INTRODUCCION

- PL/SQL es la extensión estructurada y procedimental (permite crear funciones y procedimientos) del lenguaje SQL implementada por Oracle junto a la versión 6.

- Con los scripts de SQL se tienen limitaciones como uso de variables, modularidad,etc.

- Con PL/SQL se pueden usar sentencias SQL para acceder a bases de datos Oracle y sentencias de control de flujo para procesar los datos, se pueden declarar variables y constantes, definir procedimientos, funciones, subprogramas, capturar y tratar errores en tiempo de ejecución, etc...

- En un programa escrito en PL/SQL se pueden utilizar sentencias SQL de tipo LMD (Lenguaje de Manipulación de Datos) directamente y sentencias de tipo LDD (Lenguaje de Definición de Datos) mediante la utilización de paquetes.

- Oracle incorpora un gestor PL/SQL en el servidor de bbdd y en las principales herramientas, Forms , Reports, Graphics, etc.

- Basado en Ada incorpora todas las características propias de los lenguajes de tercera generación; manejo de variables, estructura modular (procedimientos y funciones) , estructuras de control, control de excepciones,etc

- El código PL/SQL puede estar almacenado en la base de datos (procedimientos, funciones, disparadores y paquetes) facilitando el acceso a todos los usuarios autorizados.

- La ejecución de los bloques PL/SQL puede realizarse interactivamente desde herramientas como SQL*Plus, Oracle Forms, etc..., o bien, cuando el S.G.B.D. detecte determinados eventos, también llamados disparadores.

- Nosotros vamos a utilizarlos en SQL Worksheet de Oracle Live

SQL para poder probar nuestro Pl/SQL

- Los programas se ejecutan en el servidor, con el consiguiente ahorro de recursos en los clientes y disminución de tráfico en la red.

ARQUITECTURA DE PL/SQL

La arquitectura de PL/SQL consiste principalmente en los tres componentes que se citan a continuación:

1. El bloque PL/SQL
2. El motor PL/SQL
3.El servidor de la base de datos

El bloque PL/SQL:

- Es el componente que contiene el código de pl/sql
- Consiste en distintas secciones que forman la lógica del bloque
- También contiene instrucciones de sql que interactuar con el servidor de base de datos
- Hay diferentes tipos de bloques o Units de Pl/sql y son :

1. Bloques anonimos
2. Procedimientos
3. Paquetes
4. Trigger

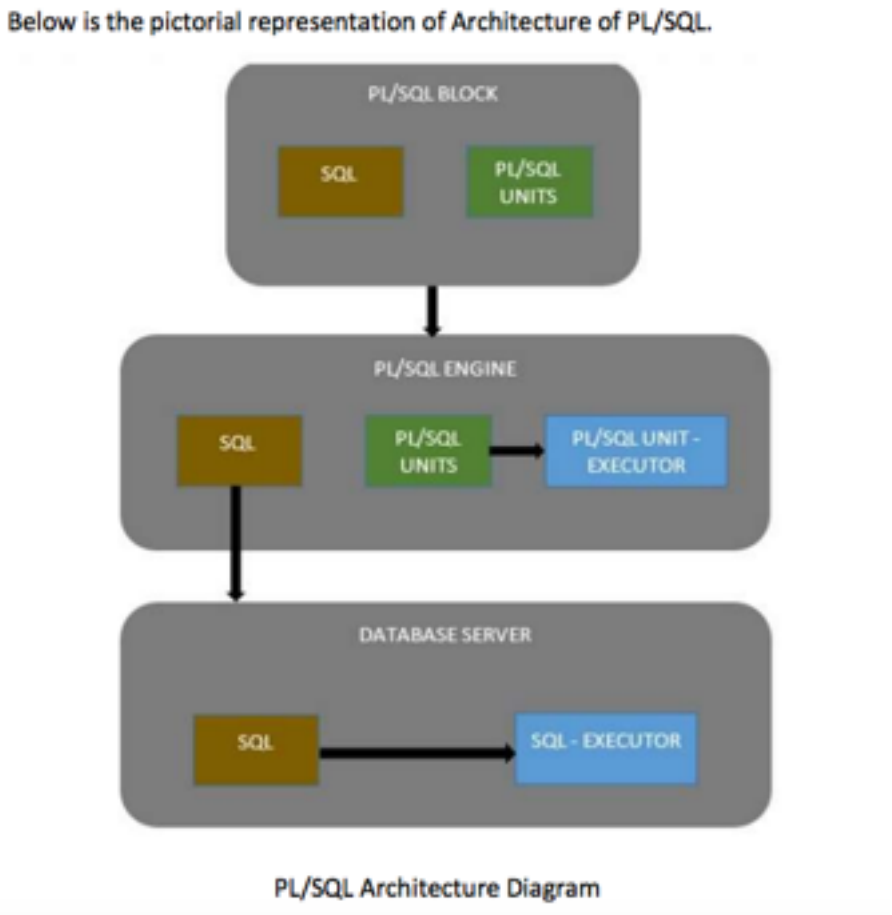
El motor PL/SQL

Es el componente donde se procesa el código
El motor PL/SQL separa las unidades de PL/SQL y las partes de SQL y este motor será el encargado de manejar los bloques PL/SQL

La parte de SQL será enviada al servidor donde interactua con la base de datos

El servidor de base de datos

Es la parte más importante pues es la que contiene los datos
El motor de The PL/SQL usa el SQL de los bloques PL/SQL para interactuar con el servidor de base de datos



Diferencias entre SQL y PL/SQL

sql	pl/sql
SQL es una consulta sencilla usada para desarrollar operaciones DML and DDL.	PL/SQL es un bloque de codigo usado para escribir programas enteros con procedimientos, bloques y funciones
Es declarativo, es decir, se dice que cosas queremos más que como se hacen	Es prodedimental, por tanto, decimos como se hacen las cosas.
Ejecuta una sentencia sencilla	Ejecuta todo un bloque
Interactua con el servidor de la base de datos	No interactua con les hervor de base de datos
No puede contener código PL/SQL	Contiene SQL en sus bloques

CARACTERÍSTICAS DEL LENGUAJE. BLOQUES

En PL/SQL, tel código no es ejecutado como una linea, siempre es ejecutado como un grupo de sentencia denominado BLOQUE.

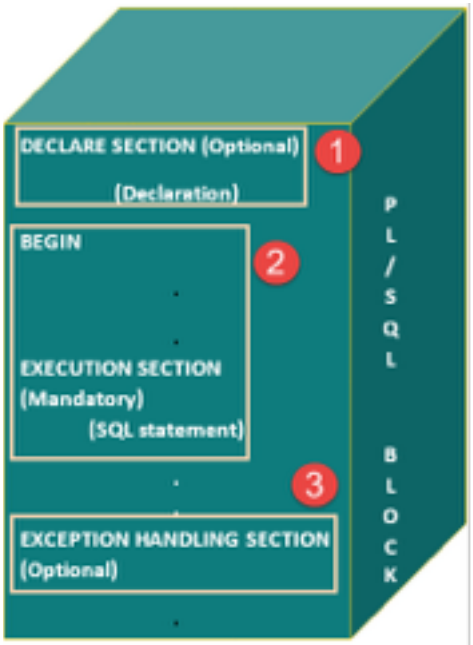
Los bloques contienen instrucciones tanto de PL/SQL como de SQL.
La estructura básica del lenguaje es el bloque.

Todos los programas se escriben en bloques de código que se dividen en diferentes secciones independientes.

Se pueden anidar bloques formando estructuras de programación más complejas

BLOQUES ANÓNIMOS: La estructura de un bloque sin nombre es la siguiente:

```
[DECLARE
      Declaraciones;]
BEGIN
      Sentencias;
  [EXCEPTION
      Excepciones;]
END;
```



- El bloque consta de tres secciones:

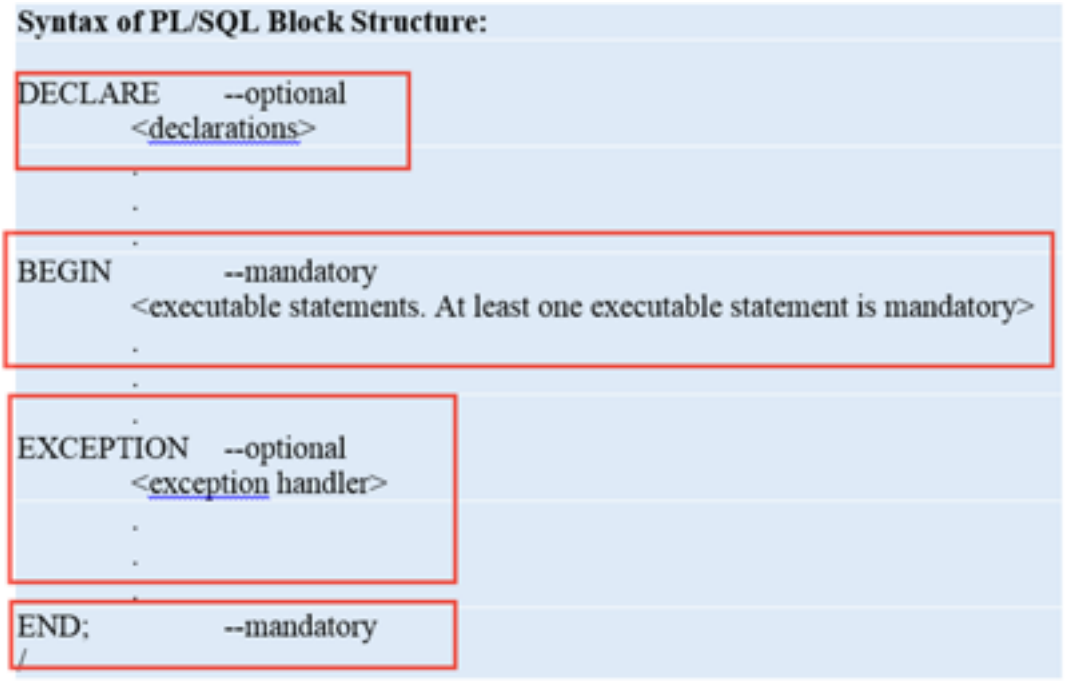
  DECLARE: Sección de declaración. En ella se declaran todos los objetos que vayamos a usar en el programa
  Es opcional, salvo cuando se deba realizar algún tipo de declaraciones de

elementos, tales como variables, constantes, cursores, etc…
Si se pone esta sección debe ser la primera.
Solo se utiliza en Bloques anónimos y en triggers pero los procedimientos no tienen esta parte Declare.

BEGIN:
Sección que contiene las sentencias que se van a ejecutar.
Es la única parte obligatoria , ya que es la parte ejecutable. Comienza con la palabra BEGIN y finaliza con END;
En esta sección es donde especificamos el código que se va a ejecutar

EXCEPTION:
Sección que contiene las sentencias para el manejo de errores.
También contiene sentencias PL/SQL .
Es opcional, salvo cuando tengamos la necesidad de tratar los errores.
Normalmente se utiliza para capturar errores ya predefinidos y hay que especificar las acciones a realizar en caso de que se produzca un error.

Por último se especifica END que indica final del bloque. Especifica final de la zona ejecutable aunque vaya después de la zona de excepciones cuando la incluya.



- Los bloques pueden contener sub-bloques, es decir, podemos tener bloques anidados. Las anidaciones se pueden realizar en la parte ejecutable y en la de manejo de excepciones, pero no en la declarativa.

```
[DECLARE
      Declaraciones;]
BEGIN

      [DECLARE
            Declaraciones;]
      BEGIN
            Sentencias:
             [EXCEPTION

            Excepciones;]

      END;

 [EXCEPTION

      Excepciones;]

END;
```

- A continuación tenemos un ejemplo de bloque PL/SQL:

```
DECLARE
sueldo NUMBER(8);
BEGIN
SELECT salario INTO sueldo
      FROM plantilla
      WHERE apellido LIKE 'MORENO';

IF sueldo < 100000 THEN
      sueldo := sueldo + 20000;
END IF;
UPDATE plantilla
      SET salario = sueldo WHERE apellido LIKE 'MORENO';
COMMIT;
END;
/
```

- Con este programa actualizamos el sueldo del empleado con apellido 'MORENO', incrementándolo en 20.000 en caso de que dicho sueldo no supere 100.000 pesetas.

- El programa tiene parte declarativa, en la que se define una variable, y parte ejecutable.

- Podemos observar que hay sentencias SQL, como SELECT y UPDATE, sentencias de control como IF... THEN y sentencias de asignación. El símbolo ':=' representa el operador de asignación.

- El bloque de código se ejecuta al encontrar el operador de ejecución '/'.

TIPOS DE BLOQUES

Se reconocen dos tipos de bloques en PL/SQL:
- Bloques anónimos: Bloques sin nombre
- Bloques nombrados: Que a su vez se subdividen en procedimientos y funciones.

Bloques anonimos:

Son bloques que no tienen nombre y necesitan construirse en una sesión para probar algún código.
Al no tener nombre no serán almacenados en la base de datos.

Son escritos y ejecutados directamente y se compilan y ejecutan en el mismo proceso.

Bloques nombrados:

Los bloques con nombre son almacenados como objetos en la base de datos.

Como son guardados en el servidor , podremos invocarlos y ejecutarlos cuando lo necesitemos.

La compilación se produce cuando los creamos y después se ejecutaran en caso de ser necesario.

Características de estos bloques:
- Pueden ser invocados por otros bloques
- Pueden tener otros bloques anidados como dijimos antes
- Se dividen en Procedimientos y Funciones, que se verán mas adelante.

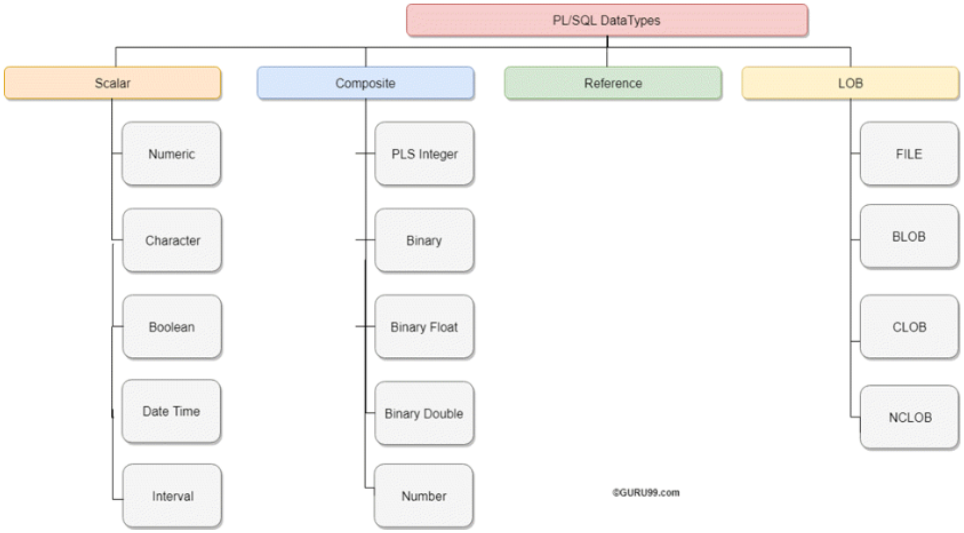
TIPOS DE DATOS

¿Qué son los tipos de datos PL/SQL?

Tipos de datos en PL/SQL se utilizan para definir cómo los datos serán almacenados, manejados y tratados por Oracle durante el almacenamiento y procesamiento de datos. Los tipos de datos están asociados con el formato de almacenamiento específico y las restricciones de rango. En Oracle, a cada valor o constante se le asigna un tipo de datos.

La principal diferencia entre PL/SQL y SQL tipos de datos es, el tipo de datos SQL se limita a la columna de la tabla, mientras que los tipos de datos PL/SQL se utilizan en la bloques PL/SQL

A continuación se muestra el diagrama de diferentes Oracle Tipos de datos PL/SQL:



Diferentes tipos de datos en PL/SQL

Tipo de datos de carácter PL/SQL

Este tipo de datos básicamente almacena caracteres alfanuméricos en formato de cadena.

Los valores literales siempre deben estar entre comillas simples al asignarlos al tipo de datos CHARACTER.

Este tipo de datos de carácter se clasifica además de la siguiente manera:

- CHAR Tipo de datos (tamaño de cadena fijo)
- VARCHAR2 Tipo de datos (tamaño de cadena variable)
- VARCHAR Tipo de datos
- NCHAR (tamaño de cadena fijo nativo)

Tipo de datos	Descripción	Sintaxis
CHAR	Este tipo de datos almacena el valor de la cadena y el tamaño de la cadena se fija en el momento de declarar el variable. • Oracle La variable se rellenaría en blanco si la variable no ocupara todo el tamaño que se ha declarado para ella, por lo tanto Oracle asignará la memoria para el tamaño declarado incluso si la variable no la ocupó por completo. • La restricción de tamaño para este tipo de datos es de 1 a 2000 bytes. • El tipo de datos CHAR es más apropiado para usar siempre que se maneje el tamaño de datos fijo.	grade CHAR; manager CHAR (10):= 'guru99'; **Explicación de sintaxis:** • La primera declaración declaró la variable "grado" del tipo de datos CHAR con el tamaño máximo de 1 byte (valor predeterminado). • La segunda declaración declaró la variable 'administrador' del tipo de datos CHAR con el tamaño máximo de 10 y asignó el valor 'guru99' que es de 6 bytes. Oracle asignará la memoria de 10 bytes en lugar de 6 bytes en este caso.
VARCHAR2	Este tipo de datos almacena la cadena, pero la longitud de la cadena no es fija. • La restricción de tamaño para este tipo de datos es de 1 a 4000 bytes para el tamaño de la columna de la tabla y de 1 a 32767 bytes para las variables. • El tamaño se define para cada variable en el momento de la declaración de la variable. • Pero Oracle asignará memoria sólo después de que se defina la variable, es decir, Oracle considerará solo la longitud real de la cadena que está almacenada en una variable para la asignación de memoria en lugar del tamaño que se ha dado para una variable en la parte de declaración. • Siempre es bueno utilizar VARCHAR2 en lugar del tipo de datos CHAR para optimizar el uso de la memoria.	manager VARCHAR2(10) := ‘guru99'; **Explicación de sintaxis:** • La declaración anterior declaró la variable 'administrador' del tipo de datos VARCHAR2 con el tamaño máximo de 10 y le asignó el valor 'guru99' que es de 6 bytes. Oracle asignará memoria de sólo 6 bytes en este caso.
VARCHAR	Esto es sinónimo del tipo de datos VARCHAR2.	manager VARCHAR(10) := ‘guru99'; **Explicación de sintaxis:**

Tipo de datos	Descripción	Sintaxis
	• Siempre es una buena práctica utilizar VARCHAR2 en lugar de VARCHAR para evitar cambios de comportamiento.	• La declaración anterior declaró la variable 'administrador' del tipo de datos VARCHAR con el tamaño máximo de 10 y asignó el valor 'guru99' que es de 6 bytes. Oracle asignará memoria de sólo 6 bytes en este caso. (Similar a VARCHAR2)
	• NVARCHAR2 (tamaño de cadena variable nativa) • LARGO y LARGO CRUDO	
NCHAR	Este tipo de datos es el mismo que el tipo de datos CHAR, pero el juego de caracteres será el juego de caracteres nacional. • Este juego de caracteres se puede definir para la sesión utilizando NLS_PARAMETERS. • El juego de caracteres puede ser UTF16 o UTF8. • La restricción de tamaño es de 1 a 2000 bytes.	native NCHAR(10); **Explicación de sintaxis:** • La declaración anterior declara la variable 'nativa' del tipo de datos NCHAR con un tamaño máximo de 10. • La longitud de esta variable depende del (número de longitudes) por byte tal como se define en el juego de caracteres.
NVARCHAR2	Este tipo de datos es el mismo que el tipo de datos VARCHAR2, pero el juego de caracteres será el juego de caracteres nacional. • Este juego de caracteres se puede definir para la sesión utilizando NLS_PARAMETERS. • El juego de caracteres puede ser UTF16 o UTF8. • La restricción de tamaño es de 1 a 4000 bytes.	Native var NVARCHAR2(10):='guru99'; **Explicación de sintaxis:** • La declaración anterior declara la variable 'Native_var' del tipo de datos NVARCHAR2 con un tamaño máximo de 10.
LARGO y LARGO	Este tipo de datos se utiliza para almacenar texto grande o datos sin procesar hasta un tamaño máximo de 2 GB.	Large_text LONG; Large_raw LONG RAW; **Explicación de sintaxis:**

	• Se utilizan principalmente en el diccionario de datos. • El tipo de datos LONG se utiliza para almacenar datos del juego de caracteres, mientras que LONG RAW se utiliza para almacenar datos en formato binario. • El tipo de datos LONG RAW acepta objetos multimedia, imágenes, etc., mientras que LONG solo funciona con datos que se pueden almacenar utilizando un juego de caracteres.	• La declaración anterior declara la variable 'Large_text' del tipo de datos LONG y 'Large_raw' del tipo de datos LONG RAW. **Nota:** No se recomienda el uso del tipo de datos LARGO Oracle. En su lugar, se debe preferir

NÚMERO PL/SQL Tipo de datos

Este tipo de datos almacena números de punto fijo o flotante de hasta 38 dígitos de precisión. Este tipo de datos se utiliza para trabajar con campos que contendrán solo datos numéricos. La variable se puede declarar con precisión y detalles de dígitos decimales o sin esta información. Los valores no necesitan estar entre comillas al asignarse para este tipo de datos.

A NUMBER(8,2);

B NUMBER(8);

C NUMBER;

Explicación de sintaxis:

- En lo anterior, la primera declaración declara que la variable 'A' es de tipo de datos numéricos con precisión total 8 y dígitos decimales 2.
- La segunda declaración declara que la variable 'B' es de tipo de datos numéricos con precisión total 8 y sin dígitos decimales.
- La tercera declaración es la más genérica, declara que la variable 'C' es de tipo numérico sin restricción en precisión o decimales. Puede tener hasta un máximo de 38 dígitos.

Tipo de datos booleano PL/SQL

Este tipo de datos almacena los valores lógicos. Oracle El tipo de datos booleano representa VERDADERO o FALSO y se utiliza principalmente en declaraciones condicionales. No es necesario que los valores estén entre comillas al asignarlos para este tipo de datos.

Var1 BOOLEAN;

Explicación de sintaxis:

- En lo anterior, la variable 'Var1' se declara como tipo de datos BOOLEANO. La salida del código será verdadera o falsa según la condición establecida.

Tipo de datos de fecha PL/SQL

Este tipo de datos almacena los valores en formato de fecha, como fecha, mes y año. Siempre que una variable se define con el tipo de datos FECHA junto con la fecha, puede contener información de hora y, de forma predeterminada, la información de hora se establece en 12:00:00 si no se especifica. Los valores deben estar entre comillas al asignarse para este tipo de datos.

El Oracle El formato de hora para entrada y salida es 'DD-MON-AA' y nuevamente se establece en NLS_PARAMETERS (NLS_DATE_FORMAT) en el nivel de sesión.

newyear DATE:='01-JAN-2015';

current_date DATE:=SYSDATE;

Explicación de sintaxis:

- En lo anterior, la variable "año nuevo" se declara como tipo de datos FECHA y se le asigna el valor del 1 de enero.st, fecha 2015.
- La segunda declaración declara la variable current_date como tipo de datos DATE y le asigna el valor con la fecha actual del sistema.
- Ambas variables contienen la información de tiempo.

Tipo de datos LOB PL/SQL

Este tipo de datos se utiliza principalmente para almacenar y manipular grandes bloques de datos no estructurados como imágenes, archivos multimedia, etc. Oracle prefiere LOB en lugar del tipo de datos LONG ya que es más flexible que el tipo de datos LONG. Las siguientes son las principales ventajas del tipo de datos LOB sobre el tipo de datos LARGO.

- El número de columnas en una tabla con tipo de datos LONG está limitado a 1, mientras que una tabla no tiene restricción en el número de columnas con tipo de datos LOB.
- La herramienta de interfaz de datos acepta el tipo de datos LOB de la tabla durante la replicación de datos, pero omite la columna LARGA de la tabla. Estas columnas LARGAS deben replicarse manualmente.
- El tamaño de la columna LARGA es de 2 GB, mientras que LOB puede almacenar hasta 128 TB.

- Oracle está mejorando constantemente el tipo de datos LOB en cada una de sus versiones de acuerdo con los requisitos modernos, mientras que el tipo de datos LONG es constante y no recibe muchas actualizaciones.

Por lo tanto, siempre es mejor utilizar el tipo de datos LOB en lugar del tipo de datos LONG. A continuación, se muestran los diferentes tipos de datos LOB. Pueden almacenar hasta un tamaño de 128 terabytes.

1. BLOB
2. CLOB y NCLOB
3. ARCHIVOB

Tipo de datos	Descripción	Sintaxis
BLOB	Este tipo de datos almacena los datos LOB en formato de archivo binario hasta un tamaño máximo de 128 TB. No almacena datos basados en los detalles del conjunto de caracteres, por lo que puede almacenar datos no estructurados, como objetos multimedia, imágenes, etc.	Binary_data BLOB; **Explicación de sintaxis:** • En lo anterior, la variable 'Binary_data' se declara como BLOB.
CLOB y NCLOB	El tipo de datos CLOB almacena los datos LOB en el juego de caracteres, mientras que NCLOB almacena los datos en el juego de caracteres nativo. Dado que estos tipos de datos utilizan almacenamiento basado en juegos de caracteres, no pueden almacenar datos como multimedia, imágenes, etc. que no se pueden poner en una cadena de caracteres. El tamaño máximo de estos tipos de datos es 128 TB.	Charac_data CLOB; **Explicación de sintaxis:** • En lo anterior, la variable 'Charac_data' se declara como tipo de datos CLOB.
ARCHIVOB	• BFILE son los tipos de datos que almacenaron los datos en formato binario no estructurado fuera de la base de datos como un archivo del sistema operativo. • El tamaño de BFILE es para un sistema operativo limitado, son archivos de solo lectura y no se pueden modificar.	

ENTORNO DE DESARROLLO

- Existen diferentes entornos de desarrollo para PL/SQL. Los principales son:

  - SQL*PLUS(ISQLPLUS)
  - Oracle Developer Procedure Builder (herramienta de desarrollo)

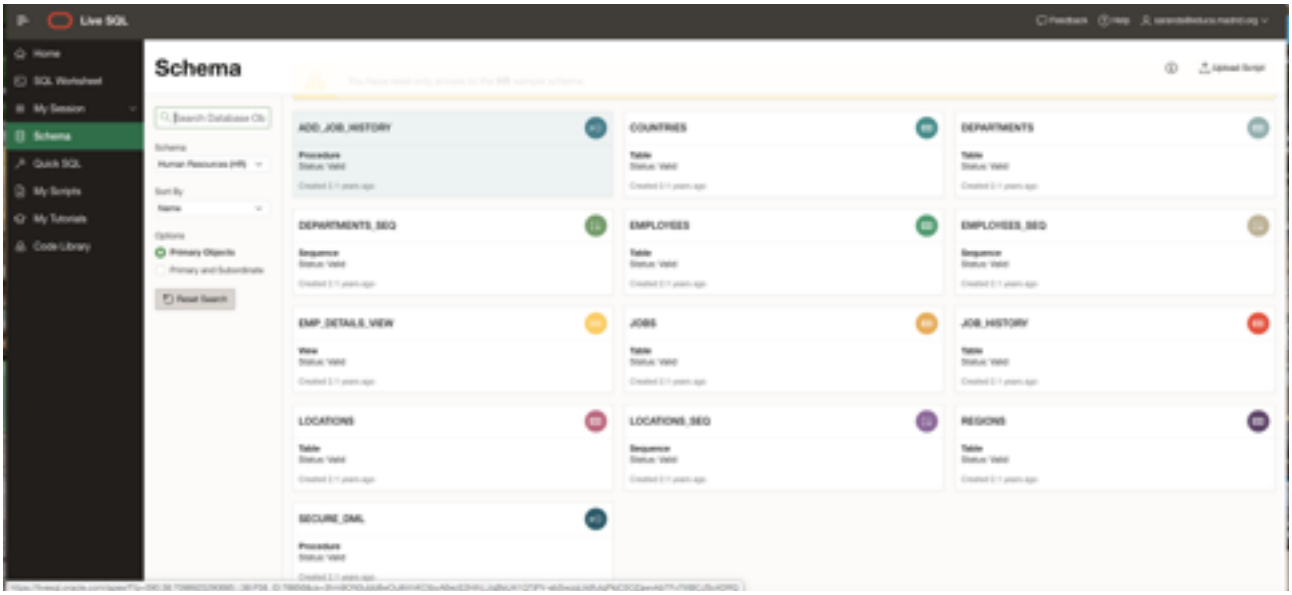
  - El primero utiliza el motor en el servidor Oracle y el segundo cuenta con las dos opciones disponibles.

  - Dispondremos de una opciones u otras en relación a la construcción de programas con bloques:

    - Un bloque anónimo se puede utilizar en cualquier entorno PL/SQl. Son un conjunto de instrucciones que se ejecutan en modo local. Normalmente se utilizan para hacer pruebas
    - Bloque nominado, es igual pero tiene una etiqueta. También se ejecuta en modo local

ORACLE LIVE SQL

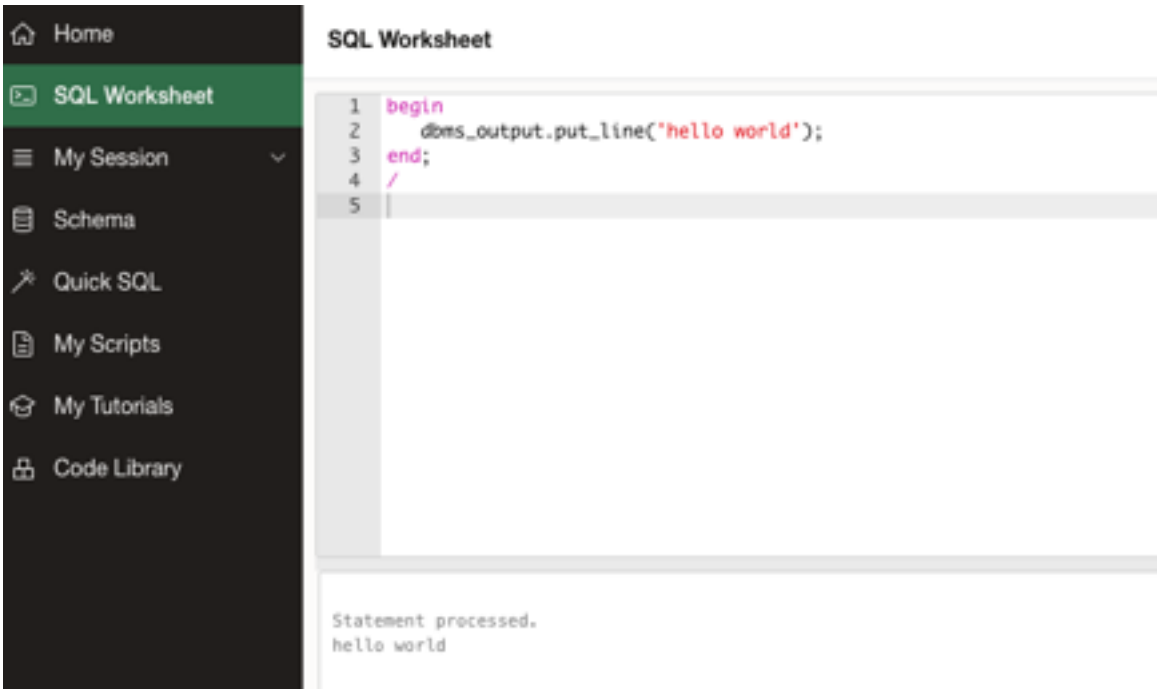
Dada la situación actual vamos a utilizar Live SQL de orca para probar nuestro código. En este entorno entras a SQL



Como veis en sql Worksheet podemos introducir nuestro código sql y PL/sql.

Os he hecho una captura del las tablas del usuario o esquema HR que viene como ejemplo en Oracle

Como vemos a continuación , acabamos de ejecutar un bloque anónimos que visualiza un "Hello World" de prueba.



Ejemplo:

```
declare
    l_today date := sysdate;
begin
    dbms_output.put_line(
            'today is '||to_char(l_today,'Day'));
exception when others then
    dbms_output.put_line(sqlerrm);
end;
/
```

El código anterior nos visualiza que hoy es miércoles (wednesday) y saltaría una excepción que nos visualiza un código de error , si se produjese algún error.

ASIGNACION DE VALORES A VARIABLES.

- La asignación de valores a una variable en la parte ejecutable de un bloque se puede realizar a través del operador := pero también con la cláusula INTO de las sentencias SELECT y con FETCH (Se estudiará posteriormente en el tema).

  Ejemplos:

  ```
  total := unidades * pts_unidad;
  SELECT salario INTO sueldo     FROM plantilla
            WHERE apellido LIKE 'MORENO';
   FECHT cursor1 INTO nombre;
  ```

  Para empezar a familiarizarnos con el SQL dentro de un bloque PL/SQL vemos que las Select ya no hacen la salida en un terminal, sino que lo que recuperan lo almacenan en la variable que pongamos detrás del Into.

  En este caso recuerda de la base de datos el sueldo de "MORENO2 y lo que recupere lo guarda en la variable sueldo.

- También es posible realizar asignaciones de valor a las variables en el momento de su declaración, para ello se utilizará la palabra clave DEFAULT, o bien, el operador de asignación.

  Ejemplos:

  ```
  importe   NUMBER (9) DEFAULT 3000;
  nombre    VARCHAR2(20) := 'Marta Parrado';
  ```

EXPRESIONES.

- Una expresión está formada por un conjunto de operando unidos por operadores, los cuales ya fueron estudiados en el Tema 3.
- Cabe mencionar algún operador que incorpora PL/SQL y la prioridad de los operadores:

Operador	Operación
**,NOT	Exponenciacion , negación
+,-	Operadores unario (positivo, negativo)
*./	Multiplicacion,división
+,-, ||	Suma, resta, concatenación
= , ! = , < > , > , < , > = , < = , I S NULL,LIKE,BETWEEN, IN	Operadores de comparación
AND	Conjunción
OR	Inclusión

VARIABLES Y CONSTANTES.

- PL/SQL permite declarar e inicializar variables y/o constantes.
- Las variables pueden ser de cualquiera de los tipos vistos en SQL o de otros tipos que veremos.
- Todas las variables han de ser declaradas antes de ser utilizadas
- Las variables se pueden utilizar:
  - Para almacenar temporalmente datos de entrada hasta que ya no se necesiten
  - Para realizar operaciones sobre los datos
  - Facilidad de mantenimiento: %ROWTYPE,%TYPE
- Para declarar una variable utilizaremos el siguiente formato:
  Nombre_variable [CONSTANT] tipo [NOT NULL]
  [{:=| DEFAULT } expresión];
- Las palabras clave NOT NULL especifican que la variable no podrá tomar valor nulo.
- La palabra clave DEFAULT permite inicializar una variable a la vez que se define.
- Expresión podrá ser un literal, otra variable o el resultado de una expresión que puede incluir operadores y funciones.
- Para los identificadores de las variables se deben utilizar las mismas reglas que para los objetos de SQL
  - Longitud máxima: 30 caracteres
  - Letras, números y caracteres solo $, _, #
  - El nombre debe empezar por una letra
  - No puede ser una palabra reservada de Oracle
- Para asignar valores por defecto podemos utilizar indistintamente := o DEFAULT
- Una variable no inicializada queda con valor NULL hasta asignarle valor en la ejecución
- Las constantes y variables declaradas como NOT NULL deben ser inicializadas
- Dos objetos pueden tener el mismo nombre siempre y cuando estén declarados en distintos bloques
- No se debe poner el mismo nombre a un identificador de variable que a la columna de la que vaya a recibir los datos. Esto evita confusiones y facilita el mantenimiento del software

Ejemplos:

```
importe    NUMBER (9);
nombre     VARCHAR2(20) NOT NULL;
descuento NUMBER(4,2) NOT NULL DEFAULT 10.27;
nombre char(20) NOT NULL := `MIGUEL';
```

casado BOOLEAN DEFAULT FALSE;

- Podemos utilizar el atributo %TYPE para dar a una variable el tipo de dato de otra variable o de una columna de la base de datos.

Nom_variable tabla.columna%type;

Ejemplo:

importe NUMBER (9);
total importe %TYPE; → Declarada total como NUMBER(9)
nuevonombre emp.nombre%TYPE -> Declara nuevonombre del mismo tipo que la columna nombre de emp

- La principal ventaja del uso de este atributo es la facilidad de mantenimiento de los programas, ya que ante cualquier cambio de tipos en la base de datos no hay que hacer ningún cambio en el código Pl/SQL

- %TYPE también puede utilizarse para declara una variable del mismo tipo que otra declarada previamente.

- **Si una columna tiene la restricción NOT NULL, la variable definida de ese tipo mediante el atributo %TYPE, no asume dicha restricción, pudiendo, por lo tanto, ponerla a NULL, a no ser que se especifique lo contrario en la declaración.**

Ejemplo: planta vecinos.piso %TYPE NOT NULL;
Declarada como NUMBER(2) y no
admitiendo valores nulos.

- También podemos utilizar %ROWTYPE que crea una variable tipo registro para cargar los valores de toda una fila.
Mifila emp%ROWTYPE

Variable tabla%ROWTYPE;

- Se pueden declarar constantes utilizando la palabra reservada CONSTANT, pero teniendo en cuenta que la constante se debe inicializar en el momento de la declaración.

- El valor de una constante, una vez determinada no modificarse. La declaración de constantes facilita la modificación de programas que contienen datos constantes.

- La declaración de una constante es igual a la de una variable, salvo que hay que utilizar CONSTANT e inicializarla.

Ejemplo: iva CONSTANT NUMBER(2) DEFAULT 16;

- En el ejemplo anterior, si no declaramos la constante 'iva', un cambio en el porcentaje del impuesto obligaría a cambiarlo en todas las expresiones donde apareciese.

- Cualquier variable o constante debe ser declarada antes de ser referenciada en otra declaración o sentencia, ya que en caso contrario se produciría un error.

  Ejemplo: total importe %TYPE; → Error
  importe NUMBER(9,2);

Procedimientos y Funciones PL/SQL

INTRODUCCION

- Los procedimientos y funciones son subprogramas que pueden ser invocados por los usuarios.
- En PL/SQL el desarrollador puede definir sus propios subprogramas o bien utilizar funciones predefinidas en PL/SQL.
- Los subprogramas pueden ser de dos tipos:
  - Funciones: subprogramas que devuelven un valor
  - Procedimientos: subprogramas que ejecutan una secuencia de instrucciones pero que el nombre del subprograma en si mismo no devuelve un valor.

FUNCIONES PREDEFINIDAS PL/SQL

PL/SQL proporciona un gran número de funciones muy útiles para ayudar a manipular la información y permite incorporar en sus expresiones casi todas las funciones disponibles en SQL. Se pueden agrupar en categorías:

- caracteres
- numéricas
- fechas
- conversión de tipo de datos
- manejo de nulos
- misceláneas
- error-reporting

Las funciones de agrupación de SQL como por ejemplo AVG, MIN, MAX, COUNT, SUM, STDDEV, y VARIANCE, no están implementadas en PL/SQL, sin embargo se pueden usar en sentencias SQL. Tampoco se pueden usar algunas otras como DECODE, DUMP, y VSIZE.

A continuación vamos a ver algunas de las más utilizadas.

FUNCIONES PREDEFINIDAS CARACTERES

- LENGTH: Devuelve la longitud de un tipo CHAR.

```
resultado := LENGTH('HOLA MUNDO’); -- Devuelve 10
```

- INSTR

Busca una cadena de caracteres (la que se indica en el segundo parámetro pasado) dentro de otra (la que se indica en el primer parámetro) y devuelve la posición de la ocurrencia de la cadena buscada dentro de la cadena. En el tercer parámetro se indica la posición desde la que se comienza a buscar (opcional) y en el cuarto el número de ocurrencia que se busca (opcional).

Su sintaxis es la siguiente: INSTR(<char>, <search_string>, <startpos>, <occurrence>)

```
resultado := INSTR('AQUI ES DONDE SE BUSCA', 'BUSCA', 1, 1); -- Devuelve 18
```

FUNCIONES PREDEFINIDAS CARACTERES

- REPLACE: Reemplaza un texto por otro en una cadena de caracteres.

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

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

```
resultado := REPLACE ('HOLA MUNDO','HOLA', 'VAYA'); -- devuelve VAYA MUNDO
```

- SUBSTR: Obtiene una parte de una cadena de caracteres, desde una posición de inicio hasta una determinada longitud.

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

```
resultado := SUBSTR('HOLA MUNDO', 6, 5); -- Devuelve MUNDO
```

FUNCIONES PREDEFINIDAS CARACTERES

- UPPER: Convierte una cadena alfanumérica a mayúsculas.

```
resultado := UPPER('hola mundo'); -- Devuelve HOLA MUNDO
```

- LOWER: Convierte una cadena alfanumérica a minúsculas.

```
resultado := LOWER('HOLA MUNDO'); -- Devuelve hola mundo
```

- RTRIM: Elimina los espacios en blanco a la derecha de una cadena de caracteres.

```
resultado := RTRIM ('Hola Mundo ');
```

- LTRIM: Elimina los espacios en blanco a la izquierda de una cadena de caracteres.

```
resultado := LTRIM (' Hola Mundo');
```

- TRIM: Elimina los espacios en blanco a la izquierda y derecha de una cadena de caracteres.

```
resultado := TRIM (' Hola Mundo ');
```

FUNCIONES PREDEFINIDAS NUMERICAS

- MOD: Devuelve el resto de la división entera entre dos números.

MOD(<dividendo>, <divisor>)

```
resultado := MOD(20,15); -- Devuelve el modulo de dividir 20/15
```

- TRUNC: Trunca un número y devuelve la parte entera.

```
resultado := TRUNC(9.99); -- Devuelve 9
```

- ROUND: Devuelve el entero más próximo.

```
resultado := ROUND(9.99); -- Devuelve 10
```

FUNCIONES PREDEFINIDAS FECHAS

- SYSDATE: Devuelve la fecha del sistema.

```
resultado := SYSDATE;
```

- TRUNC: Trunca una fecha, elimina las horas, minutos y segundos de la misma.

```
resultado := TRUNC(SYSDATE);
```

FUNCIONES PREDEFINIDAS CONVERSION TIPO DATOS

- TO_DATE: Convierte una expresión al tipo fecha.

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

El parámetro opcional formato indica el formato de entrada de la expresión no el de salida.

En este ejemplo se convierte la cadena de caracteres '01/12/2006' a una fecha (tipo DATE). El formato indica que la fecha está escrita como día/mes/año, de forma que la fecha sea el uno de diciembre y no el doce de enero.

```
resultado := TO_DATE('01/12/2006', 'DD/MM/YYYY');
```

El siguiente ejemplo muestra la conversión con formato de día y hora.

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

FUNCIONES PREDEFINIDAS CONVERSION TIPO DATOS

- TO_CHAR: Convierte una expresión al tipo CHAR.

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

El parámetro opcional formato indica el formato de salida de la expresión.

```
resultado := TO_CHAR(SYSDATE, 'DD/MM/YYYY HH24:MI:SS');
```

- TO_NUMBER: Convierte una expresión alfanumérica en numérica, se puede especificar el formato de salida (opcional).

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

```
resultado := TO_NUMBER ('10.21', ’99.99’); -- resultado: 10,21 (el separador decimal es ,)
```

FUNCIONES PREDEFINIDAS MANEJO DE NULOS

- NVL: Devuelve el valor recibido como parámetro en el caso de que expresión sea NULL o el valor de la 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;
```

FUNCIONES PREDEFINIDAS MISCELANEAS

- 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 (el último parámetro).

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;
```

FUNCIONES PREDEFINIDAS

Carácter	Numéricas	Fecha	Conversión	Manejo Nulos	Misceláneas	Error
ASCII	ABS	ADD_MONTHS	CHARTOROWID	NVL	DECODE	SQLCODE
CHR	ACOS	CURRENT_DATE	CONVERT		DUMP	SQLERRM
CONCAT	ASIN	CURRENT_TIMESTAM	HEXTORAW		GREATEST	
INITCAP	ATAN	LAST_DAY	NLS_CHARSET_ID		GREATEST_LB	
INSTR	ATAN2	LOCALTIMESTAMP	NLS_CHARSET_NAME		LEAST	
INSTRB	CEIL	MONTHS_BETWEEN	RAWTOHEX		LEAST_UB	
LENGTH	COS	NEW_TIME	ROWIDTOCHAR		UID	
LENGTHB	COSH	NEXT_DAY	TO_CHAR		USER	
LOWER	EXP	ROUND	TO_DATE		USERENV	
LPAD	FLOOR	SYSDATE	TO_LABEL		VSIZE	
LTRIM	LN	SYSTIMESTAMP	TO_MULTI_BYTE			
NLS_INITCAP	LOG	TRUNC	TO_NUMBER			
NLS_LOWER	MOD		TO_SINGLE_BYTE			
NLS_UPPER	POWER					
NLSSORT	ROUND					
REPLACE	SIGN					
RPAD	SIN					
RTRIM	SINH					
SOUNDEX	SQRT					
SUBSTR	TAN					
SUBSTRB	TANH					
TRANSLATE	TRUNC					
UPPER						

PROCEDIMIENTOS Y FUNCIONES DEFINIDOS POR EL DESARROLLADOR

Los bloques de código anónimos BEGIN .. END, son un mecanismo básico para la programación en PL/SQL, pero no están orientados a la reutilización de SCRIPTS. Por ejemplo, en caso de que se tenga un algoritmo para algún cálculo según determinados parámetros tendríamos que repetirlo cuantas veces sea necesario.

El uso de procedimientos en PL/SQL es un buen mecanismo para la reutilización de código, además de que permite dividir el código en partes funcionales más pequeñas.

Los procedimientos pueden ser declarados en bloques anónimos o almacenarnos en la base de datos.

DECLARACION DE PROCEDIMIENTOS

La creación de un procedimiento en PL/SQL es similar a la creación de un bloque anónimo. La sintaxis de un procedimiento es la siguiente:

```
PROCEDURE nom_proc[(param1[,param2 ...])]
IS
  declaraciones locales;
BEGIN
  sentencias;
[EXCEPTION
         tratamiento_de_excepciones]
END [nom_proc];
```

- nom_proc: Es el nombre del procedimiento, se usará para identificarlo.
- param: son como variables, contienen datos que se pueden especificar en el momento de llamar al procedimiento.
- declaraciones locales: Como en un bloque anónimo se pueden crear variables que sólo pueden ser usadas en código dentro del procedimiento.
- sentencias: Es el código que se ejecuta cuando se llama al procedimiento y que puede hacer uso de las variables declaradas así como de los parámetros.

DECLARACION DE PROCEDIMIENTOS

El IS es el equivalente a DECLARE en los bloques anónimos.

En el IS sí debemos indicar la longitud de las variables locales.

PARAMETROS DE LOS PROCEDIMIENTOS

Tienen la siguiente sintaxis:

Nom_param [IN|OUT|IN OUT] tipo_dato[{:=|DEFAULT }Valor]

- Cuando no se indica, los parámetros se definen por defecto de tipo IN.
- En tipo_dato sólo se especifica el tipo, sin indicar su longitud ni restricciones.
- Si un procedimiento no tienen parámetros, no es necesario poner los paréntesis en la cabecera.

- Un parámetro de entrada permite que pasemos valores al subprograma y no puede ser modificado en el subprograma. El parámetro pasado puede ser una constante o una variable.
- Un parámetro de salida permite devolver valores y en el subprograma actúa como variable no inicializada. El parámetro pasado debe ser una variable.
- Un parámetro de entrada-salida se utiliza para pasar valores al subprograma y/o para recibirlos, por lo que un parámetro formal que actúe como parámetro pasado debe ser una variable.

PROCEDIMIENTOS DENTRO DE UN BLOQUE ANONIMO

En estos casos el procedimientos se debe crear dentro de la sección DECLARE ... BEGIN.

```
DECLARE
-- El procedimiento debe ser declarado dentro de la sección DECLARE .. BEGIN
  PROCEDURE registrar_cliente(P_ID NUMBER,
                              P_NOMBRE VARCHAR2,
                              P_APELLIDOS VARCHAR2)
  IS
    declaraciones locales;
  BEGIN
    sentencias;
  [EXCEPTION
           tratamiento_de_excepciones]
  END [nom_proc];
BEGIN
-- Sentencias, código de bloque anónimo
 ...
  REGISTRAR_CLIENTE(1,'Juan', 'Rosales');
  REGISTRAR_CLIENTE(2,'Luis', 'Cabrera');
 ...
END;
/
```

La declaración de procedimientos debe ir al final de la sección DECLARE correspondiente.

PROCEDIMIENTOS ALMACENADOS

PL/SQL permite almacenar los procedimientos para ser usados desde cualquier bloque anónimo (sin que haya la necesidad de declararlo) y también desde otros procedimientos.

Para crear un procedimiento almacenado debemos poner la palabra reservada CREATE y ejecutar el código como si se tratase de un bloque PL/SQL.

El procedimiento almacenado es compilado previamente por el motor PL/SQL y si no da errores quedará almacenado y se podrá llamar.

```
CREATE PROCEDURE registrar_cliente (P_ID NUMBER,
                                    P_NOMBRE VARCHAR2,
                                    P_APELLIDOS VARCHAR2)
IS
  declaraciones locales;
BEGIN
  sentencias;
[EXCEPTION
         tratamiento_de_excepciones]
END [nom_proc];
```

PROCEDIMIENTOS ALMACENADOS

Adicionalmente, se puede añadir las palabras reservadas **OR REPLACE** para evitar errores al intentar compilar un procedimiento que ya ha sido compilado :

CREATE OR REPLACE PROCEDURE REGISTRAR_CLIENTE….

Con esta sentencia creamos un procedimiento. Si ya existía lo reemplaza.

En el caso de los procedimientos almacenados ya no es necesario declarar el procedimiento dentro de la sección DECLARE .. BEGIN de los bloques anónimos.

Un procedimiento **se puede invocar desde un** bloque u otro procedimiento/ función de PL/SQL **llamándolo** simplemente **por su nombre** y pasándole los parámetros.

PROCEDIMIENTOS ALMACENADOS

Se puede invocar un procedimiento también desde SQL*PLUS:

Sql> **execute** registrar_cliente (7902,'Antonio','Alvarez Sánchez');

Si alguno de los parámetros fuera de salida (OUT o IN OUT) se debe invocar con una variable que debe ser definida previamente:

```
// Creación del procedimiento, el segundo parámetro es de salida
CREATE OR REPLACE PROCEDURE calcular_cuadrado_procedure(P_NUMERO NUMBER, P_CUADRADO OUT NUMBER)
IS
BEGIN
  P_CUADRADO := P_NUMERO*P_NUMERO;
END calcular_cuadrado_procedure;
```

```
// Llamada al procedimiento desde consola SQL*PLUS
SQL> var num_cuadrado NUMBER -- Se define la variable Host necesaria para parámetro de salida

SQL> exec calcular_cuadrado_procedure(5,:num_cuadrado) -- Llamada a método, : antes de variable Host

SQL> PRINT num_calculo -- Muestra por pantalla la variable Host
```

METODOS DE PASO DE PARAMETROS

Notación Posicional: Se pasan los valores de los parámetros en el mismo orden en que el procedure los define.

```
BEGIN
  REGISTRAR_CLIENTE(1,'Juan', 'Rosales');
END;
```

Notación Nominal: Se pasan los valores en cualquier orden nombrando explícitamente el parámetro y su valor separados por el símbolo =>.

```
BEGIN
  REGISTRAR_CLIENTE(P_ID => 1,P_NOMBRE => 'Juan', P_APELLIDOS => 'Rosales');
END;
```

Notación Mixta: Combina las dos anteriores.

```
BEGIN
  REGISTRAR_CLIENTE(1,P_NOMBRE => 'Juan', P_APELLIDOS => 'Rosales');
END;
```

DECLARACION DE FUNCIONES

La creación de una función tiene una sintaxis similar a la de un procedimiento:

```
FUNCTION nom_funcion([param1[,param2 ...]])
RETURN [tipo de valor devuelto]
IS
  declaraciones locales;
BEGIN
  sentencias;
  RETURN(expresión);
[EXCEPTION
         tratamiento_de_excepciones]
END [nom_funcion];
```

- Los parámetros tienen la misma sintaxis que en los procedimientos.
- La cláusula RETURN de la cabecera indica el tipo de datos que devuelve la función.
- La cláusula RETURN del cuerpo hace efectivo ese retorno.

FUNCIONES ALMACENADAS

Como para los procedimientos, para crear una función almacenada:

```
CREATE OR REPLACE FUNCTION nom_funcion(([param1[,param2 ...]])
RETURN [tipo de valor devuelto]
IS
   declaraciones locales;
BEGIN
   sentencias;
   RETURN(expresión);
[EXCEPTION
        tratamiento_de_excepciones]
END [nom_funcion];
```

En el caso de las funciones almacenadas ya no es necesario declararla dentro de la sección DECLARE .. BEGIN de los bloques anónimos.

LLAMADAS A FUNCIONES

Una función **se puede invocar desde un** bloque u otro procedimiento / función de PL/SQL **llamándola** simplemente **por su nombre** y pasándole los parámetros requeridos, asignando el valor (mediante :=) a una variable del mismo tipo que devuelve la función:

```
BEGIN
  ...
  num_calculo := calcular_cuadrado(3);
  ...
END;
```

Se puede invocar también desde SQL*PLUS

- Consulta genérica:

```
SQL> select year_of_date(start_date) FROM DUAL;
```

- Utilizando exec y variables Host:

```
SQL> var num_calculo NUMBER -- Se define la variable de Host necesaria para asignar valor a la función
SQL> exec :num_calculo := calcular_cuadrado(3) -- Llamada a la función, usar : antes de variable Host

SQL> PRINT num_calculo -- Muestra por pantalla la variable Host
```

PROCEDIMIENTOS Y FUNCIONES

PL/SQL permite la **sobrecarga** en los nombres de subprogramas (aplica a procedimientos y funciones), es decir, podemos llamar a dos subprogramas con el mismo nombre y los distingue porque sus parámetros deben tener o distinto número o distintos tipo. La sobrecarga de los subprogramas se usa generalmente cuando conceptualmente se ejecuta la misma tarea (o similar) pero con un conjunto de parámetros ligeramente diferente (o con diferente definición).

También permite **programación recursiva** (ejemplo típico cálculo del factorial de un número).

PROCEDIMIENTOS Y FUNCIONES

Se pueden conocer los procedimientos y funciones definidos mediante la siguiente consulta

```
SELECT object_name FROM user_procedures;
```

Se puede eliminar un procedimiento ejecutando

```
drop procedure nombre_procedimiento;
```

Se puede eliminar una función ejecutando

```
drop function nombre_función;
```

SENTENCIAS DE CONTROL.

- PL/SQL dispone de sentencias que permiten construir las tres estructuras de control de la programación estructurada, es decir, secuencial, alternativa y repetitiva, teniendo en cuenta que para las dos últimas estructuras están permitidas las anidaciones, es decir, dentro de in if podremos encontrar otro, o dentro de un while podremos anidar por ejemplo otro while.

SENTENCIAS SECUENCIALES

- Se consideran una secuencia , al conjunto de instrucciones que se ejecutan una detrás de otra.
- Cada sentencia en PL/SQL finaliza con ;

SENTENCIAS ALTERNATIVAS

- Son las sentencias que nos permiten cambiar el flujo del programa dependiendo de que se cumpla o no una condición.
- Oracle cuenta con las siguientes sentencias alternativas

IF-THEN

- Como se aprecia en el primer gráfico que se presenta a continuación, , cuando se cumple la condición, es decir cuando devuelve True, se ejecutan una serie de sentencias (<action_block>) que no se ejecutarán si no se cumple la condición, pero en cualquiera de los dos casos, continúa la ejecución del programa después del END IF;
- La condición por tanto, siempre debe ser evaluada como true o false
- La sintaxis de un If será.

```
IF <condition: returns Boolean>
THEN
 -executed only if the condition returns TRUE
 <action_block>
END if;
```

- Cualquier condición evaluada como 'NULL', será tratada como ‘FALSE'.

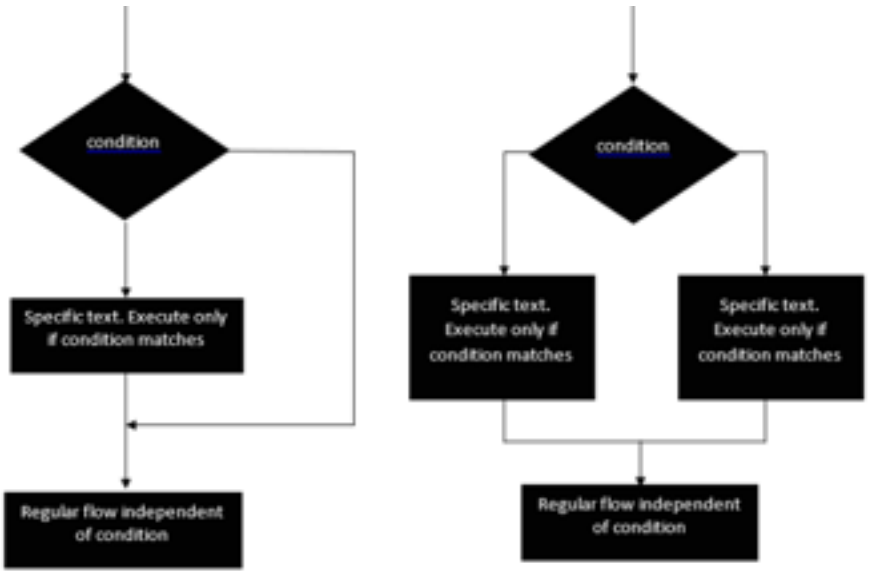
Ejemplo :

```
DECLARE
a CHAR(1) :=’u’;
BEGIN
IF UPPER(a) in ('A’,'E','I','0','U' ) THEN
dbms_output.put_line(‘El carácter es una vocal');
END IF;
END;
/
```

- Como vemos la función que coge el carácter y lo convierte a a mayusculas es UPPER(), exactamente igual que en SQL
- In , permite comprobar si el valor está entre los citados entre los ( )

IF-THEN-ELSE

- Como se aprecia en el segundo gráfico, si se cumple la condición se ejecuta un conjunto de sentencias y si no se cumple se ejecutan otra sentencia o conjunto de ellas, y después en cualquiera de los dos casos continúa con la ejecución del resto del programa



Decision Making Statement Diagram

```
IF <condition: returns Boolean>
THEN
        -executed only if the condition returns TRUE
        <action_blockl>
ELSE
        -execute if the condition failed (returns FALSE)
        <action_block2>
END if;
```

- Se ejecuta <action_blockl>cuando la condición devuelve true
- Se ejecuta <action_block2> cuando la condición devuelve False

IF-THEN-ELSIF

```
 IF <conditionl: returns Boolean>
THEN
-executed only if the condition returns TRUE <
action_blockl>
ELSIF <condition2 returns Boolean> <
action_block2>
ELSIF <condition3:returns Boolean> <
action_block3>
ELSE —optional
<action_block_else>
END if;
```

- Esta sentencia alternativa se utiliza cuando hay que seleccionar una alternativa entre un conjunto de ellas.
- La primera condición que devuelva verdadero, será la que se ejecute , y el resto no se ejecutará.
- En caso de que no se cumpla ninguna, se ejecutan las sentencias del bloque ELSE si es que existiese.

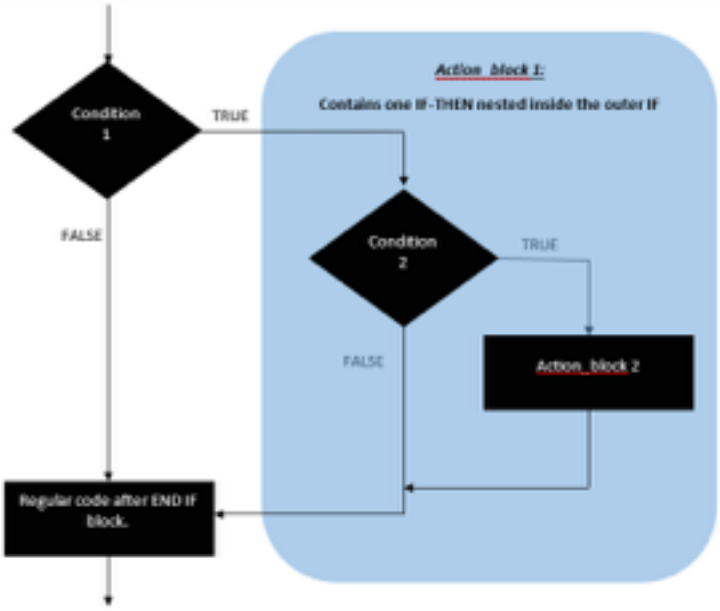
Ejemplo:

```
DECLARE
mark NUMBER :=55;
BEGIN
dbms_output.put_line('Program started.' );
IF( mark >= 70) THEN
dbms_output.put_line('Grade A');
ELSIF(mark >= 40 AND mark < 70) THEN
dbms_output.put_line('Grade B');
ELSIF(mark >=35 AND mark < 40) THEN
dbms_output.put_line('Grade C');
END IF;
dbms_output.put_line('Program completed.');
END;
/
```

Como vemos en este caso, la primera condición no la cumple IF (mark >= 70), por tanto , pasa a evaluar la siguiente . ELSIF(mark >= 40 AND mark < 70) esta si la cumple , por tanto ejecuta dbms_output.put_line('Grade B'); y dbms_output.put_line('Program completed.'); y finaliza el programa.

Bucles anidados

- En PL se pueden incluir unos if dentro de otro, anidando así los if que deseemos



IF <conditionl: returns Boolean>
THEN
—executed only if the condition returns TRUE
<action block1 starts>
IF <condition2: returns Boolean>
THEN
<action_block2>
END IF; —END IF corresponds to condition2
<action_blockl ends>
END IF; —END IF corresponds to condition1

CASE

Es simillar a a IF , pero selecciona un bloque de sentencias en función de la expresión, que ahora no tiene por qué ser un valor Booleano, puede ser un entero, cadena, etc.

El else se ejecuta cuando ninguna de las alternativas es seleccionada.

```
CASE (expression)
 WHEN <valuel> THEN action_blockl;
 WHEN <value2> THEN action_block2;
 WHEN <value3> THEN action_block3;
 ELSE action_block_default;
END CASE;
```

ejemplo:

```
DECLARE
a NUMBER :=55;
b NUMBER :=5;
arth_operation VARCHAR2(20) :='MULTIPLY’;
BEGIN
dbms_output.put_line(‘Program started.' );
CASE (arth_operation)
WHEN ‘ADD’ THEN dbms_output.put_line(‘Addition of the numbers are: '|| a+b );
WHEN ‘SUBTRACT' THEN dbms_output.put_line(‘Subtraction of the numbers are: '||a-b );
WHEN ‘MULTIPLY' THEN dbms_output.put_line(‘Multiplication of the numbers are: '|| a*b
);
WHEN ‘DIVIDE' THEN dbms_output.put_line(‘Division of the numbers are:'|| a/b);
ELSE dbms_output.put_line(‘No operation action defined. Invalid operation');
END CASE;
dbms_output.put_line(‘Program completed.' );
END;
/
```

SEARCHED CASE

Es un caso especial del CASE, pero no ponemos expresión en el CASE y las vamos poniendo en el WHEN. Cuando se cumpla una expresión se ejecuta el código asociado y finaliza el CASE.

```
CASE
WHEN <expression1> THEN action_blockl;
WHEN <expression2> THEN action_block2;
WHEN <expression3> THEN action_block3;
ELSE action_block_default;
END CASE;
```

Ejemplo:

```
DECLARE a NUMBER :=55;
b NUMBER :=5;
arth_operation VARCHAR2(20) :='DIVIDE';
BEGIN
dbms_output.put_line(‘Program started.' );
CASE
WHEN arth_operation = 'ADD'
THEN dbms_output.put_line(‘Addition of the numbers are: '||a+b );
WHEN arth_operation = ‘SUBTRACT'
THEN dbms_output.put_line(‘Subtraction of the numbers are: '|| a-b);
WHEN arth_operation = ‘MULTIPLY’
THEN dbms_output.put_line(‘Multiplication of the numbers are: '|| a*b );
WHEN arth_operation = ’DIVIDE'
THEN dbms_output.put_line(‘Division of the numbers are: '|| a/b ):
ELSE dbms_output.put_line(‘No operation action defined. Invalid operation');
END CASE;
dbms_output.put_line(‘Program completed.' );
END;
/
```