

Programación de bases de datos.

Caso práctico

Juan recuerda, de cuando estudió el Ciclo de Desarrollo de Aplicaciones Informáticas, que había muchas tareas que se podían automatizar dentro de la base de datos mediante el uso de un lenguaje de programación, e incluso que se podían programar algunas restricciones a la hora de manipular los datos. **Juan** se lo comenta a **María** y ésta se muestra ilusionada con dicha idea ya que muchas veces repiten el trabajo con la base de datos de juegos on-line que tienen entre manos (consultas, inserciones, etc. que son muy parecidas y que se podrían automatizar).



Ministerio de Educación (Uso educativo no)

Para ello hablan con **Ada** y ésta les comenta que claro que se puede hacer y que



Ministerio de Educación (Uso educativo no)

precisamente eso es lo que les toca hacer ahora. **Ada** les dice que para ese propósito existe un lenguaje de programación llamado PL/SQL que permite hacer lo que ellos quieren, así que les pasa un manual para que se lo vayan leyendo y se vayan poniendo manos a la obra con la base de datos de juegos on-line.

Ahora que ya dominas el uso de SQL para la manipulación y consulta de datos, es el momento de dar una vuelta de tuerca adicional para mejorar las aplicaciones que utilicen nuestra base de datos. Para ello nos vamos a centrar en la programación de bases de datos, utilizando el lenguaje PL/SQL. En esta unidad conoceremos qué es PL/SQL, cuál es su sintaxis y veremos cómo podemos sacarle el máximo partido a nuestra base de datos mediante su uso.

Debes conocer

La mayor parte de los ejemplos de esta unidad están basados en el modelo de datos extraído del siguiente caso de estudio:

[Caso de estudio.](#)



[Ministerio de Educación y Formación Profesional.](#) (Dominio público)

Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.

[Aviso Legal](#)

1.- Introducción.

Caso práctico

Juan y María se han puesto a repasar el manual de PL/SQL que les ha pasado Ada. Aunque no han avanzado mucho con el mismo, ya saben a qué se van a enfrentar y los beneficios que pueden obtener del uso del mismo para su aplicación de juegos on-line. Cuando hacen la primera parada de la mañana para tomarse un café, ambos se ponen a comentar las primeras conclusiones que han sacado después de su primer acercamiento a este lenguaje. Ambos están deseosos de seguir avanzando en su aprendizaje y saben que para ello cuentan con la inestimable ayuda de **Ada**.



[Miguel Martínez Monasterio \(INTEF\) \(CC BY-NC-SA\)](#)

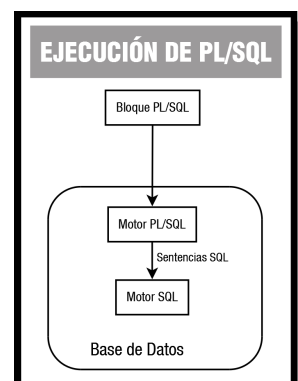
Estarás pensando que si no tenemos bastante con aprender SQL, sino que ahora tenemos que aprender otro lenguaje más que lo único que va a hacer es complicarnos la vida. Verás que eso no es cierto ya que lo más importante, que es el conocimiento de SQL, ya lo tienes. PL/SQL tiene una sintaxis muy sencilla y verás como pronto te acostumbras y luego no podrás vivir sin él.

Pero, ¿qué es realmente PL/SQL?

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

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

Como veremos, en PL/SQL podemos definir variables, constantes, funciones, procedimientos, capturar errores en tiempo de ejecución, anidar cualquier número de bloques, etc. como solemos hacer en cualquier otro lenguaje de programación. Además, por medio de PL/SQL programaremos los disparadores de nuestra base de datos, tarea que no podríamos hacer sólo con SQL.



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

Una de las grandes ventajas que nos ofrece PL/SQL es un mejor rendimiento en entornos de red cliente-servidor, ya que permite mandar bloques PL/SQL desde el cliente al servidor a través de la red, reduciendo de esta forma el tráfico y así no tener que mandar una a una las sentencias SQL correspondientes.

Para saber más

En el siguiente enlace podrás encontrar una breve historia de PL/SQL.

[Breve historia de PL/SQL.](#)

En estos enlaces podrás comprobar como los gestores de bases de datos incluyen hoy día un lenguaje procedimental para programar la base de datos muy parecido a PL/SQL.

[Procedimientos almacenados en MySQL.](#)

[Lenguaje procedimental en PostgreSQL.](#)

2.- Conceptos básicos.

Caso práctico

Juan ha avanzado muy rápido en la lectura del manual que le pasó **Ada**. **Juan** ya sabe programar en otros lenguajes de programación y por tanto la lectura de los primeros capítulos que se centran en cómo se estructura el lenguaje, los tipos de datos, las estructuras de control, etc. le han resultado muy fáciles de comprender. Sabe que lo único que tendrá que tener en cuenta son algunos aspectos en cuanto a las reglas de escritura y demás, pero la lógica es la de cualquier otro lenguaje de programación.



[INTEF \(CC BY-NC-SA\)](#)

Como **María** está un poco más verde en el tema de la programación, **Juan** se ha ofrecido a darle un repaso rápido a todo lo que él ha visto y a explicarle todo aquello en lo que tenga dudas y a ver si pronto se pueden poner manos a la obra con la base de datos de juegos on-line.

En este apartado nos vamos a ir introduciendo poco a poco en los diferentes conceptos que debemos tener claros para programar en PL/SQL. Como para cualquier otro lenguaje de programación, debemos conocer las reglas de sintaxis que podemos utilizar, los diferentes elementos de que consta, los tipos de datos de los que disponemos, las estructuras de control que nos ofrece (tanto iterativas como condicionales) y cómo se realiza el manejo de los errores.

Como podrás comprobar, es todo muy sencillo y pronto estaremos escribiendo fragmentos de código que realizan alguna tarea particular. ¡Vamos a ello!

Autoevaluación

Indica cuáles de las siguientes características que nos proporciona PL/SQL son ciertas.

- ☐ Permite reducir el tráfico en la red en entornos cliente-servidor.

- ☐ No podemos utilizar sentencias SQL dentro de un bloque PL/SQL.

- ☐ Nos ofrece las ventajas de SQL y la potencia de un lenguaje procedimental.

- ☐ Para utilizar PL/SQL debemos instalar diferentes drivers en nuestra base de datos Oracle.

Mostrar retroalimentación

Solución

1. Correcto
2. Incorrecto
3. Correcto
4. Incorrecto

2.1.- Unidades léxicas (I).

En este apartado nos vamos a centrar en conocer cuáles son las unidades léxicas que podemos utilizar para escribir código en PL/SQL. Al igual que en nuestra lengua podemos distinguir diferentes unidades léxicas como palabras, signos de puntuación, etc. En los lenguajes de programación también existen diferentes unidades léxicas que definen los elementos más pequeños que tienen sentido propio y que al combinarlos de manera adecuada, siguiendo las reglas de sintaxis, dan lugar a sentencias válidas sintácticamente.

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

Cada unidad léxica puede estar separada por espacios (debe estar separada por espacios si se trata de 2 identificadores), por saltos de línea o por tabuladores para aumentar la legibilidad del código escrito.

```
IF A=CLAVE THEN ENCONTRADO:=TRUE;ELSE ENCONTRADO:=FALSE;END IF;
```

Sería equivalente a escribir la siguiente línea:

```
if a=clave then encontrado:=true;else encontrado:=false;end if;
```

Y también sería equivalente a este otro fragmento que es más legible y facilita su comprensión.

```
IF a = clave THEN
    encontrado := TRUE;
ELSE
    encontrado := FALSE;
END IF;
```

Las unidades léxicas se pueden clasificar en:

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

Vamos a verlas más detenidamente.

Delimitadores.

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

Delimitadores en PL/SQL.

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

2.1.1.- Unidades léxicas (II).

Ya hemos visto qué son los delimitadores. Ahora vamos a continuar viendo el resto de unidades léxicas que nos podemos encontrar en PL/SQL.

Identificadores.

Los identificadores en PL/SQL, como en cualquier otro lenguaje de programación, son utilizados para nombrar elementos de nuestros programas. A la hora de utilizar los identificadores debemos tener en cuenta los siguientes aspectos:

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

Literales.

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

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

Comentarios.

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

En PL/SQL podemos utilizar dos tipos de comentarios:

- ✔ Los comentarios de una línea se expresan por medio del delimitador `--`. Ejemplo:

```
a:=b;  --asignación
```

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

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

Ejercicio resuelto

Dada la siguiente línea de código, haz su descomposición en las diferentes unidades léxicas que contenga.

```
IF A <> B  
THEN iguales := FALSE; --No son iguales
```

Mostrar retroalimentación

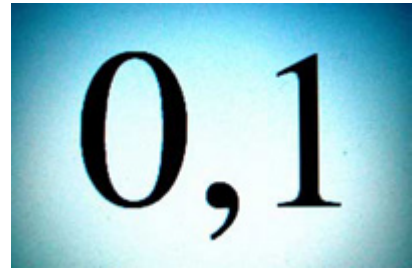
La descomposición en unidades léxicas sería la siguiente:

- ✓ Identificadores: A, B, iguales.
- ✓ Identificadores (palabras reservadas): IF, THEN.
- ✓ Delimitadores: <>, :=, ;.
- ✓ Comentarios: --No son iguales.

2.2.- Tipos de datos simples, variables y constantes.

En cualquier lenguaje de programación, las variables y las constantes tienen un tipo de dato asignado (bien sea explícitamente o implícitamente). Dependiendo del tipo de dato el lenguaje de programación sabrá la estructura que utilizará para su almacenamiento, las restricciones en los valores que puede aceptar, la precisión del mismo, etc.

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



[INTEE \(CC BY-NC-SA\)](#)

Numéricos.

- ✓ **BINARY_INTEGER**: Tipo de dato numérico cuyo rango es de -2147483647 .. 2147483647. PL/SQL además define algunos subtipos de éste: **NATURAL**, **NATURALN**, **POSITIVE**, **POSITIVEN**, **SIGNTYPE**.
- ✓ **NUMBER**: Tipo de dato numérico para almacenar números racionales. Podemos especificar su escala (-84 .. 127) y su precisión (1 .. 38). La escala indica cuándo se redondea y hacia dónde. Ejemplos. escala=2: 8.234 -> 8.23, escala=-3: 7689 -> 8000. Si se define un **NUMBER(6,2)** estamos indicando que son 6 dígitos numéricos de los cuales 2 son decimales. PL/SQL también define algunos subtipos como: **DEC**, **DECIMAL**, **DOUBLE PRECISION**, **FLOAT**, **INTEGER**, **INT**, **NUMERIC**, **REAL**, **SMALLINT**.
- ✓ **PLS_INTEGER**: Tipo de datos numérico cuyo rango es el mismo que el del tipo de dato **BINARY_INTEGER**, pero que su representación es distinta por lo que las operaciones aritméticas llevadas a cabo con los mismos serán más eficientes que en los dos casos anteriores.

Alfanuméricos.

- ✓ **CHAR(n)**: Array de n caracteres, máximo 2000 bytes. Si no especificamos longitud sería 1. Ocupa en memoria n caracteres.
- ✓ **LONG**: Array de caracteres con un máximo de 32760 bytes.
- ✓ **RAW**: Array de bytes con un número máximo de 2000.
- ✓ **LONG RAW**: Array de bytes con un máximo de 32760.
- ✓ **VARCHAR2(n)**: Tipo de dato para almacenar cadenas de longitud variable con un máximo de 32760. Ocupa en memoria hasta n caracteres.

Grandes objetos.

- ✓ **BFILE**: Puntero a un fichero del Sistema Operativo.
- ✓ **BLOB**: Objeto binario con una capacidad de 4 GB.
- ✓ **CLOB**: Objeto carácter con una capacidad de 2 GB.

Otros.

- ✓ **BOOLEAN**: **TRUE/FALSE**.
- ✓ **DATE**: Tipo de dato para almacenar valores día/hora desde el 1 enero de 4712 a.c. hasta el 31 diciembre de 4712 d.c.

Hemos visto los tipos de datos simples más usuales. Los tipos de datos compuestos los

dejaremos para posteriores apartados.

Atributo %TYPE.

Las variables PL/SQL se suelen declarar para sostener y manipular los datos almacenados en una Base de Datos. Cuando declares variables PL/SQL para guardar los valores de columnas, debes asegurarte de que la variable tenga la precisión (tamaño) y el tipo de dato correcto. Si no es así se puede producir un error durante la ejecución.

Cuando se utiliza el atributo %TYPE para declarar una variable estamos indicando que tiene el mismo tipo de datos que la variable especificada. Normalmente se relaciona con una columna de la BD indicando el nombre de la tabla de base de datos y el nombre de la columna. Si se hace referencia a una variable declarada anteriormente, se indicará el nombre de la variable previamente declarada a la variable por declarar.

Una ventaja del uso de este atributo, es que si cambia la definición de una columna de la BD, no será necesario cambiar la declaración de la variable.

Ejemplos:

```
DECLARE
```

```
    nombre_oficina oficinas.nombre%type; -- la variable nombre_oficina tomará
```

```
    vx number(5,2); -- Variable numérica de 5 dígitos, dos de los cuales son c
```

```
    vy vx%type; -- la variable vy tomará el mismo tipo de dato que tenga la v
```

Hemos visto los tipos de datos simples más usuales. Los tipos de datos compuestos los dejaremos para posteriores apartados.

Para saber más

En el siguiente enlace podrás ampliar información sobre los tipos de datos de los que disponemos en PL/SQL.

[Tipos de datos en PL/SQL.](#)

Autoevaluación

En PL/SQL cuando vamos a trabajar con enteros es preferible utilizar el tipo de dato `BINARY_INTEGER`, en vez de `PLS_INTEGER`.

- ☐ Verdadero.
- ☐ Falso.

No es correcto, ya que el tipo de dato `PLS_INTEGER` hace que nuestros programas sean más eficientes debido a la representación interna del mismo.

Efectivamente, nuestros programas serán más eficientes al utilizar este tipo de dato debido a su representación interna.

Solución

1. Incorrecto
2. Opción correcta

2.2.1.- Subtipos.

Cuántas veces no has deseado cambiarle el nombre a las cosas por alguno más común para ti. Precisamente, esa es la posibilidad que nos ofrece PL/SQL con la utilización de los subtipos.

PL/SQL nos permite definir subtipos de tipos de datos para darles un nombre diferente y así aumentar la legibilidad de nuestros programas. Los tipos de operaciones aplicables a estos subtipos serán las mismas que los tipos de datos de los que proceden. La sintaxis será:



[INTEF](#) (CC BY-NC-SA)

```
SUBTYPE subtipo IS tipo_base;
```

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

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

```
SUBTYPE id_familia IS familias.identificador%TYPE; -- Permite nombrar a la columna identifi
SUBTYPE agente IS agentes%ROWTYPE; -- Permite nombrar a las filas de la tabla agentes con e.
```

Los subtipos no podemos restringirlos, pero podemos usar un truco para conseguir el mismo efecto y es por medio de una variable auxiliar:

```
SUBTYPE apodo IS varchar2(20);           --ilegal
aux varchar2(20);
SUBTYPE apodo IS aux%TYPE;               --legal
```

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

```
DECLARE
    SUBTYPE numero IS NUMBER;
    numero_tres_digitos NUMBER(3);
    mi_numero_de_la_suerte numero;
    SUBTYPE encontrado IS BOOLEAN;
    SUBTYPE resultado IS BOOLEAN;
    lo_he_encontrado encontrado;
    resultado_busqueda resultado;
```

```

SUBTYPE literal IS CHAR;
SUBTYPE sentencia IS VARCHAR2;
literal_nulo literal;
sentencia_vacia sentencia;

BEGIN
    ...
    numero_tres_digitos := mi_numero_de_la_suerte;    --legal
    ...
    lo_he_encontrado := resultado_busqueda;           --legal
    ...
    sentencia_vacia := literal_nulo;                  --legal
    ...
END;
```

Autoevaluación

Indica la afirmación correcta.

- ☐ Los subtipos lo único que hacen es añadir complejidad a nuestros programas.
- ☐ No hay manera de restringir los subtipos con respecto a su tipo base.
- ☐ Podemos definir un subtipo cuyo tipo base sea una tabla de la base de datos.
- ☐ Podemos definir un subtipo de una variable pero no de una columna de la base de datos.

Incorrecto. Los subtipos añaden legibilidad y no complejidad.

No es cierto. Los subtipos podemos restringirlos con respecto a su tipo base de forma indirecta.

Efectivamente, veo que lo estás entendiendo.

No es correcto. Los subtipos pueden ser de una variable y también de una columna de la base de datos.

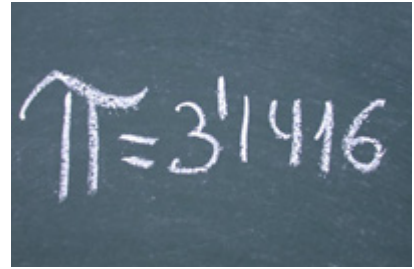
Solución

1. Incorrecto

2. Incorrecto
3. Opción correcta
4. Incorrecto

2.2.2.- Variables y constantes.

Llevamos un buen rato hablando de tipos de datos, variables e incluso de constantes y te estarás preguntando cuál es la forma adecuada de definir las. En este apartado vamos a ver las diferentes posibilidades a la hora de definir las y dejaremos para el apartado siguiente ver cuál es el lugar adecuado para hacerlo dentro de un bloque PL/SQL.



[INTEF \(CC BY-NC-SA\)](#)

Para declarar variables o constantes pondremos el nombre de la variable, seguido del tipo de datos y opcionalmente una asignación con el operador `:=`. Si es una constante antepondremos la palabra `CONSTANT` al tipo de dato (lo que querrá decir que no podemos cambiar su valor). Podremos sustituir el operador de asignación en las declaraciones por la palabra reservada `DEFAULT`. También podremos forzar a que no sea nula utilizando la palabra `NOT NULL` después del tipo y antes de la asignación. Si restringimos una variable con `NOT NULL` deberemos asignarle un valor al declararla, de lo contrario PL/SQL lanzará la excepción `VALUE_ERROR` (no te asustes que más adelante veremos lo que son las excepciones, pero como adelanto te diré que es un error en tiempo de ejecución).

```
id SMALLINT;  
hoy DATE := sysdate;  
pi CONSTANT REAL:= 3.1415;  
id SMALLINT NOT NULL; --ilegal, no está inicializada  
id SMALLINT NOT NULL := 9999; --legal  
nombre varchar2(30):= ;'Alejandro Magno';  
num INTEGER DEFAULT 4; -- también se puede utilizar DEFAULT para inicializar una variable
```

El alcance y la visibilidad de las variables en PL/SQL será el típico de los lenguajes estructurados basados en bloques, aunque eso lo veremos más adelante.

Conversión de tipos.

Aunque en PL/SQL existe la conversión implícita de tipos para tipos parecidos, siempre es aconsejable utilizar la conversión explícita de tipos por medio de funciones de conversión (`TO_CHAR`, `TO_DATE`, `TO_NUMBER`, ...) y así evitar resultados inesperados.

Precedencia de operadores.

Al igual que en nuestro lenguaje matemático se utiliza una precedencia entre operadores a la hora de realizar las operaciones aritméticas, en PL/SQL también se establece dicha precedencia para evitar confusiones. Si dos operadores tienen la misma precedencia, se evalúa de izquierda a derecha, pero lo aconsejable es utilizar los paréntesis (al igual que hacemos en nuestro lenguaje matemático) para alterar la precedencia de los mismos ya que las operaciones encerradas entre paréntesis tienen mayor precedencia. En la tabla siguiente se muestra la precedencia de los operadores de mayor a menor.

Precedencia de operadores.

| Operador. | Operación. |
|--|----------------------------------|
| ** , NOT | Exponenciación, negación lógica. |
| + , - | Identidad, negación. |
| * , / | Multiplicación, división. |
| + , - , | Suma, resta y concatenación. |
| = , != , < , > , <= , >= , IS NULL , LIKE , BETWEEN , IN | Comparaciones. |
| AND | Conjunción lógica |
| OR | Disyunción lógica. |

Autoevaluación

Rellena el hueco con el resultado de las siguientes operaciones.

✔ $5+3*2^{**}2$ es igual a: .

✔ $2^{**}3+6/3$ es igual a: .

✔ $2^{**}(3+6/3)$ es igual a: .

Enviar

2.3.- El bloque PL/SQL.

Ya hemos visto las unidades léxicas que componen PL/SQL, los tipos de datos que podemos utilizar y cómo se definen las variables y las constantes. Ahora vamos a ver la unidad básica en PL/SQL que es el bloque.

Un bloque PL/SQL consta de tres zonas:

- ✓ **Declaraciones:** definiciones de variables, constantes, cursores y excepciones.
- ✓ **Proceso:** zona donde se realizará el proceso en sí, conteniendo las sentencias ejecutables.
- ✓ **Excepciones:** zona de manejo de errores en tiempo de ejecución.

La sintaxis es la siguiente:

```
[DECLARE
    [Declaración de variables, constantes, cursores y excepciones]]
BEGIN
    [Sentencias ejecutables]
[EXCEPTION
    Manejadores de excepciones]
END;
```

Los bloques PL/SQL pueden anidarse a cualquier nivel. Como hemos comentado anteriormente el ámbito y la visibilidad de las variables es la normal en un lenguaje procedimental. Por ejemplo, en el siguiente fragmento de código se declara la variable `aux` en ambos bloques, pero en el bloque anidado `aux` con valor igual a 10 actúa de variable global y `aux` con valor igual a 5 actúa como variable local, por lo que en la comparación evaluaría a `FALSE`, ya que al tener el mismo nombre la visibilidad dominante sería la de la variable local.

```
DECLARE
    aux number := 10; -- Variable global
BEGIN
    DECLARE
        aux number := 5; -- Variable local al bloque donde es definida
    BEGIN
        ...
        IF aux = 10 THEN --evalúa a FALSE, no entraría
            ...
        END;
    END;
```

Para saber más

En el siguiente enlace podrás ampliar información sobre el ámbito y la visibilidad de las variables en PL/SQL.

[Ámbito y visibilidad en PL/SQL.](#)

Autoevaluación

En PL/SQL el bloque es la unidad básica, por lo que éstos no pueden anidarse.

- ☐ Verdadero.
- ☐ Falso.

No, los bloques se pueden anidar según nos convenga.

Efectivamente, veo que lo vas entendiendo.

Solución

1. Incorrecto
2. Opción correcta

2.4.- Estructuras de control (I).

En la vida constantemente tenemos que tomar decisiones que hacen que llevemos a cabo unas acciones u otras dependiendo de unas circunstancias o repetir una serie de acciones un número dado de veces o hasta que se cumpla una condición. En PL/SQL también podemos imitar estas situaciones por medio de las estructuras de control que son sentencias que nos permiten manejar el flujo de control de nuestro programa, y éstas son dos: condicionales e iterativas.

Control condicional.

Las estructuras de control condicional nos permiten llevar a cabo una acción u otra dependiendo de una condición. Prueba los ejemplos para entender bien el funcionamiento. El caracter / que aparece al final se utiliza para ejecutar el bloque desde SQLPlus.

SENTENCIA IF. Sus variantes son:

- ✔ **Sentencia IF-THEN:** Forma más simple de las sentencias de control condicional. Si la evaluación de la condición es **TRUE**, entonces se ejecuta la secuencia de sentencias encerradas entre el **THEN** y el final de la sentencia.

Sintaxis:

```
IF condicion THEN
secuencia_de_sentencias;
END IF;
```

Ejemplo:

```
SET SERVEROUTPUT ON
DECLARE a integer:=10;
B integer:=7;
BEGIN
IF a>b
  THEN dbms_output.put_line(a || ' es mayor'); -- Como la función put_line solo imprime un v;
END IF;
END;
/
```

- ✔ **Sentencia IF-THEN-ELSE:** Con esta forma de la sentencia ejecutaremos la primera secuencia de sentencias si la condición se evalúa a **TRUE** y en caso contrario ejecutaremos la segunda secuencia de sentencias.

Sintaxis:

```

IF condicion
THEN Secuencia_de_sentencias1;
  ELSE Secuencia_de_sentencias2;
END IF;

```

Ejemplo:

```

DECLARE
a integer:=10;
b integer:=17;
BEGIN
IF a>b THEN
  dbms_output.put_line(a || ' es mayor');
ELSE
  dbms_output.put_line(b || ' es mayor o iguales');
END IF;
END;
/

```

- ✔ **Sentencia IF-THEN-ELSIF:** Con esta última forma de la sentencia condicional podemos hacer una selección múltiple. Si la evaluación de la condición 1 da TRUE, ejecutamos la secuencia de sentencias 1, sino evaluamos la condición 2. Si esta evalúa a TRUE ejecutamos la secuencia de sentencias 2 y así para todos los ELSIF que haya. El último ELSE es opcional y es por si no se cumple ninguna de las condiciones anteriores.

Sintaxis:

```

IF condicion1 THEN
  Secuencia_de_sentencias1;
ELSIF condicion2 THEN
  Secuencia_de_sentencias2;
  ...
[ELSE
  Secuencia_de_sentencias;]
END IF;

```

Ejemplo:

```

IF (operacion = 'SUMA') THEN
  resultado := arg1 + arg2;
ELSIF (operacion = 'RESTA') THEN
  resultado := arg1 - arg2;
ELSIF (operacion = 'PRODUCTO') THEN
  resultado := arg1 * arg2;
ELSIF (arg2 <> 0) AND (operacion = 'DIVISION') THEN
  resultado := arg1 / arg2;

```

```

ELSE
    RAISE operacion_no_permitida; -- Lanza un error de ejecución
END IF;
/

```

SENTENCIA CASE

Permite representar n sentencias **IF** anidadas, y es más fácil de interpretar cuando se compara con varios valores permitiendo sustituir a las sentencias **IF** encadenadas.

Se puede utilizar de dos formas:

- ✔ Utilizando un selector y un manejador **WHEN** para cada posible valor de ese selector

```

CASE selector
    WHEN expression1 THEN
        ordenes;
    [ WHEN expression2 THEN
        ordenes;
    ...
    [WHEN expression THEN
        ordenes;]
    [ ELSE
        ordenes;]
END CASE ;

```

- ✔ Utilizando condiciones de búsqueda:

```

CASE
    WHEN condición1 THEN
        ordenes;
    [ WHEN condición2 THEN
        ordenes;
    ...
    WHEN condiciónN THEN
        ordenes;]
    [ ELSE
        ordenes;]
END CASE ;

```

En cualquiera de las dos formas, antes de terminar la sentencia **CASE** se puede especificar **ELSE** por si no se ha cumplido ninguna de las condiciones o el valor del selector no ha coincidido con ninguno de los evaluados. Las condiciones, condicion1 a condicionN, son analizadas en el mismo orden en que aparecen listadas y, en el momento en que una de estas condiciones se cumple como verdadera, la sentencia **CASE** devuelve el resultado correspondiente y deja de analizar el resto de condiciones.

Ejemplo:

```

DECLARE
    nota INTEGER:=8; -- Se podría especificar nota INTEGER:=&nota
BEGIN
    CASE
        WHEN nota in(1,2) THEN
            DBMS_OUTPUT.PUT_LINE('Muy deficiente');
        WHEN nota in (3,4) THEN
            DBMS_OUTPUT.PUT_LINE('Insuficiente');
        WHEN nota = 5 THEN
            DBMS_OUTPUT.PUT_LINE('Suficiente');
        WHEN nota=6 THEN
            DBMS_OUTPUT.PUT_LINE('Bien');
        WHEN nota in(7,8) THEN
            DBMS_OUTPUT.PUT_LINE('Notable');
        WHEN nota in (9,10) THEN
            DBMS_OUTPUT.PUT_LINE('Sobresaliente');
        ELSE
            DBMS_OUTPUT.PUT_LINE('Error, no es una nota');
        END CASE;
    END;
/

```

Para pedir datos por teclado en PL/SQL se utilizarán variables de sustitución escribiendo el caracter especial '&' y a continuación un identificador. Si el dato que se va a recibir es una cadena formada por caracteres alfanuméricos, se deben incluir las comillas al definir la expresión.

```

DECLARE
    cadena varchar2(25) :='&cad';
BEGIN
    DBMS_OUTPUT.PUT_LINE(cadena);
END;/

```

Autoevaluación

En PL/SQL no existen sentencias que nos permitan tomar una acción u otra dependiendo de una condición.

- ☐ Verdadero.
- ☐ Falso.

Incorrecto, deberías volver a leerte este apartado ya que es de lo que trata.

Efectivamente, para eso existen las sentencias de control condicional.

Solución

1. Incorrecto
2. Opción correcta

2.4.1.- Estructuras de control (II). Bucles

Ya que hemos visto las estructuras de control condicional, veamos ahora las estructuras de **control iterativo**.

Control iterativo.

Estas estructuras nos permiten ejecutar una secuencia de sentencias un determinado número de veces.

- ✓ **LOOP:** La forma más simple es el bucle infinito, cuya sintaxis es:

```
LOOP
    secuencia_de_sentencias;
END LOOP;
```

- ✓ **EXIT:** Con esta sentencia forzamos a un bucle a terminar y pasa el control a la siguiente sentencia después del bucle. Un **EXIT** no fuerza la salida de un bloque PL/SQL, sólo la salida del bucle.

```
LOOP
    ...
    IF encontrado = TRUE THEN
        EXIT;
    END IF;
END LOOP;
```

```
DECLARE
    a integer :=1;
BEGIN
    LOOP
        dbms_output.put_line(a);
        IF a>9 THEN
            EXIT;
        END IF;
        a:=a+1;
    END LOOP;
END;
/
```

- ✔ **EXIT WHEN** condicion: Fuerza a salir del bucle cuando se cumple una determinada condición.

```
LOOP
    ...
    EXIT WHEN encontrado;
END LOOP;
```

```
DECLARE
    a integer :=1;
BEGIN
    LOOP
        dbms_output.put_line(a);
        EXIT WHEN a>9;
        a:=a+1;
    END LOOP;
END;
/
```

- ✔ **WHILE LOOP**: Este tipo de bucle ejecuta la secuencia de sentencias mientras la condición sea cierta.

```
WHILE condicion LOOP
    Secuencia_de_sentencias;
END LOOP;
```

```
DECLARE
    a integer :=1;
BEGIN
    WHILE a<10 LOOP
        dbms_output.put_line(a);
        a:=a+1;
    END LOOP;
END;/
```

- ✔ **FOR LOOP**: Este bucle itera mientras el contador se encuentre en el rango definido.

```
FOR contador IN [REVERSE] limite_inferior..limite_superior LOOP
    Secuencia_de_sentencias;
END LOOP;
```

```
DECLARE
    a NUMBER;
```

```
BEGIN
FOR a IN 1..10 LOOP -- ascendente de uno en uno
    dbms_output.put_line(a);
END LOOP;
FOR a IN REVERSE 1..10 LOOP -- descendente de uno en uno
    dbms_output.put_line(a);
END LOOP;
END;
/
```

Autoevaluación

Al utilizar `REVERSE` en un bucle `FOR`, en el rango debemos poner el número mayor el primero y el menor el último.

- ☐ Verdadero.
- ☐ Falso.

No, el rango lo especificamos de misma forma, el menor el primero y el mayor el último, aunque luego se itere del mayor al menor.

Efectivamente, vas por buen camino.

Solución

1. Incorrecto
2. Opción correcta

2.5.- Manejo de errores (I).

Muchas veces te habrá pasado que surgen situaciones inesperadas con las que no contabas y a las que tienes que hacer frente. Pues cuando programamos con PL/SQL pasa lo mismo, que a veces tenemos que manejar errores debidos a situaciones diversas. Vamos a ver cómo tratarlos.

Cualquier situación de error es llamada **excepción** en PL/SQL. Cuando se detecta un error, una excepción es lanzada, es decir, la ejecución normal se para y el control se transfiere a la parte de manejo de excepciones. La parte de manejo de excepciones es la parte etiquetada como `EXCEPTION` y constará de sentencias para el manejo de dichas excepciones, llamadas **manejadores de excepciones**.



[Félix Vallés Calvo \(INTEF\)](#) (CC BY-NC-SA)

Manejadores de excepciones

Sintaxis:

```
WHEN nombre_excepcion THEN
    <sentencias para su manejo>
    ....
WHEN OTHERS THEN
    <sentencias para su manejo>
```

Ejemplo:

```
DECLARE
    supervisor agentes%ROWTYPE;
BEGIN
    SELECT * INTO supervisor FROM agentes
    WHERE categoria = 2 AND oficina = 3;
    ...
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        --Manejamos el no haber encontrado datos
    WHEN OTHERS THEN
        --Manejamos cualquier error inesperado
END;
/
```

La parte `OTHERS` captura cualquier excepción no considerada.

Las excepciones pueden estar definidas por el usuario o definidas internamente. Las excepciones predefinidas se lanzarán automáticamente asociadas a un error de Oracle. Las excepciones definidas por el usuario deberán definirse y lanzarse explícitamente.

En PL/SQL nosotros podemos definir nuestras propias excepciones en la parte `DECLARE` de

cualquier bloque. Estas excepciones podemos lanzarlas explícitamente por medio de la sentencia `RAISE nombre_excepción`.

Excepciones definidas por el usuario

Sintaxis:

```
DECLARE
    nombre_excepcion EXCEPTION;
BEGIN
    ...
    RAISE nombre_excepcion;
    ...
END;
```

Ejemplo:

```
DECLARE
    categoria_erronea EXCEPTION;
BEGIN
    ...
    IF categoria<0 OR categoria>3 THEN
        RAISE categoria_erronea;
    END IF;
    ...
EXCEPTION
    WHEN categoria_erronea THEN
        --manejamos la categoria errónea
END;
```

Debes conocer

En el siguiente enlace podrás ver las diferentes excepciones predefinidas en Oracle, junto a su código de error asociado (que luego veremos lo que es) y una explicación de cuándo son lanzadas.

[Excepciones predefinidas en Oracle.](#)

2.5.1.- Manejo de errores (II).

Ahora que ya sabemos lo que son las excepciones, cómo capturarlas y manejarlas y cómo definir y lanzar las nuestras propias. Es la hora de comentar algunos detalles sobre el uso de las mismas.

- ✓ El alcance de una excepción sigue las mismas reglas que el de una variable, por lo que si nosotros redefinimos una excepción que ya es global para el bloque, la definición local prevalecerá y no podremos capturar esa excepción a menos que el bloque en la que estaba definida esa excepción fuese un bloque nombrado, y podremos capturarla usando la sintaxis: `nombre_bloque.nombre_excepcion`.
- ✓ Las excepciones predefinidas están definidas globalmente. No necesitamos (ni debemos) redefinir las excepciones predefinidas.

```
DECLARE
    no_data_found EXCEPTION;
BEGIN
    SELECT * INTO ...
EXCEPTION
    WHEN no_data_found THEN      --captura la excepción local, no
                                --la global
END;
```

- ✓ Cuando manejamos una excepción no podemos continuar por la siguiente sentencia a la que la lanzó.

```
DECLARE
    ...
BEGIN
    ...
    INSERT INTO familias VALUES
(id_fam, nom_fam, NULL, oficina);
    INSERT INTO agentes VALUES
(id_ag, nom_ag, login, password, 0, 0, id_fam, NULL);
    ...
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        --manejamos la excepción debida a que el nombre de
        --la familia ya existe, pero no podemos continuar por
        --el INSERT INTO agentes, a no ser que lo pongamos
        --explícitamente en el manejador
END;
```

- ✔ Pero sí podemos encerrar la sentencia dentro de un bloque, y ahí capturar las posibles excepciones, para continuar con las siguientes sentencias.

```
DECLARE
    id_fam NUMBER;
    nom_fam VARCHAR2(40);
    oficina NUMBER;
    id_ag NUMBER;
    nom_ag VARCHAR2(60);
    usuario VARCHAR2(20);
    clave VARCHAR2(20);
BEGIN
    ...
    BEGIN
        INSERT INTO familias VALUES (id_fam, nom_fam, NULL, oficina);
    EXCEPTION
        WHEN DUP_VAL_ON_INDEX THEN
            SELECT identificador INTO id_fam FROM familias WHERE nombre = nom_fam;
    END;
    INSERT INTO agentes VALUES (id_ag, nom_ag, login, password, 1, 1, id_fam, null);
    ...
END;
```

Ejercicio resuelto

Supongamos que queremos reintentar una transacción hasta que no nos dé ningún error. Para ello deberemos encapsular la transacción en un bloque y capturar en éste las posibles excepciones. El bloque lo metemos en un bucle y así se reintentará la transacción hasta que sea posible llevarla a cabo.

Mostrar retroalimentación

```
DECLARE
    id_fam NUMBER;
    nombre VARCHAR2(40);
    oficina NUMBER;
BEGIN
    ...
    LOOP
        BEGIN
            SAVEPOINT inicio;
            INSERT INTO familias VALUES
(id_fam, nombre, NULL, oficina);
            ...
            COMMIT;
```



```
        EXIT;  
    EXCEPTION  
        WHEN DUP_VAL_ON_INDEX THEN  
            ROLLBACK TO inicio;  
            id_fam := id_fam + 1;  
    END;  
END LOOP;  
...  
END;
```

2.5.2.- Manejo de errores (III).

Continuemos viendo algunos detalles a tener en cuenta, relativos al uso de las excepciones.

- ✓ Cuando ejecutamos varias sentencias seguidas del mismo tipo y queremos capturar alguna posible excepción debida al tipo de sentencia, podemos encapsular cada sentencia en un bloque y manejar en cada bloque la excepción, o podemos utilizar una variable localizadora para saber qué sentencia ha sido la que ha lanzado la excepción (aunque de esta manera no podremos continuar por la siguiente sentencia).



[Elena Hervás \(INTEF\)](#) (CC BY-NC-SA)

```
DECLARE
    sentencia NUMBER := 0;
BEGIN
    ...
    SELECT * FROM agentes ...
    sentencia := 1;
    SELECT * FROM familias ...
    sentencia := 2;
    SELECT * FROM oficinas ...
    ...
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        IF sentencia = 0 THEN
            RAISE agente_no_encontrado;
        ELSIF sentencia = 1 THEN
            RAISE familia_no_encontrada;
        ELSIF sentencia = 2 THEN
            RAISE oficina_no_encontrada;
        END IF;
END;/
```

- ✓ Si la excepción es capturada por un manejador de excepción apropiado, ésta es tratada y posteriormente el control es devuelto al bloque superior. Si la excepción no es capturada y no existe bloque superior, el control se devolverá al entorno. También puede darse que la excepción sea manejada en un bloque superior a falta de manejadores para ella en los bloques internos, la excepción se propaga de un bloque al superior y así hasta que sea manejada o no queden bloques superiores con lo que el control se devuelve al entorno. Una excepción también puede ser relanzada en un manejador. En la siguiente presentación puedes ver cómo se propagan diferentes excepciones entre diferentes bloques.

[Propagación de excepciones en PL/SQL.](#) (odp - 17,90 KB)
[Resumen textual alternativo](#)

Autoevaluación

Todas las excepciones están predefinidas y nosotros no podemos definir nuevas excepciones.

- ☐ Verdadero.
- ☐ Falso.

No, también podemos definir nuestras propias excepciones.

Efectivamente, esta afirmación es falsa.

Solución

1. Incorrecto
2. Opción correcta

Las excepciones definidas por el usuario deben ser lanzadas explícitamente.

- ☐ Verdadero.
- ☐ Falso.

Efectivamente, las lanzamos explícitamente usando la sentencia **RAISE**.

No, deberías repasar los contenidos.

Solución

1. Opción correcta
2. Incorrecto

Es obligatorio declarar todas las excepciones predefinidas que vamos a usar en nuestros bloques.

- ☐ Verdadero.
- ☐ Falso.

No y además no es aconsejable hacerlo.

Efectivamente, ni es obligatorio ni debemos hacerlo.

Solución

1. Incorrecto
2. Opción correcta

2.5.3.- Manejo de errores (IV).

Oracle también permite que nosotros lancemos nuestros propios mensajes de error a las aplicaciones y asociarlos a un código de error que Oracle reserva, para no interferir con los demás códigos de error. Lo hacemos por medio del procedimiento:

```
RAISE_APPLICATION_ERROR(error_number, message [, (TRUE|FALSE)]);
```

Donde `error_number` es un entero negativo comprendido entre -20000..-20999 y `message` es una cadena que devolvemos a la aplicación. El tercer parámetro especifica si el error se coloca en la pila de errores (`TRUE`) o se vacía la pila y se coloca únicamente el nuestro (`FALSE`). Sólo podemos llamar a este procedimiento desde un subprograma.

No hay excepciones predefinidas asociadas a todos los posibles errores de Oracle, por lo que nosotros podremos asociar excepciones definidas por nosotros a errores Oracle, por medio de la directiva al compilador (o pseudoinstrucción):

```
PRAGMA_INIT( nombre_excepcion, error_Oracle )
```

Donde `nombre_excepcion` es el nombre de una excepción definida anteriormente, y `error_Oracle` es el número negativo asociado al error.

```
DECLARE
    no_null EXCEPTION;
    PRAGMA EXCEPTION_INIT(no_null, -1400);
    id familias.identificador%TYPE;
    nombre familias.nombre%TYPE;
BEGIN
    ...
    nombre := NULL;
    ...
    INSERT INTO familias VALUES (id, nombre, null, null);
EXCEPTION
    WHEN no_null THEN
        ...
END;
```

Oracle asocia 2 funciones para comprobar la ejecución de cualquier sentencia. `SQLCODE` nos devuelve el código de error y `SQLERRM` devuelve el mensaje de error asociado. Si una sentencia es ejecutada correctamente, `SQLCODE` nos devuelve 0 y en caso contrario devolverá un número negativo asociado al error (excepto `NO_DATA_FOUND` que tiene asociado el +100).

```
DECLARE
    cod number;
```

```
msg varchar2(100);  
BEGIN  
...  
EXCEPTION  
WHEN OTHERS THEN  
    cod := SQLCODE;  
    msg := SUBSTR(SQLERRM, 1, 1000);  
    INSERT INTO errores VALUES (cod, msg);  
END;
```

Autoevaluación

De las siguientes afirmaciones marca las que creas que son correctas.

- ☐ Podemos lanzar nuestros propios mensajes de error a las aplicaciones.
- ☐ Podemos acceder al código de error generado por la ejecución de una sentencia pero no a su mensaje asociado.
- ☐ Podemos asociar excepciones definidas por nosotros a códigos de error de Oracle.

Mostrar retroalimentación

Solución

1. Correcto
2. Incorrecto
3. Correcto

2.6.- Sentencias SQL en programas PL/SQL

En un bloque PL/SQL, se utilizan sentencias SQL para recuperar y modificar los datos de tablas de una BD.

Hasta ahora hemos trabajado con SQL de forma interactiva, escribíamos la sentencia y obteníamos los resultados en pantalla. Para trabajar con SQL dentro de un programa trabajaremos con SQL embebido o incrustado. Los datos devueltos de la consulta los guardaremos en variables y estructuras definidas en el lenguaje anfitrión, en nuestro caso PLSQL, para utilizarlas como convenga. Para ello añadiremos alguna cláusula a la sentencia `SELECT`.

PL/SQL soporta el Lenguaje de Manipulación de Datos (DML) y los comandos de control de transacciones. Se pueden utilizar los comandos DML para modificar los datos de una tabla de Base de Datos.

PL/SQL no soporta directamente el Lenguaje de Definición de Datos (DDL) como `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`, etc... Lo cual no puede suceder si las aplicaciones tienen que crear objetos de Base de Datos en tiempo de ejecución mediante el paso de valores. Las sentencias DDL no pueden ser ejecutadas directamente. Estas instrucciones son sentencias dinámicas de SQL. Las sentencias dinámicas de SQL se construyen como cadenas de caracteres en tiempo de ejecución y pueden contener marcadores de posición para parámetros. Por lo tanto, puede utilizar SQL dinámico para ejecutar sus sentencias DDL en PL/SQL.

PL/SQL no soporta directamente las sentencias de Lenguaje de Control de Datos (DCL), tales como `GRANT` o `REVOKE`. Puede utilizar SQL dinámico para ejecutarlas.

Si queremos recuperar datos de la BD utilizaremos la instrucción `SELECT` utilizaremos este formato.

Sintaxis:

```
SELECT lista_campos INTO {nombre_variable[, nombre_variable]...| nombre_registro}
FROM tabla
[WHERE condición];
```

donde

- la `lista_campos`, contendrá al menos una columna y puede incluir expresiones SQL, funciones de fila, o funciones de grupo.
- la cláusula `INTO` es obligatoria y se especifica entre `SELECT` y cláusula `FROM`.
- `nombre_variable`, variable donde se guardará el valor recuperado. Deben especificarse tantas variables como campos se indiquen en la lista de campos, debe haber correspondencia en posición y en tipo de datos.
- `nombre_registro`, un dato compuesto de PL/SQL donde se guardarán los valores recuperados

Las instrucciones `SELECT` dentro de un programa `PLSQL` deben devolver una sola fila. Una consulta que devuelve más de una fila o ninguna fila genera un error de tipo, respectivamente `TOO_MANY_ROWS` o `NO_DATA_FOUND`. Si queremos recuperar más de una fila necesitaremos una estructura de datos llamada cursor que se verá más adelante.

Las instrucciones `DML`, `INSERT`, `UPDATE` y `DELETE`, se ejecutan utilizando la misma sintaxis que en `SQL`.

3.- Tipos de datos compuestos.

Caso práctico

María, gracias a la ayuda de Juan, ha comprendido muy bien el manejo básico de PL/SQL. **Juan** le comenta que en la mayoría de los lenguajes de programación, además de los tipos de datos simples, existen tipos de datos compuestos que le dan mucha más versatilidad a los mismos y una gran potencia. **Juan** no conoce bien si PL/SQL dispone de estos tipos de datos y si dispone de ellos ¿de cuáles?



Ministerio de Educación (Uso educativo no)

María y **Juan** van a hablar con **Ada** para ver si les puede aclarar un poco la situación y dar unas pequeñas pautas para empezar. **Ada** les comenta que claro que PL/SQL cuenta con este tipo de datos, pero que hoy tiene una reunión importantísima y que tiene que terminar de preparársela, por lo que los remite al capítulo sobre tipos de datos compuestos del manual que les pasó. **Juan** y **María** le dicen que no se preocupe y que ya verá como a la vuelta de esa reunión son capaces de saber cuáles son e incluso de dominarlos. **Ada**, para motivarlos, les dice que si es así los invita a un aperitivo a su vuelta. Así que **Juan** y **María** se ponen con el capítulo de tipos de datos compuestos del manual para ver si pueden sorprender a **Ada** a su vuelta.

En el capítulo anterior, entre otras cosas, hemos conocido los tipos de datos simples con los que cuenta PL/SQL. Pero dependiendo de la complejidad de los problemas, necesitamos disponer de otras estructuras en las que apoyarnos para poder modelar nuestro problema. En este capítulo nos vamos a centrar en conocer los tipos de datos complejos que nos ofrece PL/SQL y cómo utilizarlos para así poder sacarle el mayor partido posible.

Citas para pensar

"Todo lo complejo puede dividirse en partes simples".

René Descartes.

3.1.- Registros.

El uso de los registros es algo muy común en los lenguajes de programación. PL/SQL también nos ofrece este tipo de datos. En este apartado veremos qué son y cómo definirlos y utilizarlos.

Un **registro** es un grupo de elementos relacionados, referenciados por un único nombre, almacenados en campos, cada uno de los cuales tiene su propio nombre y tipo de dato.

Por ejemplo, una dirección podría ser un registro con campos como calle, número, piso, puerta, código postal, ciudad, provincia y país. Los registros hacen que la información sea más fácil de organizar y representar. Para declarar un registro seguiremos la siguiente sintaxis:

```
TYPE nombre_tipo IS RECORD (decl_campo[, decl_campo] ...);
```

donde:

```
decl_campo := nombre tipo [[NOT NULL] {:=|DEFAULT} expresion]
```

El tipo del campo será cualquier tipo de dato válido en PL/SQL excepto `REF CURSOR`. La expresión será cualquier expresión que evalúe al tipo de dato del campo.

```
TYPE direccion IS RECORD
(
  calle          VARCHAR2(50),
  numero         INTEGER(4),
  piso           INTEGER(4),
  puerta         VARCHAR2(2),
  codigo_postal  INTEGER(5),
  ciudad         VARCHAR2(30),
  provincia      VARCHAR2(20),
  pais           VARCHAR2(20) := 'España'
);
mi_direccion direccion;
```

Para acceder a los campos usaremos el operador punto.

```
...
mi_direccion.calle := 'Ramirez Arellano';

mi_direccion.numero := 15;
```

...

Para asignar un registro a otro, éstos deben ser del mismo tipo, no basta que tengan el mismo número de campos y éstos emparejen uno a uno. Tampoco podemos comparar registros aunque sean del mismo tipo, ni tampoco comprobar si éstos son nulos. Podemos hacer `SELECT` en registros, pero no podemos hacer `INSERT` desde registros.

```
DECLARE
TYPE familia IS RECORD
(
    identificador    NUMBER,
    nombre           VARCHAR2(40),
    padre            NUMBER,
    oficina          NUMBER
);
TYPE familia_aux IS RECORD
(
    identificador    NUMBER,
    nombre           VARCHAR2(40),
    padre            NUMBER,
    oficina          NUMBER
);
SUBTYPE familia_fila IS familias%ROWTYPE; -- tendrá los mismos campos que tenga la tabla
mi_fam familia;
mi_fam_aux familia_aux;
mi_fam_fila familia_fila;
BEGIN
...
mi_fam := mi_fam_aux;                --ilegal
mi_fam := mi_fam_fila;                --legal
IF mi_fam IS NULL THEN ...           --ilegal
IF mi_fam = mi_fam_fila THEN ...     --ilegal
SELECT * INTO mi_fam FROM familias ... --legal
INSERT INTO familias VALUES (mi_fam_fila); --ilegal
...
END;/
```

Autoevaluación

Un registro se puede asignar a otro siempre que tenga el mismo número de campos y éstos emparejen uno a uno.

- ☐ Verdadero.
- ☐ Falso.

No, deben ser del mismo tipo para poder asignarlos.

Efectivamente, veo que lo vas entendiendo.

Solución

1. Incorrecto
2. Opción correcta

3.2.- Colecciones. Arrays de longitud variable.

Una colección es un grupo ordenado de elementos, todos del mismo tipo. Cada elemento tiene un subíndice único que determina su posición en la colección.

Una colección es un grupo ordenado de elementos, todos del mismo tipo. Cada elemento tiene un subíndice único que determina su posición en la colección.

En PL/SQL las colecciones sólo pueden tener una dimensión. PL/SQL ofrece 2 clases de colecciones: arrays de longitud variable y tablas anidadas.

Arrays de longitud variable.

Los elementos del tipo `VARRAY` son los llamados arrays de longitud variable. Son como los arrays de cualquier otro lenguaje de programación, pero con la salvedad de que a la hora de declararlos, nosotros indicamos su tamaño máximo y el array podrá ir creciendo dinámicamente hasta alcanzar ese tamaño. Un `VARRAY` siempre tiene un límite inferior igual a 1 y un límite superior igual al tamaño máximo.

Para declarar un `VARRAY` usaremos la sintaxis:

```
TYPE nombre IS {VARRAY | VARYING} (tamaño_máximo) OF tipo_elementos [NOT NULL];
```

Donde `tamaño_máximo` será un entero positivo y `tipo_elementos` será cualquier tipo de dato válido en PL/SQL, excepto `BINARY_INTEGER`, `BOOLEAN`, `LONG`, `LONG RAW`, `NATURAL`, `NATURALN`, `NCHAR`, `NCLOB`, `NVARCHAR2`, objetos que tengan como atributos `TABLE` o `VARRAY`, `PLS_INTEGER`, `POSITIVE`, `POSITIVEN`, `SIGNTYPE`, `STRING`, `TABLE`, `VARRAY`. Si `tipo_elementos` es un registro, todos los campos deberían ser de un tipo escalar.

Cuando definimos un `VARRAY`, éste es automáticamente nulo, por lo que para empezar a utilizarlo deberemos inicializarlo. Para ello podemos usar un constructor:

```
TYPE familias_hijas IS VARRAY(100) OF familia;
familias_hijas1 familias_hijas := familias_hijas( familia(100, 'Fam100', 10.
```

También podemos usar constructores vacíos.

```
familias_hijas2 familias_hijas := familias_hijas();
```

Para referenciar elementos en un **VARRAY** utilizaremos la sintaxis **nombre_colección(subíndice)**. Si una función devuelve un **VARRAY**, podemos usar la sintaxis: **nombre_funcion(lista_parametros)(subíndice)**.

```
IF familias_hijas1(i).identificador = 100 THEN ...
    IF dame_familias_hijas(10)(i).identificador = 100 THEN ...
```

Un **VARRAY** puede ser asignado a otro si ambos son del mismo tipo.

```
DECLARE

    TYPE tabla1 IS VARRAY(10) OF NUMBER;
    TYPE tabla2 IS VARRAY(10) OF NUMBER;
    mi_tabla1 tabla1 := tabla1();
    mi_tabla2 tabla2 := tabla2();
    mi_tabla tabla1 := tabla1();
BEGIN
    ...
    mi_tabla := mi_tabla1;          --legal
    mi_tabla1 := mi_tabla2;        --ilegal
    ...
END;
```

Para extender un **VARRAY** usaremos el método **EXTEND**. Sin parámetros, extendemos en 1 elemento nulo el **VARRAY**. **EXTEND(n)** añade n elementos nulos al **VARRAY** y **EXTEND(n,i)** añade n copias del i-ésimo elemento.

COUNT nos dirá el número de elementos del **VARRAY**. **LIMIT** nos dice el tamaño máximo del **VARRAY**. **FIRST** siempre será 1. **LAST** siempre será igual a **COUNT**. **PRIOR** y **NEXT** devolverá el antecesor y el sucesor del elemento.

Al trabajar con **VARRAY** podemos hacer que salte alguna de las siguientes excepciones, debidas a un mal uso de los mismos: **COLECTION_IS_NULL**, **SUBSCRIPT_BEYOND_COUNT**, **SUBSCRIPT_OUTSIDE_LIMIT** y **VALUE_ERROR**.

Ejemplos de uso de los **VARRAY**.

✔ Extender un **VARRAY**.

```
DECLARE

    TYPE tab_num IS VARRAY(10) OF NUMBER;
    mi_tab tab_num;
BEGIN
    mi_tab := tab_num();
    FOR i IN 1..10 LOOP
        mi_tab.EXTEND;
        mi_tab(i) := calcular_elemento(i);
    END LOOP;
```

END;

- Consultar propiedades VARRAY.

```
DECLARE
    TYPE numeros IS VARRAY(20) OF NUMBER;
    tabla_numeros numeros := numeros();
    num NUMBER;
BEGIN
    num := tabla_numeros.COUNT; --num := 0
    FOR i IN 1..10 LOOP
        tabla_numeros.EXTEND;
        tabla_numeros(i) := i;
    END LOOP;
    num := tabla_numeros.COUNT; --num := 10
    num := tabla_numeros.LIMIT; --num := 20
    num := tabla_numeros.FIRST; --num := 1;
    num := tabla_numeros.LAST; --num := 10;
    ...
END;
```

- Posibles excepciones.

```
DECLARE
    TYPE numeros IS VARRAY(20) OF INTEGER;
    v_numeros numeros := numeros( 10, 20, 30, 40 );
    v_enteros numeros;
BEGIN
    v_enteros(1) := 15; --lanzaría COLECTION_IS_NULL
    v_numeros(5) := 20; --lanzaría SUBSCRIPT_BEYOND_COUNT
    v_numeros(-1) := 5; --lanzaría SUBSCRIPT_OUTSIDE_LIMIT v_numeros('A') := 25; --
    lanzaría VALUE_ERROR
    ....
END;
```

Autoevaluación

Indica, de entre las siguientes, cuál es la afirmación correcta referida a VARRAY.

- ☐ Un VARRAY no hace falta inicializarlo.
- ☐ COUNT y LIMIT siempre nos devolverán el mismo valor.
- ☐ LAST y COUNT siempre nos devolverán el mismo valor.

Incorrecto, sí es necesario inicializar un `VARRAY` para poder empezar a utilizarlo.

No, esta afirmación será cierta en el caso en el que el `VARRAY` esté lleno totalmente.

Efectivamente, estás en lo cierto.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

3.2.1.- Colecciones. Tablas anidadas.

Las tablas anidadas son colecciones de elementos, que no tienen límite superior fijo, y pueden aumentar dinámicamente su tamaño. Además podemos borrar elementos individuales.

Para declararlos utilizaremos la siguiente sintaxis:

```
TYPE nombre IS TABLE OF tipo_elementos [NOT NULL];
```

Donde `tipo_elementos` tendrá las mismas restricciones que para los `VARRAY`.

Al igual que pasaba con los `VARRAY`, al declarar una tabla anidada, ésta es automáticamente nula, por lo que deberemos inicializarla antes de usarla.

```
TYPE hijos IS TABLE OF agente;  
hijos_fam hijos := hijos( agente(...) ...);
```

También podemos usar un constructor nulo.

Para referenciar elementos usamos la misma sintaxis que para los `VARRAY`.

Para extender una tabla usamos `EXTEND` exactamente igual que para los `VARRAY`. `COUNT` nos dirá el número de elementos, que no tiene por qué coincidir con `LAST`. `LIMIT` no tiene sentido y devuelve `NULL`. `EXISTS(n)` devuelve `TRUE` si el elemento existe, y `FALSE` en otro caso (el elemento ha sido borrado). `FIRST` devuelve el primer elemento que no siempre será 1, ya que hemos podido borrar elementos del principio. `LAST` devuelve el último elemento. `PRIOR` y `NEXT` nos dicen el antecesor y sucesor del elemento (ignorando elementos borrados). `TRIM` sin argumentos borra un elemento del final de la tabla. `TRIM(n)` borra `n` elementos del final de la tabla. `TRIM` opera en el tamaño interno, por lo que si encuentra un elemento borrado con `DELETE`, lo incluye para ser eliminado de la colección. `DELETE(n)` borra el `n`-ésimo elemento. `DELETE(n, m)` borra del elemento `n` al `m`. Si después de hacer `DELETE`, consultamos si el elemento existe nos devolverá `FALSE`.

Al trabajar con tablas anidadas podemos hacer que salte alguna de las siguientes excepciones, debidas a un mal uso de las mismas: `COLECTION_IS_NULL`, `NO_DATA_FOUND`, `SUBSCRIPT_BEYOND_COUNT` y `VALUE_ERROR`.

Ejemplos de uso de las tablas anidadas.

Diferentes operaciones sobre tablas anidadas.

```
DECLARE  
TYPE numeros IS TABLE OF NUMBER;  
tabla_numeros numeros := numeros();  
num NUMBER;  
BEGIN
```

```

num := tabla_numeros.COUNT;      --num := 0
FOR i IN 1..10 LOOP
    tabla_numeros.EXTEND;
    tabla_numeros(i) := i;
END LOOP;
num := tabla_numeros.COUNT;      --num := 10
tabla_numeros.DELETE(10);
num := tabla_numeros.LAST;       --num := 9
num := tabla_numeros.FIRST;      --num := 1
tabla_numeros.DELETE(1);
num := tabla_numeros.FIRST;      --num := 2
FOR i IN 1..4 LOOP
    tabla_numeros.DELETE(2*i);
END LOOP;
num := tabla_numeros.COUNT;      --num := 4
num := tabla_numeros.LAST;       --num := 9
...
END;

```

Posibles excepciones en su uso.

```

DECLARE
TYPE numeros IS TABLE OF NUMBER;
tabla_num numeros := numeros();
tabla1 numeros;
BEGIN
tabla1(5) := 0;      --lanzaría COLLECTION_IS_NULL
tabla_num.EXTEND(5);
tabla_num.DELETE(4);
tabla_num(4) := 3;   --lanzaría NO_DATA_FOUND
tabla_num(6) := 10;  --lanzaría SUBSCRIPT_BEYOND_COUNT
tabla_num(-1) := 0;  --lanzaría SUBSCRIPT_OUTSIDE_LIMIT
tabla_num('y') := 5;--lanzaría VALUE_ERROR
END;

```

Autoevaluación

Las tablas anidadas podemos hacer que crezcan dinámicamente, pero no podemos borrar elementos.

- ☐ Verdadero.
- ☐ Falso.

No, en las tablas anidadas también podemos borrar elementos.

¡Muy bien!

Solución

1. Incorrecto
2. Opción correcta

3.3.- Cursores.

En los apartados anteriores hemos visto algunos tipos de datos compuestos cuyo uso es común en otros lenguajes de programación. Sin embargo, en este apartado vamos a ver un tipo de dato, que aunque se puede asemejar a otros que ya conozcas, su uso es exclusivo en la programación de las bases de datos y que es el **cursor**.

Un **cursor** no es más que una estructura que almacena el conjunto de filas devuelto por una consulta a la base de datos.

Oracle usa áreas de trabajo para ejecutar sentencias SQL y almacenar la información procesada. Hay 2 clases de cursores: **implícitos** y **explícitos**. PL/SQL declara implícitamente un cursor para todas las sentencias SQL de manipulación de datos, incluyendo consultas que devuelven una sola fila. Para las consultas que devuelven más de una fila, se debe declarar explícitamente un cursor para procesar las filas individualmente.

En este primer apartado vamos a hablar de los cursores implícitos y de los atributos de un cursor (estos atributos tienen sentido con los cursores explícitos, pero los introducimos aquí para ir abriendo boca), para luego pasar a ver los cursores explícitos y terminaremos hablando de los cursores variables.

Cursores implícitos.

Oracle abre implícitamente un cursor para procesar cada sentencia SQL que no esté asociada con un cursor declarado explícitamente.

Con un cursor implícito no podemos usar las sentencias **OPEN**, **FETCH** y **CLOSE** para controlar el cursor. Pero sí podemos usar los atributos del cursor para obtener información sobre las sentencias SQL más recientemente ejecutadas.

Atributos de un cursor.

Cada cursor tiene 4 atributos que podemos usar para obtener información sobre la ejecución del mismo o sobre los datos. Estos atributos pueden ser usados en PL/SQL, pero no en SQL. Aunque estos atributos se refieren en general a cursores explícitos y tienen que ver con las operaciones que hayamos realizado con el cursor, es deseable comentarlas aquí y en el siguiente apartado tomarán pleno sentido.

- ✔ **%FOUND**: Después de que el cursor esté abierto y antes del primer **FETCH**, **%FOUND** devuelve **NULL**. Después del primer **FETCH**, **%FOUND** devolverá **TRUE** si el último **FETCH** ha devuelto una fila y **FALSE** en caso contrario. Para cursores implícitos **%FOUND** devuelve **TRUE** si un **INSERT**, **UPDATE** o **DELETE** afectan a una o más de una fila, o un **SELECT ... INTO ...** devuelve una o más filas. En otro caso **%FOUND** devuelve **FALSE**.
- ✔ **%NOTFOUND**: Es lógicamente lo contrario a **%FOUND**.
- ✔ **%ISOPEN**: Evalúa a **TRUE** si el cursor está abierto y **FALSE** en caso contrario. Para cursores implícitos, como Oracle los cierra automáticamente, **%ISOPEN** evalúa siempre a **FALSE**.
- ✔ **%ROWCOUNT**: Para un cursor abierto y antes del primer **FETCH**, **%ROWCOUNT** evalúa a 0. Después de cada **FETCH**, **%ROWCOUNT** es incrementado y evalúa al número de filas que hemos procesado. Para cursores implícitos **%ROWCOUNT** evalúa al número de filas afectadas por un **INSERT**, **UPDATE** o **DELETE** o el número de filas devueltas por un **SELECT ... INTO ...**

Debes conocer

Aunque todavía no hemos visto las operaciones que se pueden realizar con un cursor explícito, es conveniente que te vayas familiarizando con la evaluación de sus atributos según las operaciones que hayamos realizado con el cursor y que tomarán pleno sentido cuando veamos el siguiente apartado.

[Evaluación de los atributos de un cursor explícito.](#)

3.3.1.- Cursores explícitos.

Cuando una consulta devuelve múltiples filas, debemos declarar explícitamente un cursor para procesar las filas devueltas. Cuando declaramos un cursor, lo que hacemos es darle un nombre y asociarle una consulta usando la siguiente sintaxis:

```
CURSOR nombre_cursor [(parametro [, parametro] ...)] [RETURN tipo_devuelto] IS sentencia_sel;
```

Donde `tipo_devuelto` debe representar un registro o una fila de una tabla de la base de datos, y `parámetro` sigue la siguiente sintaxis:

```
parametro := nombre_parametro [IN] tipo_dato [{:= | DEFAULT} expresion]
```

Ejemplos:

```
CURSOR cAgentes IS SELECT * FROM agentes;  
CURSOR cFamilias RETURN familias%ROWTYPE IS SELECT * FROM familias WHERE ...
```

Además, como hemos visto en la declaración, un cursor puede tomar parámetros, los cuales pueden aparecer en la consulta asociada como si fuesen constantes. Los parámetros serán de entrada, un cursor no puede devolver valores en los parámetros actuales. A un parámetro de un cursor no podemos imponerle la restricción `NOT NULL`.

```
CURSOR c1 (cat INTEGER DEFAULT 0) IS SELECT * FROM agentes WHERE categoria = cat;
```

Cuando abrimos un cursor, lo que se hace es ejecutar la consulta asociada e identificar el conjunto resultado, que serán todas las filas que emparejen con el criterio de búsqueda de la consulta. Para abrir un cursor usamos la sintaxis:

```
OPEN nombre_cursor [(parametro [, parametro] ...)];
```

Ejemplos:

```
OPEN cAgentes;  
OPEN c1(1);  
OPEN c1;
```

La sentencia `FETCH` devuelve una fila del conjunto resultado. Después de cada `FETCH`, el cursor avanza a la próxima fila en el conjunto resultado.

```
FETCH cFamilias INTO mi_id, mi_nom, mi_fam, mi_ofi;
```

Para cada valor de columna devuelto por la consulta `SELECT` asociada al cursor, debe haber una variable que se corresponda en la lista de variables después del `INTO`.

Para procesar un cursor entero deberemos hacerlo por medio de un bucle.

Hay varias formas de hacerlo. En el formato del ejemplo siguiente es necesario abrir el cursor previamente y tras recorrerlo cerrarlo.

```
BEGIN
...
OPEN cFamilias;
LOOP
    FETCH cFamilias INTO mi_id, mi_nom, mi_fam, mi_ofi;
    EXIT WHEN cFamilias%NOTFOUND;
    ...
END LOOP;
CLOSE cFamilias;
...
END;
```

Una vez cerrado el cursor podemos reabrirlo, pero cualquier otra operación que hagamos con el cursor cerrado lanzará la excepción `INVALID_CURSOR`.

Otra forma más sencilla es utilizar los bucles para cursores, los cuales declaran implícitamente una variable índice definida como `%ROWTYPE` para el cursor, abren el cursor, extraen los valores de cada fila del cursor, almacenándolas en la variable índice, y finalmente cierran el cursor.

```
BEGIN
...
FOR cFamilias_rec IN cFamilias LOOP
    --Procesamos las filas accediendo a
    --cFamilias_rec.identificador, cFamilias_rec.nombre,
    --cFamilias_rec.familia, ...
END LOOP;
...
END;
```

Autoevaluación

En PL/SQL los cursores son abiertos al definirlos.

- ☐ Verdadero.
- ☐ Falso.

No, un cursor se debe abrir por medio de la sentencia `OPEN`.

Efectivamente, debemos abrirlos por medio de la sentencia `OPEN`.

Solución

1. Incorrecto
2. Opción correcta

3.3.2.- Cursores variables.

Oracle, además de los cursores vistos anteriormente, nos permite definir cursores variables que son como punteros a cursores, por lo que podemos usarlos para referirnos a cualquier tipo de consulta. Los cursores serían estáticos y los cursores variables serían dinámicos.



[Abraham Pérez Pérez \(INTEF\)](#) (CC BY-NC-SA)

Para declarar un cursor variable debemos seguir 2 pasos:

- ✓ Definir un tipo `REF CURSOR` y entonces declarar una variable de ese tipo.

```
TYPE tipo_cursor IS REF CURSOR RETURN agentes%ROWTYPE;
cAgentes tipo_cursor;
```

- ✓ Una vez definido el cursor variable debemos asociarlo a una consulta (notar que esto no se hace en la parte declarativa, sino dinámicamente en la parte de ejecución) y esto lo hacemos con la sentencia `OPEN-FOR` utilizando la siguiente sintaxis:

```
OPEN nombre_variable_cursor FOR sentencia_select;
OPEN cAgentes FOR SELECT * FROM agentes WHERE oficina = 1;
```

Un cursor variable no puede tomar parámetros. Podemos usar los atributos de los cursores para cursores variables.

Además, podemos usar varios `OPEN-FOR` para abrir el mismo cursor variable para diferentes consultas. No necesitamos cerrarlo antes de reabrirlo. Cuando abrimos un cursor variable para una consulta diferente, la consulta previa se pierde.

Una vez abierto el cursor variable, su manejo es idéntico a un cursor. Usaremos `FETCH` para traernos las filas, usaremos sus atributos para hacer comprobaciones y lo cerraremos cuando dejemos de usarlo.

```
DECLARE
TYPE cursor_Agentes IS REF CURSOR RETURN agentes%ROWTYPE;
cAgentes cursor_Agentes;
agente cAgentes%ROWTYPE;
BEGIN
...
OPEN cAgentes FOR SELECT * FROM agentes WHERE oficina = 1;
LOOP
    FETCH cAgentes INTO agente;
    EXIT WHEN cAgentes%NOTFOUND;
    ...
END LOOP;
CLOSE cAgentes;
...
```

END;

También puede utilizarse el bucle `FOR` con cursores variable definidos dentro del mismo bucle:

```
for va_cursor in (select ...) loop
...
end loop;
```

Autoevaluación

A los cursores variables no podemos pasarles parámetros al abrirlos.

- ☐ Verdadero.
- ☐ Falso.

Correcto, veo que lo vas entendiendo.

No, a este tipo de cursores no podemos pasarles parámetros al abrirlos.

Solución

1. Opción correcta
2. Incorrecto

Los cursores variables se abren exactamente igual que los cursores explícitos.

- ☐ Verdadero.
- ☐ Falso.

No, al abrir un cursor variable debemos asociarle la consulta por medio de la sentencia `OPEN-FOR`.

Correcto, ya que debemos abrirlo por medio de la sentencia `OPEN-FOR` con la que le asociamos la consulta.

Solución

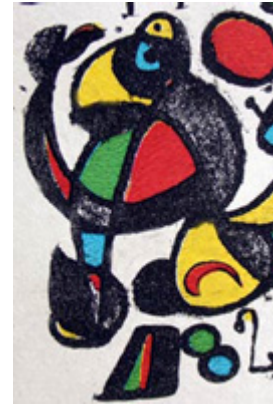
1. Incorrecto
2. Opción correcta

4.- Abstracción en PL/SQL.

Caso práctico

María, gracias a la ayuda de **Juan**, tiene bastante claro cómo programar en PL/SQL pero no entiende muy bien cómo integrar todo esto con la base de datos de juegos on-line. Sabe que puede utilizar unos tipos de datos, que hay unas estructuras de control y que se pueden manejar los errores que surjan, pero lo que no sabe es cómo y donde utilizar todo eso.

Juan le explica que lo que hasta ahora ha aprendido es el comienzo, pero que ahora viene lo bueno y que será donde le va a encontrar pleno sentido a lo aprendido anteriormente. Le explica que PL/SQL permite crear funciones y procedimientos y además agruparlos en paquetes y que eso será lo que realmente van a hacer con la base de datos de juegos on-line. Deberán ver qué es lo que utilizan más comúnmente e implementarlo en PL/SQL utilizando funciones y procedimientos según convenga. **Juan** la tranquiliza y le dice que lo primero que va a hacer es explicarle cómo se escriben dichas funciones y procedimientos y luego pasarán a implementar alguno y que así verá la potencia real de PL/SQL. **María** se queda más tranquila y está deseando implementar esa primera función o procedimiento que le resolverá la gran duda que tiene.



[INTEE \(CC BY-NC-SA\)](#)

Hoy día cualquier lenguaje de programación permite definir diferentes grados de abstracción en sus programas. La abstracción permite a los programadores crear unidades lógicas y posteriormente utilizarlas pensando en qué hace y no en cómo lo hace. La abstracción se consigue utilizando funciones, procedimientos, librerías, objetos, etc.

PL/SQL nos permite definir funciones y procedimientos. Además nos permite agrupar todas aquellas que tengan relación en paquetes. También permite la utilización de objetos. Todo esto es lo que veremos en este apartado y conseguiremos darle modularidad a nuestras aplicaciones, aumentar la reusabilidad y mantenimiento del código y añadir grados de abstracción a los problemas.

Para saber más

En los siguientes enlaces puedes ampliar información sobre la abstracción en programación.

Abstracción en los lenguajes de programación.

Programación orientada a objetos.

4.1.- Subprogramas.

Los **subprogramas** son bloques de código PLSQL, referenciados bajo un nombre, que realizan una acción determinada. Les podemos pasar parámetros y los podemos invocar. Además, los subprogramas pueden estar almacenados en la base de datos o estar encerrados en otros bloques. Si el programa está almacenado en la base de datos, podremos invocarlo si tenemos permisos suficientes y si está encerrado en otro bloque lo podremos invocar si tenemos visibilidad sobre el mismo.

Hay dos clases de subprogramas: las funciones y los procedimientos. Las funciones devuelven un valor y los procedimientos no.

Para declarar un subprograma utilizamos la siguiente sintaxis:

Sintaxis de Funciones.

```
FUNCTION nombre [(parametro [, parametro] ...)]  
    RETURN tipo_dato IS  
    [declaraciones_locales]  
BEGIN  
    sentencias_ejecutables  
[EXCEPTION  
    manejadores_de_excepciones]  
END [nombre];
```

Sintaxis de Procedimientos.

```
PROCEDURE nombre [( parametro [, parametro] ... )] IS  
    [declaraciones_locales]  
BEGIN  
    sentencias_ejecutables  
[EXCEPTION manejadores_de_excepciones]  
END [nombre];
```

Donde:

```
parametro := nombre_parametro [IN|OUT|IN OUT] tipo_dato [{:=|DEFAULT} expresion]
```

Algunas consideraciones que debes tener en cuenta son las siguientes:

- ✔ No podemos imponer una restricción **NOT NULL** a un parámetro.
- ✔ No podemos especificar una restricción del tipo:

```
PROCEDURE KK(a NUMBER(10)) IS ...          --ilegal
```

- ✔ Una función siempre debe acabar con la sentencia **RETURN**.

Podemos definir subprogramas al final de la parte declarativa de cualquier bloque. En Oracle, cualquier identificador debe estar declarado antes de usarse, y eso mismo pasa con los subprogramas, por lo que deberemos declararlos antes de usarlos.

```
DECLARE
hijos NUMBER;
FUNCTION hijos_familia( id_familia NUMBER )
    RETURN NUMBER IS
    hijos NUMBER;
BEGIN
    SELECT COUNT(*) INTO hijos FROM agentes
        WHERE familia = id_familia;
    RETURN hijos;
END hijos_familia;
BEGIN
...
END;
```

Si quisiéramos definir subprogramas en orden alfabético o lógico, o necesitamos definir subprogramas mutuamente recursivos (uno llama a otro, y éste a su vez llama al anterior), deberemos usar la definición hacia delante, para evitar errores de compilación.

```
DECLARE
    PROCEDURE calculo(...);          --declaración hacia delante
    --Definimos subprogramas agrupados lógicamente
    PROCEDURE inicio(...) IS
    BEGIN
        ...
        calculo(...);
        ...
    END;
    ...
BEGIN
    ...
END;
```

Autoevaluación

Una función siempre debe devolver un valor.

- ☐ Verdadero.
- ☐ Falso.

Efectivamente, una función obligatoriamente debe devolver un valor.

Incorrecto, creo que debería repasar este apartado.

Solución

1. Opción correcta
2. Incorrecto

En PL/SQL no podemos definir subprogramas mutuamente recursivos.

- ☐ Verdadero.
- ☐ Falso.

Incorrecto ya que podemos hacerlo usando la definición hacia delante.

¡Muy bien!

Solución

1. Incorrecto
2. Opción correcta

4.1.1.- Almacenar subprogramas en la base de datos.

Para almacenar un subprograma en la base de datos utilizaremos la misma sintaxis que para declararlo, anteponiendo `CREATE [OR REPLACE]` a `PROCEDURE` o `FUNCTION`, y finalizando el subprograma con una línea que simplemente contendrá el carácter `'/'` para indicarle a Oracle que termina ahí. Si especificamos `OR REPLACE` y el subprograma ya existía, éste será reemplazado. Si no lo especificamos y el subprograma ya existe, Oracle nos devolverá un error indicando que el nombre ya está siendo utilizado por otro objeto de la base de datos.

```
CREATE OR REPLACE FUNCTION hijos_familia(id_familia NUMBER)
    RETURN NUMBER IS hijos NUMBER;
BEGIN
    SELECT COUNT(*) INTO hijos FROM agentes
        WHERE familia = id_familia;
RETURN hijos;
END;
/
```

Cuando los subprogramas son almacenados en la base de datos, para ellos no podemos utilizar las declaraciones hacia delante, por lo que cualquier subprograma almacenado en la base de datos deberá conocer todos los subprogramas que utilice.

Para invocar un subprograma usaremos la sintaxis:

```
nombre_procedimiento [(parametro [,parametro] ...)];
variable := nombre_funcion [(parametro [, parametro] ...)];
BEGIN
...
    hijos := hijos_familia(10);
...
END;
```

Si el subprograma está almacenado en la base de datos y queremos invocarlo desde SQL*Plus usaremos la sintaxis:

```
EXECUTE nombre_procedimiento [(parametros)];
EXECUTE :variable_sql := nombre_funcion [(parametros)];
```

Cuando almacenamos un subprograma en la base de datos éste es compilado antes. Si hay algún error se nos informará de los mismos y deberemos corregirlos creándolo de nuevo por medio de la cláusula `OR REPLACE`, antes de que el subprograma pueda ser utilizado.

Hay varias vistas del diccionario de datos que nos ayudan a llevar un control de los subprogramas, tanto para ver su código, como los errores de compilación. También hay algunos comandos de SQL*Plus que nos ayudan a hacer lo mismo pero de forma algo menos engorrosa.

El comando SQLPlus `show errors` tras la compilación de un procedimiento o función nos muestra los errores si los hay.

Vistas y comandos asociados a los subprogramas.

| Información almacenada. | Vista del diccionario. | Comando. |
|-------------------------|------------------------|-------------|
| Código fuente. | USER_SOURCE | DESCRIBE |
| Errores de compilación. | USER_ERRORS | SHOW ERRORS |
| Ocupación de memoria. | USER_OBJECT_SIZE | |

También existe la vista `USER_OBJECTS` de la cual podemos obtener los nombres de todos los subprogramas almacenados.

Debes conocer

En la siguiente presentación te mostramos las vistas relacionadas con los subprogramas, la compilación de un subprograma con errores y cómo mostrar los mismos, la compilación de un subprograma correctamente y cómo mostrar su código fuente y la ejecución de un subprograma desde SQL*Plus y desde SQLDeveloper

[Presentación Subprogramas](#) (odp - 315,90 KB)
[Resumen textual alternativo](#)

Autoevaluación

Una vez que hemos almacenado un subprograma en la base de datos podemos consultar su código mediante la vista `USER_OBJECTS`.

- ☐ Verdadero.
- ☐ Falso.

No, podemos hacerlo mediante la vista `USER_SOURCE`.

Sí, lo estás captando a la primera.

Solución

1. Incorrecto
2. Opción correcta

4.1.2.- Parámetros de los subprogramas.

Ahora vamos a profundizar un poco más en los parámetros que aceptan los subprogramas y cómo se los podemos pasar a la hora de invocarlos.

Las variables pasadas como parámetros a un subprograma son llamadas **parámetros actuales**. Las variables referenciadas en la especificación del subprograma como parámetros, son llamadas **parámetros formales**.

Cuando llamamos a un subprograma, los parámetros actuales podemos escribirlos utilizando notación posicional o notación nombrada, es decir, la asociación entre parámetros actuales y formales podemos hacerla por posición o por nombre.

En la notación posicional, el primer parámetro actual se asocia con el primer parámetro formal, el segundo con el segundo, y así para el resto. En la notación nombrada usamos el operador => para asociar el parámetro actual al parámetro formal. También podemos usar notación mixta.

Los parámetros pueden ser de **entrada** al subprograma, de **salida**, o de **entrada/salida**. Por defecto si a un parámetro no le especificamos el modo, éste será de entrada. Si el parámetro es de salida o de entrada/salida, el parámetro actual debe ser una variable.

Un parámetro de entrada permite que pasemos valores al subprograma y no puede ser modificado en el cuerpo del subprograma. El parámetro actual pasado a un subprograma como parámetro formal de entrada puede ser una constante o una variable.

Un parámetro de salida permite devolver valores y dentro del subprograma actúa como variable no inicializada. El parámetro formal debe ser siempre 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 actual de entrada-salida siempre deberá ser una variable.

Los parámetros de entrada los podemos inicializar a un valor por defecto. Si un subprograma tiene un parámetro inicializado con un valor por defecto, podemos invocarlo prescindiendo del parámetro y aceptando el valor por defecto o pasando el parámetro y sobrescribiendo el valor por defecto. Si queremos prescindir de un parámetro colocado entre medias de otros, deberemos usar notación nombrada o si los parámetros restantes también tienen valor por defecto, omitirlos todos.

Debes conocer

En el siguiente enlace puedes encontrar ejemplos sobre el paso de parámetros a nuestros subprogramas.

[Paso de parámetros a subprogramas.](#)

Autoevaluación

Indica de entre las siguientes afirmaciones las que creas que son correctas.

- ☐ En PL/SQL podemos usar la notación posicional para pasar parámetros.

- ☐ No existen los parámetros de salida ya que para eso existen las funciones.

- ☐ Los parámetros de entrada los podemos inicializar a un valor por defecto.

Mostrar retroalimentación

Solución

1. Correcto
2. Incorrecto
3. Correcto

4.1.3.- Sobrecarga de subprogramas y recursividad.

PL/SQL también nos ofrece la posibilidad de sobrecargar funciones o procedimientos, es decir, llamar con el mismo nombre subprogramas que realizan el mismo cometido y que aceptan distinto número y/o tipo de parámetros. No podemos sobrecargar subprogramas que aceptan el mismo número y tipo de parámetros y sólo difieren en el modo. Tampoco podemos sobrecargar subprogramas con el mismo número de parámetros y que los tipos de los parámetros sean diferentes, pero de la misma familia, o sean subtipos basados en la misma familia.



[INTEF \(CC BY-NC-SA\)](#)

Debes conocer

En el siguiente enlace podrás ver un ejemplo de una función que es sobrecargada tres veces dependiendo del tipo de parámetros que acepta.

[Sobrecarga de subprogramas.](#)

PL/SQL también nos ofrece la posibilidad de utilizar la recursividad en nuestros subprogramas. Un subprograma es recursivo si éste se invoca a él mismo.

Debes conocer

En el siguiente enlace podrás ampliar información sobre la recursividad.

[Recursividad.](#)

En el siguiente enlace podrás ver un ejemplo del uso de la recursividad en nuestros subprogramas.

[Ejemplo del uso de la recursividad.](#)

Autoevaluación

En PL/SQL no podemos sobrecargar subprogramas que aceptan el mismo número y tipo de parámetros, pero sólo difieren en el modo.

- ☐ Verdadero.
- ☐ Falso.

Correcto, veo que lo has entendido.

No, no podemos hacerlo.

Solución

1. Opción correcta
2. Incorrecto

En PL/SQL no podemos utilizar la recursión y tenemos que imitarla mediante la iteración.

- ☐ Verdadero.
- ☐ Falso.

Incorrecto, creo que deberías mirar otra vez el segundo enlace del segundo "Debes conocer".

¡Muy bien!

Solución

1. Incorrecto
2. Opción correcta

4.2.- Paquetes.

Un paquete es un objeto que agrupa tipos, elementos y subprogramas. Suelen tener dos partes: la especificación y el cuerpo, aunque algunas veces el cuerpo no es necesario.

En la parte de especificación declararemos la interfaz del paquete con nuestra aplicación y en el cuerpo es donde implementaremos esa interfaz.



[INTEF](#) (CC BY-NC-SA)

Para crear un paquete usaremos la siguiente sintaxis:

```
CREATE [OR REPLACE] PACKAGE nombre AS
    [declaraciones públicas y especificación subprogramas]
END [nombre]
CREATE [OR REPLACE] PACKAGE BODY nombre AS
    [declaraciones privadas y cuerpo subprogramas especificados]
[BEGIN
    sentencias inicialización]
END [nombre];
```

La parte de inicialización sólo se ejecuta una vez, la primera vez que el paquete es referenciado.

Debes conocer

En el siguiente enlace te mostramos un ejemplo de un paquete que agrupa las principales tareas que realizamos con nuestra base de datos de ejemplo.

[Ejemplo de paquete.](#)

Para referenciar las partes visibles de un paquete, lo haremos por medio de la notación del punto.

```
BEGIN
    ...
    call_center.borra_agente( 10 );
    ...
END;
```

Para saber más

Oracle nos suministra varios paquetes para simplificar algunas tareas. En el siguiente enlace puedes encontrar más información sobre los mismos.

[Paquetes suministrados por Oracle.](#)

4.2.1.- Ejemplos de utilización del paquete DBMS_OUTPUT.

Oracle nos suministra un paquete público con el cual podemos enviar mensajes desde subprogramas almacenados, paquetes y disparadores, colocarlos en un buffer y leerlos desde otros subprogramas almacenados, paquetes o disparadores.

SQL*Plus permite visualizar los mensajes que hay en el buffer, por medio del comando `SET SERVEROUTPUT ON`. La utilización fundamental de este paquete es para la depuración de nuestros subprogramas.

Veamos uno a uno los subprogramas que nos suministra este paquete:

- ✔ Habilita las llamadas a los demás subprogramas. No es necesario cuando está activada la opción `SERVEROUTPUT`. Podemos pasarle un parámetro indicando el tamaño del buffer.

```
ENABLE
ENABLE( buffer_size IN INTEGER DEFAULT 2000);
```

- ✔ Deshabilita las llamadas a los demás subprogramas y purga el buffer. Como con `ENABLE` no es necesario si estamos usando la opción `SERVEROUTPUT`.

```
DISABLE
DISABLE();
```

- ✔ Coloca elementos en el buffer, los cuales son convertidos a `VARCHAR2`.

```
PUT
PUT(item IN NUMBER);
PUT(item IN VARCHAR2);
PUT(item IN DATE);
```

- ✔ Coloca elementos en el buffer y los termina con un salto de línea.

```
PUT_LINE
PUT_LINE(item IN NUMBER);
PUT_LINE(item IN VARCHAR2);
PUT_LINE(item IN DATE);
```

- ✔ Coloca un salto de línea en el buffer. Utilizado cuando componemos una línea usando varios `PUT`.

```
NEW_LINE  
NEW_LINE();
```

- ✓ Lee una línea del buffer colocándola en el parámetro `line` y obviando el salto de línea. El parámetro `status` devolverá 0 si nos hemos traído alguna línea y 1 en caso contrario.

```
GET_LINE  
GET_LINE(line OUT VARCHAR2, status OUT VARCHAR2);
```

- ✓ Intenta leer el número de líneas indicado en `numlines`. Una vez ejecutado, `numlines` contendrá el número de líneas que se ha traído. Las líneas traídas las coloca en el parámetro `lines` del tipo `CHARARR`, tipo definido el paquete `DBMS_OUTPUT` como una tabla de `VARCHAR2(255)`.

```
GET_LINES  
GET_LINES(lines OUT CHARARR, numlines IN OUT INTEGER);
```

Ejercicio resuelto

Debes crear un procedimiento que visualice todos los agentes, su nombre, nombre de la familia y/o nombre de la oficina a la que pertenece.

Mostrar retroalimentación

```
CREATE OR REPLACE PROCEDURE lista_agentes IS  
    CURSOR cAgentes IS SELECT identificador, nombre,familia, oficina F  
    fam familias.nombre%TYPE;  
    ofi oficinas.nombre%TYPE;  
    num_ag INTEGER := 0;  
BEGIN  
    DBMS_OUTPUT.ENABLE( 1000000 );  
    DBMS_OUTPUT.PUT_LINE('Agente          |Familia          |Ofi  
    DBMS_OUTPUT.PUT_LINE('-----  
    FOR ag_rec IN cAgentes LOOP  
        IF (ag_rec.familia IS NOT NULL) THEN  
            SELECT nombre INTO fam FROM familias WHERE identificador  
            ofi := NULL;  
            DBMS_OUTPUT.PUT_LINE(rpad(ag_rec.nombre,20) || '|' || rp  
            num_ag := num_ag + 1;  
        ELSIF (ag_rec.oficina IS NOT NULL) THEN  
            SELECT nombre INTO ofi FROM oficinas WHERE identificador
```

```

        fam := NULL;
        DBMS_OUTPUT.PUT_LINE(rpad(ag_rec.nombre,20) || '||' || rp
        num_ag := num_ag + 1;
    END IF;
END LOOP;
DBMS_OUTPUT.PUT_LINE('-----')
        DBMS_OUTPUT.PUT_LINE('Número de agentes: ' || num_ag);
END lista_agentes;
/

```

Recuerda que para ejecutarlo desde SQL*Plus debes ejecutar las siguientes sentencias:

```

SQL>SET SERVEROUTPUT ON;
SQL>EXEC lista_agentes;

```

4.3.- Objetos.

Hoy día, la programación orientada a objetos es uno de los paradigmas más utilizados y casi todos los lenguajes de programación la soportan. En este apartado vamos a dar unas pequeñas pinceladas de su uso en PL/SQL que serán ampliados en la siguiente unidad de trabajo.

Un tipo de objeto es un tipo de dato compuesto, que encapsula unos datos y las funciones y procedimientos necesarios para manipular esos datos. Las variables son los atributos y los subprogramas son llamados métodos. Podemos pensar en un tipo de objeto como en una entidad que posee unos atributos y un comportamiento (que viene dado por los métodos).

- ✓ Cuando creamos un tipo de objeto, lo que estamos creando es una entidad abstracta que especifica los atributos que tendrán los objetos de ese tipo y define su comportamiento.
- ✓ Cuando instanciamos un objeto estamos particularizando la entidad abstracta a una en particular, con los atributos que tiene el tipo de objeto, pero con un valor dado y con el mismo comportamiento.

Los tipos de objetos tiene 2 partes: una **especificación** y un **cuerpo**. La parte de especificación declara los atributos y los métodos que harán de interfaz de nuestro tipo de objeto. En el cuerpo se implementa la parte de especificación. En la parte de especificación debemos declarar primero los atributos y después los métodos. Todos los atributos son públicos (visibles). No podemos declarar atributos en el cuerpo, pero sí podemos declarar subprogramas locales que serán visibles en el cuerpo del objeto y que nos ayudarán a implementar nuestros métodos.

Los atributos pueden ser de cualquier tipo de datos Oracle, excepto:

- ✓ LONG y LONG RAW.
- ✓ NCHAR, NCLOB y NVARCHAR2.
- ✓ MLSLABEL y ROWID.
- ✓ Tipos específicos de PL/SQL: BINARY_INTEGER, BOOLEAN, PLS_INTEGER, RECORD, REF CURSOR, %TYPE y %ROWTYPE.
- ✓ Tipos definidos dentro de un paquete PL/SQL.

No podemos inicializar un atributo en la declaración. Tampoco podemos imponerle la restricción NOT NULL.

Un **método** es un subprograma declarado en la parte de especificación de un tipo de objeto por medio de: MEMBER. Un método no puede llamarse igual que el tipo de objeto o que cualquier atributo. Para cada método en la parte de especificación, debe haber un método implementado en el cuerpo con la misma cabecera.

Todos los métodos en un tipo de objeto aceptan como primer parámetro una instancia de su tipo. Este parámetro es SELF y siempre está accesible a un método. Si lo declaramos explícitamente debe ser el primer parámetro, con el nombre SELF y del tipo del tipo de objeto. Si SELF no está declarado explícitamente, por defecto será IN para las funciones e IN OUT para los procedimientos.

Los métodos dentro de un tipo de objeto pueden sobrecargarse. No podemos sobrecargarlos si los parámetros formales sólo difieren en el modo o pertenecen a la misma familia. Tampoco podremos sobrecargar una función miembro si sólo difiere en el

tipo devuelto.

Una vez que tenemos creado el objeto, podemos usarlo en cualquier declaración. Un objeto cuando se declara sigue las mismas reglas de alcance y visibilidad que cualquier otra variable.

Cuando un objeto se declara éste es automáticamente `NULL`. Dejará de ser nulo cuando lo inicialicemos por medio de su constructor o cuando le asignemos otro. Si intentamos acceder a los atributos de un objeto `NULL` saltará la excepción `ACCES_INTRO_NULL`.

Todos los objetos tienen constructores por defecto con el mismo nombre que el tipo de objeto y acepta tantos parámetros como atributos del tipo de objeto y con el mismo tipo. PL/SQL no llama implícitamente a los constructores, deberemos hacerlo nosotros explícitamente.

```
DECLARE
    familia1 Familia;
BEGIN
    ...
    familia1 := Familia( 10, 'Fam10', 1, NULL );
    ...
END;
```

Un tipo de objeto puede tener a otro tipo de objeto entre sus atributos. El tipo de objeto que hace de atributo debe ser conocido por Oracle. Si 2 tipos de objetos son mutuamente dependientes, podemos usar una declaración hacia delante para evitar errores de compilación.

Ejercicio resuelto

Cómo declararías los objetos para nuestra base de datos de ejemplo.

Mostrar retroalimentación

```
CREATE OBJECT Oficina;      --Definición hacia delante
CREATE OBJECT Familia AS OBJECT (
    identificador    NUMBER,
    nombre           VARCHAR2(20),
    familia_         Familia,
    oficina_         Oficina,
    ...
);
CREATE OBJECT Agente AS OBJECT (
    identificador    NUMBER,
    nombre           VARCHAR2(20),
```

```
        familia_      Familia,  
        oficina_      Oficina,  
        ...  
    );  
    CREATE OBJECT Oficina AS OBJECT (  
        identificador      NUMBER,  
        nombre              VARCHAR2(20),  
        jefe                Agente,  
        ...  
    );
```


4.3.1.- Objetos. Funciones mapa y funciones de orden.

En la mayoría de los problemas es necesario hacer comparaciones entre tipos de datos, ordenarlos, etc. Sin embargo, los tipos de objetos no tienen orden predefinido por lo que no podrán ser comparados ni ordenados ($x > y$, `DISTINCT`, `ORDER BY`, ...). Nosotros podemos definir el orden que seguirá un tipo de objeto por medio de las funciones mapa y las funciones de orden.

Una función miembro mapa es una función sin parámetros que devuelve un tipo de dato: `DATE`, `NUMBER` o `VARCHAR2` y sería similar a una función hash. Se definen anteponiendo la palabra clave `MAP` y sólo puede haber una para el tipo de objeto.

```
CREATE TYPE Familia AS OBJECT (  
    identificador      NUMBER,  
    nombre             VARCHAR2(20),  
    familia_           NUMBER,  
    oficina_          NUMBER,  
    MAP MEMBER FUNCTION orden RETURN NUMBER,  
    ...  
);  
  
CREATE TYPE BODY Familia AS  
MAP MEMBER FUNCTION orden RETURN NUMBER IS  
BEGIN  
    RETURN identificador;  
END;  
...  
END;
```

Una función miembro de orden es una función que acepta un parámetro del mismo tipo del tipo de objeto y que devuelve un número negativo si el objeto pasado es menor, cero si son iguales y un número positivo si el objeto pasado es mayor.

```
CREATE TYPE Oficina AS OBJECT (  
    identificador      NUMBER,  
    nombre             VARCHAR2(20),  
    ...  
    ORDER MEMBER FUNCTION igual ( ofi Oficina ) RETURN INTEGER,  
    ...  
);  
  
CREATE TYPE BODY Oficina AS  
    ORDER MEMBER FUNCTION igual ( ofi Oficina ) RETURN INTEGER IS  
BEGIN  
    IF (identificador < ofi.identificador) THEN  
        RETURN -1;  
    ELSIF (identificador = ofi.identificador) THEN  
        RETURN 0;  
    ELSE  
        RETURN 1;  
    END IF;  
END;
```

```
ELSE
    RETURN 1;
END IF;
END;
...
END;
```

Autoevaluación

Los métodos de un objeto sólo pueden ser procedimientos.

- ☐ Verdadero.
- ☐ Falso.

Incorrecto, creo que deberías dar un repaso a este apartado.

Efectivamente, ya que las funciones también pueden ser métodos.

Solución

1. Incorrecto
2. Opción correcta

El orden de un objeto se consigue:

- ☐ Al crearlo.
- ☐ En PL/SQL los objetos no pueden ser ordenados.
- ☐ Mediante las funciones mapa y las funciones de orden.

No, los objetos no tienen ningún orden predefinido.

No es cierto, el orden lo podemos conseguir mediante el uso de funciones mapa y funciones de orden.

¡Muy bien!

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

5.- Disparadores.

Caso práctico

Juan y María ya han hecho un paquete en el que tienen agrupadas las funciones más usuales que realizan sobre la base de datos de juegos on-line. Sin embargo, **María** ve que hay cosas que aún no pueden controlar. Por ejemplo, **María** quiere que la clave y el usuario de un jugador o jugadora no puedan ser la misma y eso no sabe cómo hacerlo. **Juan** le dice que para ese cometido están los disparadores y sin más dilaciones se pone a explicarle a **María** para qué sirven y cómo utilizarlos.



[Loren \(INTEF\) \(CC BY-NC-SA\)](#)

En este apartado vamos a tratar una herramienta muy potente e importante proporcionada por PL/SQL para programar nuestra base de datos y mantener la integridad y seguridad que son los disparadores o triggers en inglés.

Pero, ¿qué es un disparador?

Un **disparador** no es más que un procedimiento que es ejecutado cuando se realiza alguna sentencia sobre la BD, o sus tablas, y bajo unas circunstancias establecidas a la hora de definirlo.

Los disparadores pueden ser de tres tipos:

- ✔ **De tablas**
- ✔ **De sustitución**
- ✔ **De sistema**

Un disparador puede ser lanzado antes o después de realizar la operación que lo lanza. Por lo que tendremos disparadores **BEFORE** y disparadores **AFTER**.

Disparadores de Tablas

Normalmente previenen transacciones erróneas o nos permiten implementar restricciones de integridad o seguridad, o automatizar procesos. Son los más utilizados.

Se ejecutan cuando se realiza alguna sentencia de manipulación de datos sobre una tabla dada. Puede ser lanzado al insertar, al actualizar o al borrar de una tabla, por lo que tendremos disparadores **INSERT**, **UPDATE** o **DELETE** (o mezclados).

Puede ser lanzado una vez por sentencia (**FOR STATEMENT**) o una vez por cada fila a la que afecta (**FOR EACH ROW**). Por lo que tendremos disparadores de sentencia y disparadores de fila.

De sustitución

No se ejecutan ni antes ni después, sino en lugar de (se conocen como triggers INSTEAD OF). Sólo se pueden asociar a vistas y a nivel de fila.

De sistema

Se ejecutan cuando se produce una determinada operación sobre la BD, se crea una tabla, se conecta un usuario, etc. Los eventos que se pueden detectar son por ejemplo: LOGON, LOGOFF, CREATE, DROP, etc y el momento puede ser tanto BEFORE como AFTER.

Un **disparador** no es más que un procedimiento y puede ser usado para:

- ✔ Llevar a cabo auditorías sobre la historia de los datos en nuestra base de datos.
- ✔ Garantizar complejas reglas de integridad.
- ✔ Automatizar la generación de valores derivados de columnas.
- ✔ Etc.

Cuando diseñamos un disparador debemos tener en cuenta que:

- ✔ No debemos definir disparadores que dupliquen la funcionalidad que ya incorpora Oracle.
- ✔ Debemos limitar el tamaño de nuestros disparadores, y si estos son muy grandes codificarlos por medio de subprogramas que sean llamados desde el disparador.
- ✔ Cuidar la creación de disparadores recursivos.

Autoevaluación

En PL/SQL sólo podemos definir disparadores de fila.

- ☐ Verdadero.
- ☐ Falso.

No, también podemos definir disparadores de sentencia.

Efectivamente, ya que también podemos definir disparador de sentencia.

Solución

1. Incorrecto

2. Opción correcta

La diferente entre un disparador de fila y uno de sentencia es que el de fila es lanzado una vez por fila a la que afecta la sentencia y el de sentencia es lanzado una sola vez.

- ☐ Verdadero.
- ☐ Falso.

Muy bien, vas por buen camino.

Incorrecto, creo que deberías repasar este apartado otra vez.

Solución

- 1. Opción correcta
- 2. Incorrecto

5.1.- Definición de disparadores de Tablas

De los tres tipos de disparadores, veremos los asociados a tablas.

Por lo visto anteriormente, para definir un disparador deberemos indicar si será lanzado antes o después de la ejecución de la sentencia que lo lanza, si se lanzará una vez por sentencia o una vez por fila a la que afecta, y si será lanzado al insertar y/o al actualizar y/o al borrar.

La sintaxis que seguiremos para definir un disparador será la siguiente:

```
CREATE [OR REPLACE] TRIGGER nombre
momento acontecimiento ON tabla
[[REFERENCING (old AS alias_old|new AS alias_new)
FOR EACH ROW
[WHEN condicion]]
bloque_PL/SQL;
```



[Loren \(INTEF\)](#) (CC BY-NC-SA)

Donde nombre nos indica el nombre que le damos al disparador, momento nos dice cuando será lanzado el disparador (BEFORE o AFTER), acontecimiento será la acción que provoca el lanzamiento del disparador (INSERT y/o DELETE y/o UPDATE). REFERENCING y WHEN sólo podrán ser utilizados con disparadores para filas. REFERENCING nos permite asignar un alias a los valores NEW o/y OLD de las filas afectadas por la operación, y WHEN nos permite indicar al disparador que sólo sea lanzado cuando sea TRUE una cierta condición evaluada para cada fila afectada.

En un disparador de fila, podemos acceder a los valores antiguos y nuevos de la fila afectada por la operación, referenciados como :old y :new (de ahí que podamos utilizar la opción REFERENCING para asignar un alias). Si el disparador es lanzado al insertar, el valor antiguo no tendrá sentido y el valor nuevo será la fila que estamos insertando. Para un disparador lanzado al actualizar el valor antiguo contendrá la fila antes de actualizar y el valor nuevo contendrá la fila que vamos actualizar. Para un disparador lanzado al borrar sólo tendrá sentido el valor antiguo.

En el cuerpo de un disparador también podemos acceder a unos predicados que nos dicen qué tipo de operación se está llevando a cabo, que son: INSERTING, UPDATING y DELETING.

Un disparador de fila no puede acceder a la tabla asociada. Se dice que esa tabla está mutando. Si un disparador es lanzado en cascada por otro disparador, éste no podrá acceder a ninguna de las tablas asociadas, y así recursivamente.

```
CREATE TRIGGER prueba BEFORE UPDATE ON agentes
FOR EACH ROW
BEGIN
...
SELECT identificador FROM agentes WHERE ...
/*devolvería el error ORA-04091: table AGENTES is mutating, trigger/function may not see it'
```

```
...  
END;  
/
```

Si tenemos varios tipos de disparadores sobre una misma tabla, el orden de ejecución será:

- ✔ Triggers before de sentencia.
- ✔ Triggers before de fila.
- ✔ Triggers after de fila.
- ✔ Triggers after de sentencia.

Existe una vista del diccionario de datos con información sobre los disparadores:
`USER_TRIGGERS`;

```
SQL>DESC USER_TRIGGERS;  
Name                               Null?    Type  
-----  
TRIGGER_NAME                       NOT NULL VARCHAR2(30)  
TRIGGER_TYPE                       VARCHAR2(16)  
TRIGGERING_EVENT                   VARCHAR2(26)  
TABLE_OWNER                        NOT NULL VARCHAR2(30)  
TABLE_NAME                         NOT NULL VARCHAR2(30)  
REFERENCING_NAMES                  VARCHAR2(87)  
WHEN_CLAUSE                        VARCHAR2(4000)  
STATUS                             VARCHAR2(8)  
DESCRIPTION                       VARCHAR2(4000)  
TRIGGER_BODY                       LONG
```


5.2.- Ejemplos de disparadores.

Ya hemos visto qué son los disparadores, los tipos que existen, cómo definirlos y algunas consideraciones a tener en cuenta a la hora de trabajar con ellos.

Ahora vamos a ver algunos ejemplos de su utilización con los que podremos comprobar la potencia que éstos nos ofrecen.

Ejercicio resuelto

Como un agente debe pertenecer a una familia o una oficina pero no puede pertenecer a una familia y a una oficina a la vez, deberemos implementar un disparador para llevar a cabo esta restricción que Oracle no nos permite definir desde el DDL.

Mostrar retroalimentación

Para este cometido definiremos un disparador de fila que saltará antes de que insertemos o actualicemos una fila en la tabla agentes, cuyo código podría ser el siguiente:

```
CREATE OR REPLACE TRIGGER integridad_agentes
BEFORE INSERT OR UPDATE ON agentes
FOR EACH ROW
BEGIN
    IF (:new.familia IS NULL and :new.oficina IS NULL) THEN -- Si los dos son NULL
        RAISE_APPLICATION_ERROR(-20201, 'Un agente no puede ser huérfano');
    ELSIF (:new.familia IS NOT NULL and :new.oficina IS NOT NULL) THEN
        RAISE_APPLICATION_ERROR(-20202, 'Un agente no puede tener dos oficinas');
    END IF;
END;
/
```

Debes conocer

En la siguiente presentación podrás ver la descripción de la tabla USER_TRIGGERS, cómo se almacena un disparador en la base de datos, cómo

se consulta su código y cómo se lanza su ejecución. Pruébalo en tu máquina para comprobar su funcionamiento.

[Presentación sobre disparadores. Creación, consulta de su código y lanzamiento con una sentencia insert](#) (odp - 296,81 KB)
[Resumen textual alternativo](#)

Ejercicio resuelto

Supongamos que tenemos una tabla de históricos (histagentes) para agentes que nos permita auditar las familias y oficinas por la que ha ido pasando un agente. La tabla tiene la fecha de inicio y la fecha de finalización del agente en esa familia u oficina, el identificador del agente, el nombre del agente, el nombre de la familia y el nombre de la oficina. Queremos hacer un disparador que registre (inserte o actualice en la tabla histagentes) los cambios que se producen en la tabla agentes (al insertar, actualizar o borrar).

La tabla histagentes necesita ser creada antes de la creación del trigger y es la siguiente:

```
create table histagentes (  
    fecha_inicio date,  
    fecha_hasta date,  
    identificador number(6),  
    nombre varchar2(40),  
    familia varchar2(40),  
    oficina varchar2(40)  
);
```

Mostrar retroalimentación

Para llevar a cabo esta tarea definiremos un disparador de fila que saltará después de insertar, actualizar o borrar una fila en la tabla agentes, cuyo código podría ser el siguiente:

```
CREATE OR REPLACE TRIGGER historico_agentes  
AFTER INSERT OR UPDATE OR DELETE ON agentes  
FOR EACH ROW  
DECLARE  
    oficina VARCHAR2(40);  
    familia VARCHAR2(40);
```

```

        ahora DATE := sysdate;
BEGIN
    IF INSERTING THEN
        IF (:new.familia IS NOT NULL) THEN
            SELECT nombre INTO familia FROM familias WHERE identificado = :new.identificado
            oficina := NULL;
        ELSE
            SELECT nombre INTO oficina FROM oficinas WHERE identificado = :new.identificado
            familia := NULL;
        END IF;
        INSERT INTO histagentes VALUES (ahora, NULL, :new.identificado);
        COMMIT;
    ELSIF UPDATING THEN
        UPDATE histagentes SET fecha_hasta = ahora WHERE identificado = :new.identificado;
        IF (:new.familia IS NOT NULL) THEN
            SELECT nombre INTO familia FROM familias WHERE identificado = :new.identificado
            oficina := NULL;
        ELSE
            SELECT nombre INTO oficina FROM oficinas WHERE identificado = :new.identificado
            familia := NULL;
        END IF;
        INSERT INTO histagentes VALUES (ahora, NULL, :new.identificado);
        COMMIT;
    ELSE
        UPDATE histagentes SET fecha_hasta = ahora WHERE identificado = :new.identificado;
        COMMIT;
    END IF;
END;
/

```

Ejercicio resuelto

Queremos realizar un disparador que no nos permita llevar a cabo operaciones con familias si no estamos en la jornada laboral.

Mostrar retroalimentación

```

CREATE OR REPLACE TRIGGER jornada_familias
BEFORE INSERT OR DELETE OR UPDATE ON familias
DECLARE
    ahora DATE := sysdate;
BEGIN
    IF (TO_CHAR(ahora, 'DY') = 'SAT' OR TO_CHAR(ahora, 'DY') = 'SUN') THEN
        RAISE_APPLICATION_ERROR(-20301, 'No podemos manipular familia');
    END IF;

```

```
IF (TO_CHAR(ahora, 'HH24') < 8 OR TO_CHAR(ahora, 'HH24') > 18) THEN  
    RAISE_APPLICATION_ERROR(-20302, 'No podemos manipular familia:  
END IF;  
END;  
/
```

6.- Interfaces de programación de aplicaciones para lenguajes externos.

Caso práctico

Juan y María tienen la base de datos de juegos online lista, pero no tienen claro si todo lo que han hecho lo tienen que utilizar desde dentro de la base de datos o si pueden reutilizar todo ese trabajo desde otro lenguaje de programación desde el que están creando la interfaz web desde la que los jugadores y jugadoras se conectarán para jugar y desde la que también se conectará la administradora para gestionar la base de datos de una forma más amigable.

En ese momento pasa **Ada** al lado y ambos la asaltan con su gran duda. **Ada**, muy segura, les responde que para eso existen las interfaces de programación de aplicaciones o APIs, que les permiten acceder desde otros lenguajes de programación a la base de datos.



[Abraham Pérez Pérez \(INTEF\)](#)
(CC BY-NC-SA)

Llegados a este punto ya sabemos programar nuestra base de datos, pero al igual que a **Juan y María** te surgirá la duda de si todo lo que has hecho puedes aprovecharlo desde cualquier otro lenguaje de programación. La respuesta es la misma que dio **Ada**: Sí.

En este apartado te vamos a dar algunas referencias sobre las APIs para acceder a las bases de datos desde un lenguaje de programación externo. No vamos a ser exhaustivos ya que eso queda fuera del alcance de esta unidad e incluso de este módulo. Todo ello lo vas a estudiar con mucha más profundidad en otros módulos de este ciclo (o incluso ya lo conoces). Aquí sólo pretendemos que sepas que existen y que las puedes utilizar.

Las primeras APIs utilizadas para acceder a bases de datos Oracle fueron las que el mismo Oracle proporcionaba: Pro*C, Pro*Fortran y Pro*Cobol. Todas permitían embeber llamadas a la base de datos en nuestro programa y a la hora de compilarlo, primero debíamos pasar el precompilador adecuado que trasladaba esas llamadas embebidas en llamadas a una librería utilizada en tiempo de ejecución. Sin duda, el más utilizado fue Pro*C, ya que el lenguaje C y C++ tuvieron un gran auge.

Para saber más

En los siguientes enlaces podrás obtener más información sobre Pro*C y cómo crear un programa para acceder a nuestra base de datos.

[Introducción a Pro*C.](#)

[Ejemplo de programa en Pro*C.](#)

Hoy día existen muchas más APIs para acceder a las bases de datos ya que tanto los lenguajes de programación como las tecnologías han evolucionado mucho. Antes la programación para la web casi no existía y por eso todos los programas que accedían a bases de datos lo hacían bien en local o bien a través de una red local. Hoy día eso sería impensable, de ahí que las APIs también hayan evolucionado mucho y tengamos una gran oferta a nuestro alcance para elegir la que más se adecue a nuestra necesidades.

Para saber más

En los siguientes enlaces podrás ampliar información sobre algunas APIs muy comunes para acceder a bases de datos.

[Tecnologías Java para acceder a bases de datos.](#)

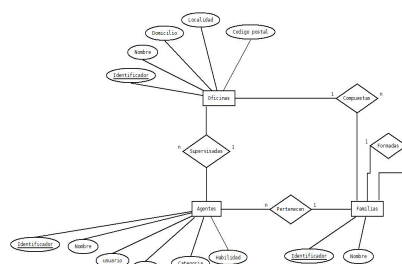
[Conectividad abierta de bases de datos. \(ODBC\)](#)

Anexo I.- Caso de estudio.

Una empresa de telefonía tiene sus centros de llamadas distribuidos por la geografía española en diferentes oficinas. Estas oficinas están jerarquizadas en familias de agentes telefónicos. Cada familia, por tanto, podrá contener agentes u otras familias. Los agentes telefónicos, según su categoría, además se encargarán de supervisar el trabajo de todos los agentes de una oficina o de coordinar el trabajo de los agentes de una familia dada. El único agente que pertenecerá directamente a una oficina y que no formará parte de ninguna familia será el supervisor de dicha oficina, cuya categoría es la 2. Los coordinadores de las familias deben pertenecer a dicha familia y su categoría será 1 (no todas las familias tienen por qué tener un coordinador y dependerá del tamaño de la oficina, ya que de ese trabajo también se puede encargar el supervisor de la oficina). Los demás agentes deberán pertenecer a una familia, su categoría será 0 y serán los que principalmente se ocupen de atender las llamadas.

- ✓ De los agentes queremos conocer su nombre, su clave y contraseña para entrar al sistema, su categoría y su habilidad que será un número entre 0 y 9 indicando su habilidad para atender llamadas.
- ✓ Para las familias sólo nos interesa conocer su nombre.
- ✓ Finalmente, para las oficinas queremos saber su nombre, domicilio, localidad y código postal de la misma.

Un posible modelo entidad-relación para el problema expuesto podría ser el siguiente:



Ministerio de Educación (Uso educativo nc)

El **Modelo Relacional** resultante sería:

OFICINAS (identificador, nombre, domicilio, localidad, codigo_postal)

FAMILIAS (identificador, nombre, familia (fk), oficina (fk))

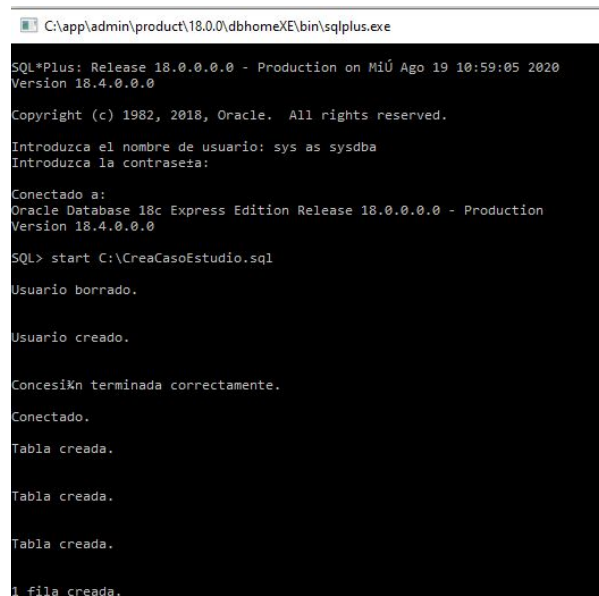
AGENTES (identificador, nombre, usuario, clave, habilidad, categoría, familia (fk), oficina (fk))

De este modelo de datos surgen tres tablas, que puedes crear en Oracle con el script del siguiente enlace

[Script CreaCasoEstudio.zip](#) (zip - 1,62 KB)

El script crea un usuario llamado **c##agencia** con clave **agencia**. Para que puedas ejecutarlo y comenzar de cero, cuantas veces quieras, el script elimina el usuario **c##agencia** y sus tablas antes de volver a crearlo.

Conecta como administrador con SYS as SYSDBA y ejecútalo anteponiendo el símbolo @ o la palabra start antes del nombre del script.



```
C:\app\admin\product\18.0.0\dbhomeXE\bin\sqlplus.exe
SQL*Plus: Release 18.0.0.0.0 - Production on MiÚ Ago 19 10:59:05 2020
Version 18.4.0.0.0

Copyright (c) 1982, 2018, Oracle. All rights reserved.

Introduzca el nombre de usuario: sys as sysdba
Introduzca la contrasea:

Conectado a:
Oracle Database 18c Express Edition Release 18.0.0.0.0 - Production
Version 18.4.0.0.0

SQL> start C:\CreaCasoEstudio.sql

Usuario borrado.

Usuario creado.

Concesión terminada correctamente.
Conectado.

Tabla creada.

Tabla creada.

Tabla creada.

1 fila creada.
```

[Oracle Corporation](#) (Todos los derechos reservados)

Habilitando la Salida/OUTPUT en Bloques PL/SQL.

PL/SQL no proporciona funcionalidad de entrada o salida directamente siendo necesario utilizar paquetes predefinidos de **Oracle** para tales fines. Para generar una salida debes realizar lo siguiente :

1. Ejecutar el siguiente comando tanto en **SQL*Plus** como en cualquier otro entorno

```
SET SERVEROUTPUT ON
```

2. En el bloque **PL/SQL**, hay que utilizar el procedimiento **PUT_LINE** del paquete **DBMS_OUTPUT** para mostrar la salida. El valor a mostrar en pantalla se pasará como argumento del procedimiento.

Ejecuta el siguiente código desde SQLPlus y desde SQLDeveloper para comprobar su funcionamiento.

```
SET SERVEROUTPUT ON
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hola mundo');
END;
```


Anexo II.- Excepciones predefinidas en Oracle.

Las excepciones predefinidas son:

Excepciones predefinidas en Oracle.

| Excepción. | SQLCODE | Lanzada cuando ... |
|-------------------------|---------|--|
| ACCES_INT0_NULL | -6530 | Intentamos asignar valor a atributos de objetos no inicializados. |
| COLECTION_IS_NULL | -6531 | Intentamos asignar valor a elementos de colecciones no inicializadas, o acceder a métodos distintos de EXISTS. |
| CURSOR_ALREADY_OPEN | -6511 | Intentamos abrir un cursor ya abierto. |
| DUP_VAL_ON_INDEX | -1 | Índice único violado. |
| INVALID_CURSOR | -1001 | Intentamos hacer una operación con un cursor que no está abierto. |
| INVALID_NUMBER | -1722 | Conversión de cadena a número falla. |
| LOGIN_DENIED | -1403 | El usuario y/o contraseña para conectarnos a Oracle no es válido. |
| NO_DATA_FOUND | +100 | Una sentencia SELECT no devuelve valores, o intentamos acceder a un elemento borrado de una tabla anidada. |
| NOT_LOGGED_ON | -1012 | No estamos conectados a Oracle. |
| PROGRAM_ERROR | -6501 | Ha ocurrido un error interno en PL/SQL. |
| ROWTYPE_MISMATCH | -6504 | Diferentes tipos en la asignación de 2 cursores. |
| STORAGE_ERROR | -6500 | Memoria corrupta. |
| SUBSCRIPT_BEYOND_COUNT | -6533 | El índice al que intentamos acceder en una colección sobrepasa su límite superior. |
| SUBSCRIPT_OUTSIDE_LIMIT | -6532 | Intentamos acceder a un rango no válido dentro de una colección (-1 por ejemplo). |
| TIMEOUT_ON_RESOURCE | -51 | Un timeout ocurre mientras Oracle espera por un recurso. |

| Excepción. | SQLCODE | Lanzada cuando ... |
|---------------|---------|--|
| TOO_MANY_ROWS | -1422 | Una sentencia <code>SELECT...INTO...</code> devuelve más de una fila. |
| VALUE_ERROR | -6502 | Ocurre un error de conversión, aritmético, de truncado o de restricción de tamaño. |
| ZERO_DIVIDE | -1476 | Intentamos dividir un número por 0. |

Anexo III.- Evaluación de los atributos de un cursor explícito.

Evaluación de los atributos de un cursor explícito según las operaciones realizadas con él.

| Operación realizada. | %FOUND | %NOTFOUND | %ISOPEN | %ROWCOUNT |
|---------------------------------|------------|------------|---------|----------------|
| Antes del OPEN | Excepción. | Excepción. | FALSE | Excepción. |
| Después del OPEN | NULL | NULL | TRUE | 0 |
| Antes del primer FETCH | NULL | NULL | TRUE | 0 |
| Después del primer FETCH | TRUE | FALSE | TRUE | 1 |
| Antes de los siguientes FETCH | TRUE | FALSE | TRUE | 1 |
| Después de los siguientes FETCH | TRUE | FALSE | TRUE | Depende datos. |
| Antes del último FETCH | TRUE | FALSE | TRUE | Depende datos. |
| Después del último FETCH | FALSE | TRUE | TRUE | Depende datos. |
| Antes del CLOSE | FALSE | TRUE | TRUE | Depende datos. |
| Después del CLOSE | Excepción. | Excepción. | FALSE | Excepción. |

Anexo IV.- Paso de parámetros a subprogramas.

Veamos algunos ejemplos del paso de parámetros a subprogramas:

✔ Notación mixta.

```
DECLARE
    PROCEDURE prueba( formal1 NUMBER, formal2 VARCHAR2) IS
    BEGIN
        ...
    END;
    actual1 NUMBER;
    actual2 VARCHAR2;
BEGIN
    ...
    prueba(actual1, actual2);           --posicional
    prueba(formal2=>actual2,formal1=>actual1);    --nombrada
    prueba(actual1, formal2=>actual2);    --mixta
END;
```

✔ Parámetros de entrada.

```
FUNCTION categoria( id_agente IN NUMBER )
RETURN NUMBER IS
    cat NUMBER;
BEGIN
    ...
    SELECT categoria INTO cat FROM agentes
WHERE identificador = id_agente;
RETURN cat;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        id_agente := -1; --ilegal, parámetro de entrada
END;
```

✔ Parámetros de salida.

```
PROCEDURE nombre( id_agente NUMBER, nombre OUT VARCHAR2) IS
BEGIN
    IF (nombre = 'LUIS') THEN    --error de sintaxis
    END IF;
    ...
END;
```

```
END;
```

- ✔ Parámetros con valor por defecto de los que podemos prescindir.

```
DECLARE
    SUBTYPE familia IS familias%ROWTYPE;
    SUBTYPE agente IS agentes%ROWTYPE;
    SUBTYPE tabla_agentes IS TABLE OF agente;
    familia1 familia;
    familia2 familia;
    hijos_fam tabla_agentes;
    FUNCTION inserta_familia( mi_familia familia,
        mis_agentes tabla_agentes := tabla_agentes() )
RETURN NUMBER IS
    BEGIN
        INSERT INTO familias VALUES (mi_familia);
        FOR i IN 1..mis_agentes.COUNT LOOP
            IF (mis_agentes(i).oficina IS NOT NULL) or (mis_agentes(i).familia != 1)
                ROLLBACK;
                RETURN -1;
            END IF;
            INSERT INTO agentes VALUES (mis_agentes(i));
        END LOOP;
        COMMIT;
        RETURN 0;
    EXCEPTION
        WHEN DUP_VAL_ON_INDEX THEN
            ROLLBACK;
            RETURN -1;
        WHEN OTHERS THEN
            ROLLBACK;
            RETURN SQLCODE;
    END inserta_familia;
BEGIN
    ...
    resultado := inserta_familia( familia1 );
    ...
    resultado := inserta_familia( familia2, hijos_fam2 );
    ...
END;
```

Anexo V.- Sobrecarga de subprogramas.

Veamos un ejemplo de sobrecarga de subprogramas en el que una misma función es sobrecargada tres veces para diferentes tipos de parámetros.

```
DECLARE
    TYPE agente IS agentes%ROWTYPE;
    TYPE familia IS familias%ROWTYPE;
    TYPE tAgentes IS TABLE OF agente;
    TYPE tFamilias IS TABLE OF familia;

    FUNCTION inserta_familia( mi_familia familia )
    RETURN NUMBER IS
    BEGIN
        INSERT INTO familias VALUES (mi_familia.identificador, mi_familia.nombre, mi_familia.oficina);
        COMMIT;
        RETURN 0;
    EXCEPTION
        WHEN DUP_VAL_ON_INDEX THEN
            ROLLBACK;
            RETURN -1;
        WHEN OTHERS THEN
            ROLLBACK;
            RETURN SQLCODE;
    END inserta_familia;

    FUNCTION inserta_familia( mi_familia familia, hijas tFamilias )
    RETURN NUMBER IS
    BEGIN
        INSERT INTO familias VALUES (mi_familia.identificador, mi_familia.nombre, mi_familia.oficina);
        IF (hijas IS NOT NULL) THEN
            FOR i IN 1..hijas.COUNT LOOP
                IF (hijas(i).oficina IS NOT NULL) or (hijas(i).familia != mi_familia.identificador) THEN
                    ROLLBACK;
                    RETURN -1;
                END IF;
                INSERT INTO familias VALUES (hijas(i).identificador, hijas(i).nombre, hijas(i).oficina);
            END LOOP;
        END IF;
        COMMIT;
        RETURN 0;
    EXCEPTION
        WHEN DUP_VAL_ON_INDEX THEN
            ROLLBACK;
            RETURN -1;
        WHEN OTHERS THEN
            ROLLBACK;
            RETURN -1;
    END inserta_familia;

    FUNCTION inserta_familia( mi_familia familia, hijos tAgentes )
    RETURN NUMBER IS
```

```

BEGIN
    INSERT INTO familias VALUES (mi_familia.identificador, mi_familia.nombre, mi_familia.
    IF (hijos IS NOT NULL) THEN
        FOR i IN 1..hijos.COUNT LOOP
            IF (hijos(i).oficina IS NOT NULL) or (hijos(i).familia != mi_familia.id)
                ROLLBACK;
                RETURN -1;
            END IF;
            INSERT INTO agentes VALUES (hijos(i).identificador, hijos(i).nombre, hi
        END LOOP;
    END IF;
    COMMIT;
    RETURN 0;
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK;
        RETURN -1;
    WHEN OTHERS THEN
        ROLLBACK;
        RETURN -1;
END inserta_familias;

mi_familia familia;
mi_familia1 familia;
familias_hijas tFamilias;
mi_familia2 familia;
hijos tAgentes;
BEGIN
    ...
    resultado := inserta_familia(mi_familia);
    ...
    resultado := inserta_familia(mi_familia1, familias_hijas);
    ...
    resultado := inserta_familia(mi_familia2, hijos);
    ...
END;
```

Anexo VI.- Ejemplo de recursividad.

Aquí tienes un ejemplo del uso de la recursividad en nuestros subprogramas.

```
DECLARE
    TYPE agente IS agentes%ROWTYPE;
    TYPE tAgentes IS TABLE OF agente;
    hijos10 tAgentes;
    PROCEDURE dame_hijos( id_familia NUMBER,
                          hijos IN OUT tAgentes ) IS
        CURSOR hijas IS SELECT identificador FROM familias WHERE familia = id_familia;
    hija NUMBER;
    CURSOR cHijos IS SELECT * FROM agentes WHERE familia = id_familia;
    hijo agente;
    BEGIN
        --Si la tabla no está inicializada -> la inicializamos
        IF hijos IS NULL THEN
            hijos = tAgentes();
        END IF;
        --Metemos en la tabla los hijos directos de esta familia
        OPEN cHijos;
        LOOP
            FETCH cHijos INTO hijo;
            EXIT WHEN cHijos%NOTFOUND;
            hijos.EXTEND;
            hijos(hijos.LAST) := hijo;
        END LOOP;
        CLOSE cHijos;
        --Hacemos lo mismo para todas las familias hijas de la actual
        OPEN hijas;
        LOOP
            FETCH hijas INTO hija;
            EXIT WHEN hijas%NOTFOUND;
            dame_hijos( hija, hijos );
        END LOOP;
        CLOSE hijas;
    END dame_hijos;
BEGIN
    ...
    dame_hijos( 10, hijos10 );
    ...
END;
```


Anexo VII.- Ejemplo de paquete.

Aquí puedes ver un ejemplo de un paquete que agrupa las principales tareas que llevamos a cabo con la base de datos de ejemplo.

```
CREATE OR REPLACE PACKAGE call_center AS      --inicialización
--Definimos los tipos que utilizaremos
SUBTYPE agente IS agentes%ROWTYPE;
SUBTYPE familia IS familias%ROWTYPE;
SUBTYPE oficina IS oficinas%ROWTYPE;
TYPE tAgentes IS TABLE OF agente;
TYPE tFamilias IS TABLE OF familia;
TYPE tOficinas IS TABLE OF oficina;

--Definimos las excepciones propias
referencia_no_encontrada exception;
referencia_encontrada exception;
no_null exception;
PRAGMA EXCEPTION_INIT(referencia_no_encontrada, -2291);
PRAGMA EXCEPTION_INIT(referencia_encontrada, -2292);
PRAGMA EXCEPTION_INIT(no_null, -1400);

--Definimos los errores que vamos a tratar
todo_bien                CONSTANT NUMBER := 0;
elemento_existente        CONSTANT NUMBER:= -1;
elemento_inexistente      CONSTANT NUMBER:= -2;
padre_existente           CONSTANT NUMBER:= -3;
padre_inexistente         CONSTANT NUMBER:= -4;
no_null_violado           CONSTANT NUMBER:= -5;
operacion_no_permitida    CONSTANT NUMBER:= -6;

--Definimos los subprogramas públicos
--Nos devuelve la oficina padre de un agente
PROCEDURE oficina_padre( mi_agente agente, padre OUT oficina );

--Nos devuelve la oficina padre de una familia
PROCEDURE oficina_padre( mi_familia familia, padre OUT oficina );

--Nos da los hijos de una familia
PROCEDURE dame_hijos( mi_familia familia, hijos IN OUT tAgentes );

--Nos da los hijos de una oficina
PROCEDURE dame_hijos( mi_oficina oficina, hijos IN OUT tAgentes );

--Inserta un agente
FUNCTION inserta_agente ( mi_agente agente )
RETURN NUMBER;

--Inserta una familia
FUNCTION inserta_familia( mi_familia familia )
RETURN NUMBER;
```

```

--Inserta una oficina
FUNCTION inserta_oficina ( mi_oficina oficina )
RETURN NUMBER;

--Borramos una oficina
FUNCTION borra_oficina( id_oficina NUMBER )
RETURN NUMBER;

--Borramos una familia
FUNCTION borra_familia( id_familia NUMBER )
RETURN NUMBER;

--Borramos un agente
FUNCTION borra_agente( id_agente NUMBER )
RETURN NUMBER;
END call_center;
/
CREATE OR REPLACE PACKAGE BODY call_center AS          --cuerpo
--Implemento las funciones definidas en la especificación

--Nos devuelve la oficina padre de un agente
PROCEDURE oficina_padre( mi_agente agente, padre OUT oficina ) IS
    mi_familia familia;
BEGIN
    IF (mi_agente.oficina IS NOT NULL) THEN
        SELECT * INTO padre FROM oficinas
        WHERE identificador = mi_agente.oficina;
    ELSE
        SELECT * INTO mi_familia FROM familias
        WHERE identificador = mi_agente.familia;
        oficina_padre( mi_familia, padre );
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        padre := NULL;
END oficina_padre;

--Nos devuelve la oficina padre de una familia
PROCEDURE oficina_padre( mi_familia familia, padre OUT oficina ) IS
    madre familia;
BEGIN
    IF (mi_familia.oficina IS NOT NULL) THEN
        SELECT * INTO padre FROM oficinas
        WHERE identificador = mi_familia.oficina;
    ELSE
        SELECT * INTO madre FROM familias
        WHERE identificador = mi_familia.familia;
        oficina_padre( madre, padre );
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        padre := NULL;
END oficina_padre;

--Nos da los hijos de una familia
PROCEDURE dame_hijos( mi_familia familia, hijos IN OUT tAgentes ) IS
    CURSOR cHijos IS SELECT * FROM agentes

```

```

WHERE familia = mi_familia.identificador;
CURSOR cHijas IS SELECT * FROM familias
WHERE familia = mi_familia.identificador;
hijo agente;
hija familia;
BEGIN
--inicializamos la tabla si no lo está
if (hijos IS NULL) THEN
    hijos := tAgentes();
END IF;
--metemos en la tabla los hijos directos
OPEN cHijos;
LOOP
    FETCH cHijos INTO hijo;
    EXIT WHEN cHijos%NOTFOUND;
    hijos.EXTEND;
    hijos(hijos.LAST) := hijo;
END LOOP;
CLOSE cHijos;
--hacemos lo mismo para las familias hijas
OPEN cHijas;
LOOP
    FETCH cHijas INTO hija;
    EXIT WHEN cHijas%NOTFOUND;
    dame_hijos( hija, hijos );
END LOOP;
CLOSE cHijas;
EXCEPTION
    WHEN OTHERS THEN
        hijos := tAgentes();
END dame_hijos;

--Nos da los hijos de una oficina
PROCEDURE dame_hijos( mi_oficina oficina, hijos IN OUT tAgentes ) IS
    CURSOR cHijos IS SELECT * FROM agentes
    WHERE oficina = mi_oficina.identificador;
    CURSOR cHijas IS SELECT * FROM familias
    WHERE oficina = mi_oficina.identificador;
    hijo agente;
    hija familia;
BEGIN
--inicializamos la tabla si no lo está
if (hijos IS NULL) THEN
    hijos := tAgentes();
END IF;
--metemos en la tabla los hijos directos
OPEN cHijos;
LOOP
    FETCH cHijos INTO hijo;
    EXIT WHEN cHijos%NOTFOUND;
    hijos.EXTEND;
    hijos(hijos.LAST) := hijo;
END LOOP;
CLOSE cHijos;
--hacemos lo mismo para las familias hijas
OPEN cHijas;
LOOP

```

```

        FETCH cHijas INTO hija;
        EXIT WHEN cHijas%NOTFOUND;
        dame_hijos( hija, hijos );
    END LOOP;
    CLOSE cHijas;
EXCEPTION
    WHEN OTHERS THEN
        hijos := tAgentes();
END dame_hijos;

--Inserta un agente
FUNCTION inserta_agente ( mi_agente agente )
RETURN NUMBER IS
BEGIN
    IF (mi_agente.familia IS NULL and mi_agente.oficina IS NULL) THEN
        RETURN operacion_no_permitida;
    END IF;
    IF (mi_agente.familia IS NOT NULL and mi_agente.oficina IS NOT NULL) THEN
        RETURN operacion_no_permitida;
    END IF;
    INSERT INTO agentes VALUES (mi_agente.identificador, mi_agente.nombre, mi_agente.oficina);
    COMMIT;
    RETURN todo_bien;
EXCEPTION
    WHEN referencia_no_encontrada THEN
        ROLLBACK;
        RETURN padre_inexistente;
    WHEN no_null THEN
        ROLLBACK;
        RETURN no_null_violado;
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK;
        RETURN elemento_existente;
    WHEN OTHERS THEN
        ROLLBACK;
        RETURN SQLCODE;
END inserta_agente;

--Inserta una familia
FUNCTION inserta_familia( mi_familia familia )
RETURN NUMBER IS
BEGIN
    IF (mi_familia.familia IS NULL and mi_familia.oficina IS NULL) THEN
        RETURN operacion_no_permitida;
    END IF;
    IF (mi_familia.familia IS NOT NULL and mi_familia.oficina IS NOT NULL) THEN
        RETURN operacion_no_permitida;
    END IF;
    INSERT INTO familias VALUES ( mi_familia.identificador, mi_familia.nombre, mi_familia.oficina );
    COMMIT;
    RETURN todo_bien;
EXCEPTION
    WHEN referencia_no_encontrada THEN
        ROLLBACK;
        RETURN padre_inexistente;
    WHEN no_null THEN
        ROLLBACK;

```

```

        RETURN no_null_violado;
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK;
        RETURN elemento_existente;
    WHEN OTHERS THEN
        ROLLBACK;
        RETURN SQLCODE;
END inserta_familia;

--Inserta una oficina
FUNCTION inserta_oficina ( mi_oficina oficina )
RETURN NUMBER IS
BEGIN
    INSERT INTO oficinas VALUES (mi_oficina.identificador, mi_oficina.nombre, mi_ofic:
    COMMIT;
    RETURN todo_bien;
EXCEPTION
    WHEN no_null THEN
        ROLLBACK;
        RETURN no_null_violado;
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK;
        RETURN elemento_existente;
    WHEN OTHERS THEN
        ROLLBACK;
        RETURN SQLCODE;
END inserta_oficina;

--Borramos una oficina
FUNCTION borra_oficina( id_oficina NUMBER )
RETURN NUMBER IS
    num_ofi NUMBER;
BEGIN
    SELECT COUNT(*) INTO num_ofi FROM oficinas
    WHERE identificador = id_oficina;
    IF (num_ofi = 0) THEN
        RETURN elemento_inexistente;
    END IF;
    DELETE oficinas WHERE identificador = id_oficina;
    COMMIT;
    RETURN todo_bien;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        RETURN SQLCODE;
END borra_oficina;

--Borramos una familia
FUNCTION borra_familia( id_familia NUMBER )
RETURN NUMBER IS
    num_fam NUMBER;
BEGIN
    SELECT COUNT(*) INTO num_fam FROM familias
    WHERE identificador = id_familia;
    IF (num_fam = 0) THEN
        RETURN elemento_inexistente;
    END IF;

```

```

        DELETE familias WHERE identificador = id_familia;
        COMMIT;
        RETURN todo_bien;
    EXCEPTION
        WHEN OTHERS THEN
            ROLLBACK;
            RETURN SQLCODE;
    END borra_familia;

--Borramos un agente
FUNCTION borra_agente( id_agente NUMBER )
RETURN NUMBER IS
    num_ag NUMBER;
BEGIN
    SELECT COUNT(*) INTO num_ag FROM agentes
    WHERE identificador = id_agente;
    IF (num_ag = 0) THEN
        RETURN elemento_inexistente;
    END IF;
    DELETE agentes WHERE identificador = id_agente;
    COMMIT;
    RETURN todo_bien;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        RETURN SQLCODE;
    END borra_agente;
END call_center;
/

```

