U.T.1 P L / S Q L

1. INTRODUCCIÓN

PL (Procedural Language)/SQL. La unidad lógica en PL es el bloque. Todos los programas están compuestos por bloques que pueden ser anclados o no. El diseño de PL se basa en el Lenguaje ADA y, consta de 3 secciones, la sección declarativa, la ejecutable y la de manejo de excepciones. Sólo la sección ejecutable es obligatoria.

Ventajas en la utilización de PL/SQL

PL/SQL es un lenguaje de procesamiento de transacciones completamente portable y con un alto rendimiento, que proporciona las siguientes ventajas al ser utilizado:

Soporte para SQL
Soporte para la programación orientada a objetos
Mejor rendimiento
Alta productividad
Completa portabilidad
Integración con Oracle garantizada
Seguridad

Soporte para SQL

SQL se ha convertido en el lenguaje estándar de bases de datos por su flexibilidad de uso y facilidad de aprenderlo. Unos pocos comandos permiten la fácil manipulación de prácticamente toda la información almacenada en una base de datos.

SQL es no-procedural, lo cual significa que es Oracle quien se preocupará de cómo ejecutar de la mejor manera un requerimiento señalado en una sentencia SQL. No es necesaria la conexión entre varias sentencias porque Oracle las ejecuta de a una a la vez.

PL/SQL le permite a usted una completa manipulación de los datos almacenados en una base Oracle, proporciona comandos de control de transacciones y permite utilizar las funciones de SQL, operadores y pseudocolumnas. Así, usted puede manipular los datos en Oracle de una manera flexible y segura. Además, PL/SQL soporta tipos de datos de SQL,

lo que reduce la necesidad de convertir los datos al pasar de una a otra aplicación.

PL/SQL también soporta SQL dinámico, una avanzada técnica de programación que convierte a sus aplicaciones en más flexibles y versátiles.

Soporte para Programación Orientada a Objetos

Los objetos se han convertido en una herramienta ideal para modelar situaciones de la vida real. Con su utilización es posible reducir el costo y tiempo de construcción de aplicaciones complejas. Otra ventaja es que utilizando una metodología de este tipo es posible mantener diferentes equipos de programadores construyendo aplicaciones basadas en el mismo grupo de objetos.

Permitir el encapsulamiento del código en bloques es el primer paso para la implementación de métodos asociados a diferentes tipos de objetos construidos también con PL/SQL.

Mejor rendimiento

Sin PL/SQL, Oracle tendría que procesar las instrucciones una a una. Cada llamada produciría un overhead considerable, sobre todo si consideramos que estas consultas viajan a través de la red.

Por el contrario, con PL/SQL, un bloque completo de sentencias puede ser enviado cada vez a Oracle, lo que reduce drásticamente la intensidad de comunicación con la base de datos. Los procedimientos almacenados escritos con PL/SQL son compilados una vez y almacenados en formato ejecutable, lo que produce que las llamadas sean más rápidas y eficientes. Además, ya que los procedimientos almacenados se ejecutan en el propio servidor, el tráfico por la red se reduce a la simple llamada y el envío de los parámetros necesarios para su ejecución.

El código ejecutable se almacena en caché y se comparte a todos los usuarios, redundando en mínimos requerimientos de memoria y disminuyendo el overhead al mínimo.

Alta productividad

Si se decide utilizar otros productos de Oracle como *Oracle Forms* y *Oracle Reports*, es posible integrar bloques completos de PL/SQL en un trigger de Oracle Forms, debido a que PL/SQL es el mismo en todos los ambientes.

Completa portabilidad

Las aplicaciones escritas con PL/SQL son portables a cualquier sistema operativo y plataforma en la cual se encuentre corriendo Oracle. En otras palabras, PL/SQL corre dondequiera que se encuentre corriendo Oracle también. Esto significa que se pueden codificar librerías que podrán ser reutilizadas en otros ambientes.

Integración con Oracle

PL/SQL y los lenguajes SQL en general se encuentran perfectamente integrados. PL/SQL soporta todos los tipos de datos de SQL. Los atributos %TYPE y %ROWTYPE integran PL/SQL con SQL, permitiendo la declaración de variables basado en tipos de columnas de tablas de la base de datos. Lo anterior provee independencia de los datos, reduce costos de mantenimiento y permite a los programas adaptarse a los cambios en la base de datos para cumplir con las nuevas necesidades del negocio.

Seguridad

Los procedimientos almacenados construidos con PL/SQL habilitan la división de la lógica del cliente con la del servidor. De esta manera, se previene que se efectúe manipulación de los datos desde el cliente. Además, se puede restringir el acceso a los datos de Oracle, permitiendo a los usuarios la ejecución de los procedimientos almacenados para los cuales tengan privilegios solamente.

Para ejecutar

SQL > {PROGRAMA} / [→ RUN]

Para ver errores

SQL > Show errors;

NORMAL

```
REM COMENTARIOS

DECLARE
      Sección declarativa
  ┌BEGIN
      Sección ejecutable
   EXCEPCTION
      Manejo excepciones
  └END
```

ANIDADOS

```
   DECLARE
      Declaraciones;
┌ BEGIN
      …
      DECLARE
            Declaraciones;
       ┌BEGIN
            …
       └END
      EXCEPTION
            Excepciones;
└ END
```

2. FUNDAMENTOS PL

PL/SQL no es un lenguaje creado para interactuar con el usuario, sino para trabajar con la BD. Prueba de ello es que no dispone de órdenes para introducir datos por teclado, ni para mostrarlos por pantalla.

Oracle incorpora el paquete DBMS_OUTPUT, que incluye el procedimiento PUT_LINE, que permite visualizar textos por pantalla, para que funcione correctamente se activa:

SQL > SET serveroutput ON;
Se podrá utilizar DBMS_OUTPUT.PUT_LINE ('Expresión')

Ejemplo1:

- Set serveroutput ON;
  Begin
  DBMS_OUTPUT.PUT_LINE ('Hola mundo');
  END;
  /

Ejemplo2:

```
> Set serveroutput ON;
DECLARE
        v_sal number;
 BEGIN
        select sal into v_Sal from emp where empno = '7654';
        DBMS_OUTPUT.PUT_LINE (v_sal);
END;
/
```

Tipos de Bloques

- **Anónimos.** Se construyen de manera dinámica y se ejecutan 1 sola vez.
- **Nominados.** Son bloques anónimos con una etiqueta que le da nombre al bloque. También se construyen de manera dinámica.
- **Subprogramas.** Son procedimientos, paquetes y funciones almacenados en la base de datos que no cambian después de su construcción y se ejecutan múltiples veces, mediante una llamada al procedimiento o función.
- **Disparadores.** Son bloques nominados que se almacenan en la base de datos que se ejecutan múltiples veces de manera implícita cada vez que tiene lugar el suceso de disparo.

Características del lenguaje: Unidades Léxicas

- A-Z, a-z
- 0 – 9
- +, -, *, /, <, >, =
- Delimitadores:
  - ( ) → Inicio fin de expresión
  - ; → fin de línea
  - % → Indicador de atributo
  - := → Asignación
  - . → Selector de componente
  - < > → Distinto → !=
  - << >> → Comienzo, fin de etiqueta
  - -- → Comentario 1 línea
  - /* → comentario de más líneas → /* linea1, linea2… */
  - || → Concatenar
  - @ → Enlace a la BD
  - ‘hdc‘ → Indicador de caracteres

Un comentario en PL/SQL puede tener dos formatos:

1. Líneas que empiezan por un doble guión (--), en cuyo caso toda la línea es tratada como un comentario.

2. Comentarios al estilo del lenguaje de programación C, como por ejemplo /* Esto es un comentario */, donde sólo el texto que hay entre "/*" y "*/" es tratado como un comentario.

Delimitadores e Identificadores

Un delimitador es un símbolo simple o compuesto que tiene un significado especial dentro de PL/SQL. Por ejemplo, es posible utilizar delimitadores para representar operaciones aritméticas, por ejemplo:

Símbolo	Significado
+	operador de suma
%	indicador de atributo
`	delimitador de caracteres
.	selector de componente
/	operador de división
(	expresión o delimitador de lista
)	expresión o delimitador de lista
:	indicador de variable host
,	separador de ítems
*	operador de multiplicación
"	delimitador de un identificador entre comillas
=	operador relacional
<	operador relacional
>	operador relacional
@	indicador de acceso remoto
;	terminador de sentencias
-	negación u operador de substracción

Los delimitadores compuestos consisten de dos caracteres, como por ejemplo:

Símbolo	Significado
:=	operador de asignación
=>	operador de asociación
||	operador de concatenación
**	operador de exponenciación
<<	comienzo de un rótulo
>>	fin de un rótulo
/*	comienzo de un comentario de varias líneas
*/	fin de un comentario de varias líneas
..	operador de rango
<>	operador relacional
!=	operador relacional
^=	operador relacional
<=	operador relacional
>=	operador relacional
--	comentario en una línea

Los identificadores incluyen constantes, variables, excepciones, cursores, subprogramas y paquetes. Un identificador se forma de una letra, seguida opcionalmente de otras letras, números, signo de moneda, underscore y otros signos numéricos.

La longitud de un identificador no puede exceder los 30 caracteres. Se recomienda que los nombres de los identificadores utilizados sean descriptivos.

Algunos identificadores especiales, llamados *palabras reservadas*, tienen un especial significado sintáctico en PL/SQL y no pueden ser redefinidos. Son palabras reservadas, por ejemplo, BEGIN, END, ROLLBACK, etc.

Declaración de variables

Nombre_var tipo [constant] [not null] [: = valor]

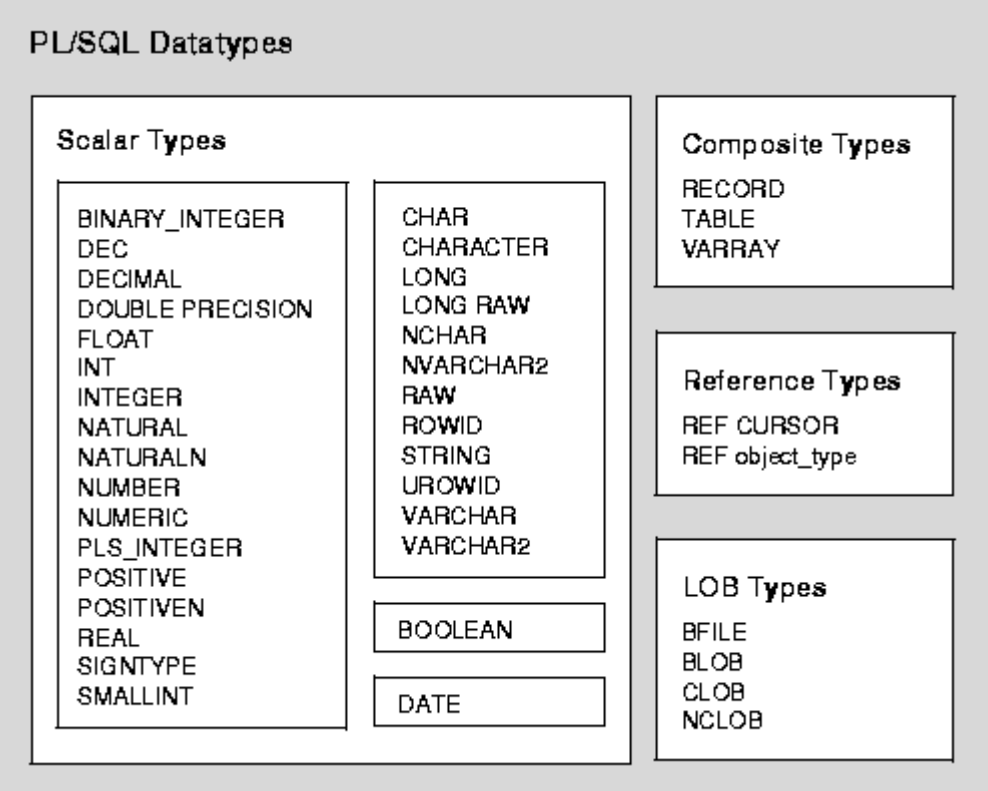
Ejemplo

```
V_num number  : = 45;
v_descrip  varchar2  (50);
v_var  number not null : = 0;
```
→ (obligatorio inicializar)

Otra forma de dar un valor a una variable:

```
Select nombre into var from tabla;
```

Tipos de datos



>> Numéricos
Number (x, y);
binary_integer; → Los valores se almacenan en formato binario con complemento a 2, lo que permite usarlas en cálculos sin necesidad de convertirlos. Se utiliza para valores temporales y que no vayan a ser almacenados en la BD (Ejemplo → Contadores).

>> Carácter
Varchar2 (nº caracteres)
Char (x)
Long
>> Fecha
Date
>> Boolean
v_var Boolean
>> Rowid

Conversiones

Algunas veces se hace necesario convertir un valor desde un tipo de dato a otro. En PL/SQL se aceptan las conversiones de datos *implícitas* y *explícitas*.

Una conversión explícita es aquella que se efectúa utilizando las funciones predefinidas. Por ejemplo, para convertir un valor de carácter a fecha o número se utiliza TO_DATE o TO_NUMBER.

Existe una cantidad limitada de funciones de conversión, que implementan esta característica de conversión explícita.

Cuando se hace necesario, PL/SQL puede convertir un tipo de dato a otro en forma implícita. Esto significa que la interpretación que se dará a algún dato será el que mejor se adecue dependiendo del contexto en que se encuentre. Tampoco significa que todas las conversiones son permitidas. Algunos ejemplos de conversión implícita más comunes se dan cuando variables de tipo *char* se operan matemáticamente para obtener un resultado numérico.

Si PL/SQL no puede decidir a qué tipos de dato de destino puede convertir una variable se generará un error de compilación.

Tabla de conversiones implícitas

Hasta / Desde	BIN_INT	CHAR	DATE	LONG	NUMBER	PLS_INT	RAW	ROWID	VARCHAR2
BIN_INT		X		X	X	X			X
CHAR	X		X	X	X	X	X	X	X
DATE		X		X					X
LONG		X					X		X
NUMBER	X	X		X		X			X
PLS_INT	X	X		X	X				X
RAW		X		X					X
ROWID		X							X
VARCHAR2	X	X	X	X	X	X	X	X	

Declaración de Constantes

En la declaración de una constante (muy similar a la de una variable), se debe incorporar la palabra reservada "constant" e inmediatamente asignar el valor deseado. En adelante, no se permitirán reasignaciones de valores para aquella constante que ya ha sido definida.

Ejemplo: credit_limit CONSTANT real := 5000.00 ;

Atributos de variables

Hay muchos casos donde las variables PL se emplean para manipular datos almacenados en una tabla en la BD. En este caso la variable debe tener el mismo tipo que la columna de la tabla. No necesitamos saber al completo el tipo de nuestra variable, sino que le asignamos el de una columna u otra variable.

- **% type**

El atributo %TYPE define el tipo de una variable utilizando una definición previa de otra variable o columna de la base de datos.

Ejemplo:

```
DECLARE
  credito        REAL(7,2);
  debito         credito%TYPE;
  …
```

También se podría declarar una variable siguiendo el tipo de un campo de alguna tabla, como por ejemplo en:

```
debito        cuenta.debe%TYPE;
```

La ventaja de esta última forma es que no es necesario conocer el tipo de dato del campo "debe" de la tabla "cuenta", manteniendo la independencia necesaria para proveer más flexibilidad y rapidez en la construcción de los programas.

```
SQL>  variable  tabla.columna  %type;
SQL>  v_nombre  emp.ename   %type;


cumple date;
cumple1 cumple %type;
```

- **% rowtype** - Creamos una variable que es una estructura o un registro que contiene las mismas columnas que la tabla a la que estamos haciendo referencia.

El atributo %ROWTYPE precisa el tipo de un registro (*record*) utilizando una definición previa de una tabla o vista de la base de datos. También se puede asociar a una variable como del tipo de la estructura retornada por un cursor.

Ejemplo:

```
DECLARE
        emp_rec   emp%ROWTYPE;
        CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
        dept_rec   c1%ROWTYPE;
```

En este ejemplo la variable *emp_rec* tomará el formato de un registro completo de la tabla *emp* y la variable *dept_rec* se define por una estructura similar a la retornada por el cursor *c1*.

Ejemplo

```
SQL> Fila  alumnos  %rowtype;
     fila.nombre;
     fila.codiop_curso;
```

Llenarla:

```
Select * into fila from ALUMNOS where codal=1222;
```

Cómo asignar valores a variables

Es posible asignar valores a las variables de dos formas. La primera utiliza el operador "**:=**". La variable se ubica al lado izquierdo y la expresión al lado derecho del símbolo.

Por ejemplo:

```
tax := price * tax_rate ;
```

Variables de acoplamiento

SQL puede asignar memoria, pudiéndose usar dichas posiciones de memoria en bloques PL. Como el almacenamiento se asigna fuera del bloque, puede usarse para bloques sucesivos. Se definen utilizando la clausula VARIABLE

Dentro del bloque se delimitan con 2 puntos delante de la variable de acoplamiento y sólo pueden ser del tipo char, varchar2 o number .

Ejemplo

```
SQL > VARIABLE v_cont number;
 BEGIN
select count ( * ) into  :v_cont from student where course = 102;
END;

SQL > print v_cont;
```

Hacer ejercicio

Contar cuantos empleados tiene el departamento 30, usando variables de acoplamiento.

```
Sql >VARIABLE v_cont number;
 BEGIN
        select count ( * ) into  :v_cont from emp  where deptno=30;
END;
SQL > print v_cont;
```

```
Procedimiento PL/SQL terminado correctamente.
SQL> print v_cont;
V_CONT
     6
```

Variables de sustitución

SQL PLUS dispone de un mecanismo que permite al usuario realizar operaciones de entrada. SQL PLUS hace la sustitución textual de la variable antes de que el bloque PL o la orden SQL se envíen al servidor. Se designan mediante el carácter (&).

No hay memoria asignada para las variables de sustitución. Por eso, sólo se puede utilizar para recoger datos del teclado (luego lo guardamos en otra variable o desaparece).

Ejemplo) Visualizar aquellos estudiantes que se apellidan igual que los introducidos por teclado.

Select * FROM students where apellido = '&ape'

Ámbito y visibilidad de una variable PL

Es aquella parte del programa en la que se puede acceder a dicha variable. Para una variable PL el ámbito va desde la definición de la variable hasta el final del bloque.

Ejemplo

```
<<1_etiqueta>>
DECLARE
      V_numero  number (3,2);

BEGIN
      DECLARE
          v_caracter varchar (20);
      BEGIN
          …   → v_caracter es visible pero v_numero NO,
                      1_etiqueta.v_numero  → Así le hago visible
      END;


END;  1_etiqueta;
```

Las referencias a un identificador son resueltas de acuerdo a su alcance y visibilidad dentro de un programa. El *alcance* de un identificador es aquella región de la unidad de programa (bloque, subprograma o paquete) desde la cual se puede referenciar al identificador.

Un identificador es visible sólo en las regiones en que se puede referenciar.

La figura muestra el alcance y visibilidad de la variable **x**, la cual está declarada en dos bloques cerrados diferentes.

Los identificadores declarados en un bloque de PL/SQL se consideran locales al bloque y globales a todos sus sub-bloques o bloques anidados. De esto se desprende que un mismo identificador no se puede declarar dos veces en un mismo bloque pero sí en varios bloques diferentes, cuantas veces se desee.

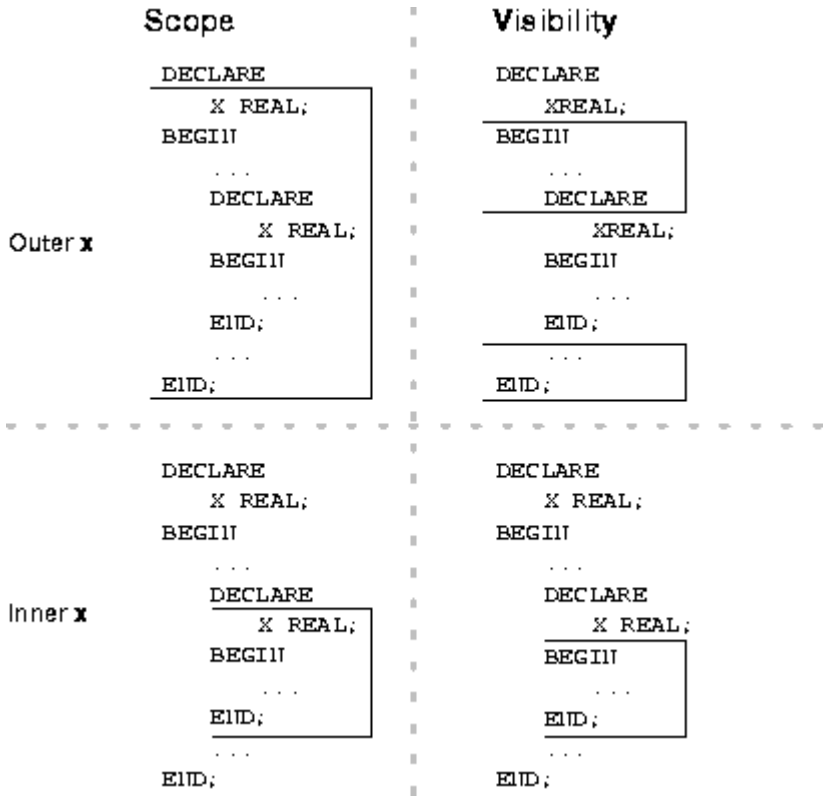


Figura Alcance y Visibilidad de Identificadores

Este ejemplo ilustra el alcance y visibilidad (o posibilidad de ser referenciada) de una determinada variable **x**, que ha sido declarada en dos bloques anidados. La variable más externa tiene un alcance más amplio pero cuando es referenciada en el bloque en que se ha declarado otra variable con el mismo nombre, es esta última la que puede ser manipulada y no la primera.

Estructura de control

Sentencia IF

A menudo es necesario tomar alternativas de acción dependiendo de las circunstancias. La sentencia IF permite ejecutar una secuencia de acciones condicionalmente. Esto es, si la secuencia es ejecutada o no depende del valor de la condición a evaluar. Existen tres modos para esta instrucción: IF – THEN, IF – THEN – ELSE y IF – THEN – ELSIF.

IF – THEN

Este es el modo más simple y consiste en asociar una condición con una secuencia de sentencias encerradas entre las palabras reservadas THEN y END IF (no ENDIF).

Ejemplo:

```
IF condición THEN
      secuencia_de_sentencias
END IF;
```

La secuencia de sentencias es ejecutada sólo si la condición es verdadera. Si la condición es falsa o nula no realiza nada. Un ejemplo real de su utilización es la siguiente:

```
IF condición THEN
      calcular_bonus (emp_id)
      UPDATE sueldos SET pago = pago + bonus WHERE emp_no =
      emp_id;
END IF;
```

IF – THEN – ELSE

Esta segunda modalidad de la sentencia IF adiciona una nueva palabra clave: ELSE, seguida por una secuencia alternativa de acciones:

```
IF condición THEN
        secuencia_de_sentencias_1
ELSE
        secuencia_de_sentencias_2
END IF;
```

La secuencia de sentencias en la cláusula ELSE es ejecutada solamente si la condición es falsa o nula. Esto implica que la presencia de la cláusula ELSE asegura la ejecución de alguna de las dos secuencias de estamentos. En el ejemplo siguiente el primer UPDATE es ejecutado cuando la condición es verdadera, en el caso que sea falsa o nula se ejecutará el segundo UPDATE:

```
IF tipo_trans = 'CR' THEN
        UPDATE cuentas SET balance = balance + credito WHERE …
ELSE
        UPDATE cuentas SET balance = balance – debito WHERE …
END IF;
```

Las cláusulas THEN y ELSE pueden incluir estamentos IF, tal como lo indica el siguiente ejemplo:

```
IF tipo_trans = 'CR' THEN
        UPDATE cuentas SET balance = balance + credito WHERE …
ELSE
        IF  nuevo_balance >= minimo_balance THEN
UPDATE cuentas SET balance = balance – debito WHERE …
        ELSE
RAISE fondos_insuficientes;
        END IF;
END IF;
```

IF – THEN – ELSIF

Algunas veces se requiere seleccionar una acción de una serie de alternativas mutuamente exclusivas. El tercer modo de la sentencia IF utiliza la clave ELSIF (no ELSEIF) para introducir condiciones adicionales, como se observa en el ejemplo siguiente:

```
IF condición_1 THEN
        secuencia_de_sentencias_1
ELSIF condición_2 THEN
        secuencia_de_sentencias_2
ELSE
        secuencia_de_sentencias_3
END IF;
```

Si la primera condición es falsa o nula, la cláusula ELSIF verifica una nueva condición. Cada sentencia IF puede poseer un número indeterminado de cláusulas ELSIF; la palabra clave ELSE que se encuentra al final es opcional.

Las condiciones son evaluadas una a una desde arriba hacia abajo. Si alguna es verdadera, la secuencia de sentencias que corresponda será ejecutada. Si cada una de las condiciones analizadas resultan ser falsas, la secuencia correspondiente al ELSE será ejecutada:

```
BEGIN
        …
  IF  sueldo > 50000 THEN
     bonus : = 1500;
  ELSIF sueldo > 35000 THEN
     bonus : = 500;
  ELSE
     bonus : = 100;
  END IF;
  INSERT INTO sueldos VALUES (emp_id, bonus, );
END;
```

Si el valor de sueldo es mayor que 50.000, la primera y segunda condición son verdaderas, sin embargo a *bonus* se le asigna 1500, ya que la segunda condición jamás es verificada. En este caso sólo se verifica la primera condición para luego pasar el control a la sentencia INSERT.

Ejemplo

```
DECLARE
    v_numeroasientos  rooms.number_seat  %type;
    v_comentario  varchar2 (3B);

BEGIN
    select number_seat into v_numeroasientos from rooms
where room.id = 999;
    IF  v_numeroasientos < 50 then
        v_comentario : = 'Muy pequeña';
        insert into temp_table (char_col) values ('para grupos
de 50');
    ELSIF  v_numeroasientos >100 then
        v_comentario : = 'muy grande';
        insert into temp_table (char_col) values ('para grupos
de 100');
    ELSE
        v_comentario : = 'normal';

    insert into temp_table (char_col) values ('normal');
    END IF;
END;
```

BUCLES

Existen tres modalidades para esta instrucción: LOOP, WHILE – LOOP y FOR – LOOP.

1. bucle LOOP

El modo básico (o infinito) de LOOP encierra una serie de acciones entre las palabras clave LOOP y END LOOP, como en el siguiente ejemplo:

```
LOOP
secuencia_de_instrucciones
END LOOP;
```

Condición de parada

Con cada iteración del ciclo las sentencias son ejecutadas. Para terminar estos ciclos de ejecución se utiliza la palabra clave EXIT. Es posible ubicar innumerables EXIT dentro del loop, obviamente ninguno fuera de él. Existen dos modalidades para utilizar esta sentencia: EXIT y EXIT – WHEN.

EXIT

La cláusula EXIT obliga al loop a concluir incondicionalmente. Cuando se encuentra un EXIT en el código, el loop es completado inmediatamente y pasa el control a la próxima sentencia.

```
LOOP
    IF ranking_credito < 3 THEN
                                    ...
EXIT;  --Termina el loop inmediatamente
    END IF;
END LOOP;
```

Es necesario recordar que esta sentencia debe estar dentro del loop. Para completar un bloque PL/SQL antes de que su final natural sea alcanzado, es posible utilizar la instrucción RETURN.

EXIT – WHEN

Esta sentencia permite terminar el loop de manera condicional. Cuando se encuentra un EXIT la condición de la cláusula WHEN es evaluada. Si la condición es verdadera el loop es terminado y el control es pasado a la próxima sentencia.

Ejemplo)

```
LOOP
FECTH c1 INTO …
EXIT WHEN c1%NOTFOUND; -- termina el loop si la condición es verdadera
  …
END LOOP;
CLOSE c1;
```

Hasta que la condición no sea verdadera el loop no puede completarse, esto implica que necesariamente dentro de las sentencias el valor de la condición debe ir variando. En el ejemplo anterior si la ejecución de FETCH retorna una fila la condición es falsa. Cuando FETCH falla al retornar una fila, la condición es verdadera por lo que el loop es completado y el control es pasado a la sentencia CLOSE.

La sentencia EXIT – WHEN reemplaza la utilización de un IF. A modo de ejemplo se pueden comparar los siguientes códigos:

```
IF count > 100 THEN          |          EXIT WHEN count > 100;
    EXIT;                    |
END IF;                      |
```

Ambos códigos son equivalentes, pero el EXIT – WHEN es más fácil de leer y de entender.

Ejemplo: Bucle que inserta 50 filas en la tabla temporal

```
DECLARE
v_cont binary_integer := 1;

BEGIN
    LOOP
        INSERT INTO temp values (v_cont);
         v_cont : = v_cont +1;
           exit when v_cont > 50;
      END LOOP;
END;
```

2. Bucle while

WHILE condición LOOP
secuencia ordenes;
END LOOP;

La condición se evalúa antes de cada iteración. Si es verdadera se ejecuta la secuencia de órdenes y, si es falsa, el bucle termina. Si la condición del bucle no toma un valor 'true', el bucle no llega a ejecutarse.

Ejemplo)

```
DECLARE
        v_cont  binary integer : = 1;
BEGIN
        while v_cont <=50 loop
                insert into temp values (v_cont);
                v_cont : = v_cont +1;
        end loop;
END;
```

3. Bucle FOR

FOR contador_bucle IN [REVERSE] limite inferior .. limite superior LOOP
secuencia de ordenes;
END LOOP;

Contador_bucle es el índice del bucle que por defecto es binary_integer. No es necesario declararlo.

Los límites del bucle, solo se evalúan una vez, y determinan el nº total de iteraciones, en las que el índice varía, entre el límite superior e interior, incrementándose en una unidad.

[REVERSE] → sirve para inicializar al límite superior e ir decrementando en una unidad.

Ejemplo)

```
BEGIN
        FOR v_contador IN 1..50  Loop
                insert into temp values (v_contador);
        end loop;
END;
```

Ejemplo)

```
DECLARE
      v_lowValue  number := 10;
      v_HighValue  number := 40;

BEGIN
      FOR v_contador IN REVERSE v_LowValue .. v_HighValue Loop
            Insert INTO temp values (v_contador, 'rangos específicos');
      END LOOP;
END;
```

Nombre de las Variables

El motor de PL cuando encuentra una condición de tipo expr1 = expr2. Primero comprueba si se corresponden con columnas de la tabla con la que se está operando y después se comprueba si se trata variables del bloque PL.

```
<< e_bloqueborrar >>
DECLARE
    department  char  (3);

BEGIN
    Department  :=  'cs'
    Delete FROM classes where department = Department;


//* No diferencia entre Mayús. y Minús.  borra todas las
            filas de la tabla clases en vez de las que son 'CS'
como indica   la variable.

END;
```

Si al bloque le ponemos una etiqueta se puede usar para una variable el mismo nombre que una columna, anteponiendo la etiqueta.

e_bloqueborrar.Departament

Secuencias

CREATE SECUENCE <nombre> START WITH Nº INCREMENT BY Nº;

Ejemplo) create sequence student_sq start with 10000 increment by 2;

Una secuencia es un objeto de ORACLE que se emplea para generar series de números.

Sequence.CURRVAL → Devuelve el valor actual de una secuencia.

Sequence.NEXTVAL → Incrementa la secuencia y devuelve el siguiente valor.

Ejemplo) Insert into students (id, first_name, last_name) values (student_sq.nextval, 'Pepe', 'López');

Procesamiento de Transacciones

Existen técnicas básicas que permiten salvaguardar la consistencia de la base de datos de forma explícita, es decir, manejable por el usuario o programador.

Las tareas o *jobs* que maneja Oracle son llamadas *sesiones*. Una sesión de usuario comienza cuando se corre un programa o se conecta a la base a través de una herramienta. Oracle administra la concurrencia con los mecanismos de control adecuados, que garantizan que se mantenga siempre la integridad de los datos.

Oracle también permite la habilitación de bloqueos para controlar el acceso concurrente a los datos. Un bloqueo permite el acceso exclusivo a ciertos datos por un breve periodo de tiempo, ya sea a través de una tabla o fila de datos.

Una transacción es una serie de sentencias SQL de manipulación de datos que provee una unidad lógica de trabajo. Cada sentencia SQL corresponde a una transacción. Esta unidad es reconocida por Oracle con la finalidad de proveer la característica de asegurar las transacciones efectuadas en la base de datos (*commit*) o deshacerlas (*rollback*). Si un programa falla a la mitad de una transacción, la base de datos se recupera automáticamente hasta el último punto guardado.

Las sentencias *commit* y *rollback* permiten asegurar que todos los cambios efectuados sobre la base de datos se guardarán permanentemente o se descartarán en forma definitiva. Todas las sentencias que se ejecutan desde la ocurrencia del último *commit* o *rollback* comprenden la transacción (o grupo de transacciones) activa. La sentencia *savepoint* establece un punto de procesamiento dentro de una transacción y funciona de manera similar a un rótulo

Uso de COMMIT

La sentencia *commit* finaliza la transacción actual efectúa los cambios en la base de datos de forma permanente. Mientras un usuario no efectúa el *commit*, el resto de usuarios que accedan la misma base en forma concurrente no verán los cambios que este primer usuario ha estado efectuando. Sólo después de ejecutada la sentencia todos los usuarios de la base estarán en condiciones de ver los cambios implementados por el usuario que hace el *commit*.

Ejemplo:

```
BEGIN
  …
  UPDATE cuenta SET bal = mi_bal – debito WHERE num_cta = 7715 ;
  …
UPDATE cuenta SET bal = mi_bal + credito WHERE num_cta = 7720 ;
      COMMIT WORK;
END;
```

Uso de ROLLBACK

La sentencia *rollback* finaliza la transacción actual y deshace todos los cambios realizados en la base de datos en la transacción activa. Considérese el caso del ejemplo siguiente, donde se inserta información en tres tablas diferentes y se toma la precaución de que si se trata de insertar un valor duplicado en una clave primaria, se genera un error (controlado con la sentencia *rollback*).

3. REGISTROS y TABLAS

Son un tipo compuesto que debe ser definido primero y luego declarar variables del tipo así definido. La sintaxis de la instrucción:

```
TYPE tipo_reg IS RECORD(
        Campo1 tipo1 [ NOT NULL ] [ : = expr1 ],
        Campo2 tipo2 [ NOT NULL ] [ : = expr2 ],
        … );
```

Ejemplo)

```
Declare
TYPE t_regestud IS RECORD(
        Student_id number(5) not null := 9999,
        Nombre VarChar2 (20),
        Apellidos varchar(20));

v_student1  t_regestud;
v_student2  t_regestud;
```

Referenciar un campo

Nombre_reg**.**nombre_campo

Ejemplo) ***v_student1.nombre_campo;***

Asignación de registros

Para asignar un registro a otro ambos deben ser del mismo tipo.

Ejemplo

```
v_student1 := v_student2;
```

Ejemplo)

```
Declare
    type t_regtype1 is record (
        campo1 number,
        campo2 varchar2(5));
    type t_regtype2 is record (
        campo1  number,
        campo2  varcha2r(5));
V_reg1 t_regtype1;
V_reg2 t_regtype2;
V_reg1 := V_reg2   esto da error
V_reg1.campo1:=v_reg2.campo1;
V_reg1.campo2:=v_reg2.campo2;
```

También se puede asignar un valor a un registro mediante la orden SELECT, que extraería los datos de la DB y los almacenaría en el registro.

Los campos del registro deben corresponderse con los campos en la lista de selección de la consulta.

Ejem)

```
Declare
Type t_regestudiante IS RECORD (
                        ape1 students.first_name  %type;
                        ape2 students.last_name % type;
                        especialidad students.major%type);


 v_student t_registroestudiante;

Begin
Select first_name, last_name, major
      into v_student from students where id=10000;

END;
```

Registros Anidados

Un registro puede tener un campo que a su vez sea un registro.

Ejemplo)

```
Declare
type t_domicilio IS RECORD (
                calle VarChar(35),
                numero number,
                localidad Varchar (25));

Type t_datospersona IS RECORD (
                nombre varchar2(35),
                domicilio  t_domicilio,
                fecha_nac date);

  v_persona t_datospersona;
Begin
        ...
        v_persona.nombre:='pepe';
        v_persona.domicilio.calle := 'sol, 21';
   END;
```

TABLAS

Las tablas se tratan como si fueran matrices. Una tabla PL es similar a una tabla de DB que tuviese dos columnas, una sería la clave que es del tipo Binary_Integer y un valor Value que es del tipo especificado en la definición.

Las tablas no están restringidas. El número de filas sólo tiene el límite que tenga el tipo Binary_Integer.

1. Definir tipo.
2. Declarar Var de ese tipo.

```
TYPE t_tabla IS TABLE OF  tipo  INDEX BY
BINARY_INTEGER;
```

Referenciar un elemento

Nombre_tabla (Indice)

Ejemplo)

```
Declare
    Type t_tablacaracteres IS TABLE of varchar2(10) index by binary_integer;
    v_caracteres  t_tablacaracteres;
    Type  t_name  IS  TABLE  of  students.first_name%type    index  by
binary_integer;

    v_names t_name;
    …
    v_names(2) := 'pepe';
```

Ejemplo) Cada elemento de v_student es un registro. Recupera el registro con id=10001 y lo almacena en v_students(10001).

```
Declare
   Type t_studenttable IS TABLE OF students%Rowtype  INDEX BY
binay_integer;

   v_students t_studenttable;

   Begin
      select * into v_student (10001) from students where id=10001;
End;
```

V_students(10001).first_name:='pepe';

id	First name	Last name	Current credit	id	First name	Last name	Current credit	id	First name	Last name	Current credit	……..

Al ser cada elemento de la tabla un registro, para referenciar los campos dentro del registro la sintaxis es:

Tabla(indice).campo

ATRIBUTOS DE LA TABLA

Tabla.atributo

ATRIBUTO	TIPO	DESCRIPCION
COUNT	Number	Nº filas de la tabla
DELETE	x	Borra filas
EXISTS	Boolean	Devuelve TRUE si existe en la tabla el elemento especificado
FIRST	Binary_integer	Devuelve el indice de la 1ª fila de la tabla
LAST	Binary_integer	Devuelve el indice de la ultima fila de la tabla-
NEXT (indice)	Binary_integer	Devuelve el indice de la fila de la tabla que sigue a la fila especificada.
PRIOR	Binary_integer	Devuelve el indice de la fila de la tabla que antecede a la fila especificada.

Ejercicio) Insertar 50 filas en una tabla

```
Declare
        type t_numbertable is TABLE of number index by binary_integer;
        v_number  t_numbertable;
        v_total number;

Begin
        for v_counter IN 1 .. 50 loop
            v_number (v_counter) := v_counter;
        end loop;
        v_total := v_number.count;

End;
```

devuelve 50

Ejemplo) Recorre todas las filas de la tabla y las inserta en una tabla temporal

```
Declare
    type t_majortable  IS TABLE of students.major%type_
                                        Index by binary_integer;
    v_major t_majortable;
    v_index binary_integer;

Begin
    V_major (-7) := 'Matematicas';
    v_major (4) := 'Historia';
    v_major (5) := 'Economia';
    v_index :=  v_majors.FIRST;
    Loop
        insert into temp values (v_index, v_major (v_index))
        exit when v_index = v_majors.LAST
        v_index := v_major.NEXT (v_index);
    End Loop;
End;
```

Ejercicio) Llenar una tabla Pl con los nombres de los alumnos

```
Declare
    type t_nombres IS TABLE OF Alumnos.nombre%type index by …;
    v_nombre_alumno  t_nombres;
    filas  alumnos%Rowtype;
    x  binary_integer := 0;

Begin
    FOR filas in (select nombre from alumnos) Loop
        x := x+1;
        v_nombre_alumnos (x) := filas.nombre;
    End Loop;
```

- insertar en una tabla TEMP los datos del array

End;

FOR EN UNA TABLA → Se necesita una variable de tipo Rowtype. Se utiliza para recorrer una tabla mediante un for.

4. CURSORES

Para procesar una orden SQL, ORACLE asigna un área de memoria llamada Área de Contexto. Un cursor es un puntero a esta área.

Los cursores permiten manejar grupos de datos que se obtienen como resultado de una consulta SQL que retorna una o más filas.

PL/SQL utiliza dos tipos de cursores: implícitos y explícitos. Siempre declara un cursor implícito para cualquier sentencia de manipulación de datos, incluyendo aquellas que retornan sólo una fila.

Sin embargo, para las queries que retornan más de una fila, usted debe declarar un cursor explícito para ser usado en una instrucción FOR.

No se pueden usar sentencias de control para cursores implícitos, como en el caso de los cursores explícitos, por lo que no se revisarán en este capítulo.

Los Cursores Explícitos son aquellos que devuelven cero, una o más filas, dependiendo de los criterios con que hayan sido construidos. Un cursor puede ser declarado en la primera sección de un programa PL/SQL ("declare").

Existen tres comandos para controlar un cursor: OPEN, FETCH y CLOSE. En un principio, el cursor se inicializa con la instrucción OPEN. Enseguida, se utiliza la instrucción FETCH para recuperar la primera fila o conjunto de datos. Se puede ejecutar FETCH repetidas veces hasta que todas las filas hayan sido recuperadas. Cuando la última fila ya ha sido procesada, el cursor se puede liberar con la sentencia CLOSE.

Es posible procesar varias queries en paralelo, declarando y abriendo múltiples cursores.

Tipos de Cursores

1. **Cursores Explícitos**: Para procesar un cursor explicito:
   1) Declarar cursor

      DECLARE

      CURSOR *nombre_cursor* [(parámetro1 [, parámetro2]…)]

      [RETURN tipo_de_retorno] IS sentencia_select ;

   2) Abrirlo
      - **OPEN** nombre_cursor
   3) Recoger los datos en Var PL
      - **FETCH** nombre_cursor **INTO** Lista_varPL ó registroPL;
   4) Cerrarlo
      - **CLOSE nombre_cursor**;

Ejemplo) *Declaración*

```
Declare
        V_dep classes.departament %type;
        V_curso  classes.course  %type;
        CURSOR  c_classes IS select * from classes where Department =
                                        V_dep AND course = v_curso;
```

Ejemplo)

```
Declare
        V_dep classes.department %type;
        V_curso  classes.course  %type;
        CURSOR  c_todasclases IS select * from classes;

        V_classesrecord    c_todaslasclase%rowtype;



BEGIN
        OPEN c_todasclases;
        FETCH c_todasclases INTO v_classesrecord;
        …
        FETCH c_todasclases  INTO v_dept, v_curso;   → ERROR el Select devuelve 7 columnas.
        CLOSE c_todasclases;
```

- La asignación de variables se debe hacer antes de la apertura del cursor. Si se asigna después no tendrá ningún efecto. Para poder examinar los nuevos datos habría que cerrar y abrir de nuevo el cursor

Ejemplo)

```
DECLARE
        v_building rooms.building%type;
   v_dep classes.departament%type;
         v_room_id classes.room%type;
        v_curso  classes.course%type
        CURSOR  c_buildings IS select * FROM rooms, classes where
                                  rooms.room_id = classes.room_id AND
                                  department = v_dep AND course = v_curso;
    BEGIN
        v_dep := 'His';
        v_curso := 101;
        OPEN c_buildings;
        …   * Ojo al asignar aquí no tendría efecto.
        CLOSE c_buildings;

    END;
```

Atributos de los cursores

- **%FOUND** → Es un atributo booleano que devuelve TRUE si la ultima orden FETCH devolvió una fila, y FALSE en caso contrario. Antes del primer FETCH está a NULL.

Ejemplo1)

```
Fetch…
  WHILE  cursor%FOUND Loop
          …
          Fetch…
  End Loop;
```

Ejemplo2)

```
Loop
   Fetch cursor INTO…
   If cursor%FOUND then
          …
   Else
          …
   End If
 End Loop
```

- **%NOTFOUND** → Atributo booleano que devuelve TRUE cuando el último FETCH no ha recuperado ninguna fila.
- **%ISOPEN** → Se utiliza para determinar si el cursor asociado está abierto o no.
- **%ROWCOUNT** → Devuelve el numero de filas extraídas por el cursor hasta el momento.

Cursores con Parámetros

Existen cursores que admiten parámetros al igual que los procedimientos.

Ejemplo)

```
DECLARE
Cursor c_classes (P_department classes.department %type, p_course
classes.course %type IS select * from classes where department =
p_department AND course=p_course);


Pasamos los valores reales al cursor
Open c_classes('his',101) ;
```

Bucles con cursores

Utiliza un cursor que recupera información de los estudiantes de historia, procesa las filas recuperadas, matriculando a cada estudiante de 'historia' y código '301' y registrándolo en la tabla 'registered_student', registra también el nombre y los apellidos en una tabla temporal.

- LOOP

```
DECLARE
    v_studentid  students.id %type;
    v_firstname students.firstname %type;
    v_lastname  students.lastname %type;
    cursor  c_historia is Select id, firstname, lastname FROM students
where
                                                major= 'history';
BEGIN
    open c_historia;
    LOOP
        Fetch c_historia INTO v_studentid, v_firstname, v_lastname;
        EXIT when c_historia %NOTFOUND;
        INSERT INTO registered_students (student_id, department,
course)
                                        VALUES (v_studentid, 'historia', 301);
        INSERT INTO temp VALUES (v_firstname, v_lastname);
    END LOOP;
    close c_historia;
END;
```

- WHILE… LOOP

```
DECLARE
  cursor  c_historia  IS Select  *  FROM students where major=
'history';
  v_studentdata  c_historia%ROWTYPE;

BEGIN
  open c_historia;

  Fetch c_historia into v_studentdata;
  While c_historia  %Found  Loop
        insert into registered_student (student_id, department,
course) values
                                        (v_studentdata.id, 'History',
301);
        insert into temp_table  values (v_studentdata.id,
v_studentdata.firstname,

  v_studentdata.lastname);
        fetch  c_historia into v_studentdata;
  End Loop;
  close  c_historia;
END;
```

- FOR

  El for abre el cursor, extrae los datos y se cierra de modo implícito; y además comprueba automáticamente si ya no quedan más filas.

```
FOR  var_tipo_cursor   IN CURSOR  LOOP
      …
END LOOP
```

Ejemplo)

```
Declare
 cursor c_historia is select id, firstname, lastname from students where
                                                    major='historia';

Begin
 For  v_studentdata in c_historia loop
        insert into registered_students (students_id, department, course)
                                        values (v_studentdata.id, 'his',301);
        insert into temp_table values  (v_studentdata.id,
                        v_studentdata.firstname,v_studentdata.lastname);
          End Loop;
        End;
```

2. Cursores Implícitos: (SQL%NOTFOUND)

Son aquellos en los que el programa ni abre ni cierra el cursor, sino que se abre de modo implícito. Únicamente sirve para procesar órdenes Insert, Update y Delete.

Ejemplo) El siguiente bloque ejecutará una orden INSERT si la orden UPDATE no encuentra una fila coincidente.

```
BEGIN
    UPDATE rooms SET number_seat = 100 where room_id=99990;
    IF SQL%NOTFOUND then
        INSERT INTO rooms (roomed, number_Seat) VALUES (99980, 100);
    END IF
END
```

Ejemplo2) Este ejemplo ilustra que no es conveniente emplear SQL NOTFOUND con ordenes Select INTO, ya que dará error cuando no encuentre ninguna fila coincidente y hará que *el control pase a la sección de excepciones del bloque. Y no se lleve a cabo la comprobación SQL NOTFOUND.*

```
DECLARE
    v_roomdata  rooms  %Rowtype;

BEGIN
    Select * INTO v_roomdata From rooms where room_id = -1;
    IF SQL%NotFound then
        Insert Into temp_table VALUES  ('No encontrado');
    END IF;
    Exception
        When no_data_found   then
            Insert INTO temp_table VALUES ('Estamos en la excepción');
 END;
```

SQW%ROWCOUNT cuenta el número de filas actualizadas.*Poner enunciado a este ejemplo*

```
DECLARE

    v_num_empelados NUMBER(2);
BEGIN
    INSERT INTO dept VALUES (99, 'PROVISIONAL', NULL);
  UPDATE emp SET deptno=99 WHERE deptno = 20;
  v_num_empleados := SQL%ROWCOUNT;
  DELETE FROM dept
    WHERE deptno = 20;
 DBMS_OUTPUT.PUT_LINE(v_num_empleados ||
   ' Empleados ubicados en PROVISIONAL');
EXCEPTION
    WHEN OTHERS THEN
```

```
RAISE_APPLICATION_ERROR(-20000, 'Error en aplicación');
END;
/
```

Clausula FOR UPDATE OF

Efectúa bloqueos exclusivos antes que termine la orden open, esto evita que otras sesiones cambien las filas del conjunto activo antes de que la transacción se confirme.

Si otra sesión ha establecido bloqueos con anterioridad sobre las filas del conjunto activo, entonces la operación select for update espera a que sean levantados.

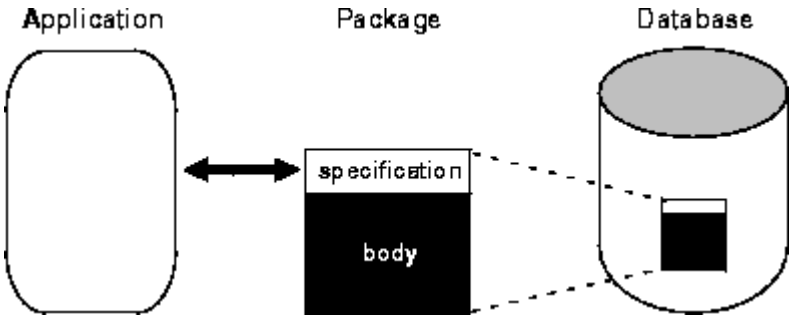
Si se declara el cursor con la clausula FOR UPDATE se puede emplear **WHERE CURRENT OF CURSOR**, que hace referencia a la fila recien extraida por el cursor.

Ejemplo) actualizar los creditos actuales de todos los alumnos matriculados en curso historia 101.

5. PROCEDIMIENTOS, FUNCIONES Y SUBPROGRAMAS

Los subprogramas son bloques de instrucciones de PL/SQL que pueden ser invocados por otros y recibir parámetros. En PL/SQL existen dos tipos de subprogramas: Los *Procedimientos* y las *Funciones*. Por regla general, se utiliza un procedimiento para ejecutar una acción específica y una función para calcular un valor.

Los subprogramas también constan de una sección de declaraciones, un cuerpo que se ejecuta y una sección opcional de manejo de excepciones.



Ventajas de los subprogramas

Los subprogramas proveen *extensibilidad*, es decir, permite crear nuevos programas cada vez que se necesiten, los cuales pueden ser invocados fácilmente y sus resultados utilizados de inmediato.

También aportan *modularidad*. Esto es, permite dividir un gran programa en módulos lógicos más pequeños y fáciles de manejar. Esto apoya el diseño de programas utilizando la metodología top-down.

Además, los subprogramas proveen las características de *reusabilidad* y *mantenibilidad*. Una vez construido, un subprograma puede ser utilizado en

cualquier número de aplicaciones. Si la definición del tema que implementa es cambiada, entonces sólo se debe alterar el subprograma y no todos los lugares donde es referenciado.

Finalmente, construir subprogramas agregan abstracción, lo que implica que es preciso conocer sólo qué es lo que hacen y no cómo están implementados necesariamente.

PROCEDIMIENTOS

Un procedimiento es un subprograma que ejecuta una acción específica. Un procedimiento es un bloque PL con sección declarativa, ejecutable y manejo de secciones.

```
CREATE [ OR Replace ] Procedure nombre_proc [(Argumento [ IN | OUT | IN OUT] tipo)] IS | AS

        'cuerpo procedimiento';

End nombre_proc;
```

- **OR REPLACE** → Sirve para poder cambiar el código de un procedimiento sin tener que borrarlo y crearlo de nuevo.

Ejemplo)

```
Create or replace procedure nombre_procedimiento AS
   / *  sección declarativa  * /    ← NO SE PONE DECLARE

Begin
   / * sección de ejecutable * /

Exception
   / * sección de excepciones * /

End nombre proced;
```

Borrar un `DROP PROCEDURE nombre` **procedimiento**

Parámetros formales

Ejemplo)

```
Create or replace procedure addnewStudent (p_firstname students.firstname
%type,p_lastname  students.lastname  %type,
p_major  students.major %type) AS
BEGIN
insert into students id, firstname, lastname, major, current_credits values
 (student_sequence.nextual, p_firstname, p_lastname, p_major, 0);
End AddnewStudents;
```

- La llamada al procedimiento:

```
BEGIN
        AddnewStudents ('David', 'Perez', 'Musica');← parámetros reales
End
```

Los parámetros reales

Los parámetros reales son valores o variables que contienen valores que se pasan como argumentos al procedimiento cuando este es invocado. Los valores de los parámetros reales son los que se utilizan dentro del procedimiento.

Los parámetros formales

Los parámetros formales son los parámetros que están en la declaración del procedimiento y son meros contenedores para los valores de los parámetros reales.

MODOS

- **IN →** El valor del parámetro real se pasa al procedimiento cuando este es invocado. Dentro del procedimiento el parámetro formal se considera de sólo lectura y no se puede cambiar.
- **OUT →** Se ignora cualquier valor que tenga el parámetro real en su llamada al procedimiento. Dentro del procedimiento el parámetro formal se considera de solo escritura y no puede ser leído. Sólo se le pueden asignar valores. Al devolver el control al entorno que lo llamó, los contenidos del parámetro formal se asignan al parámetro real.
- **IN OUT →** Es una combinación de los dos anteriores: el parámetro real se pasa al procedimiento y dentro de él el parámetro formal puede ser leído y escrito.

-

IN	OUT	IN OUT
es el tipo por defecto	debe ser especificado	debe ser especificado
pasa valores a un subprograma	retorna valores a quien lo llamó	pasa valores iniciales al subprograma y retorna un valor actualizado a quien lo llamó
un parámetro formal actúa como una constante	parámetros formales actúan como variables	parámetros formales actúan como una variable inicializada
a un parámetro formal no se le puede asignar un valor	a un parámetro formal debe asignársele un valor	a un parámetro formal podría asignársele un valor
los parámetros actuales pueden ser constantes, variables inicializadas, literales o expresiones	los parámetros actuales deben ser variables	los parámetros actuales deben ser variables

Ejemplo)

```
Create or replace procedure ejemplos ( p_inparameter IN number,
                                       p_outparameter OUT number,
                                       p_inoutparameter INOUT number) AS
v_localvariable  number;

  BEGIN
     v_localvariable :=  p_inparameter;   ✔ correcto leemos de un parametro
IN
     p_inparameter  := 7; ✘ error escribimos en un parámetro IN
     p_outparameter := 7;  ✔ correcto
     v_localvariable := p_outparameter;  ✘ error leemos de un parámetro OUT
     v_localvariable := p_inoutparameter;  ✔
     p_inoutparameter := 7;  ✔
```

- Los parámetros Formales no se pueden restringir.

Ejemplo)

```
Create or replace procedure  ejemplo2 (
        p_param1  in out varchar2 (10),
        p_param2  in out number (3,2)) AS
    Begin
        ----
End procedure emplo2;
```

P_param in out number (3,2) → error los parámetros formales no se pueden restringir

Se puede usar % type: adquiere el valor que tuviese el campo en la tabla del parámetro que acompaña

Valores predeterminados en los parámetros

Teniendo en cuenta que si los parámetros formales tienen un valor predeterminado no deben ser pasados valores desde el entorno que realiza la llamada. Si esto ocurre, se utilizara el valor del parámetro real en lugar del predeterminado

Ejemplo) Asignar como valor predeterminado la especialidad de 'Economía' a todos los nuevos estudiantes salvo que se indique lo contrario mediante el argumento.

```
Create or Replace procedure AddNewStudent  (p_firstname students.firstname%type,
                    p_lastname  students.lastname%type,
                    p_major  students.major%type := 'Economia') AS
BEGIN
    insert into students values  (student_sequence.nextual, p_firstname,
                                                p_lastname, p_major, 0);
End AddNewStudent;
```

- **Llamada**

```
BEGIN
        AddNewStudent ('Barbara', 'Lopez');
        AddNewStudent ('Antonio', 'Gomez', 'History');
END;
```

FUNCIONES

Una llamada a un procedimiento es una llamada PL en Sí misma, mientras que una llamada a una función se realiza como parte de una expresión. Una función es un rvalor.

CREATE [OR Replace] FUNCTION nombre_func [(Argumento [IN | OUT | IN OUT] tipo)] RETURN tipo_retorno {IS | AS}

'cuerpo función;

End nombre_funcion;

Ejemplo)

Create or replace function nombre_funcion(argumentos) RETURN tipo AS

*/ * sección declarativa * / ← NO SE PONE DECLARE*

Begin

*/ * sección ejecutable * /*

Exception

*/ * sección de excepciones * /*

End nombre_funcion;

- **RETURN expresión →** se emplea para devolver el control y un valor al entorno que realizó la llamada. *Expresión* es el valor que la función devuelve, el cual se convierte al tipo especificado en la clausula RETURN de la declaración de la función, si es que no es ya de dicho tipo. Puede haber más de un RETURN en una función, aunque sólo se ejecutará uno de ellos.

- **Llamada a una función:**
  Var:=nombre_funcion(parámetros);

Ejemplo) Función que devuelve True si la clase especificada tiene una ocupación mayor al 90% y False en caso contrario.

```
Create or replace FUNCTION casilleno (p_departament  classes.departament
%type,
                              p_ course classes.course %type) RETURN Boolean IS
   V_currentstudents number;
   V_maxstudents number;
   V_returnvalue Boolean;
   V_fullpercent constant number:=90;


      BEGIN

      SELECT current_student, max_student INTO v_current_students,
      v_maxstudents from      classes where department=p_departament  and
      course = p_course;

     IF(v_currentstudents/v_maxstudents*100)>v_fullpercent then
     V_returnvalue:=TRUE;
     ELSE
     V_returnvalue:=FALSE;
     END IF;
     RETURN v_returnvalue;

End casilleno;
```

SUBPROGRAMAS LOCALES

Se pueden definir procedimientos y funciones dentro de la sección declarativa de un bloque anónimo, se llama subprograma local.

Ejemplo)

```
DECLARE
        CURSOR C_allstudents is SELECT first_name, last_name FROM students;
       V_formattedName  varchar2(50);

    FUNCTION concaten (p_firstname IN VARCHAR2,
                       P_lastname IN VARCHAR2)RETURN varchar2
    IS
    BEGIN
            RETURN p_firstname || ' ' || p_lastname;
    END concaten;

BEGIN
  FOR v_studentRecord IN  C_allstudents LOOP
     V_formatName:=concaten ( v_stuendtRecord.first_name,
     V_studentRecord.last_name);
     Insert into temp values(V_formatName);


  END LOOP;
END;
```

El nombre de la función es un identificador PL/SQL y por lo tanto tiene las mismas reglas de ámbito y visibilidad que cualquier otro identificador, la función sólo es visible en el bloque en el que ha sido declarada.

Los subprogramas locales deben ser declarados al final de la sección declarativa, sino dan error.

6. PAQUETES

Un paquete es una estructura PL/SQL que permite almacenar una serie de objetos relacionados. Un paquete consta de la cabecera del paquete y el cuerpo.

- Cabecera – Contiene información acerca del contenido del paquete
- Cuerpo – Es un objeto del diccionario de datos, distinto de la cabecera, que no puede ser compilado hasta que no se haya compilado la cabecera, y contiene el código correspondiente a los prototipos incluidos en la cabecera. El cuerpo es opcional si la cabecera no tiene ningún procedimiento o función declarada.

Esta técnica es útil para declarar variables globales, ya que todos los objetos del paquete son visibles fuera de él.

```
CREATE[ORREPLACE] PACKAGE nombre_paquete AS
    [ especificación procedim
            "       funciones
      declaración   var
            "       tipos
            "       excepciones
            "       cursores ]
END  [ nombre_paquete ];
/
   CREATE [OR REPLACE] PACKAGE BODY nombre_paquete AS
   ……..

END  [ nombre_paquete ];
```

REGLAS

Las reglas sintácticas para la declaración de un paquete son las mismas que para la sección declarativa de un bloque anónimo excepto lo que se refiere a la declaración de funciones y procedimientos. Estas reglas son las siguientes:

1. Los elementos del paquete pueden aparecer en cualquier orden pero deben ser declarados antes de poderlos referenciar.
2. La declaración de procedimientos y funciones deben ser declaraciones formales. A diferencia de la sección declarativa de un bloque anónimo, que podía contener tanto prototipos como código de procedimientos o función, en los paquetes dicho código estará en el cuerpo.
3. La declaración formal (o prototipo) y la del cuerpo del paquete deben ser iguales.

Ámbito de Paquetes

Cualquier objeto declarado en la cabecera de un paquete esta dentro del ámbito y es visible fuera del paquete sin mas que cualificar el objeto con el nombre del paquete.

Ejemplo de llamada)

Begin

classpackage.RemoveStudents (1006, 'his', 101);

END;

Ejercicio)

Crearemos un paquete que contendrá un procedimiento para añadir un nuevo estudiante al curso especificado, otro para eliminar un estudiante del curso especificado, una excepción de usuario, un tipo tabla (array) para almacenar información del estudiante y otro procedimiento que nos devuelve una tabla (Array) que contiene los estudiantes actuales del curso especificado.*

```
CREATE OR REPLACE PACKAGE classpackage AS
    Procedure AddStudent  (p_studentid in student.id%type,
                           p_department in classes.department%type,
                           p_course in classes.course%type):

    Procedure RemoveStudent (p_studentid IN student.id%type,
                           p_department in classes.department%type,
                           p_course in classes.course%type);

    E_studentnotRegistered EXCEPTION;
    Type t_studentidtable IS table of student.id%type Index by
    binary_integer;
    Procedure classList (p_department  IN  classes department%type,
                                       p_course in classes.course%type,
                                       p_id OUT t_studentidtable);
    END classpackage;
    /
```

```
CREATE OR REPLACE PACKAGE BODY classpackage AS
Procedure AddStudent (p_studentid in student.id%type,
                                    p_department in classes.department%type,
                                    p_course in classes.course%type)  AS
 BEGIN
 insert into registered_student (student.id, department, course) values
                                                    (p_studentid, p_department,
p_course);
End AddStudent;

 Procedure RemoveStudent (p_studentid IN student.id%type,
                                    p_department in classes.department%type,
                                    p_course in classes.course%type)  AS
BEGIN
delete from registered_students where studentid = P_studentid and
                                department = p_department as course = p_course;
IF  SQL % NOTFOUND then
                raise e_studentnotRegistered;
End if;
End RemoveStudent;

Procedure ClassList (      p_department in classes.department %type,
                                    p_course in classes.course%type,
                                    p_id OUT t_studentidtable)  IS
 v_numstudents  binary_integer;
 v_studentid  registered_student.student_id % type;
 cursor c_registeredStudents IS select student_id from registeredStudents where
department = p_department and course = p_course;

 BEGIN
                        v_numstudents := 0;
                        OPEN c_registeredStudents;
                        LOOP
                                Fetch c_registeredStudents INTO v_studentid;
                                EXIT when c_registeredStudents %NOTFOUND;
                                v_numStudents := v_numStudents +1;
                                p_id (v_numStudents) := v_studentid;
                        END LOOP;
                END classlist;
END classpackage;
        /
```

Sobrecarga de los subprogramas de un paquete

Dentro de un paquete pueden sobrecargarse los procedimientos y funciones, es decir, puede haber mas de un procedimiento o función con el mismo nombre pero con distintos parámetros. Esto es útil ya que permite aplicar la misma operación a objetos de tipos diferentes.

Ejemplo) Añadir un estudiante a una clase, bien especificando el ID del estudiante o bien especificando el nombre y los apellidos.

```
Create or replace package classpackage AS
 procedure AddStudent (p_department IN classes.department %type,
                                    p_studentid IN student.id %type,
                                    p_course IN classes.course %type);

Procedure AddStudent (p_firstname IN students.first_name %type,
                                p_lastname IN students.last_name %type,
                                p_department IN classes.department %type,
                                p_course IN classes.course %type);
End Classpackage;

/

CREATE OR REPLACE PACKAGE BODY classpackage AS


 Procedure AddStudent (p_department IN classes.department %type,
                                p_studentid IN student.id %type,
                                p_course IN classes.course %type) AS
  BEGIN
 insert into registered_students (student_id, department, course) VALUES
                                    (p_studentid, p_department, p_course);
  End AddStudent;

Procedure AddStudent (p_firstname IN students.first_name %type,
                        p_lastname IN students.last_name %type,
                        p_department IN classes.department %type,
                        p_course IN classes.course %type) AS
                        v_studentid  students.id %type;
  BEGIN
 Select id into v_studentid from students where first_name = p_firstname  and
last_name = p_lastname;


Insert into registered_students (student_id, department, course) VALUES
                                    (v_studentid, p_department, p_course);
 End AddStudent;
 END Classpackage;
```

- LLamadas:
  a) Begin
  classpackage.Addstudent (1000, 'Mus', 410);
  End;
  b)Begin
  classpackage.AddStudent ('Barbara', 'Perez', 'Mus', 410);
  End;

7. DISPARADORES (triggers)

Se asemejan a los procedimientos y funciones, en que son bloques PL-SQL nominados. Tienen sección declarativa, ejecutable y manejo de excepciones.

Los disparadores, como los paquetes, deben almacenarse en la base de datos y no pueden ser locales a un bloque. Un disparador se ejecuta de forma implícita cada vez que tiene lugar el suceso de disparo. El suceso de disparo es una operación DML (Insert, Update ó Delete).

Los disparadores se usan para varias cosas, entre ellas, para el mantenimiento de restricciones de integridad compleja. También para crear auditorias sobre tablas y como aviso automático para otros programadores para indicarles que deben llevar a cabo una acción.

CREATE [OR REPLACE] TRIGGER nombre_disp { BEFORE / AFTER } suceso_disparo [OF COL] ref_tabla [FOR EACH ROW [When cond_disp]]
Declare
…
Begin
…..;
END nombre_disp;

El suceso de disparo es insert, update or delete

Tipos de Disparadores

CATEGORIA	VALOR	DESCRIPCION
Orden	Insert, update,delete	Orden DML que provoca su activación
Temporalización	Before,after	Define si el disparo se activa antes o después de ejecutar la orden
Nivel	Fila u orden	Los disparadores a nivel de fila activan por cada fila afectada por la orden que provocó el disparo (for each row) ---------- Los disparadores con nivel de orden se activan solo una vez antes o después de la orden de provocar el disparo

Restricciones de los disparadores

El cuerpo de un disparador es un bloque PL y cualquier orden legal en un bloque, también lo es en un disparador.

1. El cuerpo del disparador no puede contener ninguna variable de tipo LONG.
2. Ningún disparador puede emitir ninguna orden de control de transacciones.

Borrar

DROP TRIGGER nombre;

Des/Activar

ALTER TRIGGER nombre_disp { DISABLE / ENABLE };

Ejemplo) Crear un disparador para mantener una serie de estadísticas acerca de las distintas especialidades, incluyendo el número de estudiantes matriculados y la cantidad total de créditos seleccionados.

```
CREATE OR REPLACE TRIGGER Estadoespec AFTER INSERT OR DELETE OR
        UPDATE ON students
declare
        V_major students.major %type;
        v_total_students  students.current_credit %type;
        v_total_credits  number (9);
        CURSOR c_stadisticas IS Select major, count (*), sum (current_credit)
FROM students group by major;
 Begin
 open c_stadisticas;
 Loop
 Fetch c_stadisticas into v_major, v_total_students, v_total_credits;
 Exit when c_stadisticas%NOTFOUND;
 Update major_Stat SET total_credits = v_total_credits, total_students =
v_total_students where major = v_major;
 IF SQL%NotFound then
insert into major_stat (major, total_credits,total_students)
        VALUES (v_major, v_total_credits, v_total_student);
 END IF;

End Loop;
 close c_stadistic;
END Estadoespec;
```

DESPUES DE ejecutar un insert, update o delete sobre la tabla Students es cuando es lanzaría este disparador y por lo tanto se ejecutaría este código.

Orden de activación de los disparadores

1. Ejecutar si se encuentra el disparador tipo BEFORE con nivel orden
2. Por cada fila a la que afecte la orden
   a) Ejecutar si se encuentra el disparador BEFORE con nivel de fila.
   b) Ejecutar la propia orden.
   c) Ejecutar si se encuentra el disparador tipo AFTER con nivel de fila.
3. Ejecutar si se encuentra el disparador tipo AFTER nivel de orden.

Ejercicio) Definir cuatro tipo de disparadores UPDATE sobre la tabla 'classes'.

```
→ CREATE SEQUENCE trigger_seq  Start with 1 increment by 1;

→ CREATE OR REPLACE TRIGGER class_B BEFORE UPDATE ON classes
      begin
            insert into temp values (trigger_seq.NEXTVAL, 'antes');
      End;

→ CREATE OR REPLACE TRIGGER classes_A AFTER Update on classes
      begin
            insert into temp value (trigger_seq.Nextval, 'despues');
End;

→ CREATE OR REPLACE TRIGGER classes_Bfila Before Update on classes for
each Row
      begin
            insert into tempValues (trigger_seq.Nextval, 'antes nivel de fila');
End;

→ CREATE OR REPLACE TRIGGER classes_Afila AFTER Update on classes for
each Row
      begin
            insert into tempValues (trigger_seq.Nextval, 'despues nivel de fila');
End;
```

si se ejecuta por ejemplo esta orden

```
update classes set num_credits= 8 where departament in ('his', 'cs');
```

esto afecta a cuatro filas , los disparadores previo y posterior con nivel de orden se ejecutan una sola vez, los disparadores previo y posterior con nivel de fila se ejecutan 4 veces.

Resultado de la ejecución:

```
 1 antes
 2 antes nivel de fila
 3 después nivel de fila
 4 antes nivel de fila
 5 después nivel de fila
 6 antes nivel de fila
 7 después nivel de fila
 8 antes nivel de fila
 9 después nivel de fila
10 después
```

Utilizacion de :OLD y :NEW con disparadores con nivel de fila (for each row)

Un disparador a nivel de fila se ejecuta una vez por cada fila procesada por la orden que provoca el disparo. Dentro del disparador puede accederse a la fila que esta siendo actualmente procesada utilizando para ello dos pseudoregistros:

- :OLD
- :NEW

Sintacticamente se tratan a old y new como registros del tipo tabla_disparo % rowtype, realmente no son registros, hay operaciones que son validas con registros, pero no los son con old y new, no se pueden realizar asignaciones como registros completos solo con los campos individuales que las componen.

Ejemplo)

```
Declare
v_temprec.   Temp._table %rowtype;
v_temrec :=:old;  ERROR
v_temprec.char_col :=:old.char_col;    ← CORRECTO (campo concreto)
```

ORDEN DE DISPARO	:OLD	:NEW
Insert	-------	Valores que serán insertados cuando se complete la orden
Update	Valores originales de fila antes de actualizar	Nuevos valores serán escritos cuando se complete la orden
Delete	Valores originales antes del borrado de la fila	---------

Utilizacion de predicados de los disparadores (Inserting, Updating, Deleting)

En un disparador insert, update o delete existen 3 funciones booleanas que sirven para determinar de que operación se trata.

1. Inserting: devuelve true si la orden de disparo es un Insert.
2. Updating: devuelve true si la orden de disparo es un Update.
3. Deleting: devuelve true si la orden de disparo es un Delete.

Ejemplo 1) Crear un disparador que utiliza estos predicados para registrar todos los cambios hechos en la tabla registered_students, también almacena la identidad del usuario que lo realizó, todos los registros se almacenaran en rs_audit.

Create table RS_audit(

Old_student_id number(5);
Old_departament char(3),
Old_course number(3),
Old_grade char(1),
New_student_id number(5),
New_departament char(3),
New_grade char(1),
New_course number(3),
Change_type char(1),
Change_by_user varchar2(8)
Date_sist date);

```
Create or replace trigger Cambios before insert or delete or update on
registered_student for each row
Declare
V_change_type char (1);
Begin
If inserting then
   V_changetype:='I';
   Insert into rs_audit(changetype, change by user, data_sist, new_departament,
   new_course, new_grade) values (v_changetype, user,
   sysdate,:new.student_id,:new.departament, :new.course, :new.grade);
Elsif  updating then
      V_changetype:='U';
         Insert into rs_audit values(v_changetype, user,syspdate,
         :old.student_id,:old.departament, :old.course, :old.grade,:new.student_id,
         :new.departament,:new.course,:new.grade);
Else
      V_changetype:='D';
            Insert into rs_audit(changetype, change_by_user,
            date_sist,old_student_id,  old_departament,old_course,old_grade)
            values
            (v_change_type,user,sysdate,:old.student_id,:old.departament,:old.cour
            se,:old_grade);
End if
End Cambios;
```

Tablas Mutantes

Hay restricciones sobre las tablas y columnas a las que puede acceder el cuerpo de un disparador.

Tabla mutante – Es una tabla que se está modificando actualmente por una orden DML. Para un disparador, esta es la tabla sobre la que está definido. Las tablas que pueden necesitar ser actualizadas como resultado de restricciones de integridad referencial también son mutantes

Tabla de restricción – Es una tabla de la que puede ser necesario leer para una restricción de integridad referencial.

Ejemplo)

```
Create table registered student ( student_id number(5) not null,
                                  department char (3),
                                  course number (3),
                                  grade char (1)
Constraint rs_Student_id foreign key (student_id) references students (id),

Constraint rs_department_course foreign key (department, course) refenrences
classes (deparment, couse));
```

- Registerd_student tiene 2 restricciones de integridad referencial declarativa, por lo tanto, las tablas Student y Classes son tablas de restricción.
  La propia registered_student MUTA cuando se ejecuta una orden DML sobre ella.

Restricciones para las ordenes SQL en el cuerpo de un trigger

1. No se puede leer ni modificar ninguna tabla mutante de la orden que provoca el disparo, esto incluye a la propia tabla del disparador.
2. No se puede leer o modificar las columnas de clave primaria de una tabla de restricción de la tabla de disparo.

Estas restricciones se aplican en todos los disparadores con nivel de fila.

Ejemplo 1) Este disparador modifica tanto la tabla Student como Classes. El proceso es legal ya que las columnas modificadas no son las columnas clave (2).

```
Create or replace trigger CascadeRSInsert before insert on registered_Student
                                                    for each row
Declare
            V_credits     classes.num_credits%type;

Begin
Select num_credit into v_credits from classes where deparment
                        =:new.deparment and course =: new.course;
Update students set current_credit = current_credit + v_credits where
                                        id =: new.student_id;
Update classes set current_students = current_students + 1 where department
                                        =:new.department and course =: new.course;
End cascadeRSInsert
```

Ejemplo 2) Limitar a 5 el numero de estudiantes de cada especialidad.

```
Create or replace trigger LimiteEspec after insert or update of major on students
for each row

 Declare
          V_MaxStudents number : = 5;
          V_currentStudent number;
                                                          ERROR

 Begin
 Select count (*) into v_currentStudents from STUDENTS where major
=:new.major;
 If v_currentStudents +1 > V_MaxStudents then
  Raise_Application_error(-20000,'Demasiados estudiantes en'||:new.major);
              End if
 End LimitEspec
```

tabla Student solo es mutante para los disparadores a nivel de fila, pero sí se puede consultar la tabla de disparo utilizando disparadores o triggers con nivel de orden. No lo podemos transformar en un disparador a nivel de orden porque estamos utilizando el pseudo-registro new.

La solución es crear dos disparadores, uno a nivel de fila, en el que registramos el valor :New.Major, pero sin consultar la tabla Students; y, segundo, otro disparador a nivel de orden, que sí que realiza la consulta a la tabla Students pero utilizando el valor almacenado de la consulta anterior. Para almacenar este valor, es necesario usar una tabla PL (array) dentro de un paquete.

8. EXCEPCIONES

Las excepciones son unas características de PL y que ha heredado del lenguaje ADA, del cual proviene. El objetivo de las excepciones es el tratamiento de los errores que se producen en tiempo de ejecución y no de compilación.

Cuando se produce un error, se genera una excepción. Cuando esto sucede, el control pasa al gestor de excepciones, que es una sección independiente del programa.

Las excepciones se declaran en la sección declarativa de un bloque, se generan en la sección ejecutable y se tratan en la sección de excepciones.

```
Declare
  e_nombreExcep  EXCEPTION;      ← definidos por el usuario

BEGIN
  ---raise e_nombreExcep;

EXCEPTION
        when exception_1 then
             ordenesPL --- ;
        when e_nombreExcep then
             ordenesPL ;
        When others then
             ---;

END;
```

Tipos de excepciones

- **Definidas por el usuario** - Se declaran en la sección declarativa. El ámbito de la excepción es igual al de cualquier variable declarada en la misma sección declarativa. Para lanzar o arrancar la excepción de usuario utilizamos ***RAISE nombre_excep;***

Ejemplo) Introducir los nombres y los apellidos de los alumnos en la columna 3 de una tabla temporal. Si alguno de los apellidos esta vacío disparamos un error y abandonamos el bloque.

```
Declare
        cursor c_Alu is selected ape1, nombre from alumnos;
        error_ape EXCEPTION;

Begin
        for x in c_Alu loop
                if x.ape1 is null then
                        raise error_ape;
                end if;
                insert into temp (col3) values (x.ape1 || x.nombre)
        End loop;

Exception
        when error_ape then
                insert into temp (col3) values (" Error apellido vacío ");
END;
```

- **Excepciones Predefinidas**

Oracle ha definido diversas excepciones que se corresponden con los errores Oracle más comunes Siempre están disponibles para el programa y es necesario declararlas.

Se generan de manera implícita cuando ocurre el error asociado. Las excepciones predefinidas pueden ser generadas por la orden ***Raise*** si así se desea. Cuando se genera la excepción el control pasa inmediatamente a la sección de excepciones. Una vez aquí no hay forma de volver a la sección ejecutable del bloque.

Tipos de predefinidas

- NO_DATA_FOUND → Orden Select Into no devuelve ninguna fila.
- TOO_MANY_ROOMS → Select … Into que devuelve mas de una fila.
- ZERO_DIVIDE → División por 0.
- INVALID_NUMBER → Faltó la conversión a un numero
- VALUE_ERROR → Error de truncamiento aritmético o de conversión.
- CURSOR_ALREADY_OPEN → Se intentó abrir un cursor que ya estaba abierto.

Nombre Excepción		SQLCODE
ACCESS_INTO_NULL	El programa intentó asignar valores a los atributos de un objeto no inicializado	-6530
COLLECTION_IS_NULL	El programa intentó asignar valores a una tabla anidada aún no inicializada	-6531
CURSOR_ALREADY_OPEN	El programa intentó abrir un cursor que ya se encontraba abierto. Recuerde que un cursor de ciclo FOR automáticamente lo abre y ello no se debe especificar con la sentencia OPEN	-6511
DUP_VAL_ON_INDEX	El programa intentó almacenar valores duplicados en una columna que se mantiene con restricción de integridad de un índice único (unique index)	-1
INVALID_CURSOR	El programa intentó efectuar una operación no válida sobre un cursor	-1001
INVALID_NUMBER	En una sentencia SQL, la conversión de una cadena de caracteres hacia un número falla cuando esa cadena no representa un número válido	-1722
LOGIN_DENIED	El programa intentó conectarse a Oracle con un nombre de usuario o password inválido	-1017
NO_DATA_FOUND	Una sentencia SELECT INTO no devolvió valores o el programa referenció un elemento no inicializado en una tabla indexada	+100
NOT_LOGGED_ON	El programa efectuó una llamada a Oracle sin estar conectado	-1012
PROGRAM_ERROR	PL/SQL tiene un problema interno	-6501

ROWTYPE_MISMATCH	Los elementos de una asignación (el valor a asignar y la variable que lo contendrá) tienen tipos incompatibles. También se presenta este error cuando un parámetro pasado a un subprograma no es del tipo esperado	-6504
SELF_IS_NULL	El parámetro SELF (el primero que es pasado a un método MEMBER) es nulo	-30625
STORAGE_ERROR	La memoria se terminó o está corrupta	-6500
SUBSCRIPT_BEYOND_COUNT	El programa está tratando de referenciar un elemento de un arreglo indexado que se encuentra en una posición más grande que el número real de elementos de la colección	-6533
SUBSCRIPT_OUTSIDE_LIMIT	El programa está referenciando un elemento de un arreglo utilizando un número fuera del rango permitido (por ejemplo, el elemento "-1")	-6532
SYS_INVALID_ROWID	La conversión de una cadena de caracteres hacia un tipo *rowid* falló porque la cadena no representa un número	-1410
TIMEOUT_ON_RESOURCE	Se excedió el tiempo máximo de espera por un recurso en Oracle	-51
TOO_MANY_ROWS	Una sentencia SELECT INTO devuelve más de una fila	-1422
VALUE_ERROR	Ocurrió un error aritmético, de conversión o truncamiento. Por ejemplo, sucede cuando se intenta calzar un valor muy grande dentro de una variable más pequeña	-6502
ZERO_DIVIDE	El programa intentó efectuar una división por cero	-1476

Ejemplo) Encontrar el alumno codigo 10. Si se localiza almacenar su matricula en la columna 1 de la tabla temporal y sino se dispara una excepcion del tipo NO_DATA_FOUND y en la columna 3 de la tabla temporal se escribe un mensaje de error.

```
Declare
        fila alumnos%RowType;

Begin
        Select * into fila from alumnos where cod = 10;
        Insert into temp (col1) values (fila.matricula);

Exception
        When no_data_found  then
        insert into temp (col3) values (" no existe el alumno con codigo 10 ");

End;
```

Funciones SQLCODE, SQLERRM

Se utilizan para saber que error Oracle dio lugar a la excepción dentro de un gestor *others.*

SQLCODE → Devuelve el código de error actual. Siempre devuelve un numero negativo, menos con no_data_found, que devuelve 100, y en excepciones de usuario que devuelve siempre 1.

SQLERRM → Devuelve el texto del mensaje de error.

Ejemplo) Hallar el número de estudiantes matriculados y el número máximo de estudiantes permitidos. Cuando haya demasiados matriculados, se insertará un mensaje que explique lo ocurrido. Con cualquier otro error insertaremos en una tabla temporal el código del error y el texto del mismo.

```
Declare
        e_toomanyStudents  EXCEPTION;
        v_currentStudents  Number (3);
        V_maxStudents  number (3);
        v_errorcode  number;
        v_errortext  varchar2(200);

Begin
 Select current_students, max_students into v_currentStudents, v_maxStudents
                              from classes where deparment = 'His' and course = 101;
        if v_currentStudents > v_maxStudents then
              Raise e_toomanyStudents;
        End if;

 Exception
 when e_toomanyStudents then
 insert into log_table (info) values ("Historia 101 tiene" || v_currentStudents ||
"el Maximo Permitido es" || v_maxStudents);


 when others then
      v_errorcode := SQLCODE;
      v_errortext := SUBSTR (SQLERRM, 1, 200);
      insert into Log_table (code, message, info) values (v_errocode, v_errortext,
'ocurrió un error oracle');

End;
```

Uso de Raise_Application_Error

Se puede utilizar la function predefinida Raise_Application_Error para crear nuestros propios mensajes de error.

Sintaxis: ***Raise_application_error (nº error, mensaje de error);***

Nº de error va desde -20000 a -20999

Ojo un insert delante de ***Raise_application_error*** *no funciona*

Ejemplo) Comprobar si hay suficiente espacio en un curso antes de matricular a un estudiante

```
Create or replace procedure register (p_student in students.id%type,
p_department in
            classes.department%type, p_course in classes.course%type) AS
            v_currentStudents number;
            v_maxStudents number;

Begin
 Select current_students, mas_students into v_currentStudents, v_maxStudents
            from classes where course = p_course and department =p_department;
 if v_currentStudents + 1 > v_maxStudents then
      Raise_application_error (-20000, 'no se puede añadir más estudiantes');
 End if
      classpackage.addStudent (p_studentid, p_department, p_course);

Exception
 when no_data_found then
 Raise_application_error (-20001, p_department || '.' || p_course || 'no existe ');

End;
```

Ámbito y Propagación de excepciones

El ámbito de las excepciones es igual que el de las variables. Si una excepción se propaga fuera de su ámbito no podrá ser referenciada por su nombre. Para propagar un error definido por el usuario fuera del bloque lo mejor es definir la excepción dentro de un paquete para que continúe siendo visible fuera del bloque.

Las excepciones pueden producirse en la sección declarativa, ejecutable o en la sección de excepciones.

Sección ejecutable

Ejemplo 1)

```
Declare
        A exception
Begin

        Begin
              Raise A     → Excepción A generada en un sub-bloque, se trata en el
sub-bloque y se devuelve el control al bloque externo.
        Exception
          when A then
                  ---
        End;

END;
```

Ejemplo 2)

```
            Declare
                        A exception;
                        B exception;
                  Begin
                        Begin
                        Raise B;→ Excepción B generada en el sub-bloque, no
            existe gestor para B.
                        Exception
                              when A then
                        End;
                  Exception
                        when B then      → Se propaga al bloque de nivel superior
            y se trata allí.
             END;
```

Ejemplo 3)

```
Declare
        A exception;
        B exception;
        C exception;
Begin       Begin
            Raise C        → Excepción C no existe un gestor para C.
        Exception
        when A then
        End
 Exception
 when B then → Va al bloque nivel Superior, y como tampoco existe acaba el
                                                programa con una excepción sin
tratar.
        END;
```

Sección declarativa

Si una asignación en la sección declarativa genera una excepción se propaga inmediatamente al bloque de nivel superior, aunque exista un gestor others este no se ejecuta y allí se aplican las mismas reglas dadas en la sección anterior.

Ejemplo 1)

```
Declare
        v_number  number(3) := 'ABC';  →genera un value_error
Begin

Exception
        when other then   → aunque existe un gestor others no se ejecuta

End;
```

Ejemplo 2)

```
Begin
        Declare
                v_number  number := 'ABC';   ← Error, value error
        Begin

        Exception
                when others then  ← no se ejecuta

        End;
Exception
        when others then  ← si se trata

End;
```

Excepciones producidas en la Sección de excepciones

La excepción se propaga inmediatamente al bloque de nivel superior. Esto es así porque solo puede haber una excepción activa en cada momento en la sección de excepciones.

Ejemplo)

```
Declare
        A exception;
        B exception;
Begin
        Raise A; → Se arranca la excepcion A
        Exception
        when A then→ se trata y se lanza la excepcion B dentro del gestor A
                        Raise B;
         when B then
                        ---
End;
```

** Aunque hay un gestor para B, no se ejecuta y pasa el control al bloque de nivel superior, acaba el programa con una excepcion sin tratar.*

Ejemplo)

```
Begin
        Declare
                A exception;
                B exception;
        Begin
                Raise A
        Exception
        When A then→ Se trata la excepcion A y se lanza B dentro del gestor A
                Raise B;
        When B then → No se ejecuta y el control para al bloque de nivel superior
        End;
 Exception
        when B then → La excepcion B se trata en este bloque externo

 End;  → El programa acaba bien
```