1. Tutorial PL/SQL.

1.1. Introducción.

SQL es un lenguaje de consulta para los sistemas de bases de datos relacionales, pero que no posee la potencia de los lenguajes de programación. No permite el uso de variables, estructuras de control de flujo, bucles, entre otros; además elementos característicos de la programación. No es de extrañar, **SQL es un lenguaje de consulta, no un lenguaje de programación.**

Sin embargo, SQL es la herramienta ideal para trabajar con bases de datos. Cuando se desea realizar una aplicación completa para el manejo de una base de datos relacional, resulta necesario utilizar alguna herramienta que soporte la capacidad de consulta del SQL y la versatilidad de los lenguajes de programación tradicionales. PL/SQL es el lenguaje de programación que proporciona Oracle para extender el SQL estándar con otro tipo de instrucciones y elementos propios de los lenguajes de programación.

Para abordar el presente tutorial con mínimo de garantías es necesario conocer previamente SQL.

Cuando se desea realizar una aplicación completa para el manejo de una base de datos relacional, resulta necesario utilizar alguna herramienta que soporte la capacidad de consulta del SQL y la versatilidad de los lenguajes de programación tradicionales. PL/SQL es el lenguaje de programación que proporciona **Oracle** para extender el SQL estándar con otro tipo de instrucciones.

¿Que vamos a necesitar?

Para poder seguir este tutorial correctamente necesitaremos tener los siguientes elementos:

- Una instancia de ORACLE 8i o superior funcionando correctamente.
- Herramientas cliente de ORACLE, en particular SQL*Plus para poder ejecutar los ejemplo.
- Haber configurado correctamente una conexión a ORACLE.

1.2. Programación PL/SQL.

Con PL/SQL vamos a poder programar las unidades de programa de la base de datos ORACLE, están son:

- Procedimientos almacenados
- Funciones
- Triggers
- Scripts

Pero además PL/SQL nos permite realizar programas sobre las siguientes herramientas de ORACLE:

- Oracle Forms
- Oracle Reports
- Oracle Graphics
- Oracle Aplication Server

2. Fundamentos de PL/SQL.

2.1. Primeros pasos con PL/SQL.

Para programar en PL/SQL es necesario conocer sus fundamentos.

Como introducción vamos a ver algunos elementos y conceptos básicos del lenguaje.

- PL/SQL no es CASE-SENSITIVE, es decir, no diferencia mayúsculas de minúsculas como otros lenguajes de programación como C o Java. **Sin embargo debemos recordar que ORACLE es CASE-SENSITIVE en la búsquedas de texto.**

- Una linea en PL/SQL contiene grupos de caracteres conocidos como UNIDADES LEXICAS, que pueden ser clasificadas como:

    - DELIMITADORES
    - IDENTIFICADORES
    - LITERALES
    - COMENTARIOS
    - EXPRESIONES

- DELIMITADOR: Es un símbolo simple o compuesto que tiene una función especial en PL/SQL. Estos pueden ser:

    - Operadores Aritméticos
    - Operadores Lógicos
    - Operadores Relacionales

- IDENTIFICADOR: Son empleados para nombrar objetos de programas en PL/SQL así como a unidades dentro del mismo, éstas unidades y objetos incluyen:

    - Constantes
    - Cursores
    - Variables
    - Subprogramas
    - Excepciones
    - Paquetes

- 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:

```
-- Linea simple
/*
Conjunto de Lineas
*/
```

2.2. Tipos de datos en PL/SQL.

Cada constante y variable tiene un tipo de dato en el cual se especifica el formato de almacenamiento, restricciones y rango de valores válidos.

PL/SQL proporciona una variedad predefinida de tipos de datos . Casi todos los tipos de datos manejados por PL/SQL son similares a los soportados por SQL. A continuación se muestran los TIPOS de DATOS más comunes:

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

```
    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 máxima 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
   posicones */
/* 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), que veremos en sus apartados correspondientes.

Existen por supuesto más tipos de datos, la siguiente tabla los muestra:

Tipo de dato / Sintáxis	Oracle 8i	Oracle 9i	Descripción
dec(p, e)	La precisión máxima es de 38 dígitos.	La precisión máxima es de 38 dígitos.	Donde **p** es la precisión y **e** la escala. **Por ejemplo:** dec(3,1) es un número que tiene 2 dígitos antes del decimal y un dígito después del decimal.
decimal(p, e)	La precisión máxima es de 38 dígitos.	La precisión máxima es de 38 dígitos.	Donde **p** es la precisión y **e** la escala. **Por ejemplo:** decimal(3,1) es un número que tiene 2 dígitos antes del decimal y un dígito después del decimal.
double precision			
float			
int			
integer			

Tipo de dato / Sintáxis	Oracle 8i	Oracle 9i	Descripción
numeric(p, e)	La precisión máxima es de 38 dígitos.	La precisión máxima es de 38 dígitos.	Donde **p** es la precisión y **e** la escala. **Por ejemplo:** numeric(7,2) es un número que tiene 5 dígitos antes del decimal y 2 dígitos después del decimal.
number(p, e)	La precisión máxima es de 38 dígitos.	La precisión máxima es de 38 dígitos.	Donde **p** es la precisión y **e** la escala. **Por ejemplo:** number(7,2) es un número que tiene 5 dígitos antes del decimal y 2 dígitos después del decimal.
real			
smallint			
char (tamaño)	Hasta 32767 bytes en PLSQL. Hasta 2000 bytes en Oracle 8i.	Hasta 32767 bytes en PLSQL. Hasta 2000 bytes en Oracle 9i.	Donde **tamaño** es el número de caracteres a almacenar. Son cadenas de ancho fijo. Se rellena con espacios.
varchar2 (tamaño)	Hasta 32767 bytes en PLSQL.	Hasta 32767 bytes en PLSQL.	Donde **tamaño** es el número de caracteres a

Tipo de dato / Sintáxis	Oracle 8i	Oracle 9i	Descripción
	Hasta 4000 bytes en Oracle 8i.	Hasta 4000 bytes en Oracle 9i.	almacenar. Son cadenas de ancho variable.
long	Hasta 2 gigabytes.	Hasta 2 gigabytes.	Son cadenas de ancho variable.
raw	Hasta 32767 bytes en PLSQL. Hasta 2000 bytes en Oracle 8i.	Hasta 32767 bytes en PLSQL. Hasta 2000 bytes en Oracle 9i.	Son cadenas binarias de ancho variable.
long raw	Hasta 2 gigabytes.	Hasta 2 gigabytes.	Son cadenas binarias de ancho variable.
date	Una fecha entre el 1 de Enero de 4712 A.C. y el 31 de Diciembre de 9999 D.C.	Una fecha entre el 1 de Enero de 4712 A.C. y el 31 de Diciembre de 9999 D.C.	
timestamp (*fractional seconds precision*)	No soportado por Oracle 8i.	**fractional seconds precision** debe ser un número entre 0 y 9. (El valor por defecto es 6)	Incluye año, mes día, hora, minutos y segundos. **Por ejemplo:** timestamp(6)
timestamp (*fractional seconds precision*) with time zone	No soportado por Oracle 8i.	**fractional seconds precision** debe ser un número entre 0 y 9. (El valor por defecto es 6)	Incluye año, mes día, hora, minutos y segundos; con un valor de desplazamiento de zona horaria. **Por ejemplo:** timestamp(5) with time zone

Tipo de dato / Sintáxis	Oracle 8i	Oracle 9i	Descripción
timestamp (*fractional seconds precision*) with local time zone	No soportado por Oracle 8i.	**fractional seconds precision** debe ser un número entre 0 y 9. (El valor por defecto es 6)	Incluye año, mes día, hora, minutos y segundos; con una zona horaria expresada como la zona horaria actual. **Por ejemplo:** timestamp(4) with local time zone
interval year (*year precision*) to month	No soportado por Oracle 8i.	**year precision** debe ser un número entre 0 y 9. (El valor por defecto es 2)	Período de tiempo almacenado en años y meses. **Por ejemplo:** interval year(4) to month
interval day (*day precision*) to second (*fractional seconds precision*)	No soportado por Oracle 8i.	**day precision** debe ser un número entre 0 y 9. (El valor por defecto es 2) **fractional seconds precision** debe ser un número entre 0 y 9. (El valor por defecto es 6)	Incluye año, mes día, hora, minutos y segundos. **Por ejemplo:** interval day(2) to second(6)

Tipo de dato / Sintáxis	Oracle 8i	Oracle 9i	Descripción
rowid	El formato del campo rowid es: BBBBBBB.RRRR.FFFF donde BBBBBBB es el bloque en el fichero de la base de datos; RRRR es la fila del bloque; FFFFF es el fichero de la base de datos.	El formato del campo rowid es: BBBBBBB.RRRR.FFFFF donde BBBBBBB es el bloque en el fichero de la base de datos; RRRR es la fila del bloque; FFFFF es el fichero de la base de datos.	Datos binarios de ancho fijo. Cada registro de la base de datos tiene una dirección física o rowid.
urowid [tamaño]	Hasta 2000 bytes.	Hasta 2000 bytes.	Rowid universal. Donde **tamaño** es opcional.
boolean	Válido en PLSQL, este tipo de datos no existe en Oracle 8i.	Válido en PLSQL, este tipo de datos no existe en Oracle 9i.	
nchar (tamaño)	Hasta 32767 bytes en PLSQL. Hasta 2000 bytes en Oracle 8i.	Hasta 32767 bytes en PLSQL. Hasta 2000 bytes en Oracle 9i.	Donde **tamaño** es el número de caracteres a almacenar. Cadena NLS de ancho fijo.
nvarchar2 (tamaño)	Hasta 32767 bytes en PLSQL. Hasta	Hasta 32767 bytes en PLSQL. Hasta 4000	Donde **tamaño** es el número de

Tipo de dato / Sintáxis	Oracle 8i	Oracle 9i	Descripción
	4000 bytes en Oracle 8i.	bytes en Oracle 9i.	caracteres a almacenar. Cadena NLS de ancho variable.
bfile	Hasta 4 gigabytes.	Hasta 4 gigabytes.	Localizadores de archivo apuntan a un objeto binario de sólo lectura fuera de la base de datos.
blob	Hasta 4 gigabytes.	Hasta 4 gigabytes.	Localizadores LOB apuntan a un gran objeto binario dentro de la base de datos.
clob	Hasta 4 gigabytes.	Hasta 4 gigabytes.	Localizadores LOB apuntan a un gran objeto de caracteres dentro de la base de datos.
nclob	Hasta 4 gigabytes.	Hasta 4 gigabytes.	Localizadores LOB apuntan a un gran objeto NLS de caracteres dentro de la base de datos.

2.3. Operadores en PL/SQL.

La siguiente tabla ilustra los operadores de 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 (negación) OR (o lógico)
Operador de concatenación	||

3. Estructuras de control en PL/SQL.

3.1. Estrucuturas de control de flujo.

En PL/SQL solo disponemos de la estructura condicional IF. Su sintaxis se muestra a continuación:

```
IF (expresion) THEN
            -- Instrucciones
        ELSIF (expresion) THEN
            -- Instrucciones
        ELSE
            -- Instrucciones
        END IF;
```

Un aspecto a tener en cuenta es que la instrucción condicional anidada es *ELSIF* y no "*ELS***E***IF*".

Sentencia GOTO

PL/SQL dispone de la sentencia GOTO. La sentencia GOTO desvía el flujo de ejecución a una determinada etiqueta.

En PL/SQL las etiquetas se indican del siguiente modo: << *etiqueta* >>

El siguiente ejemplo ilustra el uso de GOTO.

```
DECLARE
  flag NUMBER;
BEGIN
  flag :=1 ;
  IF (flag = 1) THEN
     GOTO paso2;
  END IF;
<<paso1>>
         dbms_output.put_line('Ejecucion de paso 1');
<<paso2>>
         dbms_output.put_line('Ejecucion de paso 2');
```

```
END;
```

3.2. Bucles ó ciclos.

En PL/SQL tenemos a nuestra disposición los siguientes iteradores o bucles:

- LOOP
- WHILE
- FOR

El bucle **LOOP,** se repite tantas veces como sea necesario hasta que se fuerza su salida con la instrucción **EXIT.** Su sintaxis es la siguiente:

```
LOOP
    -- Instrucciones
     IF (expresion) THEN
         -- Instrucciones
         EXIT;
     END IF;
END LOOP;
```

El bucle **WHILE,** se repite mientras que se cumpla la *expresión*.

```
WHILE (expresion) LOOP
    -- Instrucciones
END LOOP;
```

El bucle **FOR**, se repite tanta veces como le indiquemos en los identificadores *inicio* y *final.*

```
FOR contador IN [REVERSE] inicio..final LOOP

    -- Instrucciones

END LOOP;
```

En el caso de especificar **REVERSE** el bucle se recorre en sentido inverso.

3.3. Bloques PL/SQL.

Un programa de PL/SQL está compuesto por bloques. Un programa está compuesto como mínimo de un bloque.

Los bloques de PL/SQL pueden ser de los siguientes tipos:

- Bloques anónimos
- Subprogramas

3.3.1. Estructura de un Bloque.

Los bloques PL/SQL presentan una estructura específica compuesta de tres partes bien diferenciadas:

- La sección declarativa en donde se declaran todas las constantes y variables que se van a utilizar en la ejecución del bloque.
- La sección de ejecución que incluye las instrucciones a ejecutar en el bloque PL/SQL.
- La sección de excepciones en donde se definen los manejadores de errores que soportará el bloque PL/SQL.

Cada una de las partes anteriores se delimita por una palabra reservada, de modo que un bloque PL/SQL se puede representar como sigue:

```
[ declare | is | as ]
   /*Parte declarativa*/

begin
        /*Parte de ejecucion*/

[ exception ]
        /*Parte de excepciones*/

end;
```

De las anteriores partes, únicamente la sección de ejecución es obligatoria, que quedaría delimitada entre las cláusulas **BEGIN** y **END**. Veamos un ejemplo de bloque PL/SQL muy genérico. Se trata de un bloque anónimos, es decir no lo identifica ningún nombre. Los bloques anónimos identifican su parte declarativa con la palabra reservada **DECLARE**.

```
DECLARE
        /*Parte declarativa*/

        nombre_variable DATE;

BEGIN
        /*Parte de ejecucion
          * Este código asigna el valor de la columna "nombre_columna"
         * a la variable identificada por "nombre_variable"
          */

        SELECT SYSDATE
        INTO nombre_variable
        FROM DUAL;

EXCEPTION
        /*Parte de excepciones*/
        WHEN OTHERS THEN
        dbms_output.put_line('Se ha producido un error');

END;
```

3.3.2. Sección de Declaración de Variables.

En esta parte se declaran las variables que va a necesitar nuestro programa. Una variable se declara asignándole un nombre o "identificador" seguido del tipo de valor que puede contener. También se declaran cursores, de gran utilidad para la consulta de datos, y excepciones definidas por el usuario. También podemos especificar si se trata de una constante, si puede contener valor nulo y asignar un valor inicial.

La sintaxis genérica para la declaración de constantes y variables es:

```
```

```
nombre_variable [CONSTANT] <tipo_dato> [NOT NULL][:=valor_inicial]
```

donde:

- tipo_dato: es el tipo de dato que va a poder almacenar la variable, este puede ser cualquiera de los tipos soportados por ORACLE, es decir **NUMBER**, **DATE**, **CHAR**, **VARCHAR, VARCHAR2, BOOLEAN ...** Además para algunos tipos de datos (NUMBER y VARCHAR) podemos especificar la longitud.

- La cláusula CONSTANT indica la definición de una constante cuyo valor no puede ser modificado. Se debe incluir la inicialización de la constante en su declaración.

- La cláusula NOT NULL impide que a una variable se le asigne el valor nulo, y por tanto debe inicializarse a un valor diferente de NULL.

- Las variables que no son inicializadas toman el valor inicial NULL.

- La inicialización puede incluir cualquier expresión legal de PL/SQL, que lógicamente debe corresponder con el tipo del identificador definido.

- Los tipos escalares incluyen los definidos en SQL más los tipos VARCHAR y BOOLEAN. Este último puede tomar los valores TRUE, FALSE y NULL, y se suele utilizar para almacenar el resultado de alguna operación lógica. VARCHAR es un sinónimo de CHAR.

- También es posible definir el tipo de una variable o constante, dependiendo del tipo de otro identificador, mediante la utilización de las cláusulas **%TYPE** y **%ROWTYPE**. Mediante la primera opción se define una variable o constante escalar, y con la segunda se define una variable fila, donde identificador puede ser otra variable fila o una tabla. Habitualmente se utiliza **%TYPE** para definir la variable del mismo tipo que tenga definido un campo en una tabla de la base de datos, mientras que **%ROWTYPE** se utiliza para declarar variables utilizando cursores.

Ejemplos:

3.3.3. Estructura de un bloque anónimo.

```
    DECLARE
       /* Se declara la variable de tipo VARCHAR2(15) identificada por v_location
y se le asigna
                         el valor "Granada"*/
       v_location VARCHAR2(15) := 'Granada';
                    /*Se declara la constante  de tipo NUMBER identificada por PI
y se le asigna
                         el valor 3.1416*/
       PI CONSTANT NUMBER := 3.1416;
                    /*Se declara la variable del mismo tipo que tenga el campo
nombre de la tabla tabla_empleados
          identificada por v_nombre y no se le asigna ningún valor */
       v_nombre tabla_empleados.nombre%TYPE;
                    /*Se declara la variable del tipo registro correspondiente a
un supuesto cursor, llamado
                       micursor, identificada por reg_datos*/
              reg_datos micursor%ROWTYPE;

    BEGIN
           /*Parte de ejecucion*/

    EXCEPTION
           /*Parte de excepciones*/

    END;
```

3.3.4. Estructura de un subprograma:

```
    CREATE PROCEDURE simple_procedure IS
       /* Se declara la variable de tipo VARCHAR2(15) identificada por v_location
y se le asigna el valor "Granada"*/
       v_location VARCHAR2(15) := 'Granada';
                   /*Se declara la constante  de tipo NUMBER identificada por PI
y se le asigna
                      el valor 3.1416*/
       PI CONSTANT NUMBER := 3.1416;
                   /*Se declara la variable del mismo tipo que tenga el campo
nombre de la tabla tabla_empleados
          identificada por v_nombre y no se le asigna ningún valor */
       v_nombre tabla_empleados.nombre%TYPE;
                   /*Se declara la variable del tipo registro correspondiente a
un supuesto cursor, llamado
                      micursor, identificada por reg_datos*/
              reg_datos micursor%ROWTYPE;

    BEGIN
           /*Parte de ejecucion*/

    EXCEPTION
           /*Parte de excepciones*/

    END;
```

4. Cursores en PL/SQL.

4.1. Introducción a cursores PL/SQL.

PL/SQL utiliza cursores para gestionar las instrucciones **SELECT**. Un cursor es un conjunto de registros devuelto por una instrucción SQL. Técnicamente los cursores son fragmentos de memoria que reservados para procesar los resultados de una consulta **SELECT**.

Podemos distinguir dos tipos de cursores:

- **Cursores implícitos**. Este tipo de cursores se utiliza para operaciones **SELECT INTO**. Se usan cuando la consulta devuelve un único registro.
- **Cursores explícitos**. Son los cursores que son declarados y controlados por el programador. Se utilizan cuando la consulta devuelve un conjunto de registros. Ocasionalmente también se utilizan en consultas que devuelven un único registro por razones de eficiencia. Son más rápidos.

Un cursor se define como cualquier otra variable de PL/SQL y debe nombrarse de acuerdo a los mismos convenios que cualquier otra variable. Los cursores implícitos no necesitan declaración.

El siguiente ejemplo declara un cursor explícito:

```
declare
  cursor c_paises is
  SELECT CO_PAIS, DESCRIPCION
  FROM PAISES;
begin
/* Sentencias del bloque ...*/
end;
```

Para procesar instrucciones SELECT que devuelvan más de una fila, son necesarios cursores explícitos combinados con un estructura de bloque.

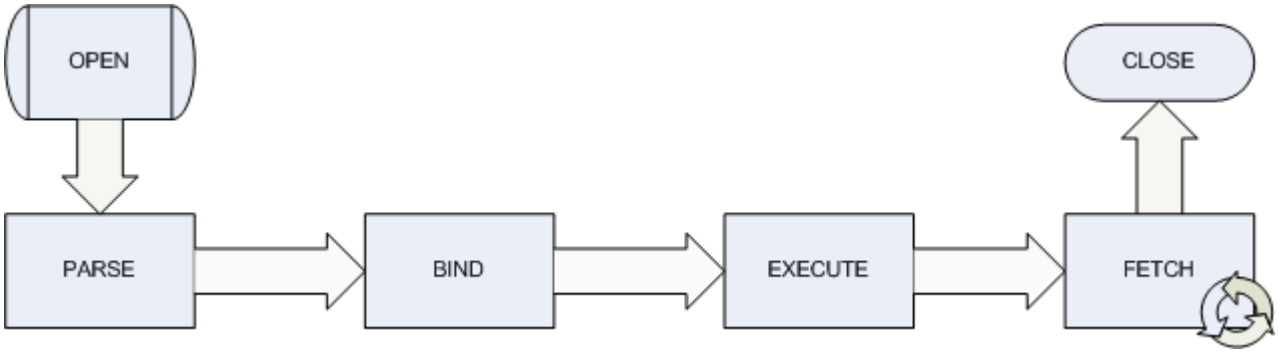
Un cursor admite el uso de parámetros. Los parámetros deben declararse junto

con el cursor.

El siguiente ejemplo muestra la declaración de un cursor con un parámetro, identificado por p_continente.

```
declare
  cursor c_paises (p_continente IN VARCHAR2) is
  SELECT CO_PAIS, DESCRIPCION
  FROM PAISES
  WHERE CONTINENTE = p_continente;
begin
/* Sentencias del bloque ...*/
end;
```

El siguiente diagrama representa como se procesa una instrucción SQL a través de un cursor.



Fases para procesar una instrucción SQL

4.2. Cursores Implícitos.

Declaración de cursores implícitos.

Los cursores implícitos se utilizan para realizar consultas **SELECT** que devuelven un único registro.

Deben tenerse en cuenta los siguientes puntos cuando se utilizan cursores implícitos:

- Con cada cursor implícito debe existir la palabra clave **INTO**.
- Las variables que reciben los datos devueltos por el cursor tienen que contener el mismo tipo de dato que las columnas de la tabla.
- Los cursores implícitos solo pueden devolver una única fila. En caso de que se devuelva más de una fila (o ninguna fila) se producirá una excepción. No se preocupe si aún no sabe que es una excepción, le valdrá conocer que es el medio por el que PL/SQL gestiona los errores.

El siguiente ejemplo muestra un cursor implícito:

```
declare
vdescripcion VARCHAR2(50);
begin
       SELECT DESCRIPCION
       INTO vdescripcion
       from PAISES
       WHERE CO_PAIS = 'ESP';

             dbms_output.put_line('La lectura del cursor es: ' ||
vdescripcion);
end;
```

La salida del programa generaría la siguiente línea:

```
La lectura del cursor es: ESPAÑA
```

4.2.1. Excepciones asociadas a los cursores implícitos.

Los cursores implícitos sólo pueden devolver una fila, por lo que pueden producirse determinadas excepciones. Las más comunes que se pueden encontrar son **no_data_found** y **too_many_rows**. La siguiente tabla explica brevemente estas excepciones.

Excepción	**Explicación**
NO_DATA_FOUND	Se produce cuando una sentencia SELECT intenta recuperar datos pero ninguna fila satisface sus condiciones. Es decir, cuando "*no hay datos*"
TOO_MANY_ROWS	Dado que cada cursor implícito sólo es capaz de recuperar una fila , esta excepción detecta la existencia de más de una fila.

4.3. Cursores Explícitos en PL/SQL

4.3.1. Declaración de cursores explícitos.

Los cursores explícitos se emplean para realizar consultas **SELECT** que pueden devolver cero filas, o más de una fila.

Para trabajar con un cursor explicito necesitamos realizar las siguientes tareas:

- Declarar el cursor.
- Abrir el cursor con la instrucción **OPEN**.
- Leer los datos del cursor con la instrucción **FETCH**.
- Cerrar el cursor y liberar los recursos con la instrucción **CLOSE**.

Para declarar un cursor debemos emplear la siguiente sintaxis:

```
CURSOR nombre_cursor IS
 instrucción_SELECT
```

También debemos declarar los posibles parámetros que requiera el cursor:

```
CURSOR nombre_cursor(param1 tipo1, ..., paramN tipoN) IS
 instrucción_SELECT
```

Para abrir el cursor

```
 OPEN nombre_cursor;

 o bien (en el caso de un cursor con parámetros)

 OPEN nombre_cursor(valor1, valor2, ..., valorN);
```

Para recuperar los datos en variables PL/SQL.

```
FETCH nombre_cursor INTO lista_variables;

-- o bien ...

FETCH nombre_cursor INTO registro_PL/SQL;
```

Para cerrar el cursor:

```
CLOSE nombre_cursor;
```

El siguiente ejemplo ilustra el trabajo con un cursor explícito. Hay que tener en cuenta que al leer los datos del cursor debemos hacerlo sobre variables del mismo tipo de datos de la tabla (o tablas) que trata el cursor.

```
DECLARE
  CURSOR cpaises
  IS
  SELECT CO_PAIS, DESCRIPCION, CONTINENTE
  FROM PAISES;

  co_pais VARCHAR2(3);
  descripcion VARCHAR2(50);
  continente  VARCHAR2(25);
BEGIN
  OPEN cpaises;
  FETCH cpaises INTO co_pais,descripcion,continente;
  CLOSE cpaises;
END;
```

Podemos simplificar el ejemplo utilizando el atributo de tipo **%ROWTYPE** sobre el cursor.

```
DECLARE
  CURSOR cpaises
  IS
  SELECT CO_PAIS, DESCRIPCION, CONTINENTE
  FROM PAISES;
    registro cpaises%ROWTYPE;
BEGIN
  OPEN cpaises;
  FETCH cpaises INTO registro;
  CLOSE cpaises;
```

```
END;
```

El mismo ejemplo, pero utilizando parámetros:

```
DECLARE
  CURSOR cpaises (p_continente VARCHAR2)
  IS
  SELECT CO_PAIS, DESCRIPCION, CONTINENTE
  FROM PAISES
  WHERE CONTINENTE = p_continente;

  registro cpaises%ROWTYPE;
BEGIN
  OPEN cpaises('EUROPA');
  FETCH cpaises INTO registro;
  CLOSE cpaises;
END;
```

Cuando trabajamos con cursores debemos considerar:

- Cuando un cursor está cerrado, no se puede leer.
- Cuando leemos un cursor debemos comprobar el resultado de la lectura utilizando los atributos de los cursores.
- Cuando se cierra el cursor, es ilegal tratar de usarlo.
- Es ilegal tratar de cerrar un cursor que ya está cerrado o no ha sido abierto

4.3.2. Atributos de cursores.

Toman los valores TRUE, FALSE o NULL dependiendo de la situación:

Atributo	Antes de abrir	Al abrir	Durante la recuperación	Al finalizar la recuperación	Después de cerrar
%NOTFOUND	**ORA-1001**	NULL	FALSE	TRUE	ORA-1001
%FOUND	ORA-1001	NULL	TRUE	FALSE	ORA-1001
%ISOPEN	FALSE	TRUE	TRUE	TRUE	FALSE
%ROWCOUNT	ORA-1001	0	*	**	ORA-1001

* Número de registros que ha recuperado hasta el momento
** Número de total de registros

4.3.3. Manejo del cursor.

Por medio de ciclo LOOP podemos iterar a través del cursor. Debe tenerse cuidado de agregar una condición para salir del bucle:

Vamos a ver varias formas de iterar a través de un cursor. La primera es utilizando un bucle LOOP con una sentencia EXIT condicionada:

```
OPEN nombre_cursor;
LOOP
    FETCH nombre_cursor INTO lista_variables;
    EXIT WHEN nombre_cursor%NOTFOUND;
    /* Procesamiento de los registros recuperados */
END LOOP;
CLOSE nombre_cursor;
```

Aplicada a nuestro ejemplo anterior:

```
DECLARE

    CURSOR cpaises

    IS

    SELECT CO_PAIS, DESCRIPCION, CONTINENTE

    FROM  PAISES;

     co_pais VARCHAR2(3);

    descripcion VARCHAR2(50);

    continente VARCHAR2(25);

BEGIN

    OPEN cpaises;

    LOOP

        FETCH cpaises INTO co_pais,descripcion,continente;

        EXIT WHEN cpaises%NOTFOUND;

        dbms_output.put_line(descripcion);

    END LOOP;

    CLOSE cpaises;
```

```
    END;
```

Otra forma es por medio de un bucle WHILE LOOP. La instrucción FECTH aparece dos veces.

```
   OPEN nombre_cursor;
   FETCH nombre_cursor INTO lista_variables;
   WHILE nombre_cursor%FOUND
   LOOP
       /* Procesamiento de los registros recuperados */
       FETCH nombre_cursor INTO lista_variables;
   END LOOP;
   CLOSE nombre_cursor;
```

```
DECLARE CURSOR cpaises

        IS    SELECT CO_PAIS, DESCRIPCION, CONTINENTE

        FROM PAISES;

        co_pais VARCHAR2(3);

        descripcion VARCHAR2(50);

        continente VARCHAR2(25);

    BEGIN

        OPEN cpaises;

        FETCH cpaises INTO co_pais,descripcion,continente;

        WHILE cpaises%found

        LOOP

            dbms_output.put_line(descripcion);

            FETCH cpaises INTO co_pais,descripcion,continente;

        END LOOP;

        CLOSE cpaises;

    END;
```

Por último podemos usar un bucle FOR LOOP. Es la forma más corta ya que el cursor implícitamente ejecutan las instrucciones OPEN, FECTH y CLOSE.

```
FOR variable IN nombre_cursor LOOP
    /* Procesamiento de los registros recuperados */
END LOOP;
```

```
BEGIN
    FOR REG IN (SELECT * FROM PAISES)
    LOOP
        dbms_output.put_line(reg.descripcion);
    END LOOP;
END;
```

4.3.4. Cursores de actualización.

Declaración y utilización de cursores de actualización.

Los cursores de actualización se declaran igual que los cursores explícitos, añadiendo **FOR UPDATE** al final de la sentencia select.

```
CURSOR nombre_cursor IS
 instrucción_SELECT
FOR UPDATE
```

Para actualizar los datos del cursor hay que ejecutar una sentencia **UPDATE** especificando la clausula **WHERE CURRENT OF** *<cursor_name>*.

```
UPDATE <nombre_tabla> SET
<campo_1> = <valor_1>
[,<campo_2> = <valor_2>]
WHERE CURRENT OF <cursor_name>
```

El siguiente ejemplo muestra el uso de un cursor de actualización:

```
DECLARE
    CURSOR cpaises IS
    select CO_PAIS, DESCRIPCION, CONTINENTE
    from paises
    FOR UPDATE;
    co_pais VARCHAR2(3);
    descripcion VARCHAR2(50);
    continente VARCHAR2(25);
BEGIN
    OPEN cpaises;
    FETCH cpaises INTO co_pais,descripcion,continente;
    WHILE cpaises%found
    LOOP
        UPDATE PAISES
        SET CONTINENTE = CONTINENTE || '.'
        WHERE CURRENT OF cpaises;
        FETCH cpaises INTO co_pais,descripcion,continente;
    END LOOP;
    CLOSE cpaises;
    COMMIT;
END;
```

Cuando trabajamos con cursores de actualización debemos tener en cuenta que éstos cursores de actualización generan bloqueos en la base de datos.

5. Excepciones en PL/SQL.

5.1. Manejo de excepciones.

En PL/SQL una advertencia o condición de error es llamada una excepción.

Las excepciones se controlan dentro de su propio bloque.La estructura de bloque de una excepción se muestra a continuación.

```
DECLARE
 -- Declaraciones
BEGIN
  -- Ejecucion
EXCEPTION
  -- Excepcion
END;
```

Cuando ocurre un error, se ejecuta la porción del programa marcada por el bloque **EXCEPTION**, transfiriéndose el control a ese bloque de sentencias.

El siguiente ejemplo muestra un bloque de excepciones que captura las excepciones **NO_DATA_FOUND** y **ZERO_DIVIDE.** Cualquier otra excepción será capturada en el bloque **WHEN OTHERS THEN**.

```
DECLARE
 -- Declaraciones
BEGIN
  -- Ejecucion
EXCEPTION
WHEN NO_DATA_FOUND THEN
  -- Se ejecuta cuando ocurre una excepcion de tipo NO_DATA_FOUND
WHEN ZERO_DIVIDE THEN
  -- Se ejecuta cuando ocurre una excepcion de tipo ZERO_DIVIDE
WHEN OTHERS THEN
  -- Se ejecuta cuando ocurre una excepcion de un tipo no tratado
  -- en los bloques anteriores
END;
```

Como ya hemos dicho cuando ocurre un error, se ejecuta el bloque **EXCEPTION**, transfiriéndose el control a las sentencias del bloque. Una vez finalizada la ejecución del bloque de **EXCEPTION** no se continua ejecutando el bloque anterior.

Si existe un bloque de excepción apropiado para el tipo de excepción se ejecuta dicho bloque. Si no existe un bloque de control de excepciones adecuado al tipo de excepcion se ejecutará el bloque de excepción **WHEN OTHERS THEN** (si existe!). **WHEN OTHERS** debe ser el último manejador de excepciones.

Las excepciones pueden ser definidas en forma interna o explícitamente por el usuario. Ejemplos de excepciones definidas en forma interna son la división por

cero y la falta de memoria en tiempo de ejecución. Estas mismas condiciones excepcionales tienen sus propio tipos y pueden ser referenciadas por ellos: **ZERO_DIVIDE** y **STORAGE_ERROR**.

Las excepciones definidas por el usuario deben ser alcanzadas explícitamente utilizando la sentencia **RAISE**.

Con las excepciones se pueden manejar los errores cómodamente sin necesidad de mantener múltiples chequeos por cada sentencia escrita. También provee claridad en el código ya que permite mantener las rutinas correspondientes al tratamiento de los errores de forma separada de la lógica del negocio.

5.2. Excepciones predefinidas.

PL/SQL proporciona un gran número de excepciones predefinidas que permiten controlar las condiciones de error más habituales.

Las excepciones predefinidas no necesitan ser declaradas. Simplemente se utilizan cuando estas son lanzadas por algún error determinado.

La siguiente es la lista de las excepciones predeterminadas por PL/SQL y una breve descripción de cuándo son accionadas:

Excepción	Se ejecuta ...	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

Excepción	Se ejecuta ...	SQLCODE
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	-6532

Excepción	Se ejecuta ...	SQLCODE
	rango permitido (por ejemplo, el elemento “-1”)	
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

5.3. Excepciones definidas por el usuario.

PL/SQL permite al usuario definir sus propias excepciones, las que deberán ser declaradas y lanzadas explícitamente utilizando la sentencia **RAISE**.

Las excepciones deben ser declaradas en el segmento **DECLARE** de un bloque, subprograma o paquete. Se declara una excepción como cualquier otra variable, asignándole el tipo **EXCEPTION**. Las mismas reglas de alcance aplican tanto sobre variables como sobre las excepciones.

```
DECLARE
 -- Declaraciones
    MyExcepcion EXCEPTION;
BEGIN
```

```
  -- Ejecucion
EXCEPTION
  -- Excepcion
END;
```

5.4. Reglas de Alcance.

Una excepción es válida dentro de su ámbito de alcance, es decir el bloque o programa donde ha sido declarada. Las excepciones predefinidas son siempre válidas.

Como las variables, una excepción declarada en un bloque es local a ese bloque y global a todos los sub-bloques que comprende.

5.5. La sentencia RAISE

La sentencia **RAISE** permite lanzar una excepción en forma explícita. Es posible utilizar esta sentencia en cualquier lugar que se encuentre dentro del alcance de la excepción.

```
DECLARE
 -- Declaramos una excepcion identificada por VALOR_NEGATIVO
    VALOR_NEGATIVO EXCEPTION;
    valor NUMBER;
BEGIN
  -- Ejecucion
valor := -1;
        IF valor < 0 THEN
                RAISE VALOR_NEGATIVO;
        END IF;
EXCEPTION
  -- Excepcion
WHEN VALOR_NEGATIVO THEN
        dbms_output.put_line('El valor no puede ser negativo');
END;
```

Con la sentencia **RAISE** podemos lanzar una excepción definida por el usuario o predefinida, siendo el comportamiento habitual lanzar excepciones definidas por el usuario.

Recordar la existencia de la excepción **OTHERS**, que simboliza cualquier condición de excepción que no ha sido declarada. Se utiliza comúnmente para controlar cualquier tipo de error que no ha sido previsto. En ese caso, es común observar la sentencia **ROLLBACK** en el grupo de sentencias de la excepción o alguna de las funciones **SQLCODE – SQLERRM**, que se detallan en el próximo punto.

5.6. Uso de SQLCODE y SQLERRM.

Al manejar una excepción es posible usar las funciones predefinidas **SQLCode** y **SQLERRM** para aclarar al usuario la situación de error acontecida.

SQLcode devuelve el número del error de Oracle y un 0 (cero) en caso de éxito al ejecutarse una sentencia SQL.

Por otra parte, **SQLERRM** devuelve el correspondiente mensaje de error.

Estas funciones son muy útiles cuando se utilizan en el bloque de excepciones, para aclarar el significado de la excepción **OTHERS**.

Estas funciones no pueden ser utilizadas directamente en una sentencia SQL, pero sí se puede asignar su valor a alguna variable de programa y luego usar esta última en alguna sentencia.

```
DECLARE
  err_num NUMBER;
  err_msg VARCHAR2(255);
  result  NUMBER;

BEGIN

  SELECT 1/0 INTO result
  FROM DUAL;

EXCEPTION
WHEN OTHERS THEN

  err_num := SQLCODE;
  err_msg := SQLERRM;

  DBMS_OUTPUT.put_line('Error:'||TO_CHAR(err_num));
  DBMS_OUTPUT.put_line(err_msg);
END;
```

También es posible entregarle a la función **SQLERRM** un número negativo que represente un error de Oracle y ésta devolverá el mensaje asociado.

```
DECLARE
  msg VARCHAR2(255);
BEGIN
  msg := SQLERRM(-1403);
  DBMS_OUTPUT.put_line(MSG);
END;
```

5.7. Excepciones personalizadas en PL/SQL.

RAISE_APPLICATION_ERROR.

En ocasiones queremos enviar un mensaje de error personalizado al producirse una excepción PL/SQL.
Para ello es necesario utilizar la instrucción **RAISE_APPLICATION_ERROR**;

La sintaxis general es la siguiente:

```
RAISE_APPLICATION_ERROR(<error_num>,<mensaje>);
```

Siendo:

- error_num es un entero negativo comprendido entre -20001 y -20999
- mensaje la descripción del error

```
DECLARE
  v_div NUMBER;
BEGIN
     SELECT 1/0 INTO v_div FROM DUAL;
EXCEPTION
  WHEN OTHERS THEN
  RAISE_APPLICATION_ERROR(-20001,'No se puede dividir por cero');
END;
```

5.8. Propagación de excepciones en PL/SQL.

Una de las características más interesantes de la excepciones es la propagación de excepciones.

Cuando se lanza una excepción, el control se transfiere hasta la sección **EXCEPTION** del bloque donde se ha producido la excepción. Entonces se busca un manejador válido de la excepción (**WHEN** <*excepcion*>

En el caso de que no se encuentre ningún manejador válida el control del programa se desplaza hasta el bloque **EXCEPTION** del bloque que ha realizado la llamada PL/SQL.

Observemos el siguiente bloque de PL/SQL (Nótese que se ha añadido una clausula WHERE 1=2 para provocar una excepción **NO_DATA_FOUND**).

```
DECLARE

fecha DATE;
       FUNCTION fn_fecha RETURN DATE
       IS
         fecha DATE;
       BEGIN
            SELECT SYSDATE INTO fecha
            FROM DUAL
            WHERE 1=2;
            RETURN fecha;
       EXCEPTION
       WHEN ZERO_DIVIDE THEN
       dbms_output.put_line('EXCEPCION ZERO_DIVIDE CAPTURADA
                            EN fn_fecha');
       END;

BEGIN

   fecha := fn_fecha();
   dbms_output.put_line('La fecha es '||TO_CHAR(fecha,
'DD/MM/YYYY'));

EXCEPTION
    WHEN NO_DATA_FOUND THEN
    dbms_output.put_line('EXCEPCION NO_DATA_FOUND CAPTURADA EN
                          EL BLOQUE PRINCIPAL');
END;
```

La excepción **NO_DATA_FOUND** se produce durante la ejecución de la función fn_fecha, pero como no existe ningún manejador de la excepción en dicha función, la excepción se propaga hasta el bloque que ha realizado la llamada. En ese momento se captura la excepción.

6. Subprogramas en PL/SQL.

Como hemos visto anteriormente los bloques de PL/SQL pueden ser bloques anónimos (scripts) y subprogramas.

Los subprogramas son bloques de PL/SQL a los que asignamos un nombre identificativo y que normalmente almacenamos en la propia base de datos para su posterior ejecución.

Los subprogramas pueden recibir parámetros.

Los subprogramas pueden ser de varios tipos:

- Procedimientos almacenados
- Funciones
- Triggers
- Subprogramas.

6.1. Procedimientos almacenados.

Un procedimiento es un subprograma que ejecuta una acción especifica y que no devuelve ningún valor. Un procedimiento tiene un nombre, un conjunto de parámetros (opcional) y un bloque de código.

La sintaxis de un procedimiento almacenado es la siguiente:

```
CREATE [OR REPLACE]
PROCEDURE <procedure_name> [(<param1> [IN|OUT|IN OUT] <type>,
                             <param2> [IN|OUT|IN OUT] <type>,
...)]
IS
  -- Declaracion de variables locales
BEGIN
  -- Sentencias
[EXCEPTION]
  -- Sentencias control de excepcion
END [<procedure_name>];
```

El uso de OR REPLACE permite sobreescribir un procedimiento existente. Si se omite, y el procedimiento existe, se producirá, un error.

La sintaxis es muy parecida a la de un bloque anónimo, salvo porque se reemplaza la sección **DECLARE** por la secuencia **PROCEDURE ... IS** en la especificación del procedimiento.

Debemos especificar el tipo de datos de cada parámetro. **Al especificar el tipo de dato del parámetro no debemos especificar la longitud del tipo**.

Los parámetros pueden ser de entrada (**IN**), de salida (**OUT**) o de entrada salida (**IN OUT**). El valor por defecto es **IN**, y se toma ese valor en caso de que no especifiquemos nada.

```
CREATE OR REPLACE
PROCEDURE Actualiza_Saldo(cuenta NUMBER,
                          new_saldo NUMBER)
IS
  -- Declaracion de variables locales
BEGIN
  -- Sentencias
  UPDATE SALDOS_CUENTAS
        SET SALDO = new_saldo,
            FX_ACTUALIZACION = SYSDATE
  WHERE CO_CUENTA = cuenta;
END Actualiza_Saldo;
```

También podemos asignar un valor por defecto a los parámetros, utilizando la clausula **DEFAULT** o el operador de asignación (:=) .

```
CREATE OR REPLACE
PROCEDURE Actualiza_Saldo(cuenta NUMBER,
                          new_saldo NUMBER DEFAULT 10 )
IS
  -- Declaracion de variables locales
BEGIN
  -- Sentencias
  UPDATE SALDOS_CUENTAS
        SET SALDO = new_saldo,
            FX_ACTUALIZACION = SYSDATE
  WHERE CO_CUENTA = cuenta;
END Actualiza_Saldo;
```

Una vez creado y compilado el procedimiento almacenado podemos ejecutarlo. Si el sistema nos indica que el procedimiento se ha creado con errores de compilación podemos ver estos errores de compilación con la orden **SHOW ERRORS** en SQL *Plus.

Existen dos formas de pasar argumentos a un procedimiento almacenado a la hora de ejecutarlo (en realidad es válido para cualquier subprograma). Estas son:

- **Notación posicional**: Se pasan los valores de los parámetros en el mismo orden en que el procedimiento los define.

```
BEGIN
        Actualiza_Saldo(200501,2500);
        COMMIT;
END;
```

- **Notación nominal**:Se pasan los valores en cualquier orden nombrando explícitamente el parámetro.

```
BEGIN
        Actualiza_Saldo(cuenta => 200501,new_saldo  => 2500);
        COMMIT;
```

```
    END;
```

7. Funciones en PL/SQL.

Una función es un subprograma que devuelve un valor. La sintaxis para construir funciones es la siguiente:

```
CREATE [OR REPLACE]
FUNCTION <fn_name>[(<param1> IN <type>, <param2> IN <type>,
...)] RETURN <return_type>
IS
    result <return_type>;
BEGIN
 return(result);
 [EXCEPTION]
-- Sentencias control de excepcion
END [<fn_name>];
```

El uso de OR REPLACE permite sobreescribir una función existente. Si se omite, y la función existe, se producirá, un error.

La sintaxis de los parámetros es la misma que en los procedimientos almacenado, exceptuando que solo pueden ser de entrada.

Ejemplo:

```
CREATE OR REPLACE
    FUNCTION fn_Obtener_Precio(p_producto VARCHAR2)
    RETURN NUMBER
    IS
      result NUMBER;
    BEGIN
      SELECT PRECIO INTO result
      FROM PRECIOS_PRODUCTOS
      WHERE CO_PRODUCTO = p_producto;
      return(result);
    EXCEPTION
    WHEN NO_DATA_FOUND THEN
      return 0;
    END ;
```

Si el sistema nos indica que el la función se ha creado con errores de compilación podemos ver estos errores de compilación con la orden **SHOW ERRORS** en SQL *Plus.

Una vez creada y compilada la función podemos ejecutarla de la siguiente forma:

```
    DECLARE
      Valor NUMBER;
    BEGIN
            Valor := fn_Obtener_Precio('000100');
    END;
```

Las funciones pueden utilizarse en sentencias SQL de manipulación de datos (SELECT, UPDATE, INSERT y DELETE):

```
    SELECT CO_PRODUCTO,
           DESCRIPCION,
           fn_Obtener_Precio(CO_PRODUCTO)
    FROM PRODUCTOS;
```

8. Triggers (Disparadores).

8.1. Declaración de triggers (Disparadores).

Un trigger es un bloque PL/SQL asociado a una tabla, que se ejecuta como consecuencia de una determinada instrucción SQL (una operación DML: INSERT, UPDATE o DELETE) sobre dicha tabla.

La sintaxis para crear un trigger es la siguiente:

```
CREATE [OR REPLACE] TRIGGER <nombre_trigger>
{BEFORE|AFTER}
                    {DELETE|INSERT|UPDATE [OF col1, col2, ...,
colN]
                [OR {DELETE|INSERT|UPDATE [OF col1, col2, ...,
colN]...]}
ON <nombre_tabla>
[FOR EACH ROW [WHEN (<condicion>)]]
DECLARE
  -- variables locales
BEGIN
  -- Sentencias
[EXCEPTION]
  -- Sentencias control de excepción
END <nombre_trigger>;
```

El uso de OR REPLACE permite sobreescribir un trigger existente. Si se omite, y el trigger existe, se producirá, un error.

Los triggers pueden definirse para las operaciones INSERT, UPDATE o DELETE, y pueden ejecutarse antes o después de la operación. El modificador BEFORE AFTER indica que el trigger se ejecutará antes o después de ejecutarse la sentencia SQL definida por DELETE, INSERT, ó UPDATE. Si incluimos el modificador OF el trigger solo se ejecutará cuando la sentencia SQL afecte a los campos incluidos en la lista.

El alcance de los disparadores puede ser la fila o de orden. El modificador FOR EACH ROW indica que el trigger se disparará cada vez que se realizan operaciones sobre una fila de la tabla. Si se acompaña del modificador WHEN, se establece una restricción; el trigger solo actuará, sobre las filas que satisfagan la restricción.

La siguiente tabla resume los contenidos anteriores.

Valor	Descripción
INSERT, DELETE, UPDATE	Define qué tipo de orden DML provoca la activación del disparador.
BEFORE , AFTER	Define si el disparador se activa antes o después de que se ejecute la orden.
FOR EACH ROW	Los disparadores con nivel de fila se activan una vez por cada fila afectada por la orden que provocó el disparo. Los disparadores con nivel de orden se activan sólo una vez, antes o después de la orden. Los disparadores con nivel de fila se identifican por la cláusula FOR EACH ROW en la definición del disparador.

La cláusula WHEN sólo es válida para los disparadores con nivel de fila.

Dentro del ámbito de un trigger disponemos de las variables OLD y NEW. Estas variables se utilizan del mismo modo que cualquier otra variable PL/SQL, con la salvedad de que no es necesario declararlas, son de tipo **%ROWTYPE** y contienen una copia del registro antes (OLD) y después(NEW) de la acción SQL (INSERT, UPDATE, DELETE) que ha ejecutado el trigger. Utilizando esta variable podemos acceder a los datos que se están insertando, actualizando o borrando.

El siguiente ejemplo muestra un trigger que inserta un registro en la tabla

PRECIOS_PRODUCTOS cada vez que insertamos un nuevo registro en la tabla PRODUTOS:

```
CREATE OR REPLACE TRIGGER TR_PRODUCTOS_01
  AFTER INSERT ON PRODUCTOS
  FOR EACH ROW
DECLARE
  -- local variables
BEGIN
  INSERT INTO PRECIOS_PRODUCTOS
  (CO_PRODUCTO,PRECIO,FX_ACTUALIZACION)
  VALUES
  (:NEW.CO_PRODUCTO,100,SYSDATE);
END ;
```

El trigger se ejecutará cuando sobre la tabla PRODUCTOS se ejecute una sentencia INSERT.

```
INSERT INTO PRODUCTOS
(CO_PRODUCTO, DESCRIPCION)
VALUES
('000100','PRODUCTO 000100');
```

8.2. Orden de ejecución de los triggers.

Una misma tabla puede tener varios triggers. En tal caso es necesario conocer el orden en el que se van a ejecutar.

Los disparadores se activan al ejecutarse la sentencia SQL.

- Si existe, se ejecuta el disparador de tipo BEFORE (disparador previo) con nivel de orden.
- Para cada fila a la que afecte la orden:

    - Se ejecuta si existe, el disparador de tipo BEFORE con nivel de fila.
    - Se ejecuta la propia orden.
    - Se ejecuta si existe, el disparador de tipo AFTER (disparador posterior) con nivel de fila.
- Se ejecuta, si existe, el disparador de tipo AFTER con nivel de orden.

8.3. Restricciones de los triggers.

El cuerpo de un trigger es un bloque PL/SQL. Cualquier orden que sea legal en un bloque PL/SQL, es legal en el cuerpo de un disparador, con las siguientes restricciones:

- Un disparador no puede emitir ninguna orden de control de transacciones: **COMMIT**, **ROLLBACK** o **SAVEPOINT**. El disparador se activa como parte de la ejecución de la orden que provocó el disparo, y forma parte de la misma transacción que dicha orden. Cuando la orden que provoca el disparo es confirmada o cancelada, se confirma o cancela también el trabajo realizado por el disparador.

- Por razones idénticas, ningún procedimiento o función llamado por el disparador puede emitir órdenes de control de transacciones.

- El cuerpo del disparador no puede contener ninguna declaración de variables LONG o LONG RAW

8.4. Utilización de :OLD y :NEW.

Dentro del ámbito de un trigger disponemos de las variables OLD y NEW . Estas variables se utilizan del mismo modo que cualquier otra variable PL/SQL, con la salvedad de que no es necesario declararlas, son de tipo **%ROWTYPE** y contienen una copia del registro antes (OLD) y después(NEW) de la acción SQL (INSERT, UPDATE, DELTE) que ha ejecutado el trigger. Utilizando esta variable podemos acceder a los datos que se están insertando, actualizando o

borrando.

La siguiente tabla muestra los valores de OLD y NEW.

ACCION SQL	OLD	NEW
INSERT	No definido; todos los campos toman valor NULL.	Valores que serán insertados cuando se complete la orden.
UPDATE	Valores originales de la fila, antes de la actualización.	Nuevos valores que serán escritos cuando se complete la orden.
DELETE	Valores, antes del borrado de la fila.	No definidos; todos los campos toman el valor NULL.

Los registros OLD y NEW son sólo válidos dentro de los disparadores con nivel de fila.

Podemos usar OLD y NEW como cualquier otra variable PL/SQL.

8.5. Utilización de predicados de los triggers: INSERTING, UPDATING y DELETING.

Dentro de un disparador en el que se disparan distintos tipos de órdenes DML (INSERT, UPDATE y DELETE), hay tres funciones booleanas que pueden emplearse para determinar de qué operación se trata. Estos predicados son INSERTING, UPDATING y DELETING.

Su comportamiento es el siguiente:

Predicado	**Comportamiento**
INSERTING	TRUE si la orden de disparo es INSERT; FALSE en otro caso.
UPDATING	TRUE si la orden de disparo es UPDATE; FALSE en otro caso.
DELETING	TRUE si la orden de disparo es DELETE; FALSE en otro caso.

9. Subprogramas en bloques anónimos.

Dentro de la sección DECLARE de un bloque anónimo podemos declarar funciones y procedimientos almacenados y ejecutarlos desde el bloque de ejecución del script.

Este tipo de subprogramas son menos conocidos que los procedimientos almacenados, funciones y triggers, pero son enormemente útiles.

El siguiente ejemplo declara y ejecuta utiliza una función (fn_multiplica_x2).

```
DECLARE
      idx NUMBER;

       FUNCTION fn_multiplica_x2(num NUMBER)
       RETURN NUMBER
       IS
         result NUMBER;
       BEGIN
         result := num *2;
         return result;
       END fn_multiplica_x2;
BEGIN
     FOR idx IN 1..10
     LOOP
       dbms_output.put_line
      ('Llamada a la funcion ... '||
TO_CHAR(fn_multiplica_x2(idx)));
     END LOOP;

END;
```

Nótese que se utiliza la función TO_CHAR para convertir el resultado de la función fn_multiplica_x2 (numérico) en alfanumérico y poder mostrar el resultado por pantalla.

10. Packages en PL/SQL.

Un paquete es una estructura que agrupa objetos de PL/SQL compilados (procedimientos, funciones, variables, tipos ...) en la base de datos. Esto nos permite agrupar la funcionalidad de los procesos en programas.

Lo primero que debemos tener en cuenta es que los paquetes están formados por dos partes: la **especificación** y el **cuerpo**. La especificación del un paquete y su cuerpo se crean por separado.

La especificación es la interfaz con las aplicaciones. En ella es posible declarar los tipos, variables, constantes, excepciones, cursores y subprogramas disponibles para su uso posterior desde fuera del paquete. En la especificación del paquete sólo se declaran los objetos (procedimientos, funciones, variables ...), no se implementa el código. Los objetos declarados en la especificación del paquete son accesibles desde fuera del paquete por otro script de PL/SQL o programa. Haciendo una analogía con el mundo de C, la especificación es como el archivo de cabecera de un programa en C.

Para crear la especificación de un paquete la sintaxis general es la siguiente:

```
CREATE [OR REPLACE] PACKAGE <pkgName>
IS

  -- Declaraciones de tipos y registros públicas
  {[TYPE <TypeName> IS <Datatype>;]}

  -- Declaraciones de variables y constantes publicas
  -- También podemos declarar cursores
  {[<ConstantName> CONSTANT <Datatype> := <valor>;]}
  {[<VariableName> <Datatype>;]}

   -- Declaraciones de procedimientos y funciones
públicas
  {[FUNCTION <FunctionName>(<Parameter> <Datatype>,...)
    RETURN <Datatype>;]}
   {[PROCEDURE <ProcedureName>(<Parameter> <Datatype>,
...);]}
```

```
END <pkgName>;
```

El cuerpo es la implementación del paquete. El cuerpo del paquete debe implementar lo que se declaró inicialmente en la especificación. Es el donde debemos escribir el código de los subprogramas. En el cuerpo de un package podemos declarar nuevos subprogramas y tipos, pero estos serán privados para el propio package.

La sintaxis general para crear el cuerpo de un paquete es muy parecida a la de la especificación, tan solo se añade la palabra clave **BODY**, y se implementa el código de los subprogramas.

```
CREATE [OR REPLACE] PACKAGE BODY <pkgName>
IS

  -- Declaraciones de tipos y registros privados
  {[TYPE <TypeName> IS <Datatype>;]}

  -- Declaraciones de variables y constantes privadas
  -- También podemos declarar cursores
  {[<ConstantName> CONSTANT <Datatype> := <valor>;]}
  {[<VariableName> <Datatype>;]}

  -- Implementacion de procedimientos y funciones
  FUNCTION <FunctionName>(<Parameter> <Datatype>,...)
  RETURN <Datatype>
  IS
    -- Variables locales de la funcion
  BEGIN
    -- Implementeacion de la funcion
    return(<Result>);
  [EXCEPTION]
    -- Control de excepciones
  END;

   PROCEDURE  <ProcedureName>(<Parameter>  <Datatype>,
...)
  IS
   -- Variables locales de la funcion
  BEGIN
    -- Implementacion de procedimiento
  [EXCEPTION]
```

```
    -- Control de excepciones
  END;

END <pkgName>;
```

El siguiente ejemplo crea un paquete llamado *PKG_CONTABILIDAD*.

Para crear la especificación del paquete:

```
CREATE OR REPLACE PACKAGE PKG_CONTABILIDAD
IS
     -- Declaraciones de tipos y registros públicas
  TYPE Cuenta_contable IS RECORD
  (
    codigo_cuenta VARCHAR2(6),
    naturaleza    VARCHAR2(2) ,
    actividad     VARCHAR2(4) ,
    debe_haber    VARCHAR2(1)
  );
    -- Declaraciones de variables y constantes publicas
  DEBE  CONSTANT VARCHAR2(1) := 'D';
  HABER CONSTANT VARCHAR2(1) := 'D';
   ERROR_CONTABILIZAR  EXCEPTION;   -- Declaracines de
procedimientos y funciones públicas

  PROCEDURE Contabilizar (mes VARCHAR2) ;
   FUNCTION   fn_Obtener_Saldo(codigo_cuenta  VARCHAR2)
RETURN NUMBER;

END PKG_CONTABILIDAD;
```

Aquí sólo hemos declarado las variables y constantes, prototipado las funciones y procedimientos públicos . Es en el cuerpo del paquete cuando escribimos el código de los subprogramas *Contabilizar* y *fn_Obtener_Saldo*.

```
CREATE PACKAGE BODY PKG_CONTABILIDAD IS

 FUNCTION fn_Obtener_Saldo(codigo_cuenta VARCHAR2)

 RETURN NUMBER
 IS
   saldo NUMBER;
 BEGIN
        SELECT SALDO INTO saldo
        FROM SALDOS
        WHERE CO_CUENTA = codigo_cuenta;
        return (saldo);
 END;
  PROCEDURE Contabilizar(mes VARCHAR2)
 IS
   CURSOR cDatos(vmes VARCHAR2)
   IS
   SELECT *
   FROM FACTURACION
   WHERE FX_FACTURACION = vmes
    AND PENDIENTE_CONTABILIZAR = 'S';
   fila cDatos%ROWTYPE;
 BEGIN
   OPEN cDatos(mes);
     LOOP FETCH cDatos INTO fila;
     EXIT WHEN cDatos%NOTFOUND;
     /* Procesamiento de los registros recuperados */
    END LOOP;
  CLOSE cDatos;
 EXCEPTION
 WHEN OTHERS THEN
   RAISE ERROR_CONTABILIZAR;
 END Contabilizar;

END PKG_CONTABILIDAD;
```

Es posible modificar el cuerpo de un paquete sin necesidad de alterar por ello la especificación del mismo.

Los paquetes pueden llegar a ser programas muy complejos y suelen almacenar gran parte de la lógica de negocio.

10. Registros PL/SQL.

Cuando vimos los tipos de datos, omitimos intencionadamente ciertos tipos de datos.

Estos son:

- Registros
- Tablas de PL
- Varray

10.1. Declaración de un registro.

Un registro es una estructura de datos en PL/SQL, almacenados en campos, cada uno de los cuales tiene su propio nombre y tipo y que se tratan como una sola unidad lógica.

Los campos de un registro pueden ser inicializados y pueden ser definidos como NOT NULL. Aquellos campos que no sean inicializados explícitamente, se inicializarán a NULL.

La sintaxis general es la siguiente:

```
TYPE <nombre> IS RECORD
(
campo <tipo_datos> [NULL | NOT NULL]
[,<tipo_datos>...]
);
```

El siguiente ejemplo crea un tipo PAIS, que tiene como campos el código, el nombre y el continente.

```
TYPE PAIS IS RECORD
(
CO_PAIS     NUMBER  ,
DESCRIPCION VARCHAR2(50),
CONTINENTE  VARCHAR2(20)
```

```
);
```

Los registros son un tipo de datos, por lo que podremos declarar variables de dicho tipo de datos.

```
DECLARE

  TYPE PAIS IS RECORD
  (
    CO_PAIS     NUMBER  ,
    DESCRIPCION VARCHAR2(50),
    CONTINENTE  VARCHAR2(20)
  );

/* Declara una variable identificada por miPAIS de tipo PAIS
   Esto significa que la variable miPAIS tendrá los campos
   ID, DESCRIPCION y CONTINENTE.
 */
  miPAIS PAIS;
BEGIN
/* Asignamos valores a los campos de la variable.
 */
  miPAIS.CO_PAIS := 27;
  miPAIS.DESCRIPCION := 'ITALIA';
  miPAIS.CONTINENTE  := 'EUROPA';
END;
```

Los registros pueden estar anidados. Es decir, un campo de un registro puede ser de un tipo de dato de otro registro.

```
DECLARE
 TYPE PAIS IS RECORD
 (CO_PAIS     NUMBER  ,
  DESCRIPCION VARCHAR2(50),
  CONTINENTE  VARCHAR2(20)
 );

 TYPE MONEDA IS RECORD
 ( DESCRIPCION VARCHAR2(50),
   PAIS_MONEDA PAIS );
```

```
 miPAIS   PAIS;
 miMONEDA MONEDA;
BEGIN
     /* Sentencias
      */
END;
```

Pueden asignarse todos los campos de un registro utilizando una sentencia SELECT.

En este caso hay que tener cuidado en especificar las columnas en el orden conveniente según la declaración de los campos del registro. Para este tipo de asignación es muy frecuente el uso del atributo %ROWTYPE que veremos más adelante.

```
SELECT  CO_PAIS, DESCRIPCION, CONTINENTE
INTO miPAIS
FROM PAISES
WHERE CO_PAIS = 27;
```

Puede asignarse un registro a otro cuando sean del mismo tipo:

```
DECLARE

  TYPE PAIS IS RECORD ...
  miPAIS PAIS;
  otroPAIS PAIS;
BEGIN
  miPAIS.CO_PAIS := 27;
  miPAIS.DESCRIPCION := 'ITALIA';
  miPAIS.CONTINENTE  := 'EUROPA';
  otroPAIS := miPAIS;
END;
```

10.1.1. Declaración de registros con el atributo %ROWTYPE.

Se puede declarar un registro basándose en una colección de columnas de una

tabla, vista o cursor de la base de datos mediante el atributo **%ROWTYPE**.

Por ejemplo, si tengo una tabla PAISES declarada como:

```
CREATE TABLE PAISES(
CO_PAIS          NUMBER,
DESCRIPCION      VARCHAR2(50),
CONTINENTE       VARCHAR2(20) );
```

Puedo declarar una variable de tipo registro como PAISES**%ROWTYPE**;

```
DECLARE
  miPAIS PAISES%ROWTYPE;
BEGIN
  /* Sentencias ... */
END;
```

Lo cual significa que el registro miPAIS tendrá la siguiente estructura: CO_PAIS NUMBER, DESCRIPCION VARCHAR2(50), CONTINENTE VARCHAR2(20).

De esta forma se crea el registro de forma dinamic y se podrán asignar valores a los campos de un registro a través de un select sobre la tabla, vista o cursor a partir de la cual se creo el registro.

10.2. Tablas PL/SQL.

10.2.1. Declaración de tablas de PL/SQL.

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.

La sintaxis general para declarar una tabla de PL es la siguiente:

```
TYPE <nombre_tipo_tabla> IS TABLE OF
<tipo_datos> [NOT NULL]
INDEX BY BINARY_INTEGER ;
```

Una vez que hemos definido el tipo, podemos declarar variables y asignarle valores.

```
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;
```

No es posible inicializar las tablas en la inicialización.

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.

10.2.2. 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;
```

10.2.3. 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;
```

- **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;
```

- **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;
```

- **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;
```

- **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 es:' || misCiudades.NEXT(1));
END;
```

- **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.

10.3. VARRAYS

10.3.1. Definición de VARRAYS.

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('Aitor',  'Alicia',
'Pedro','','');
BEGIN
```

```
        v_lista(4) := 'Tita';
        v_lista(5) := 'Ainhoa';
    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 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.

10.3.2. Varrays en la base de datos.

Los VARRAYS pueden almacenarse en las columnas de la base de datos. Sin embargo, un varray sólo puede manipularse en su integridad, no pudiendo modificarse sus elementos individuales de un varray.

Para poder crear tablas con campos de tipo VARRAY debemos crear el VARRAY como un objeto de la base de datos.

La sintaxis general es:

```
CREATE [OR REPLACE]
TYPE <nombre_tipo> IS VARRAY (<tamaño_maximo>) OF <tipo_elementos>;
```

Una vez que hayamos creado el tipo sobre la base de datos, podremos utilizarlo como un tipo de datos más en la creación de tablas, declaración de variables ....

Véase el siguiente ejemplo:

```
CREATE OR REPLACE TYPE PACK_PRODUCTOS AS VARRAY(10) OF
VARCHAR2(60);
CREATE TABLE OFERTAS
(
CO_OFERTA NUMBER,
PRODUCTOS PACK_PRODUCTOS,
PRECION   NUMBER
);
```

Para modificar un varray almacenado, primero hay que seleccionarlo en una variable PL/SQL. Luego se modifica la variable y se vuelve a almacenar en la tabla.

La utilización de VARRAYS en la base de datos está completamente desaconsejada.

10.4. BULK COLLECT.

PL/SQL nos permite leer varios registros en una tabla de PL con un único acceso a través de la instrucción **BULK COLLECT**.

Esto nos permitirá reducir el número de accesos a disco, por lo que optimizaremos el rendimiento de nuestras aplicaciones. Como contrapartida el consumo de memoria será mayor.

```
DECLARE
  TYPE t_descripcion IS TABLE OF PAISES.DESCRIPCION%TYPE;
  TYPE t_continente  IS TABLE OF PAISES.CONTINENTE%TYPE;

  v_descripcion t_descripcion;
  v_continente  t_continente;

BEGIN
  SELECT DESCRIPCION,
         CONTINENTE
  BULK COLLECT INTO v_descripcion, v_continente
  FROM PAISES;

  FOR i IN v_descripcion.FIRST .. v_descripcion.LAST LOOP
    dbms_output.put_line(v_descripcion(i) || ', ' ||
v_continente(i));
  END LOOP;

END;
/
```

Podemos utilizar **BULK COLLECT** con registros de PL.

```
DECLARE
  TYPE PAIS IS RECORD (CO_PAIS     NUMBER  ,
                       DESCRIPCION VARCHAR2(50),
                       CONTINENTE  VARCHAR2(20));
  TYPE t_paises IS TABLE OF PAIS;

  v_paises t_paises;

BEGIN
  SELECT  CO_PAIS, DESCRIPCION, CONTINENTE
  BULK COLLECT INTO v_paises
  FROM PAISES;

  FOR i IN v_paises.FIRST .. v_paises.LAST LOOP
    dbms_output.put_line(v_paises(i).DESCRIPCION ||
```

```
                                                ', ' ||
v_paises(i).CONTINENTE);
   END LOOP;

 END;
```

También podemos utilizar el atributo **ROWTYPE**.

```
 DECLARE

   TYPE t_paises IS TABLE OF PAISES%ROWTYPE;

   v_paises t_paises;

 BEGIN
   SELECT  CO_PAIS, DESCRIPCION, CONTINENTE
   BULK COLLECT INTO v_paises
   FROM PAISES;

   FOR i IN v_paises.FIRST .. v_paises.LAST LOOP
     dbms_output.put_line(v_paises(i).DESCRIPCION ||
                                                ', ' ||
v_paises(i).CONTINENTE);
   END LOOP;

 END;
 /
```

11. Transacciones.

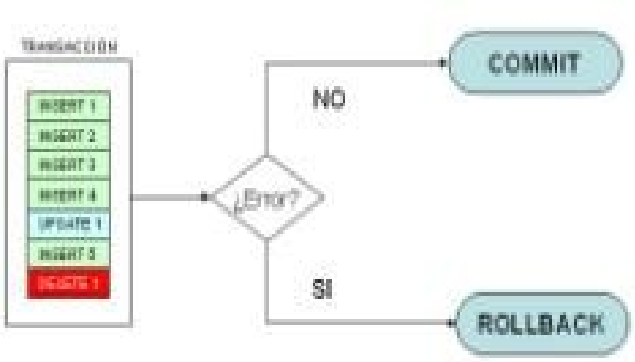
Una transacción es un conjunto de operaciones que se ejecutan en una base de datos, y que son tratadas como una única unidad lógica por el SGBD.

Es decir, una transacción es una o varias sentencias SQL que se ejecutan en una base de datos como una única operación, confirmándose o deshaciéndose en grupo.

No todas las operaciones SQL son transaccionales. Sólo 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.



Un ejemplo clásico de transacción son las transferencias bancarias. Para realizar una transferencia de dinero entre dos cuentas bancarias debemos descontar el dinero de una cuenta, realizar el ingreso en la otra cuenta y grabar las operaciones y movimientos necesarios, actualizar los saldos ... .Si en alguno de estos puntos se produce un fallo en el sistema podríamos hacer descontado el dinero de una de las cuentas y no haberlo ingresado en la otra. Por lo tanto, todas estas operaciones deben ser correctas o fallar todas. En estos casos, al confirmar la transacción (COMMIT) o al deshacerla (ROLLBACK) garantizamos que todos los datos quedan en un estado consistente.

En una transacción los datos modificados no son visibles por el resto de usuarios hasta que se confirme la transacción.

El siguiente ejemplo muestra una supuesta transacción bancaria:

```
DECLARE
   importe NUMBER;
   ctaOrigen VARCHAR2(23);
   ctaDestino VARCHAR2(23);
BEGIN
     importe := 100;
     ctaOrigen  := '2530 10 2000 1234567890';
     ctaDestino := '2532 10 2010 0987654321';
     UPDATE CUENTAS SET SALDO = SALDO - importe
     WHERE CUENTA = ctaOrigen;
     UPDATE CUENTAS SET SALDO = SALDO + importe
     WHERE CUENTA = ctaDestino;
     INSERT INTO MOVIMIENTOS
            (CUENTA_ORIGEN,        CUENTA_DESTINO,IMPORTE,
FECHA_MOVIMIENTO)
     VALUES
     (ctaOrigen, ctaDestino, importe*(-1), SYSDATE);
     INSERT INTO MOVIMIENTOS
            (CUENTA_ORIGEN,        CUENTA_DESTINO,IMPORTE,
FECHA_MOVIMIENTO)
     VALUES
     (ctaDestino,ctaOrigen, importe, SYSDATE);
     COMMIT;
EXCEPTION
WHEN OTHERS THEN
      dbms_output.put_line('Error  en  la  transaccion:'||
SQLERRM);
          dbms_output.put_line('Se      deshacen      las
modificaciones);
     ROLLBACK;
END;
```

Si alguna de las tablas afectadas por la transacción tiene triggers, las operaciones que realiza el trigger están dentro del ámbito de la transacción, y son confirmadas o deshechas conjuntamente con la transacción.

Durante la ejecución de una transacción, una segunda transacción no podrá ver los cambios realizados por la primera transacción hasta que esta se confirme.

ORACLE es completamente transaccional. Siempre debemos especificar si que

queremos deshacer o confirmar la transacción.

11.1. Transacciones autónomas.

En ocasiones es necesario que los datos escritos por parte de una transacción sean persistentes a pesar de que la transacción se deshaga con **ROLLBACK**.

PL/SQL permite marcar un bloque con

PRAGMA AUTONOMOUS_TRANSACTION.

Con esta directiva marcamos el subprograma para que se comporte como transacción diferente a la del proceso principal, llevando el control de **COMMIT** o **ROLLBACK** independiente**.**

Obsérvese el siguiente ejemplo. Primero creamos un procedimiento y lo marcamos con **PRAGMA AUTONOMOUS_TRANSACTION.**

```
CREATE OR REPLACE PROCEDURE Grabar_Log(descripcion VARCHAR2)
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO LOG_APLICACION
  (CO_ERROR, DESCRIPICION, FX_ERROR)
  VALUES
  (SQ_ERROR.NEXTVAL, descripcion, SYSDATE);

  COMMIT; -- Este commit solo afecta a la transaccion autonoma
END ;
```

A continuación utilizamos el procedimiento desde un bloque de PL/SQL:

```
DECLARE
  producto PRECIOS%TYPE;
BEGIN
     producto := '100599';
     INSERT INTO PRECIOS
     (CO_PRODUCTO, PRECIO, FX_ALTA)
     VALUES
     (producto, 150, SYSDATE);
     COMMIT;
EXCEPTION
WHEN OTHERS THEN
    Grabar_Log(SQLERRM);
    ROLLBACK;
    /* Los datos grabados por "Grabar_Log" se escriben en la base
       de datos a pesar del ROLLBACK, ya que el procedimiento está
       marcado como transacción autonoma.
     */
END;
```

Es muy común que, por ejemplo, en caso de que se produzca algún tipo de error queramos insertar un registro en una tabla de log con el error que se ha producido y hacer ROLLBACK de la transacción. Pero si hacemos ROLLBACK de la transacción también lo hacemos de la inserción del log.

12. SQL Dinámico.

12.1. Sentencias DML con SQL dinámico.

PL/SQL ofrece la posibilidad de ejecutar sentencias SQL a partir de cadenas de caracteres. Para ello debemos emplear la instrucción **EXECUTE IMMEDIATE**.

Podemos obtener información acerca de número de filas afectadas por la instrucción ejecutada por **EXEXUTE IMMEDIATE** utilizando **SQL%ROWCOUNT**.

El siguiente ejemplo muestra la ejecución de un comando SQL dinámico.

```
DECLARE
  ret NUMBER;
  FUNCTION fn_execute RETURN NUMBER IS
    sql_str VARCHAR2(1000);
  BEGIN
    sql_str := 'UPDATE DATOS SET NOMBRE = ''NUEVO NOMBRE''
                WHERE CODIGO = 1';
    EXECUTE IMMEDIATE sql_str;
    RETURN SQL%ROWCOUNT;
  END fn_execute ;

BEGIN
     ret := fn_execute();
     dbms_output.put_line(TO_CHAR(ret));
END;
```

Podemos además parametrizar nuestras consultas a través de variables host. Una variable host es una variable que pertenece al programa que está ejecutando la sentencia SQL dinámica y que podemos asignar en el interior de la sentencia SQL con la palabra clave **USING** . Las variables host van precedidas de dos puntos ":".

El siguiente ejemplo muestra el uso de variables host para parametrizar una sentencia SQL dinámica.

```
DECLARE
  ret NUMBER;
  FUNCTION fn_execute (nombre VARCHAR2, codigo NUMBER)
RETURN NUMBER
  IS
    sql_str VARCHAR2(1000);
  BEGIN
    sql_str := 'UPDATE DATOS SET NOMBRE = :new_nombre
                WHERE CODIGO = :codigo';
    EXECUTE IMMEDIATE sql_str USING nombre, codigo;
    RETURN SQL%ROWCOUNT;
  END fn_execute ;

BEGIN
     ret := fn_execute('Devjoker',1);
     dbms_output.put_line(TO_CHAR(ret));
END;
```

12.2. Cursores con SQL dinámico.

Con SQL dinámico también podemos utilizar cursores.

Para utilizar un cursor implícito solo debemos construir nuestra sentencia SELECT en una variable de tipo caracter y ejecutarla con EXECUTE IMMEDIATE utilizando la palabra clave INTO.

```
DECLARE
       str_sql VARCHAR2(255);
       l_cnt   VARCHAR2(20);
BEGIN
     str_sql := 'SELECT count(*) FROM PAISES';
     EXECUTE IMMEDIATE str_sql INTO l_cnt;
     dbms_output.put_line(l_cnt);
END;
```

Trabajar con cursores explícitos es también muy fácil. Únicamente destacar el uso de **REF CURSOR** para declarar una variable para referirnos al cursor

generado con SQL dinámico.

```
DECLARE
  TYPE CUR_TYP IS REF CURSOR;
  c_cursor   CUR_TYP;
  fila PAISES%ROWTYPE;
  v_query     VARCHAR2(255);
BEGIN
  v_query := 'SELECT * FROM PAISES';

  OPEN c_cursor FOR v_query;
  LOOP
    FETCH c_cursor INTO fila;
    EXIT WHEN c_cursor%NOTFOUND;
    dbms_output.put_line(fila.DESCRIPCION);
  END LOOP;
  CLOSE c_cursor;
END;
```

Las varibles host también se pueden utilizar en los cursores.

```
DECLARE
  TYPE cur_typ IS REF CURSOR;
  c_cursor    CUR_TYP;
  fila PAISES%ROWTYPE;
  v_query     VARCHAR2(255);
  codigo_pais VARCHAR2(3) := 'ESP';
BEGIN
  v_query := 'SELECT * FROM PAISES WHERE CO_PAIS = :cpais';
  OPEN c_cursor FOR v_query USING codigo_pais;
  LOOP
    FETCH c_cursor INTO fila;
    EXIT WHEN c_cursor%NOTFOUND;
    dbms_output.put_line(fila.DESCRIPCION);
  END LOOP;
  CLOSE c_cursor;
END;
```

13. Funciones integradas de PL/SQL.

PL/SQL tiene un gran número de funciones incorporadas, sumamente útiles. A continuación vamos a ver algunas de las más utilizadas.

SYSDATE

Devuelve la fecha del sistema:

```
SELECT SYSDATE FROM DUAL;
```

NVL

Devuelve el valor recibido como parámetro en el caso de que expresión sea NULL,o expresión en caso contrario.

```
NVL(<expresion>, <valor>)
```

El siguiente ejemplo devuelve 0 si el precio es nulo, y el precio cuando está informado:

```
SELECT CO_PRODUCTO, NVL(PRECIO, 0) FROM PRECIOS;
```

DECODE

Decode proporciona la funcionalidad de una sentencia de control de flujo **if-elseif-else.**

```
DECODE(<expr>, <cond1>, <val1>[, ..., <condN>, <valN>], <default>)
```

Esta función evalúa una expresión "*<expr>*", si se cumple la primera condición

"<*cond1*>" devuelve el valor1 "<*val1*>", en caso contrario evalúa la siguiente condición y así hasta que una de las condiciones se cumpla. Si no se cumple ninguna condición se devuelve el valor por defecto.

Es muy común escribir la función DECODE indentada como si se tratase de un bloque IF.

```
SELECT DECODE (co_pais, /* Expresion a evaluar */
               'ESP', 'ESPAÑA', /* Si co_pais = 'ESP' ==>
'ESPAÑA' */
               'MEX', 'MEXICO', /* Si co_pais = 'MEX' ==>
'MEXICO' */
               'PAIS '||co_pais)/* ELSE ==> concatena */
FROM PAISES;
```

TO_DATE

Convierte una expresión al tipo fecha. El parámetro opcional formato indica el formato de entrada de la expresión no el de salida.

```
TO_DATE(<expresion>, [<formato>])
```

En este ejemplo convertimos la expresión '01/12/2006' de tipo CHAR a una fecha (tipo DATE). Con el parámetro formato le indicamos que la fecha está escrita como día-mes-año para que devuelve el uno de diciembre y no el doce de enero.

```
SELECT TO_DATE('01/12/2006',
               'DD/MM/YYYY')
FROM DUAL;
```

Este otro ejemplo muestra la conversión con formato de día y hora.

```
SELECT TO_DATE('31/12/2006 23:59:59',
               'DD/MM/YYYY HH24:MI:SS')
FROM DUAL;
```

TO_CHAR

Convierte una expresión al tipo CHAR. El parámetro opcional formato indica el formato de salida de la expresión.

```
TO_CHAR(<expresion>, [<formato>])
```

```
SELECT TO_CHAR(SYSDATE, 'DD/MM/YYYYY')
FROM DUAL;
```

TO_NUMBER

Convierte una expresión alfanumérica en numérica. Opcionalmente podemos especificar el formato de salida.

```
TO_NUMBER(<expresion>, [<formato>])
```

```
SELECT TO_NUMBER ('10')
FROM DUAL;
```

TRUNC

Trunca una fecha o número.

Si el parámetro recibido es una fecha elimina las horas, minutos y segundos de la misma.

```
SELECT TRUNC(SYSDATE)FROM DUAL;
```

Si el parámetro es un número devuelve la parte entera.

```
SELECT TRUNC(9.99)FROM DUAL;
```

LENGTH

Devuelve la longitud de un tipo CHAR.

```
SELECT LENGTH('HOLA MUNDO')FROM DUAL;
```

INSTR

Busca una cadena de caracteres dentro de otra. Devuelve la posición de la ocurrencia de la cadena buscada.

Su sintaxis es la siguiente:

```
INSTR(<char>, <search_string>, <startpos>, <occurrence> )
```

```
SELECT INSTR('AQUI ES DONDE SE BUSCA', 'BUSCA', 1, 1 )
FROM DUAL;
```

REPLACE

Reemplaza un texto por otro en un expresión de búsqueda.

```
REPLACE(<expresion>, <busqueda>, <reemplazo>)
```

El siguiente ejemplo reemplaza la palabra 'HOLA' por 'VAYA' en la cadena 'HOLA MUNDO'.

```
SELECT REPLACE ('HOLA MUNDO','HOLA', 'VAYA')-- devuelve VAYA
MUNDO
FROM DUAL;
```

SUBSTR

Obtiene una parte de una expresión, desde una posición de inicio hasta una determinada longitud.

```
SUBSTR(<expresion>, <posicion_ini>, <longitud> )
```

```
SELECT SUBSTR('HOLA MUNDO', 6, 5) -- Devuelve MUNDO
 FROM DUAL;
```

UPPER

Convierte una expresión alfanumérica a mayúsculas.

```
SELECT UPPER('hola mundo') -- Devuelve HOLA MUNDO
FROM DUAL;
```

LOWER

Convierte una expresión alfanumérica a minúsculas.

```
SELECT LOWER('HOLA MUNDO') -- Devuelve hola mundo
FROM DUAL;
```

ROWIDTOCHAR

Convierte un ROWID a tipo caracter.

```
SELECT ROWIDTOCHAR(ROWID)
FROM DUAL;
```

RPAD

Añade N veces una determinada cadena de caracteres a la derecha una expresión. Muy util para generar ficheros de texto de ancho fijo.

```
RPAD(<expresion>, <longitud>, <pad_string> )
```

```

```

El siguiente ejemplo añade puntos a la expresión 'Hola mundo' hasta alcanzar una longitud de 50 caracteres.

```
SELECT RPAD('Hola Mundo', 50, '.')
FROM DUAL;
```

LPAD

Añade N veces una determinada cadena de caracteres a la izquierda de una expresión. Muy útil para generar ficheros de texto de ancho fijo.

```
LPAD(<expresion>, <longitud>, <pad_string> )
```

El siguiente ejemplo añade puntos a la expresión 'Hola mundo' hasta alcanzar una longitud de 50 caracteres.

```
SELECT LPAD('Hola Mundo', 50, '.')
FROM DUAL;
```

RTRIM

Elimina los espacios en blanco a la derecha de una expresión

```
SELECT RTRIM ('Hola Mundo     ')
FROM DUAL;
```

LTRIM

Elimina los espacios en blanco a la izquierda de una expresión

```
SELECT LTRIM ('       Hola Mundo')
FROM DUAL;
```

TRIM

Elimina los espacios en blanco a la izquierda y derecha de una expresión

```
SELECT TRIM ('       Hola Mundo         ')
FROM DUAL;
```

MOD

Devuelve el resto de la división entera entre dos números.

```
   MOD(<dividendo>, <divisor> )
```

```
SELECT MOD(20,15) -- Devuelve el modulo de dividir 20/15
FROM DUAL
```

14. Secuencias.

ORACLE proporciona los objetos de secuencia para la generación de códigos numéricos automáticos.

Las secuencias son una solución fácil y elegante al problema de los códigos auto generados.

La sintaxis general es la siguiente:

```
CREATE SEQUENCE <secuence_name>
[MINVALUE <min_val>]
[MAXVALUE <max_val>]
[START WITH <ini_val>]
[INCREMENT BY <inc_val>]
[NOCACHE | CACHE <cache_val>]
[CYCLE]
[ORDER];
```

El siguiente ejemplo crea una secuencia *SQ_PRODUCTOS.*

```
CREATE SEQUENCE SQ_PRODUCTOS
MINVALUE 1
MAXVALUE 999999999999999999999999999
START WITH 1
INCREMENT BY 1
CACHE 20;
```

Se puede simplificar la orden, tomando los valores por defecto. El ejemplo anterior quedaría del siguiente modo:

```
CREATE SEQUENCE SQ_PRODUCTOS;
```

Para obtener el siguiente valor de una secuencia debemos utilizar la función **NEXTVAL**. **NEXTVAL** se puede utilizar el cualquier sentencia SQL DML (SELECT, INSERT, UPDATE).

```
SELECT SQ_PRODUCTOS.NEXTVAL
FROM DUAL;
```

Podemos obtener el último valor generado por la secuencia con la función **CURRVAL**. Para poder ejecutar la función CURRVAL debemos haber ejecutado previamente la función **NEXTVAL**.

```
SELECT SQ_PRODUCTOS.CURRVAL
FROM DUAL;
```

Para eliminar una secuencia definitivamente de la base de datos debemos utilizar la sentencia **DROP**.

```
DROP SEQUENCE SQ_PRODUCTOS ;
```

15. PL/SQL y Java.

Otra de la virtudes de PL/SQL es que permite trabajar conjuntamente con Java.

PL/SQL es un excelente lenguaje para la gestion de información pero en ocasiones, podemos necesitar de un lenguaje de programación más potente. Por ejemplo podríamos necesitar consumir un servicio Web, conectar a otro servidor, trabajar con Sockets .... Para estos casos podemos trabajar conjuntamente con PL/SQL y Java.

Para poder trabajar con Java y PL/SQL debemos realizar los siguientes pasos:

- Crear el programa Java y cargarlo en la base de datos.
- Crear un programa de recubrimiento (Wrapper) de PL/SQL.

Creación de Objetos Java en la base de datos ORACLE.

ORACLE incorpora su propia versión de la máquina virtual Java y del JRE. Esta versión de Java se instala conjuntamente con **ORACLE**.

Para crear objetos Java en la base de datos podemos utilizar la utilidad LoadJava de **ORACLE** desde linea de comandos o bien crear objetos **JAVA SOURCE** en la propia base de datos.

La sintaxis para la creación de **JAVA SOURCE** en **ORACLE** es la siguiente.

```
CREATE [OR REPLACE] AND COMPILE
JAVA SOURCE
NAMED <JavaSourceName>
AS
public class <className>
{
        <java code>
        ...
};
```

El siguiente ejemplo crea y compila una clase Java OracleJavaClass en el interior de **JAVA SOURCE (**FuentesJava). **Un aspecto importante a tener en cuenta es que los métodos de la clase java que queramos invocar desde PL/SQL**

deben ser estáticos.

```
CREATE OR REPLACE AND COMPILE
JAVA SOURCE
NAMED FuentesJava
AS
public class OracleJavaClass
{
  public static String Saluda(String nombre)
  {
        return ("Hola desde Java" + nombre);
  }
}
;
```

Un mismo **JAVA SOURCE** puede contener varias clases de Java.

```
CREATE OR REPLACE AND COMPILE
JAVA SOURCE
NAMED FuentesJava
AS
public class OracleJavaClass
{
  public static String Saluda(String nombre)
  {
        return ("Hola desde Java" + nombre);
  }
}

public class OracleJavaMejorada
{
  public static String SaludoMejorado(String nombre)
  {
         return ("Saludo mejorado desde Java para:" +
nombre);
  }
}
;
```

La otra opción sería guardar nuestro codigo java en el archivo

OracleJavaClass.java, compilarlo y cargarlo en **ORACLE** con LoadJava.

A continuación se muestran ejemplos del uso de la utilidad LoadJava

```
loadJava -help

    loadJava -u usario/password -v -f -r OracleJavaClass.class

    loadJava -u usario/password -v -f -r OracleJavaClass.java
```

Ejecución de programas Java con PL/SQL

Una vez que tenemos listo el programa de Java debemos integrarlo con PL/SQL. Esto se realiza a través de subprogramas de recubrimiento llamados Wrappers.

No podemos crear un Wrapper en un bloque anónimo.

La sintaxis general es la siguiente:

```
CREATE [OR REPLACE]
FUNCTION|PROCEDURE <name> [(<params>,...)]
[RETURN <tipo>]
IS|AS
LANGUAGE JAVA NAME
'<clase>.<metodo> [return <tipo>]' ;
```

El siguiente ejemplo muestra el Wrapper para nuestra función Saludo.

```
CREATE OR REPLACE
FUNCTION Saludo_wrap (nombre VARCHAR2)
RETURN VARCHAR2
AS
LANGUAGE JAVA NAME
'OracleJavaClass.Saluda(java.lang.String)              return
java.lang.String';
```

Una vez creado el wrapper, podremos ejecutarlo como cualquier otra función o procedure de PL/SQL. Debemos crear un wrapper por cada función java que

queramos ejecutar desde PL/SQL.

Cuando ejecutemos el wrapper, es decir, la función "Saluda_wrap", internamente se ejecutará la clase java y se invocará el método estático "OracleJavaClass.Saluda".

Un aspecto a tener en cuenta es que es necesario proporcionar el nombre del tipo java completo, es decir, debemos especificar java.lang.String en lugar de únicamente String.

```
SELECT SALUDA_WRAP('DEVJOKER')
FROM DUAL;
```

La ejecución de este ejemplo en SQL*Plus genera la siguiente salida:

```
SQL> SELECT SALUDA_WRAP('DEVJOKER') FROM DUAL;

SALUDA_WRAP('DEVJOKER')
-----------------------------------------
Hola desde JavaDEVJOKER
```

Una recomendación de diseño sería agrupar todos los Wrapper en un mismo paquete.

En el caso de que nuestro programa Java necesitase de packages Java adicionales, deberíamos cargarlos en la base de datos con la utilidad LoadJava.