos
- Abre el cursor, recupera filas en cada iteración y cierra automáticamente el cursor.
- La variable **nombreRegistro** es de tipo **nombreCursor%ROWTYPE** y queda declarada implícitamente con el FOR, no siendo necesario definirla en DECLARE y desplazando automáticamente el cursor a la siguiente fila.
- El registro no puede ser referenciado fuera del bucle FOR.

Ejemplo-32

```
DECLARE
    CURSOR empleCursor IS SELECT apell, depnume FROM empleado;

BEGIN
    FOR vRegis IN empleCursor LOOP
        -- la variable de registro vReisg queda definida implícitamente
        DBMS_OUTPUT.PUT_LINE (vRegis.apell || ' '||vRegis.depnume);
    END LOOP;
END;
```



3.2.5.- Uso de parámetros en el cursor explícito.

El uso de parámetros permite pasar valores a un cursor cuando éste es abierto para que se utilicen cuando se evalúa la sentencia **SELECT**.

Permiten devolver un juego activo distinto cada vez.

Sintaxis

```
CURSOR nombreCursor [(parametro1  tipo_dato, parametro2  tipo_dato,...)] IS  SELECT.....;
```

Dónde:

Tipo_dato: corresponde solo al tipo de dato, no al tamaño

Para utilizar el cursor:

```
OPEN nombreCursor (valor1, valor2,...);
```

Donde

valor1, valor2: Son los valores con los que se va a ejecutar la consulta SELECT

Ejemplo-33

```
DECLARE
    .....
    CURSOR   empleCursor  (vDepnume  NUMBER,  vOficio VARCHAR)  IS
        SELECT apell, salario FROM empleado WHERE depnume= vDepnume  and oficio=vOficio;
    ...
BEGIN
    .....
    OPEN empCursor (10,'ANALISTA');  -- se ejecuta el SELECT con estos datos
    ....
END;
```

3.2.6.- Utilización de cursores para modificación y eliminación de filas de una tabla.

PL/SQL permite actualizar o borrar las filas seleccionadas por un cursor explícito.

Cuando queremos actualizar filas con un cursor hay que tener en cuenta que las filas a ser actualizadas o borradas, quedan bloqueadas tan pronto se abra el cursor y serán desbloqueadas al terminar las actualizaciones.

Para crearlos será necesario añadir **FOR UPDATE** al final de la declaración del cursor, y en la sentencia de actualización **UPDATE** o **DELETE** será necesario añadir la cláusula **WHERE CURRENT OF nombreCursor** para actualizar SOLO la fila recuperada con **FETCH.**

Sintaxis:

```
CURSOR nombreCursor IS  SELECT ...      .. FOR UPDATE;
En el cuerpo para realizar la actualización solo de la fila donde está el cursor
{UPDATE | DELETE} ..... WHERE CURRENT OF nombreCursor;
```

Ejemplo-34

Realizar un bloque que permita subir el salario de todos los empleados de un departamento un tanto por ciento. Ambos valores se introducirán por teclado.

```
ACCEPT numerodep PROMPT 'Introduce el número de departamento: '
ACCEPT subida PROMPT 'Introduce el porcentaje a subir: '
DECLARE
      CURSOR empleCursor IS
      SELECT oficio, salario FROM empleado WHERE depnume = &numerodep FOR UPDATE;
      vRegis  empleCursor%ROWTYPE;
      vIncremento   number(12,2);
BEGIN
     OPEN empleCursor;
     FETCH empleCursor into vRegis;
     WHILE empleCursor%FOUND LOOP
          vIncremento:=(vRegis .salario/100)* &subida;
          UPDATE empleado SET salario = salario + vIncremento
                WHERE CURRENT OF empleCursor;
          FETCH empleCursor INTO vRegis;
     END LOOP;
END;
```