PL/SQL

1. Introducción a PL/SQL

•Lenguaje de procesamiento de transacciones completamente portable y con un alto rendimiento.

•PL/SQL relaciona conceptos de bases de datos para manejarlos con estructuras de control.

•Se utiliza dentro del Administrador de BD ORACLE.

Principales Características

•Se utilizan sentencias SQL:

–Para manipular datos

–Sentencias de control de flujo para organizar la manipulación

•Se permite:

–Declaración de constantes y variables

–Definir procedimientos y funciones

–"Atrapar" errores en tiempos de ejecución

Estructuras de Bloque

•PL/SQL es un *lenguaje estructurado en bloques.*

•La unidad básica de codificación son bloques lógicos que pueden contener sub-bloques.

•Un bloque agrupa en forma lógica un grupo de sentencias.

•Se pueden efectuar declaraciones de variables que sólo tendrán validez donde éstas se definan.

•Cada bloque tiene 3 partes:

–**Declarativa:** Introducida por la palabra DECLARE, donde se declaran los elementos del programa que van a necesitarse.

–**Ejecutable:** Introducida por la palabra BEGIN, con las instrucciones. Única parte obligatoria.

–**Manejo de Excepciones:** Introducida por la palabra EXCEPTION, errores producidos durante la ejecución

•Los Bloques se escriben en SQL*PLUS.

•La estructura es:

```
DECLARE
-- declaraciones
BEGIN
-- sentencias
EXCEPTION
--manejadores de excepción
END;
```

•Es posible anidar sub-bloques en la sección ejecutable y de excepciones, pero NO en la sección de declaraciones.

Variables y Constantes

•Condición: Cada variable o constante debe estar declarada antes de ser utilizada en una expresión

•Variables: Pueden corresponder a cualquier tipo de datos SQL, tal como *char, date o number*, o algún tipo de PL/SQL como *boolean o binary_integer*.

Asignar valores a variables:

•Existen dos formas:

–Con el operador :=. Variable en el lado izquierdo y la expresión al lado derecho.
Bono := salario * 0.15;

–Obtener valores directamente desde la BD:
```
SELECT salario * 0.15
INTO bono
FROM empleado WHERE num_emp = XXXX;
```

Declaración de Constantes:

•Se incorpora la palabra reservada constant e inmediatamente se asigna el valor deseado. Posteriormente, no se permiten reasignaciones de valores para aquella constante.

Limite_credito CONSTANT real := 5000;

Cursores

•Áreas de trabajo que permiten ejecutar sentencias SQL y procesar la información obtenida de ellos.

•Dos tipos: Implícitos y Explícitos

•PL/SQL declara implícitamente un cursor para todas las sentencias de manipulación de datos, incluyendo las consultas que retornan una sola fila.

•Para consultas que devuelven más de una fila, es posible declarar explícitamente un cursor que procese las filas en forma individual

•Ejemplo:

```
DECLARE
CURSOR curs IS
SELECT num_emp, nom_emp, empleo FROM empleado WHERE depto = 20;
```

•El conjunto de filas retornado se denomina “set de resultados”. Su tamaño está determinado por el número de filas que cumplen con el criterio de selección que implementa el cursor.

•Las filas son procesadas de a una cada vez

•Se detallarán las características más adelante

Manejo de Errores

•PL/SQL provee una manera de detectar y procesar ciertas condiciones de error predefinidas o definidas por el usuario, llamadas excepciones.

•Al ocurrir un error se procesa una excepción, es decir, se detiene la ejecución normal del programa y se transfiere el control a un segmento especial del programa que tiene por objeto manejar estas situaciones excepcionales. Reciben el nombre de exception handlers

•Las excepciones predefinidas se gatillan automáticamente por el sistema cuando ocurre un error de cierta naturaleza.

•El usuario puede definir excepciones con la sentencia *raise*.

Subprogramas

•Dos tipos de subprogramas (manejan parámetros de entrada y de salida):
–Procedimientos
–Funciones

•Un subprograma es un programa más pequeño, que tiene un encabezado, una sección opcional de declaraciones, una sección de ejecución y una sección opcional de manejo de excepciones, como cualquier programa PL/SQL

Paquetes

•Se permite almacenar lógicamente un conjunto de tipos de datos relacionados, variables, cursores e incluso subprogramas dentro de un *paquete*. Cada paquete involucra la definición y tratamiento de todos los elementos recién mencionados.

•Se descomponen en: **Especificación** y **Cuerpo.**

•Especificación o package specification: Idéntica a una sección de declaración de aplicaciones.

•Es posible declarar tipos, constantes, variables, excepciones, cursores y subprogramas disponibles para su uso en el *cuerpo* del paquete.

•El cuerpo del paquete define la implementación de esos subprogramas declarados en el punto anterior.

Ejemplo de uso de paquetes- Especificación

```
CREATE PACKAGE emp_actions AS                    -- package specification
  PROCEDURE hire_employee (empno NUMBER, ename CHAR, …) ;

  PROCEDURE fire_employee (empid NUMBER) ;
END emp_actions ;
```

Ejemplo de uso de paquetes - Cuerpo

```
CREATE PACKAGE BODY emp_actions AS
-- package body

 PROCEDURE hire_employee (empno NUMBER, ename CHAR, …) IS
 BEGIN
   INSERT INTO emp VALUES (empno, ename, …);
 END hire_employee;
```

```
  PROCEDURE fire_employee (emp_id NUMBER) IS
  BEGIN
    DELETE FROM emp WHERE empno = emp_id;
  END fire_employee;

END emp_actions;
```

•Los paquetes pueden ser compilados y almacenados en una BD ORACLE y su contenido puede ser compartido por varias aplicaciones.

Ventajas en la utilización de PL/SQL

•Soporte para:
– SQL
–la programación OO
•Mejor rendimiento
•Alta productividad
•Completa portabilidad
•Integración con Oracle garantizada
•Seguridad

Soporte para SQL

•SQL es no-procedural, Oracle se preocupa del 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 proporciona comandos de control de transacciones y permite utilizar las funciones de SQL, operadores y pseudocolumnas.

•PL/SQL soporta tipos de datos de SQL, lo que reduce la necesidad de convertir los datos al pasar de una a otra aplicación.

Soporte para Programación OO

•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.
–overhead considerable
–consultas viajan a través de la red.

•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: llamadas sean más rápidas y eficientes.

•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.

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 mantención 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.

2. **Fundamentos del Lenguaje**

Este tema se centra en algunos aspectos del lenguaje, tal como el grupo de caracteres válidos, las palabras reservadas, signos de puntuación y otras reglas de formación de sentencias que es preciso conocer antes de empezar a trabajar con el resto de funcionalidades.

Set de Caracteres y Unidades Léxicas

•Las instrucciones del lenguaje deben ser escritas utilizando un grupo de caracteres válidos.
•PL/SQL no es sensible a mayúsculas o minúsculas.
•El grupo de caracteres incluye los siguientes:

–Letras mayúsculas y minúsculas de la A a la Z
–Números del 0 al 9
–Los símbolos ( ) + - * / < > = ! ~ ^ ; . ‘ @ % , “ # $ & _ | { } ? [ ]
–Tabuladores, espacios y saltos de carro

•Una línea de texto en un programa contiene lo que se conoce como unidades léxicas, los que se clasifican como sigue:
–Delimitadores (símbolos simples y compuestos)
–Identificadores (incluye palabras reservadas)
–Literales
–Comentarios

•Por ejemplo en la instrucción:
bonus := salary * 0.10; -- cálculo del bono

•se observan las siguientes unidades léxicas:
–Los identificadores *bonus* y *salary*
–El símbolo compuesto **:=**
–Los símbolos simples ***** y **;**
–El literal numérico *0.10*
–El comentario “*cálculo del bono*”

•Para asegurar la fácil comprensión del código se pueden añadir espacios entre identificadores o símbolos o bien utilizar saltos de línea e identaciones para permitir una mejor legibilidad.
•Ejemplo:

```
IF x>y THEN max := x; ELSE max := y; END IF;
```

•Puede reescribirse de la siguiente manera para mejorar el aspecto y legibilidad:

```
IF x > y THEN
        max := x;
ELSE
        max := y;
END IF;
```

Delimitadores e Identificadores

•Un delimitador es un símbolo simple o compuesto que tiene un significado especial dentro de PL/SQL.

–Operadores Aritméticos
–Operadores Lógicos
–Operadores Relacionales

•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

Operadores en PL/SQL

Tipo de operador	Operadores
Operador de asignación	:= (dos puntos + igual)
Operadores aritméticos	+ (suma) - (resta) * (multiplicación) / (división) ** (exponente)
Operadores relacionales o de comparación	= (igual a) <> (distinto de) < (menor que) > (mayor que) >= (mayor o igual a) <= (menor o igual a)
Operadores lógicos	AND (y lógico) NOT (negacion) OR (o lógico)
Operador de concatenación	||

Identificadores

•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.

Literal y Comentario

•LITERAL: Es un valor de tipo numérico, caracter, cadena o lógico no representado por un identificador (es un valor explícito).

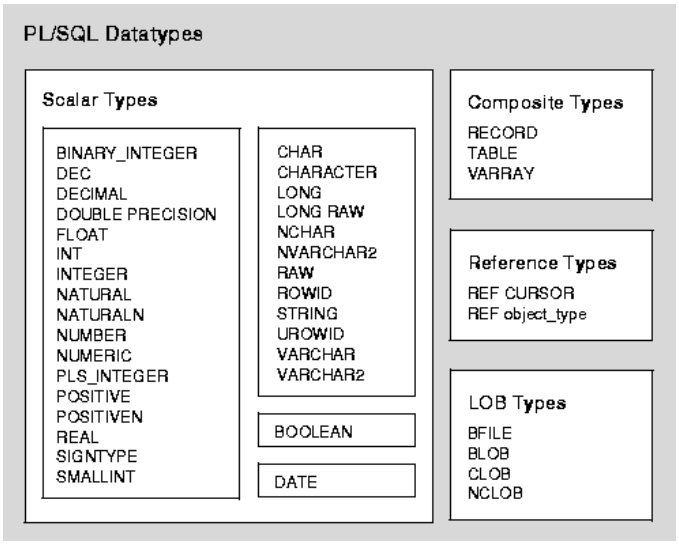
•COMENTARIO: Es una aclaración que el programador incluye en el código. Son soportados 2 estilos de comentarios, el de línea simple y de multilínea, para lo cual son empleados ciertos caracteres especiales como son:

```
-- Línea simple


/*
Conjunto de Líneas
*/
```

Tipos de Datos y Conversiones

•Cada constante y variable posee un tipo de dato el cual especifica su forma de almacenamiento, restricciones y rango de valores válidos.
•Con PL/SQL se proveen diferentes tipos de datos predefinidos.
•Un tipo *escalar* no tiene componentes internas
•Un tipo *compuesto* tiene otras componentes internas que pueden ser manipuladas individualmente.
•Un tipo de *referencia* almacena valores, llamados *punteros*, que designan a otros elementos de programa.
•Un tipo *lob* (large object) especifica la ubicación de un tipo especial de datos que se almacenan de manera diferente.



•**NUMBER** (*Numérico*): Almacena números enteros o de punto flotante, virtualmente de cualquier longitud, aunque puede ser especificada la precisión (Número de digitos) y la escala que es la que determina el número de decimales.

-- NUMBER [(precision, escala)]

saldo **NUMBER**(16,2);
/* Indica que puede almacenar un valor numérico de 16
posiciones, 2 de ellas decimales. Es decir, 14 enteros
y dos decimales */

•**CHAR** (*Caracter*): Almacena datos de tipo caracter con una longitud maxima de 32767 y cuyo valor de longitud por default es 1

-- CHAR [(longitud_maxima)]

nombre **CHAR**(20);
/* Indica que puede almacenar valores alfanuméricos de 20
posiciones */

•**VARCHAR2** (*Caracter de longitud variable*): Almacena datos de tipo caracter empleando sólo la cantidad necesaria aún cuando la longitud máxima sea mayor.

-- VARCHAR2 (longitud_maxima)

nombre **VARCHAR2**(20);
/* Indica que puede almacenar valores alfanuméricos de hasta 20 posiciones */
/* Cuando la longitud de los datos sea menor de 20 no se rellena con blancos */

•**BOOLEAN** (*lógico*): Se emplea para almacenar valores TRUE o FALSE.

hay_error **BOOLEAN**;

•**DATE** (*Fecha*): Almacena datos de tipo fecha. Las fechas se almacenan internamente como datos numéricos, por lo que es posible realizar operaciones aritméticas con ellas.

•Atributos de tipo. Un atributo de tipo PL/SQL es un modificador que puede ser usado para obtener información de un objeto de la base de datos. El atributo **%TYPE** permite conocer el tipo de una variable, constante o campo de la base de datos. El atributo **%ROWTYPE** permite obtener los tipos de todos los campos de una tabla de la base de datos, de una vista o de un cursor.

•PL/SQL también permite la creación de tipos personalizados (registros) y colecciones (tablas de PL/SQL).

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	

Uso de %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.

Uso de %ROWTYPE

•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*.

REGISTROS

•PL/SQL permite utilizar tipos compuestos definidos por el usuario

•La sintaxis general para definir un tipo registro es:

```
TYPE tipo_registro IS RECORD(
 campo1 tipo1 [NOT NULL][:= expr1],
 campo2 tipo2 [NOT NULL][:= expr2],
 .....
 campoN tipoN [NOT NULL][:= exprN]);
```

```
DECLARE
TYPE t_EjRegistro IS RECORD(
 Cont NUMBER (4),
 Nombre VARCHAR(10) NOT NULL := 'Ana',
 Fecha DATE,
 Descripcion VARCHAR (45) := 'Ejemplo');

v_Ejemplo1 t_EjRegistro;
v_Ejemplo2 t_Ejregistro;
```

•Para hacer referencia a los campos de un registro se utiliza un punto (.)
nombre_registro.nombre_campo

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

•También se pueden asignar valores a un registro completo mediante la orden SELECT que extraería los datos de la BD y los almacenaría en el registro.

```
DECLARE
  --Define un registro con algunos campo de la tabla students

  TYPE t_StudentRecord IS RECORD (
    FirstName  students.first_name%TYPE,
    LastName   students.last_name%TYPE,
    Major      students.major%TYPE);

  -- Declara una variable para recibir los datos
  v_Student  t_StudentRecord;
BEGIN
  -- Recupera información del estudiante conID 10,000.
    SELECT first_name, last_name, major
    INTO v_Student
    FROM students
    WHERE ID = 10000;
END;
/
```

Tablas

•Las tablas de PL/SQL son tipos de datos que nos permiten almacenar varios valores del mismo tipo de datos.

Una tabla PL/SQL :

•Es similar a un array

•Tiene dos componentes: Un índice de tipo BINARY_INTEGER que permite acceder a los elementos en la tabla PL/SQL y una columna de escalares o registros que contiene los valores de la tabla PL/SQL

•Puede incrementar su tamaño dinámicamente.

•Para poder declarar una tabla es necesario primero definir su tipo y luego una variable de dicho tipo.

•La sintaxis general para definir un tipo de tabla es:

TYPE nombre_tipotabla **IS TABLE OF** tipo[NOT NULL] **INDEX BY BINARY_INTEGER;**

Donde:

•**tipotabla.-**es el nombre del nuevo tipo que está siendo definido

•**tipo.-**es un tipo escalar predefinido o una referencia a un tipo escalar mediante %TYPE

•Una vez declarados el tipo y la variable, podemos hacer referencia a un elemento determinado de la tabla PL/SQL mediante la sintaxis:

nombretabla (índice)

•Donde nombretabla es el nombre de una tabla, e índice es una variable de tipo BINARY_INTEGER

•Ejemplo:

```
DECLARE
TYPE tipo_tabla IS TABLE OF VARCHAR2(9)
INDEX BY BINARY_INTEGER;
V_tabla tipo_tabla;
BEGIN
V_tabla(1):='';
...
END;
```

```
DECLARE
  /* Definimos el tipo PAISES como tabla PL/SQL */
  TYPE PAISES IS TABLE OF NUMBER INDEX BY BINARY_INTEGER ;
  /* Declaramos una variable del tipo PAISES */
  tPAISES PAISES;
BEGIN
  tPAISES(1) := 1;
  tPAISES(2) := 2;
  tPAISES(3) := 3;
END;
```

•El rango de binary integer es
–2147483647.. 2147483647, por lo tanto el índice puede ser negativo, lo cual indica que el índice del primer valor no tiene que ser necesariamente el cero

Tablas PL/SQL de registros

•Es posible declarar elementos de una tabla PL/SQL como de tipo registro.

```
DECLARE

 TYPE PAIS IS RECORD
 (
  CO_PAIS     NUMBER NOT NULL ,
  DESCRIPCION VARCHAR2(50),
  CONTINENTE  VARCHAR2(20)
 );
 TYPE PAISES IS TABLE OF PAIS INDEX BY BINARY_INTEGER ;
 tPAISES PAISES;
BEGIN

  tPAISES (1).CO_PAIS := 27;
 tPAISES (1).DESCRIPCION := 'ITALIA';
 tPAISES (1).CONTINENTE  := 'EUROPA';

END;
```

Funciones para el manejo de tablas PL/SQL

•Cuando trabajamos con tablas de PL podemos utilizar las siguientes funciones:
–**FIRST**. Devuelve el menor índice de la tabla. NULL si está vacía.
–**LAST**. Devuelve el mayor índice de la tabla. NULL si está vacía.

•El siguiente ejemplo muestra el uso de FIRST y LAST :

```
DECLARE
 TYPE ARR_CIUDADES IS TABLE OF VARCHAR2(50) INDEX BY
BINARY_INTEGER;
 misCiudades ARR_CIUDADES;
BEGIN
    misCiudades(1) := 'MADRID';
    misCiudades(2) := 'BILBAO';
    misCiudades(3) := 'MALAGA';

    FOR i IN misCiudades.FIRST..misCiudades.LAST
    LOOP
       dbms_output.put_line(misCiudades(i));
    END LOOP;
END;
```

En SQL*PLUS:

```
SQL> DECLARE
  2   TYPE ARR_CIUDADES IS TABLE OF VARCHAR2(50) INDEX BY
BINARY_INTEGER;
  3   misCiudades ARR_CIUDADES;
  4  BEGIN
  5      misCiudades(1) := 'MADRID';
  6      misCiudades(2) := 'BILBAO';
  7      misCiudades(3) := 'MALAGA';
  8
  8      FOR i IN misCiudades.FIRST..misCiudades.LAST
  9      LOOP
 10          dbms_output.put_line(misCiudades(i));
 11       END LOOP;
 12  END;
 13  /
MADRID
BILBAO
MALAGA

Procedimiento PL/SQL terminado con éxito.
```

•**EXISTS**(i). Utilizada para saber si en un cierto índice hay almacenado un valor. Devolverá TRUE si en el índice i hay un valor.

```
DECLARE
 TYPE ARR_CIUDADES IS TABLE OF VARCHAR2(50) INDEX BY
BINARY_INTEGER;
 misCiudades ARR_CIUDADES;
BEGIN
    misCiudades(1) := 'MADRID';
    misCiudades(3) := 'MALAGA';

    FOR i IN misCiudades.FIRST..misCiudades.LAST
    LOOP
       IF misCiudades.EXISTS(i) THEN
         dbms_output.put_line(misCiudades(i));
       ELSE
         dbms_output.put_line('El elemento no existe:'||TO_CHAR(i));
       END IF;
    END LOOP;
END;
```

En SQL*PLUS:

```
SQL> DECLARE
  2   TYPE ARR_CIUDADES IS TABLE OF VARCHAR2(50) INDEX BY
BINARY_INTEGER;
  3   misCiudades ARR_CIUDADES;
  4  BEGIN
  5      misCiudades(1) := 'MADRID';
  6      misCiudades(3) := 'MALAGA';
  7
  7      FOR i IN misCiudades.FIRST..misCiudades.LAST
  8      LOOP
  9         IF misCiudades.EXISTS(i) THEN
 10           dbms_output.put_line(misCiudades(i));
 11         ELSE
 12           dbms_output.put_line('El elemento no existe:'||TO_CHAR(i));
 13         END IF;
 14      END LOOP;
 15  END;
 16  /
MADRID
El elemento no existe:2
MALAGA

Procedimiento PL/SQL terminado con éxito.
```

•**COUNT**. Devuelve el número de elementos de la tabla PL/SQL.

```
DECLARE
 TYPE ARR_CIUDADES IS TABLE OF VARCHAR2(50) INDEX BY
BINARY_INTEGER;
 misCiudades ARR_CIUDADES;
BEGIN
    misCiudades(1) := 'MADRID';
    misCiudades(3) := 'MALAGA';
    /* Devuelve 2, ya que solo hay dos elementos con valor */
    dbms_output.put_line(
    'El número de elementos es:'||misCiudades.COUNT);
END;
```

En SQL*PLUS:

```
SQL> DECLARE
  2   TYPE ARR_CIUDADES IS TABLE OF VARCHAR2(50) INDEX BY
BINARY_INTEGER;
  3   misCiudades ARR_CIUDADES;
  4  BEGIN
  5       misCiudades(1) := 'MADRID';
  6       misCiudades(3) := 'MALAGA';
  7       /* Devuelve 2, ya que solo hay dos elementos con valor */
  8       dbms_output.put_line(
  9       'El número de elementos es:'||misCiudades.COUNT);
 10  END;
 11  /
El número de elementos es:2

Procedimiento PL/SQL terminado con éxito.
```

•**PRIOR** (n). Devuelve el número del índice anterior a n en la tabla.

```
DECLARE
  TYPE ARR_CIUDADES IS TABLE OF VARCHAR2(50) INDEX BY
BINARY_INTEGER;
  misCiudades ARR_CIUDADES;
BEGIN
  misCiudades(1) := 'MADRID';
  misCiudades(3) := 'MALAGA';
  /* Devuelve 1, ya que el elemento 2 no existe */
  dbms_output.put_line(
  'El elemento previo a 3 es:' || misCiudades.PRIOR(3));
END;
```

En SQL*PLUS:

```
SQL> DECLARE
  2    TYPE ARR_CIUDADES IS TABLE OF VARCHAR2(50) INDEX BY
BINARY_INTEGER;
  3    misCiudades ARR_CIUDADES;
  4  BEGIN
  5    misCiudades(1) := 'MADRID';
  6    misCiudades(3) := 'MALAGA';
  7    /* Devuelve 1, ya que el elemento 2 no existe */
  8    dbms_output.put_line(
  9    'El elemento previo a 3 es:' || misCiudades.PRIOR(3));
 10  END;
 11  /
El elemento previo a 3 es:1

Procedimiento PL/SQL terminado con éxito.
```

•**NEXT** (n). Devuelve el número del índice posterior a n en la tabla.

```
DECLARE
  TYPE ARR_CIUDADES IS TABLE OF VARCHAR2(50) INDEX BY
BINARY_INTEGER;
  misCiudades ARR_CIUDADES;
BEGIN
  misCiudades(1) := 'MADRID';
  misCiudades(3) := 'MALAGA';
  /* Devuelve 3, ya que el elemento 2 no existe */
  dbms_output.put_line(
  'El elemento siguiente a 1 es:' || misCiudades.NEXT(1));
END;
```

En SQL*PLUS:

```
SQL> DECLARE
  2    TYPE ARR_CIUDADES IS TABLE OF VARCHAR2(50) INDEX BY
BINARY_INTEGER;
  3    misCiudades ARR_CIUDADES;
  4  BEGIN
  5    misCiudades(1) := 'MADRID';
  6    misCiudades(3) := 'MALAGA';
  7    /* Devuelve 3, ya que el elemento 2 no existe */
  8    dbms_output.put_line(
  9    'El elemento siguiente a 1 es:' || misCiudades.NEXT(1));
 10  END;
 11  /
El elemento siguiente a 1 es:3
```

Procedimiento PL/SQL terminado con éxito.

•**TRIM**. Borra un elemento del final de la tabla PL/SQL.
•**TRIM**(n) borra n elementos del final de la tabla PL/SQL.
•**DELETE**. Borra todos los elementos de la tabla PL/SQL.
•**DELETE**(n) borra el correspondiente al índice n.
•**DELETE**(m,n) borra los elementos entre m y n

Tipo VARRAY

Similares a las tablas de PL/SQL definidas por el usuario, pero se implementan distinto

•Un varray se manipula de forma muy similar a las tablas de PL, pero se implementa de forma diferente.

•Los elementos en el varray se almacenan comenzando en el índice 1 hasta la longitud máxima declarada en el tipo varray.

•La sintaxis general es la siguiente:

TYPE <*nombre_tipo*> **IS VARRAY** (<*tamaño_maximo*>) **OF** <*tipo_elementos*>;

•Una consideración a tener en cuenta es que en la declaración de un varray el tipo de datos no puede ser de los siguientes tipos de datos:

•BOOLEAN
•NCHAR
•NCLOB
•NVARCHAR(n)
•REF CURSOR
•TABLE
•VARRAY

•Sin embargo se puede especificar el tipo utilizando los atributos **%TYPE** y **%ROWTYPE**.
•Los **VARRAY** deben estar inicializados antes de poder utilizarse.
•Para inicializar un **VARRAY** se utiliza un constructor (podemos inicializar el VARRAY en la sección DECLARE o bien dentro del cuerpo del bloque):

```
DECLARE

/* Declaramos el tipo VARRAY de cinco elementos VARCHAR2*/
TYPE t_cadena IS VARRAY(5) OF VARCHAR2(50);

/* Asignamos los valores con un constructor */
v_lista t_cadena:= t_cadena('Héctor', 'Alicia', 'Pedro',",");
BEGIN
v_lista(4) := 'Cristina';
```

v_lista(5) := ‘Cecilia';
END;

•El tamaño de un VARRAY se establece mediante el número de parámetros utilizados en el constructor

•Si declaramos un VARRAY de cinco elementos pero al inicializarlo pasamos sólo tres parámetros al constructor, el tamaño del VARRAY será tres.

•Si se hacen asignaciones a elementos que queden fuera del rango se producirá un error.

•El tamaño de un VARRAY podrá aumentarse utilizando la función EXTEND, pero nunca con mayor dimensión que la definida en la declaración del tipo.

•Por ejemplo, la variable v_lista que sólo tiene 3 valores definidos por lo que se podría ampliar hasta cinco elementos pero no más allá.

•Un VARRAY comparte con las tablas de PL/SQL todas las funciones válidas para ellas, pero añade las siguientes:

•**LIMIT** . Devuelve el número máximo de elementos que admite el VARRAY.

•**EXTEND** .Añade un elemento al VARRAY.

•**EXTEND**(n) .Añade (n) elementos al VARRAY.

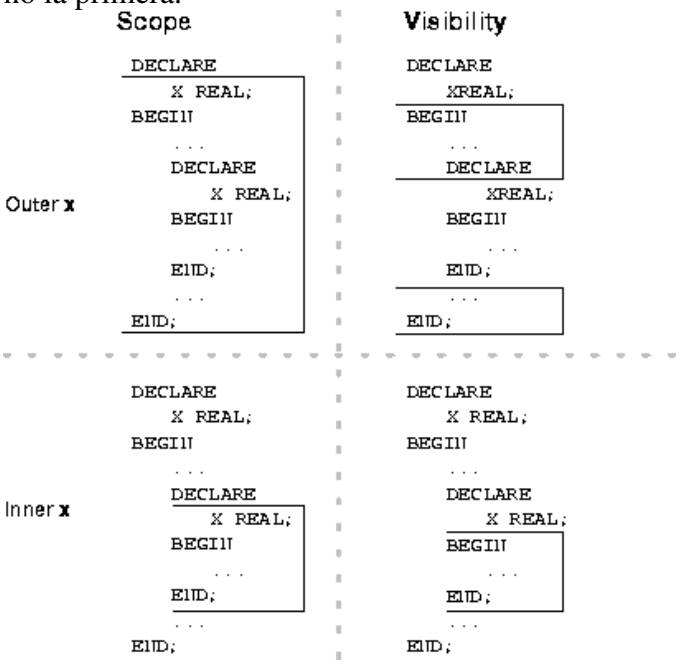
Alcance y Visibilidad

•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.

•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.

•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.

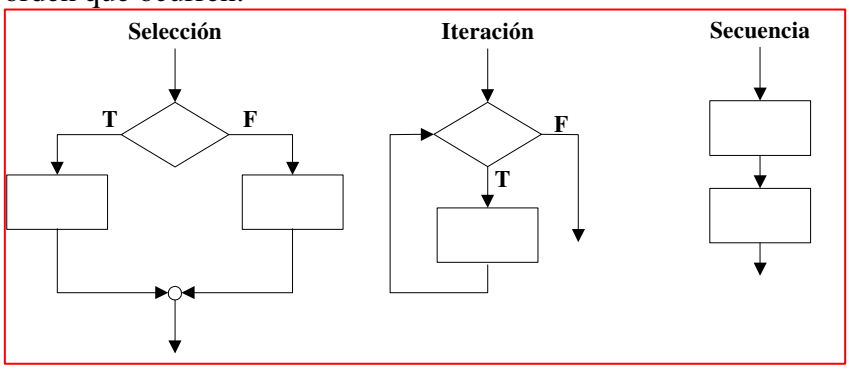


3. **Estructuras de control en PL/SQL**

Este tema se centra en estructurar el flujo de control dentro de un programa PL/SQL. Se podrá entender como las distintas sentencias se encuentran conectadas mediante un poderoso y simple control de estructuras que constan de un punto de entrada y uno de salida. En su conjunto estas estructuras pueden manejar cualquier situación y permiten una correcta estructuración del programa.

Antecedentes

- De acuerdo con el *Teorema de la Estructura*, cualquier programa computacional puede ser escrito utilizando las estructuras básicas de control
- Estas se pueden combinar de todas las maneras necesarias para alcanzar la solución de un problema dado.
- Las estructuras de selección verifican cierta condición, después ejecutan cierta secuencia de expresiones dependiendo si la condición resultó ser verdadera o falsa.
- Una *condición* es cualquier variable o expresión que retorna un valor booleano (TRUE o FALSE).
- Las estructuras de iteración ejecutan una secuencia de sentencias repetidamente mientras la condición permanezca verdadera.
- Las estructuras de secuencia simplemente ejecutan una secuencia de estamentos en el orden que ocurren.



Estructuras de Control

Control Condicional: Sentencia IF

- 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
IF – THEN – ELSIF

Control Condicional: Sentencia 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
        UPDATE sueldos SET pago = pago + bonus
        WHERE emp_no = emp_id;
END IF;
```

Sentencia: 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;
```

CASE

- Para elegir entre varios valores o cursos de acción, se puede utilizar CASE. La expresión CASE evalúa una condición y devuelve un valor para cada caso. La sintaxis es:

```
CASE [valor_a_comparar]
WHEN [valor1 | condicion1] THEN {sentencias;}
WHEN [valor2 | condicion2] THEN {sentencias;}
:
ELSE {sentencias};
END CASE;
```

- Por ejemplo:

```
DECLARE
  grado CHAR(1);
BEGIN
  grado := 'B';
  CASE grado
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excelente');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Muy Bien');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Bueno');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Insuficiente');
     ELSE DBMS_OUTPUT.PUT_LINE('No existe el grado');
  END CASE;
END;
```

- CASE con IF –THEN –ELSE

Otra forma…:

```
CASE
   WHEN empleo = 'analista' THEN
      IF sal < 3000 THEN comision := 0.1;
        ELSE comision := 0.05;
      END IF;
   WHEN empleo = 'vendedor' THEN
      IF sal < 4000 THEN comision := 0.06;
        ELSE comision := 0.05;
      END IF;
   ELSE
     Dbms_output.put_line ('no existe el empleo');
END CASE;
```

Controles de Iteración: La sentencia LOOP

•La sentencia LOOP permite ejecutar una secuencia de acciones múltiples veces. Todas ellas gobernadas por una condición que regula la ejecución de la iteración.

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

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;
```

•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;
```

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

EXIT – WHEN
•La sentencia 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.

WHILE - LOOP
•Esta sentencia se asocia a una condición con una secuencia de sentencias encerradas por las palabras clave LOOP y END LOOP, como sigue:

```
WHILE condición
LOOP
  secuencia_de_sentencias;
END LOOP;
```

•Antes de cada iteración del ciclo se evalúa la condición. Si ésta es verdadera se ejecuta la secuencia de sentencias y el control se devuelve al inicio del loop. Si la condición es falsa o nula, el ciclo se rompe y el control se transfiere a la próxima instrucción, fuera del loop.

FOR - LOOP

•En las instrucciones anteriores el número de iteraciones es desconocido, mientras no se evalúa la condición del ciclo.
•Con una instrucción del tipo FOR-LOOP, la iteración se efectúa un número finito (y conocido) de veces. La sintaxis de esta instrucción es la siguiente:

```
FOR contador IN [REVERSE] valor_minimo..valor_maximo
LOOP
  secuencia_de_sentencias;
END LOOP;
```

•El contador no necesita ser declarado porque por defecto se crea para el bloque que involucra el ciclo y luego se destruye.
•Por defecto, la iteración ocurre en forma creciente, es decir, desde el menor valor aportado hasta el mayor. Sin embargo, si se desea alterar esta condición por defecto, se debe incluir explícitamente en la sentencia la palabra REVERSE.
•Los límites de una iteración pueden ser literales, variables o expresiones, pero que deben evaluarse como números enteros.
•Un contador de loop tiene validez sólo dentro del ciclo. No es posible asignar un valor a una variable contadora de un loop, fuera de él.

•Ejemplo:

```
FOR cont IN 1..10
LOOP
  …
END LOOP;
sum := cont + 1 ;-- Esto no está permitido
```

•La sentencia EXIT también puede ser utilizada para abortar la ejecución del loop en forma prematura. Por ejemplo, en el siguiente trozo de programa la secuencia normal debería completarse luego de 10 veces de ejecutarse, pero la aparición de la cláusula EXIT podría hacer que ésta termine antes:

```
FOR j IN 1..10
LOOP
  FETCH c1 INTO emprec;
  EXIT WHEN c1%NOTFOUND;
  …
```

END LOOP;

Etiquetas

•En todos los bloques escritos en PL/SQL, los ciclos pueden ser rotulados. Un rótulo es un identificador encerrado entre los signos dobles **<<** y **>>** y debe aparecer al comienzo de un loop, como se muestra a continuación:

```
<<rótulo>>
LOOP
  secuencia de sentencias;
END LOOP;
```

•La última sentencia puede cambiarse también por *END LOOP rótulo*;

Controles de Secuencia: Las sentencias GOTO y NULL

GOTO

•La sentencia GOTO obliga a saltar a un rótulo del programa en forma incondicional. El rótulo debe ser único dentro de su alcance y debe preceder a una sentencia ejecutable o a un bloque PL/SQL. Cuando es ejecutada, esta instrucción transfiere el control a la sentencia o bloque rotulada.

•Los siguientes ejemplos ilustran una forma válida de utilizar la sentencia GOTO y otra no válida.

•Ejemplo válido:

```
BEGIN
…
<<actualiza>>
  BEGIN
    UPDATE emp SET…
    …
  END;
…
GOTO <<<actualiza>>
…
END;
```

•Ejemplo NO válido:

```
DECLARE
  done BOOLEAN;
BEGIN
  …
  FOR i IN 1..50
  LOOP
    IF done THEN
        GOTO fin_loop;
    END IF;
    …
    <<fin_loop>> -- Ilegal
  END LOOP;
-- Esta no es una sentencia ejecutable
END;
```

Restricciones

•una sentencia de este tipo no puede saltar dentro de una sentencia IF, LOOP o un sub-bloque.

•No se puede utilizar GOTO dentro del bloque de excepciones para salir de él.

NULL

•La sentencia NULL especifica explícitamente inacción. No hace nada más que pasar el control del programa a la siguiente sentencia. También sirve como un comodín para hacer el código más entendible, advirtiendo que la alternativa señalada no requiere codificación.

•Ejemplo:

```
EXCEPTION
  WHEN zero_divide THEN
        Rollback;
   WHEN value_error THEN
        INSERT INTO errores VALUES…
        Commit;
  WHEN others THEN
        NULL;
END;
```

Procesamiento de Transacciones

•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.

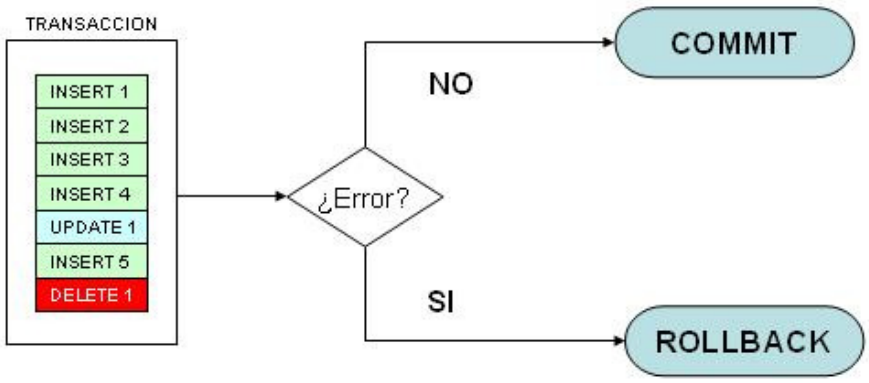
•Todas las sentencias que se ejecutan desde la ocurrencia del último *commit* o *rollback* comprenden la transacción (o grupo de transacciones) activa.

•La sentencia *savepoint* establece un punto de procesamiento dentro de una transacción y funciona de manera similar a un rótulo

•Son transaccionales las operaciones correspondiente al DML, es decir, sentencias SELECT, INSERT, UPDATE y DELETE

•Para confirmar una transacción se utiliza la sentencia **COMMIT**. Cuando realizamos **COMMIT** los cambios se escriben en la base de datos.

• Para deshacer una transacción se utiliza la sentencia **ROLLBACK**. Cuando realizamos **ROLLBACK** se deshacen todas las modificaciones realizadas por la transacción en la base de datos, quedando la base de datos en el mismo estado que antes de iniciarse la transacción.



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 accesan 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;
END;
```

•En el ejemplo anterior la sentencia COMMIT libera todas las filas bloqueadas de la tabla "cuenta".

•Note que la palabra *end* al final del código indica el final del bloque, no el fin de la transacción.

•Una transacción puede abarcar más de un bloque, como también dentro de un mismo bloque pueden ocurrir muchas transacciones.

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*).

•Ejemplo:

```
DECLARE
  emp_id  integer;
  …
BEGIN
  SELECT empno, … INTO emp_id, … FROM new_emp WHERE …
  …
  INSERT INTO emp VALUES (emp_id, …);
  INSERT INTO tax VALUES (emp_id, …);
  INSERT INTO pay VALUES (emp_id, …);
  …
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK;           --deshace todos los insert
  …
END;
```

Uso de SAVEPOINT

•Con la sentencia *savepoint* es posible nombrar y marcar un punto determinado donde se podrá retornar el control luego de ejecutarse una sentencia *rollback*.

•Ejemplo:

```
DECLARE
  emp_id   emp.empno%TYPE;
BEGIN
  UPDATE emp SET … WHERE empno=emp_id;
  DELETE FROM emp WHERE …
  …
  SAVEPOINT do_insert;
  …
  INSERT INTO emp VALUES (emp_id, …);
  …
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK TO do_insert;
END;
```

Excepciones Predefinidas (se verá con mayor profundidad más adelante)

Excepción	Descripción
TOO_MANY_ROWS	Hay más de una fila que corresponde a una orden SELECT..INTO
VALUE_ERROR	Error de truncamiento , aritmético o de conversión
ZERO_DIVIDE	División por cero
COLLECTION_IS_NULL	Se ha intentado aplicar métodos de conversión distintos de EXISTS a una tabla con valor NULL
CURSOR_ALREADY_OPEN	Se ha intentado abrir un cursos que ya estaba abierto
DUP_VAL_ON_INDEX	Violación de una restricción de unicidad.
INVALID_CURSOR	Operación ilegal con un cursor.
INVALID_NUMBER	Falló la conversión a un número.
LOGIN_DENIED	Nombre de usuario o contraseña no valido.
NO_DATA_FOUND	No se ha encontrado ningún dato.
NOT_LOGGED_ON	No existe conexión con Oracle.
PROGRAM_ERROR	Error interno PL/SQL
ACCESS_INTO_NULL	Se ha intentado asignar valores a los atributos de un Objeto que tiene el valor NULL
ROWTYPE_MISMATCH	Una variable de cursor del host y una variable de cursor de PL/SQL tienen tipos de filas incompatibles
SELF_IS_NULL	El programa ha intentado llamar a un método MEMBER con el primer parámetro null
STORAGE_ERROR	Error interno PL/SQL se queda sin memoria
SUBSCRIPT_BEYOND_COUNT	Una referencia a una tabla anidada o índice de varray mayor que el número de elementos de la colección
SUBSCRIPT_OUTSIDE_LIMIT	Una referencia a una tabla anidada o índice de varray fuera del rango declarado
SYS_INVALID_ROWID	Fallo al convertir un string de caracteres a un tipo ROWID

- Es posible establecer puntos de control (*savepoint*) en programas recursivos.
- En ese caso, cada instancia de la llamada al programa mantendrá sus propios puntos de control y el que se encuentre activo corresponderá a la instancia del programa que se esté ejecutando.