



# MANUAL DEL CURSO

## INDICE

---

|  |           |
|--|-----------|
| <b>CAPÍTULO 1: NOCIONES DE MODELAMIENTO DE DATOS .....</b> | <b>2</b>  |
| ENTIDADES .....  | 4         |
| RELACIONES .....   | 5         |
| CARDINALIDAD .....   | 6         |
| <b>CAPÍTULO 2: INTRODUCCIÓN A PL/SQL .....</b>             | <b>10</b> |
| ¿QUÉ ES PL/SQL? .....                                      | 10        |
| ESTRUCTURAS DE BLOQUE .....                                | 10        |
| VARIABLES Y CONSTANTES .....                               | 11        |
| CURSORES .....   | 12        |
| MANEJO DE ERRORES .....                                    | 13        |
| SUBPROGRAMAS .....   | 13        |
| PAQUETES .....   | 14        |
| VENTAJAS EN LA UTILIZACIÓN DE PL/SQL .....                 | 15        |
| <b>CAPÍTULO 3: FUNDAMENTOS DEL LENGUAJE .....</b>          | <b>18</b> |
| SET DE CARACTERES Y UNIDADES LÉXICAS .....                 | 18        |
| DELIMITADORES E IDENTIFICADORES .....                      | 19        |
| TIPOS DE DATOS Y CONVERSIONES .....                        | 21        |
| ALCANCE Y VISIBILIDAD .....                                | 24        |
| <b>CAPÍTULO 4: ESTRUCTURAS DEL LENGUAJE .....</b>          | <b>26</b> |
| CONTROL CONDICIONAL: SENTENCIA IF .....                    | 27        |
| CONTROLES DE ITERACIÓN: LAS SENTENCIAS LOOP Y EXIT .....   | 30        |
| CONTROLES DE SECUENCIA: LAS SENTENCIAS GOTO Y NULL .....   | 34        |
| SENTENCIAS SQL .....                                       | 36        |
| PROCESAMIENTO DE TRANSACCIONES .....                       | 42        |
| <b>CAPÍTULO 5: MANEJO DE CURSORES .....</b>                | <b>46</b> |
| DECLARACIÓN DE CURSORES .....                              | 47        |
| APERTURA DE UN CURSOR .....                                | 48        |
| RECUPERACIÓN DE FILAS .....                                | 49        |
| CIERRE DE UN CURSOR .....                                  | 51        |
| <b>CAPÍTULO 6: MANEJO DE ERRORES .....</b>                 | <b>52</b> |
| EXCEPCIONES PREDEFINIDAS .....                             | 52        |
| EXCEPCIONES DEFINIDAS POR EL USUARIO .....                 | 54        |
| USO DE SQLCODE Y SQLERRM .....                             | 56        |
| <b>CAPÍTULO 7: SUBPROGRAMAS .....</b>                      | <b>57</b> |
| PROCEDIMIENTOS .....                                       | 59        |
| FUNCIONES .....  | 60        |
| USO DE PARÁMETROS .....                                    | 62        |
| RECURSIVIDAD .....   | 64        |
| POLIMORFISMO .....   | 66        |
| <b>CAPÍTULO 8: PAQUETES .....</b>                          | <b>68</b> |
| VENTAJAS DE LA UTILIZACIÓN DE PAQUETES .....               | 69        |
| ESPECIFICACIÓN DE PAQUETES .....                           | 71        |
| CUERPO DE UN PAQUETE .....                                 | 71        |
| ANEXOS .....   | 73        |

## CAPÍTULO 1: NOCIONES DE MODELAMIENTO DE DATOS

---

Al diseñar un sistema de información o un proyecto de tecnología se debe tener en cuenta varios factores que intervienen en el desarrollo del mismo. El éxito del proyecto dependerá de la calidad con que se desarrollen todas las etapas que se identifiquen.

Algunas consideraciones se relacionan con reconocer ciertos componentes que participan en el diseño de una solución tecnológica, donde se incluye el análisis, el diseño lógico y físico de la solución y posteriormente su implantación.

En el ámbito de los sistemas de información, se reconoce que el estudio de los procesos de negocio deberá desembocar prontamente en el establecimiento de un modelo lógico de datos, que refleje de la mejor forma posible la complejidad que puede llegar a adquirir el sistema real.

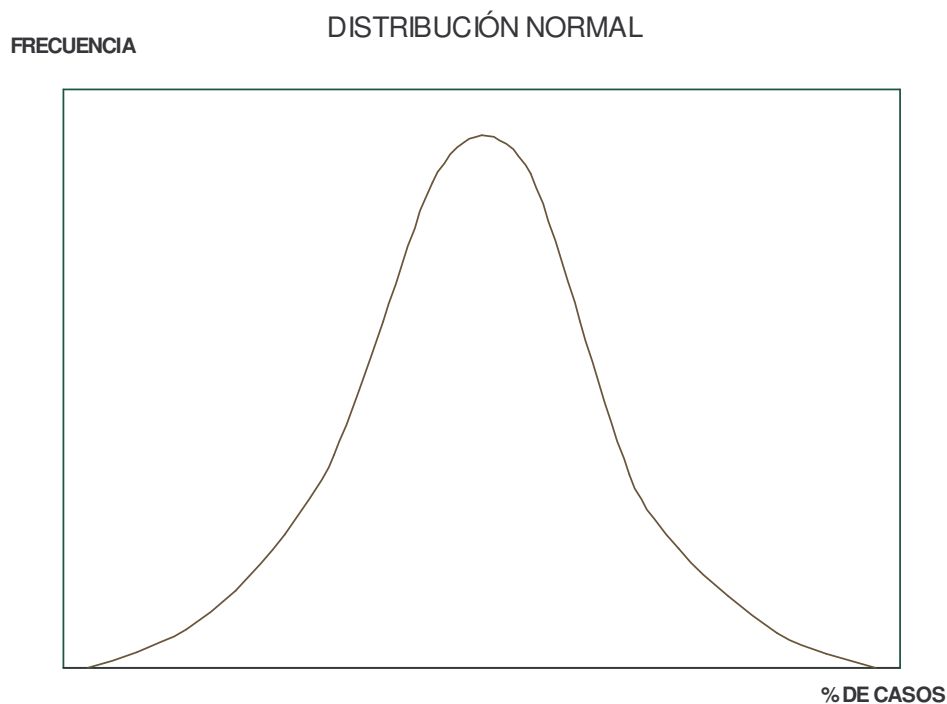


Figura 1-1 Curva de estudio de casos

Cuando se estudia un proceso de negocio o una situación cualquiera, se presenta una importante diversidad de componentes o casos, unos muy frecuentes, otros menos y algunos eventuales.

Cuando se trata de crear un sistema que refleje este proceso, surge la pregunta ¿cuáles son los componentes que voy a considerar en el diseño?.

Cuanto mayor sea el número de componentes, casos o situaciones, considerados, mayor será la complejidad del diseño. Particularmente, si deben considerarse las situaciones excepcionales, el diseño será muy complejo y por ende caro.

### *Abstracción*

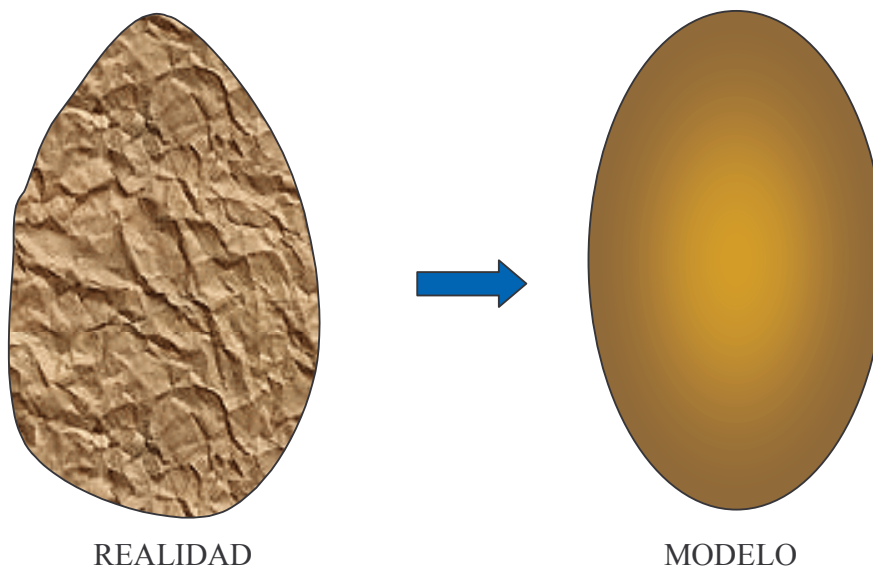


Figura 1-2: Abstracción de la realidad para generar un modelo

Abstracción es la actividad por la cual se recogen las características comunes más relevantes y esenciales de una **realidad**, para generar un **modelo**.

Este modelo deberá tener un **comportamiento**, desde el punto de vista de quién lo usará, semejante a la realidad que representa. Por esta razón el modelo deberá poseer tantos atributos de la realidad, como corresponda a su operación interna y a su relación con otros modelos.

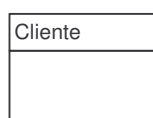
## Entidades

Entonces, cuando se modela una realidad, surge la necesidad de identificar aquellas *entidades* que conforman la situación en estudio y representarlas con objetos genéricos que tengan el nivel de abstracción adecuado que permitan reconocer las características del objeto que se está modelando. Estas entidades poseerán entonces características o atributos que las determinarán en forma detallada y que ayudarán incluso a relacionarlas con otras entidades del mismo modelo.

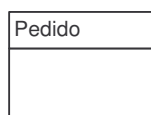
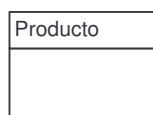
### Ejemplo:

Sea la entidad CLIENTE en una cadena de tiendas, que representa a las personas que compran los productos que son ofrecidos por la empresa.

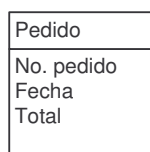
La notación más común para representar una entidad, en la génesis de un modelo de datos, es la siguiente:



Si la tienda ofrece determinados productos y la venta de éstos es registrada en un pedido, entonces las entidades PRODUCTO y PEDIDO también pertenecerán al modelo de datos lógico:



Ahora, cuando debemos detallar las características de estas entidades que son relevantes para el proceso de negocio que se está analizando, las escribimos en el recuadro inferior. En este ejemplo, las características relevantes de las entidades definidas podrían ser las siguientes:



### *Clave Primaria*

En todas las entidades es preciso identificar aquel atributo que identifica unívocamente a cada ocurrencia dentro de ella. Es la que llamamos *clave principal* o *primaria*.

Si observamos nuestro modelo, es claro que los atributos principales de cada entidad deberían ser los siguientes:

Cliente     RUT

Pedido     No. de Pedido

Producto     Código del Producto

Hasta aquí se han establecido ciertas entidades que, luego del análisis practicado a la situación en estudio, se han definido como las más representativas del modelo de negocio. También se han precisado las características más importantes de cada una de ellas, las cuales se conocen con el nombre de *atributos* en un modelo relacional.

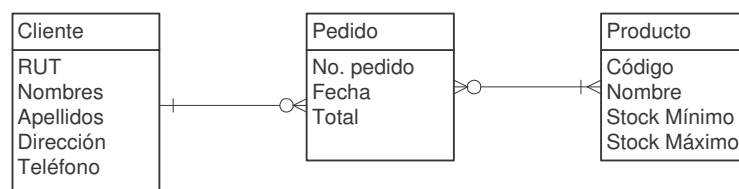
### **Relaciones**

A continuación, cuando ya se encuentra definida cada entidad en forma individual, el siguiente paso es descubrir las relaciones que existen entre ellas, las cuales también se determinan por las *reglas del negocio* que se está modelando.

En el ejemplo, se entiende que los clientes son los que realizan los pedidos. Además, cada pedido puede involucrar uno o más artículos (productos) ofrecidos por la empresa.

De esta manera, se desprende que las relaciones se producen entre las entidades, las cuales simbolizamos con una línea que las une.

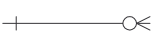
#### **Ejemplo:**



Al observar detenidamente estas líneas que unen a nuestras entidades podremos darnos cuenta que tienen formas muy diferentes. Esto no ha sido un capricho del editor de este manual, sino una nomenclatura muy conocida para representar las cardinalidades o número de veces en que las ocurrencias de una entidad participan sobre otra.

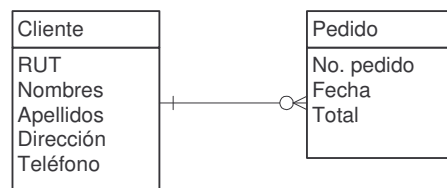
### **Cardinalidad**

Existen varias formas de anotar la cardinalidad de las relaciones que se identifican en un modelo. Todas ellas son dependientes de la posición, es decir, un mismo símbolo dibujado al revés significa una cosa totalmente diferente en el modelo. En este manual estudiaremos la notación siguiente:

1. 

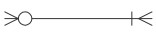
Este símbolo representa que cada ocurrencia de la entidad de la izquierda participa cero o muchas veces en una relación con algún ítem de la entidad de la derecha (<=). Al mismo tiempo, indica que una ocurrencia de la entidad de la derecha siempre se relaciona con algún ítem en la entidad de la izquierda (+).

Por ejemplo, en la relación:

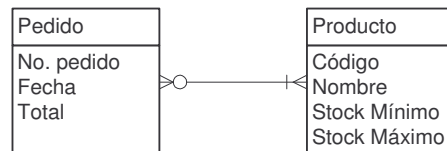




La cardinalidad indicada, al leerla de izquierda a derecha, asegura que “cada Cliente realiza 0 ó más pedidos a la tienda”. Por otro lado, también aclara que “un pedido sólo se asocia con un solo Cliente” (al leer de derecha a izquierda).

En una implementación física (un paso más adelante que este diseño) se podrá observar que algunos atributos o características de algunas entidades deben ser traspasadas a otras para mantener la integridad de los datos que se modelan en esta etapa.

2. 

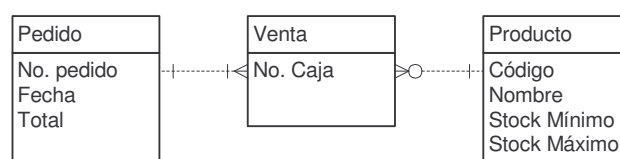
Este símbolo es especial porque está representando una relación de muchos ítems de una entidad sobre muchos otros de la otra entidad. En palabras, aterrizando a nuestro pequeño modelo, se leería de la siguiente forma:



- a) “Un pedido debe contener por lo menos un producto y como máximo varios de ellos”  
(+.
- b) “Un mismo producto puede aparecer ninguna o muchas veces en un mismo pedido”  
(.

La lectura es fácil, pero su implementación física requerirá de un paso adicional, que es romper esta relación de “muchos a muchos” y crear una nueva entidad que relacione de manera simple a cada una de las entidades “Producto” y “Pedido”.

Si se tuviera en cuenta este mismo punto al momento de efectuar el modelo lógico, podríamos haber propuesto un esquema como el siguiente:



En este nuevo modelo (o submodelo) se *rompió* la relación anterior y fabricamos dos nuevas relaciones más sencillas y que aportan tanta o más información que la anterior. Por ejemplo, ahora podemos conocer el número de la caja donde se efectuó la venta, que no era un atributo ni del pedido ni de los productos. Sin embargo, no era necesario buscar un atributo a la nueva entidad, puesto que ésta podría estar vacía si no existe un dato que la caracterice en forma adecuada dentro de nuestro modelo del negocio.



Finalmente, según nuestra manera natural de ver las cosas, entendemos que las entidades y relaciones revisadas en las páginas anteriores, aclaran la participación de los objetos del negocio para ejemplificar los temas revisados hasta aquí. Sin embargo, debo demostrar ahora que esta misma lógica no puede ser entendida por un sistema computacional o un administrador de bases de datos, por muy poderoso que éste sea.

La razón es muy simple y se ejemplificará de la siguiente manera:

Suponga las siguientes ocurrencias de las entidades anteriormente definidas:

#### Cliente

| Atributo         | Valores          |                   |                |
|------------------|------------------|-------------------|----------------|
|                  | A                | B                 | C              |
| <b>RUT</b>       | 5.555.555-5      | 6.666.666-6       | 7.777.777-7    |
| <b>Nombres</b>   | Sergio           | Marco Antonio     | Luis Fernando  |
| <b>Apellidos</b> | Contreras Ruiz   | Fernández Ríos    | Cárcamo Vera   |
| <b>Dirección</b> | Los Copihues #48 | Las Encinas #1000 | Apoquindo #777 |
| <b>Teléfono</b>  |                  | 6666666           | 7777777        |

#### Producto

| Atributo            | Valores    |                   |       |
|---------------------|------------|-------------------|-------|
|                     | A          | B                 | C     |
| <b>Código</b>       | A100       | B123              | C430  |
| <b>Nombre</b>       | Microondas | Mesa de Televisor | Silla |
| <b>Stock Mínimo</b> | 15         | 10                | 10    |
| <b>Stock Máximo</b> | 40         | 60                | 60    |

*Pregunta:*

Si el Cliente “A” efectúa un pedido de 3 sillas y un microondas. ¿Cómo debería representarlo la relación de “Pedido”?

*Respuesta:*

Antes de conocer la respuesta final, observemos cómo debería reflejarlo el modelo:

Supongamos el pedido número 1000, efectuado el 15 de diciembre por un total de \$115.000 (suponga que el microondas vale \$70.000 y cada silla \$15.000).

### Pedido

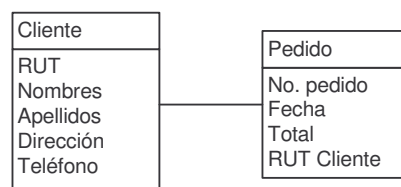
| Atributo          | Valores     |
|-------------------|-------------|
| <b>No. Pedido</b> | 1000        |
| <b>Fecha</b>      | 15-Dic-2000 |
| <b>Total</b>      | \$115.000   |

Al revisar esta ocurrencia en la entidad de “Pedido” es obvio que no podemos relacionarla de ninguna manera con algún cliente. Sin embargo, al modelarlo estuvo correcto, pero era el *dibujo* y la *cardinalidad* de la relación quienes nos informaban de que los pedidos los efectuaban los clientes en tal o cual cantidad.

Entonces, al implementar la solución debemos considerar también lo siguiente:

“Cada vez que encontremos una relación de uno-a-uno, simbolizada por  $\rightarrow$ , como en el caso de los pedidos y los clientes en la figura siguiente, se entenderá que el atributo principal de la entidad que es afectada por esta cardinalidad deberá traspasarse a la otra entidad”.

Como en nuestro modelo este tipo de relación se da entre “Cliente” y “Pedido”, entonces el modelo físico de esa relación se vería de la siguiente manera:



Y de esa manera, al pasar el atributo principal de “Cliente” (RUT) a la entidad “Pedido”, se logra establecer claramente un nexo físico en la base de datos.

## CAPÍTULO 2: INTRODUCCIÓN A PL/SQL

---

### ¿Qué es PL/SQL?

PL/SQL provee una manera muy cómoda de relacionar los conceptos de bases de datos y manejarlos mediante ciertas estructuras de control, dentro del contexto de una herramienta netamente de programación.

Su utilización es dentro del administrador de bases de datos “Oracle” y sus principales características son la posibilidad que brinda de utilizar sentencias SQL para manipular datos en Oracle y sentencias de control de flujo para organizar esta manipulación de datos.

Dentro del lenguaje, es posible declarar constantes y variables, definir procedimientos y funciones y atrapar errores en tiempo de ejecución. Así visto, PL/SQL combina la el poder de la manipulación de datos, con SQL, y las facilidades del procesamiento de los mismos, tal como en los más modernos lenguajes de programación.

### Estructuras de Bloque

PL/SQL es un lenguaje *estructurado en bloques*, lo que quiere decir que la unidad básica de codificación son bloques lógicos, los que a su vez pueden contener otros sub-bloques dentro de ellos, con las mismas características.

Un bloque (o sub-bloque) permite agrupar en forma lógica un grupo de sentencias. De esta manera se pueden efectuar declaraciones de variables que sólo tendrán validez en los bloques donde éstas se definan.

Un bloque PL/SQL tiene tres partes: una sección de *declaración*, una sección de *ejecución* y otra de manejo de *excepciones*. Sólo el bloque de ejecución es obligatorio en un programa PL/SQL.

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

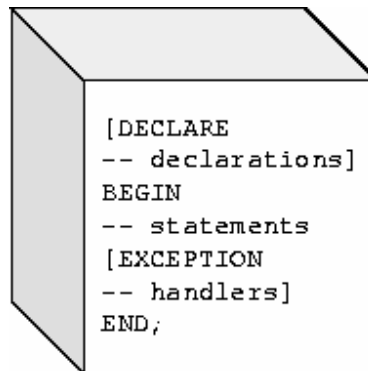


Figura 2-1: Estructura de bloques de un programa PL/SQL

### Variables y Constantes

PL/SQL permite declarar constantes y variables para ser utilizadas en cualquier expresión dentro de un programa. La única condición exigida por PL/SQL es que cada variable (o constante) debe estar declarada antes de ser utilizada en una expresión.

Las variables pueden corresponder a cualquier tipo de dato de SQL, tal como *char*, *date* o *number*, o algún tipo de PL/SQL, como *boolean* o *binary\_integer*.

Por ejemplo, si desea declarar una variable llamada “part\_no” que almacene cuatro dígitos numéricos y otra variable “in\_stock” de tipo booleano, es decir, que almacene solamente los valores True o False, la declaración se vería como sigue:

```
part_no      number(4) ;  
in_stock     boolean ;
```

#### *Cómo asignar valores a variables*

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

Por ejemplo:

```
tax := price * tax_rate ;  
bonus := current_salary * 0.10 ;  
amount := TO_NUMBER(SUBSTR('750 dólares', 1, 3)) ;  
valid := False ;
```

La segunda forma de asignar valores a variables es obtener valores directamente desde la base de datos, como en:

```
SELECT sal * 0.10 INTO bonus FROM emp WHERE empno = emp_id ;
```

### *Declaración de Constantes*

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

Ejemplo: *credit\_limit CONSTANT real := 5000.00 ;*

## **Cursores**

Los cursores son áreas de trabajo que permiten ejecutar sentencias SQL y procesar la información obtenida de ellos.

Hay dos tipos de cursores: implícitos y explícitos. PL/SQL declara implícitamente un cursor para todas las sentencias de manipulación de datos, incluyendo las consultas que retornan sólo una fila. Para consultas que devuelven más de una fila, es posible declarar explícitamente un cursor que procese las filas en forma individual.

Por ejemplo:

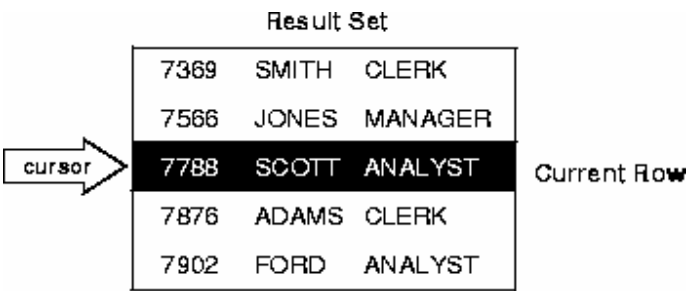
```
DECLARE
```

```
    CURSOR curs_01 IS
```

```
        SELECT empno, ename, job FROM emp WHERE deptno=20;
```

El conjunto de filas retornado se denomina “set de resultados”. Su tamaño está determinado por el número de filas que calzan con el criterio de selección de la query que implementa el cursor. Las filas son procesadas de a una cada vez.

En el capítulo dedicado exclusivamente a estos componentes del lenguaje, se detallarán las características de las diferentes modalidades de utilización de los cursores.



The diagram illustrates a cursor pointing to the current row in a result set. A box labeled 'Result Set' contains a table with five rows. An arrow labeled 'cursor' points to the third row, which is highlighted in black. To the right of the table, the text 'Current Row' is displayed.

| Result Set |       |         |
|------------|-------|---------|
| 7369       | SMITH | CLERK   |
| 7566       | JONES | MANAGER |
| 7788       | SCOTT | ANALYST |
| 7876       | ADAMS | CLERK   |
| 7902       | FORD  | ANALYST |

Current Row

Figura 2-2: Recuperación de filas a través de un cursor

### Manejo de Errores

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

Cuando ocurre un error se procesa una excepción, esto es, se detiene la ejecución normal del programa y se transfiere el control a un segmento especial del programa que tiene por objeto manejar estas situaciones excepcionales. Estas rutinas que se codifican en forma separada se conocen con el nombre de *exception handlers*.

Las excepciones predefinidas son gatilladas automáticamente por el sistema cuando ocurre un error de cierta naturaleza. Además, es posible alcanzar excepciones definidas con el usuario, simplemente haciendo un llamado a éstas utilizando la sentencia *raise*.

### Subprogramas

En PL/SQL existen dos tipos de subprogramas, llamados procedimientos y funciones, los que pueden manejar parámetros de entrada y de salida. Un subprograma es un programa en miniatura, que comienza con un encabezado, una sección opcional de declaraciones, una sección de ejecución y una sección opcional de manejo de excepciones, como cualquier otro programa de PL/SQL.

## **Paquetes**

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

Los paquetes se descomponen en dos partes: una **especificación** y un **cuerpo**.

La *especificación* (package specification) es idéntica a una sección de declaración de aplicaciones. En esta especie de encabezado es posible declarar tipos, constantes, variables, excepciones, cursores y subprogramas disponibles para su uso en el *cuerpo* del paquete. De esta manera, el cuerpo (package body) define la implementación de esos subprogramas declarados en el apartado anterior.

Ejemplo de uso de paquetes:

```
CREATE PACKAGE emp_actions as                -- package specification
    PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) ;
    PROCEDURE fire_employee (empid NUMBER) ;
END emp_actions ;
```

```
CREATE PACKAGE BODY emp_actions AS           -- package body
    PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) IS
    BEGIN
        INSERT INTO emp VALUES (empno, ename, ...);
    END hire_employee;
    PROCEDURE fire_employee (emp_id NUMBER) IS
    BEGIN
        DELETE FROM emp WHERE empno = emp_id;
    END fire_employee;
END emp_actions;
```

Los paquetes pueden ser compilados y almacenados en una base de datos Oracle y su contenido puede ser compartido por varias aplicaciones. Cuando un paquete es llamado para su ejecución, éste se almacena completamente en memoria la primera vez. Las siguientes llamadas no requieren efectuar este procedimiento cada vez y por esto aumentan la eficiencia de los programas.

### **Ventajas en la utilización de PL/SQL**

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

Soporte para SQL

Soporte para la programación orientada a objetos

Mejor rendimiento

Alta productividad

Completa portabilidad

Integración con Oracle garantizada

Seguridad

#### *Soporte para SQL*

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

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

PL/SQL le permite a usted una completa manipulación de los datos almacenados en una base Oracle, proporciona comandos de control de transacciones y permite utilizar las funciones de SQL, operadores y pseudocolumnas. Así, usted puede manipular los datos en Oracle de una manera flexible y segura. Además, PL/SQL soporta tipos de datos de SQL, lo que reduce la necesidad de convertir los datos al pasar de una a otra aplicación.

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

#### *Soporte para Programación Orientada a Objetos*

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



diferentes equipos de programadores construyendo aplicaciones basadas en el mismo grupo de objetos.

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

### *Mejor rendimiento*

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

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

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

### *Alta productividad*

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

### *Completa portabilidad*

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

*Integración con Oracle*

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

*Seguridad*

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

## CAPÍTULO 3: FUNDAMENTOS DEL LENGUAJE

---

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

### Set de Caracteres y Unidades Léxicas

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

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

Una línea de texto en un programa contiene lo que se conoce como unidades léxicas, los que se clasifican como sigue:

- Delimitadores (símbolos simples y compuestos)
- Identificadores (incluye palabras reservadas)
- Literales
- Comentarios

Por ejemplo en la instrucción:

```
bonus := salary * 0.10; -- cálculo del bono
```

se observan las siguientes unidades léxicas:

- Los identificadores *bonus* y *salary*

- El símbolo compuesto `:=`
- Los símbolos simples `*` y `;`
- El literal numérico `0.10`
- El comentario “*cálculo del bono*”

Para asegurar la fácil comprensión del código se pueden añadir espacios entre identificadores o símbolos. También es una buena práctica utilizar saltos de línea e indentaciones para permitir una mejor legibilidad.

Ejemplo:

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

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

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

### **Delimitadores e Identificadores**

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

| Símbolo | Significado               |
|---------|---------------------------|
| +       | operador de suma          |
| %       | indicador de atributo     |
| ‘       | delimitador de caracteres |
| .       | selector de componente    |
| /       | operador de división      |

|   |  |
|---|--|
| ( | expresión o delimitador de lista               |
| ) | expresión o delimitador de lista               |
| : | indicador de variable host                     |
| , | separador de ítems                             |
| * | operador de multiplicación                     |
| “ | delimitador de un identificador entre comillas |
| = | operador relacional                            |
| < | operador relacional                            |
| > | operador relacional                            |
| @ | indicador de acceso remoto                     |
| ; | terminador de sentencias                       |
| - | negación u operador de substracción            |

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

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

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

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

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

### **Tipos de Datos y Conversiones**

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

En la figura 3-1 se muestran los diferentes tipos de datos predefinidos y disponibles para ser utilizados.

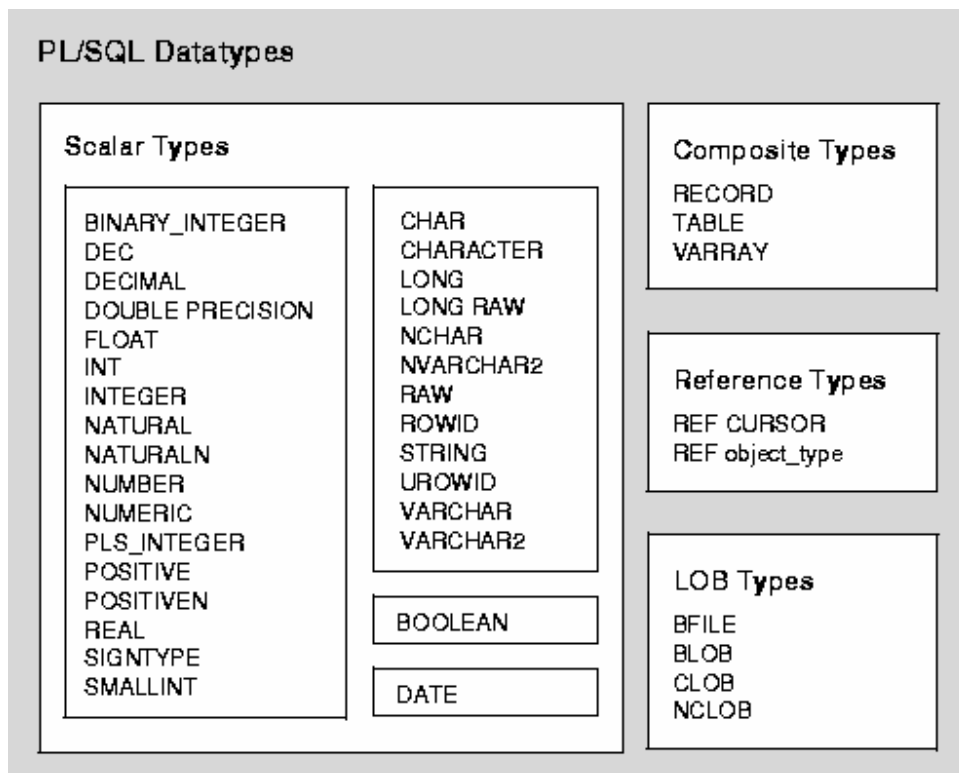


Figura 3-1: Tipos de datos de PL/SQL

### Conversiones

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

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

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

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

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

**Tabla de conversiones implícitas**

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

### *Uso de %TYPE*

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

#### Ejemplo:

DECLARE

credito REAL(7,2);

debito credito%TYPE;

...

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

debito cuenta.debe%TYPE;

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



### *Uso de %ROWTYPE*

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

#### Ejemplo:

```
DECLARE
```

```
    emp_rec    emp%ROWTYPE;
```

```
    CURSOR c1 IS SELECT deptno, dname, loc FROM dept;
```

```
    dept_rec    c1%ROWTYPE;
```

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

### **Alcance y Visibilidad**

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

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

La [figura 3-2](#) muestra el alcance y visibilidad de la variable *x*, la cual está declarada en dos bloques cerrados diferentes.

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

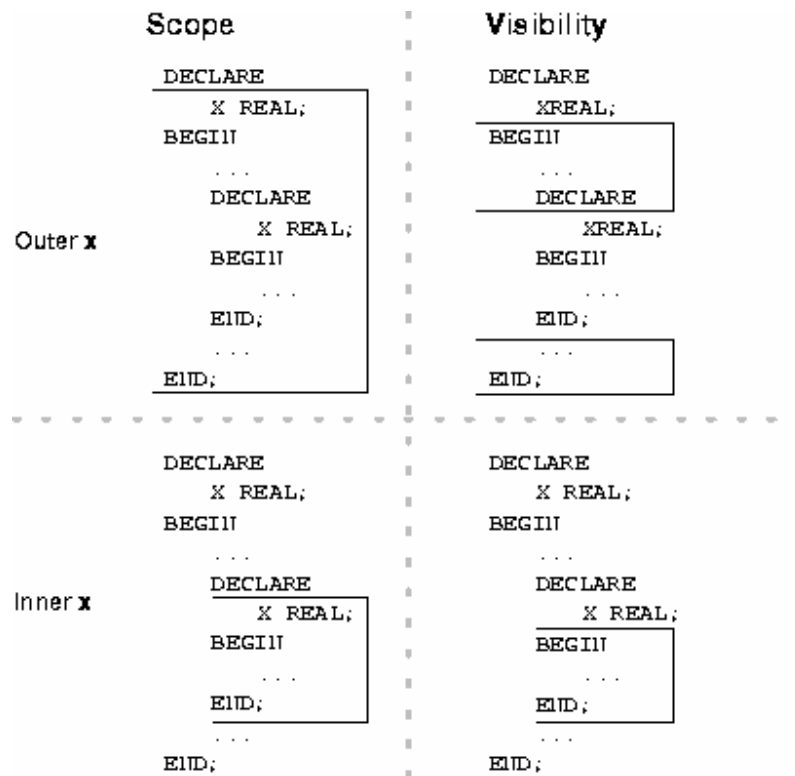


Figura 3-2: Alcance y Visibilidad de Identificadores

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

## CAPÍTULO 4: ESTRUCTURAS DEL LENGUAJE

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

De acuerdo con el *Teorema de la Estructura*, cualquier programa computacional puede ser escrito utilizando las estructuras básicas de control que se muestran en la [figura 4-1](#). Estas se pueden combinar de todas las maneras necesarias para alcanzar la solución de un problema dado.

Las estructuras de selección verifican cierta condición, después ejecutan cierta secuencia de expresiones dependiendo si la condición resultó ser verdadera o falsa. Una *condición* es cualquier variable o expresión que retorna un valor booleano (TRUE o FALSE). Las estructuras de iteración ejecutan una secuencia de sentencias repetidamente mientras la condición permanezca verdadera. Las estructuras de secuencia simplemente ejecutan una secuencia de estamentos en el orden que ocurren.

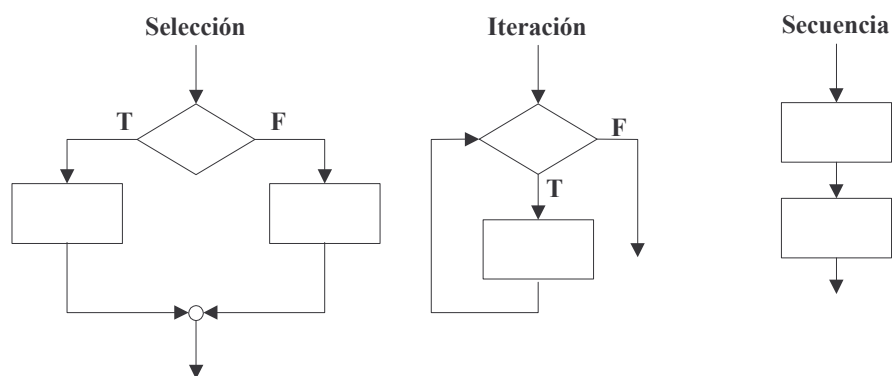


Figura 4-1: Estructuras de Control

### **Control Condicional: Sentencia IF**

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

#### **IF – THEN**

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

#### **Ejemplo:**

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

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

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

#### **IF – THEN – ELSE**

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

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

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

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

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

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

### IF – THEN – ELSIF

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

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

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

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

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

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

### **Controles de Iteración: Las sentencias LOOP y EXIT**

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

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

#### **LOOP**

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

```
LOOP
    secuencia_de_instrucciones
END LOOP;
```

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

#### **EXIT**

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

## LOOP

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

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

## EXIT – WHEN

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

### Ejemplo:

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

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

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



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

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

### *Etiquetas*

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

<<rótulo>>

LOOP

    secuencia de sentencias

END LOOP;

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

### WHILE - LOOP

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

WHILE condición LOOP

    secuencia\_de\_sentencias

END LOOP;

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

### FOR - LOOP

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

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

El contador no necesita ser declarado porque por defecto se crea para el bloque que involucra el ciclo y luego se destruye.

Por defecto, la iteración ocurre en forma creciente, es decir, desde el menor valor aportado hasta el mayor. Sin embargo, si se desea alterar esta condición por defecto, se debe incluir explícitamente en la sentencia la palabra REVERSE.

Los límites de una iteración pueden ser literales, variables o expresiones, pero que deben evaluarse como números enteros.

Un contador de loop tiene validez sólo dentro del ciclo. No es posible asignar un valor a una variable contadora de un loop, fuera de él.

#### Ejemplo:

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

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

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

O también se puede utilizar:

```
<<externo>>
FOR i IN 1..5 LOOP
    ...
    FOR j IN 1..10 LOOP
        FETCH c1 INTO empref;
        EXIT externo WHEN c1%NOTFOUND;      -- sale de ambos ciclos
    ...
    END LOOP;
END LOOP externo;
-- el control retorna a esta línea
```

### **Controles de Secuencia: Las sentencias GOTO y NULL**

Ocasionalmente podría ser útil la utilización de una sentencia de este tipo. A pesar de que es sabido que la sentencia GOTO redundante en un código más complejo y desordenado a veces podría cooperar en la implementación de un programa. La sentencia nula puede cooperar con la fácil lectura del código cuando este sobrepasa una cantidad determinada de instrucciones y torna más difícil su comprensión.

#### **GOTO**

La sentencia GOTO obliga a saltar a un rótulo del programa en forma incondicional. El rótulo debe ser único dentro de su alcance y debe preceder a una sentencia ejecutable o a un

bloque PL/SQL. Cuando es ejecutada, esta instrucción transfiere el control a la sentencia o bloque rotulada.

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

Ejemplo válido:

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

Ejemplo no válido:

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

### *Restricciones*

Algunas restricciones en la utilización de un GOTO son las siguientes: una sentencia de este tipo no puede saltar dentro de una sentencia IF, LOOP o un sub-bloque. Tampoco se puede utilizar GOTO dentro del bloque de excepciones para salir de él.

### NULL

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

#### Ejemplo:

EXCEPTION

WHEN zero\_divide THEN

Rollback;

WHEN value\_error THEN

INSERT INTO errores VALUES...

Commit;

WHEN others THEN

NULL;

END;

### **Sentencias SQL**

#### SENTENCIA de selección - SELECT

La selección sobre una tabla consiste en elegir un subconjunto de filas que cumplan (o no) algunas condiciones determinadas.

La sintaxis de una sentencia de este tipo es la siguiente:

SELECT \* / <columna1, columna2,...>

FROM <nombre-tabla>

[WHERE <condición>]

[GROUP BY <columna1, columna2,...> ]

[HAVING <condición-selección-grupos> ]

[ORDER BY <columna1 [DESC], columna2 [DESC]...> ]

*\*/<columna1, columna2,...>*

Si se escribe \*, selecciona todas las columnas. Si se desea seleccionar sólo algunas columnas de la tabla, se debe poner los nombres de cada una de ellas, separadas por una coma.

*<nombre-tabla>*

Nombre de la(s) tabla(s) de la(s) cual(es) se van a seleccionar los valores.

*WHERE <condición>*

Cláusula opcional que se utiliza cuando se desea establecer una o varias condiciones para la sentencia “Select”.

Las condiciones podrán tener:

- operadores aritméticos: =, <, >, >=, <=, <=>
- operadores booleanos: AND, OR
- operadores especiales como:

BETWEEN, que permite obtener todas las filas que se encuentran en un intervalo de valores.

Formato: nombre-columna BETWEEN limite-inferior AND limite-superior

IN, que permite obtener todas las filas que sean iguales a alguno de los valores descritos por extensión.

Formato: nombre-columna IN (valor1, valor2, .....)

LIKE, que permite imponer condiciones sobre series de caracteres o parte de ellos.

- El símbolo “%” se utiliza como carácter de sustitución para indicar un número indeterminado de caracteres. Depende de la base de datos en la que se esté trabajando.
- El símbolo “\_” se utiliza como carácter de sustitución para indicar un carácter en una determinada posición. Depende de la base de datos en la que se esté trabajando.

(Los caracteres % y \_ pueden combinarse)

Formato: nombre-columna LIKE ‘A%’

NOT, todas las condiciones que se pueden utilizar, pueden negarse anteponiendo la partícula NOT delante de los operadores especiales: IN, LIKE, BETWEEN.

IS NULL, obtiene todas las filas que contengan un valor nulo en una determinada columna.

Formato: nombre-columna IS NULL

También se puede negar con la partícula NOT, IS NOT NULL

Una sentencia “Select” puede tener una o más condiciones unidas por un operador booleano.

*GROUP BY <columna1, columna2,...>*

Se utiliza para agrupar resultados por una determinada columna, específicamente cuando se utilizan funciones de columna y los resultados se desean obtener por grupos (SQL lanza un *sort* para generar los grupos).

*HAVING <condición-selec-grupos>*

Se utiliza con la cláusula “GROUP BY”, cuando se quiere poner condiciones al resultado de un grupo.

*ORDER BY <column1 [DESC], column2 [DESC]...>*

Sirve para ordenar el resultado. Todas las columnas por las que se desee realizar el orden tienen que encontrarse en la sentencia “Select” de la consulta.

El orden de las columnas puede ser ascendente, (por omisión, ASC), o descendente, (DESC).

### SENTENCIA SELECT (JOIN)

Consiste en la unión de campos de dos o más tablas. Dichas tablas tendrán por lo menos una columna común que sirva de nexo del join.

SELECT <columna1, columna2,...>

FROM <nombre-tabla1>, <nombre-tabla2>

*<columna1, columna2,...>*

Para diferenciar las columnas con el mismo nombre se antepondrá el nombre de la tabla a la que pertenecen, utilizando el punto como separador.

Por ejemplo:

```
SELECT Tabla1.Columna2, Tabla2.Columna2, Columna3.....
```

```
FROM Tabla1, Tabla2
```

```
WHERE Tabla1.Columna1 = Tabla2.Columna1
```

La Columna1 de cada una de las tablas respectivas son las columnas de nexo o columnas de join.

### SENTENCIA SELECT DISTINCT

Recupera las filas de una tabla eliminando los valores de la columna duplicados.

```
SELECT DISTINCT <columna1>, <columna2,...>
```

```
FROM <nombre-tabla1>, <nombre-tabla2>
```

```
[GROUP BY <columna1, columna2....> ]
```

```
[HAVING <condición-selección-grupos>]
```

```
[ORDER BY <columna1 [DESC], columna2 [DESC]...>]
```

### FUNCIONES SOBRE COLUMNAS

- COUNT. Indica el número de filas que cumplen una determinada condición, o el número de valores diferentes que posee una columna.

Formato: COUNT(\*) o COUNT(DISTINCT <columna>)

- SUM. Suma los valores de una columna.

Formato: SUM(<columna>)

- AVG. Entrega la media de los valores de una columna.

Formato: AVG(<columna>)

- MIN. Entrega el valor mínimo de una columna.

Formato: MIN(<columna>)

- MAX. Entrega el valor máximo de una columna.



Formato: MAX(<columna>)

### SUBSELECTS

Permite realizar comparaciones con valores obtenidos en otra sentencia select anidada, a la que se denomina “Subselect” o “Subselect interna”.

```
SELECT <columna1>, <columna2,...>
FROM   <nombre-tabla1>, <nombre-tabla2>
WHERE  <columna1> = (SELECT <columna1>
                     FROM <nombre-tabla1>, <nombre-tabla2>
                     WHERE <condición>)
```

(Cuando en la condición se pone el operador =, la subselect deberá recuperar un sólo registro).

### CLÁUSULA UNION

Mezcla los resultados de dos o más consultas individuales en una única tabla resultado, eliminando las filas duplicadas, si existieran.

```
SELECT <columna1>, <columna2,...>
FROM   <nombre-tabla1>, <nombre-tabla2>
WHERE  <condición>
UNION [ALL]
SELECT <columna1>, <columna2,...>
FROM   <nombre-tabla1>, <nombre-tabla2>
WHERE  <condición>
```

### *ALL*

Si se especifica ALL, el resultado de la query no elimina las filas duplicadas, si existieran.

Primero realiza cada una de las Select escritas, generando una tabla resultado por cada una de las consultas. Luego, une las dos tablas en una. Las columnas de la tabla resultado poseen los nombres de las columnas de la primera sentencia “Select” que se ejecute.

### SENTENCIA INSERT

Añade filas a una tabla. Posee varios formatos posibles:

- INSERT INTO <nombre-tabla> VALUES (<serie de valores>)

El orden en el que se asignen los valores en la cláusula VALUES tiene que coincidir con el orden en que se definieron las columnas en la creación del objeto tabla, dado que los valores se asignan por posicionamiento relativo.

- INSERT INTO <nombre-tabla> (<columna1>, <columna2>.....) VALUES (<valor1>, <valor2>....)

En este caso los valores se asignarán a cada una de las columnas mencionadas por posicionamiento relativo, siendo necesario que por lo menos se asignen valores a todas aquellas columnas que no admiten valores nulos en la tabla.

### SENTENCIA INSERT CON MÚLTIPLES FILAS

Para insertar un subconjunto de filas de una tabla en otra se escribe una sentencia INSERT con una SUBSELECT interna. Los formatos posibles son:

- INSERT INTO <nombre-tabla> (<columna1>, <columna2>.....) SELECT (<sentencia Select>)

Asigna a las columnas los valores recuperados en la sentencia Select. Inserta en la tabla todas las filas que se recuperen en la Select.

- INSERT INTO <nombre-tabla> SELECT \* FROM <nombre-tabla-fuente>

En este caso las estructuras de las tablas tienen que ser iguales.

### SENTENCIA UPDATE

Actualiza valores de una o más columnas para un subconjunto de filas de una tabla.

```
UPDATE <nombre-tabla>
```

```
    SET <columna1> = valor1 [, <columna2> = valor2 ...]
```

```
[WHERE <condición>]
```

Actualiza los campos correspondientes junto con los valores que se le asignen, en el subconjunto de filas que cumplan la condición de selección. Si no se pone condición de selección, la actualización se da en todas las filas de la tabla.

Si se desea actualizar a nulos, se asignará el valor NULL.

### SENTENCIA DELETE

Borra una o más filas de una tabla. La sintaxis es la siguiente:

```
DELETE FROM <nombre-tabla>
```

```
[WHERE <condición>]
```

Si no se pone condición de selección, borra todas las filas de la tabla.

## **Procesamiento de Transacciones**

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

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

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

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

Las sentencias *commit* y *rollback* permiten asegurar que todos los cambios efectuados sobre la base de datos se guardarán permanentemente o se descartarán en forma definitiva. Todas las sentencias que se ejecutan desde la ocurrencia del último *commit* o *rollback* comprenden la transacción (o grupo de transacciones) activa. La sentencia *savepoint* establece un punto de procesamiento dentro de una transacción y funciona de manera similar a un rótulo (ver capítulo 3).

#### Uso de COMMIT

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

#### Ejemplo:

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

La sentencia COMMIT libera todas las filas bloqueadas de la tabla “cuenta”. La palabra clave “work” no tiene otro efecto que permitir la fácil lectura de la instrucción, pero es perfectamente prescindible dentro del lenguaje.

Note que la palabra *end* al final del código indica el final del bloque, no el fin de la transacción. Una transacción puede abarcar más de un bloque, como también dentro de un mismo bloque pueden ocurrir muchas transacciones.

### Uso de ROLLBACK

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

#### Ejemplo:

```
DECLARE
```

```
    emp_id integer;
```

```
    ...
```

```
BEGIN
```

```
    SELECT empno, ... INTO emp_id, ... FROM new_emp WHERE ...
```

```
    ...
```

```
    INSERT INTO emp VALUES(emp_id, ...);
```

```
    INSERT INTO tax VALUES(emp_id, ...);
```

```
    INSERT INTO pay VALUES(emp_id, ...);
```

```
    ...
```

```
EXCEPTION
```

```
    WHEN DUP_VAL_ON_INDEX THEN
```

```
        ROLLBACK;
```

```
    ...
```

```
END;
```

### Uso de SAVEPOINT

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

#### Ejemplo:

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

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

## **CAPÍTULO 5: MANEJO DE CURSORES**

---

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

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

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

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

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

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

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

### **Declaración de Cursores**

Los cursores deben ser declarados antes de ser utilizados en otras sentencias. Cuando se declara un cursor, a éste se le da un nombre y se asocia con una consulta específica usando la sintaxis siguiente:

```
DECLARE  
CURSOR nombre_cursor [ (parámetro1 [, parámetro2]...) ]  
[RETURN tipo_de_retorno] IS sentencia_select ;
```

Donde *tipo\_de\_retorno* debe representar a un registro o una fila en una tabla de la base y los parámetros siguen la siguiente sintaxis:

```
nombre_del_parametro [IN] tipo_de_dato [ { := | DEFAULT } expresión ]
```

Por ejemplo, los cursores **c1** y **c2** se pueden declarar como sigue:

```
DECLARE  
    CURSOR c1 IS SELECT empno, ename, job, sal FROM emp WHERE sal>1000 ;  
    CURSOR c2 RETURN dept%ROWTYPE IS  
        SELECT * from dept WHERE deptno = 10 ;  
    ...
```

El nombre del cursor (*c1* y *c2* en el ejemplo) corresponde a un identificador no declarado, no a un nombre de variable de PL/SQL. No se pueden asignar valores a un nombre de cursor ni utilizarlos en una expresión.

Un cursor puede recibir parámetros, los cuales deben ser declarados con la cláusula IN para formalizar su incorporación. Los parámetros pueden ser inicializados con algún valor, pero estos pueden ser cambiados en cualquier oportunidad.

El alcance de los parámetros es local al cursor, lo que significa que ellos sólo pueden ser referenciados dentro de la consulta especificada en la declaración del mismo. Estos valores son utilizados por la query cuando el cursor se abre.



### **Apertura de un Cursor**

Al abrir un cursor se ejecuta inmediatamente la consulta e identifica el conjunto resultado, el que consiste de todas las filas que concuerdan con el criterio de selección de éste. Para los cursores que se abren con la cláusula “For Update”, la sentencia de apertura del cursor además bloquea esas filas retornadas. Un ejemplo es el siguiente:

```
DECLARE
    CURSOR c1 IS SELECT ename, job FROM emp WHERE sal > 3000;
    ...
BEGIN
    OPEN c1;
    ...
END;
```

### *Pasaje de Parámetros*

Se utiliza también la sentencia OPEN para pasar los parámetros al cursor, en caso de que éste los requiera. Por ejemplo:

```
DECLARE
    emp_name          emp.name%TYPE;
    salary             emp.sal%TYPE;
    CURSOR c1 (name VARCHAR2, salary NUMBER) IS SELECT...
```

Cualquiera de estas sentencias abre el cursor:

```
OPEN c1(emp_name, 3000);
OPEN c1(“John”, 1500);
OPEN c1(emp_name, salary);
```

Obsérvese que en este ejemplo, cuando se utiliza la variable “salary” en la declaración se refiere al nombre del segundo parámetro del cursor. Sin embargo, cuando es usada en una sentencia OPEN se refiere a la variable del programa.

### **Recuperación de Filas**

La sentencia FETCH permite recuperar los conjuntos de datos de a uno a la vez. Después de cada recuperación y carga de un set de datos el cursor avanza a la fila siguiente.

#### **Ejemplo:**

```
FETCH c1 INTO my_empno, my_ename, my_deptno;
```

Para cada columna retornada en un cursor y especificada en la declaración del mismo debe existir una variable compatible en tipo en la lista INTO.

Típicamente se utiliza la sentencia FETCH dentro de un conjunto de instrucciones como el siguiente:

```
LOOP
```

```
    FETCH c1 INTO mi_registro;
```

```
    EXIT WHEN c1%NOTFOUND;
```

```
    --- se procesa el registro
```

```
END LOOP;
```

Eventualmente, la sentencia FETCH fallará, es decir, no retornará ningún conjunto de datos. Cuando esto sucede, no se gatilla una excepción, luego se debe detectar esta condición utilizando los atributos del cursor %FOUND y/o %NOTFOUND.

Algunos atributos de los cursores se detallan a continuación:

#### *Uso de %FOUND*

Luego de que un curso ha sido abierto, pero antes de recuperar la primera fila el valor del atributo %FOUND es nulo. A continuación, tomará el valor TRUE cada vez que obtenga una fila del set de resultados (en cada FETCH que se haga) y sólo alcanzará el valor FALSE cuando ya no existan más filas para mostrar en el set de resultados.

Ejemplo:

## LOOP

```
    FETCH c1 INTO ...
    IF c1%FOUND THEN      -- fetch exitoso
        ...
    ELSE                  -- fetch falló; se sale del loop
        EXIT;
    END IF;
END LOOP;
```

*Uso de %NOTFOUND*

Es el opuesto lógico de %FOUND. Cada vez que una sentencia FETCH retorne una fila válida, este atributo devolverá FALSO. Sólo alcanzará el valor TRUE cuando no haya más filas en un cursor y se ejecute la sentencia FETCH (sin éxito por lo tanto).

Ejemplo:

## LOOP

```
    FETCH c1 INTO ...
    EXIT WHEN c1%NOTFOUND;
    ...
END LOOP;
```

*Uso de %ISOPEN*

Este atributo toma el valor verdadero (TRUE) cuando un cursor se encuentra abierto. De otra manera, retorna FALSO.

*Uso de %ROWCOUNT*

Cuando un cursor es abierto, este atributo es seteado en 0 (cero). En adelante, cada vez que se recuperen filas exitosamente con un FETCH, este valor se irá incrementando en uno.

Cuando se utiliza con cursores implícitos, este atributo devuelve el total de filas afectadas por una instrucción del tipo INSERT, UPDATE o DELETE.

**Cierre de un Cursor**

La sentencia que deshabilita un cursor, CLOSE, se utiliza de la siguiente manera:

```
CLOSE c1;
```

Una vez que un cursor ya ha sido cerrado, es posible volverlo a abrir sin tener que declararlo otra vez. Cualquier otra operación que se desee efectuar sobre un cursor no operativo (cerrado) provocará una excepción del tipo “invalid\_cursor”.

## CAPÍTULO 6: MANEJO DE ERRORES

---

En PL/SQL una advertencia o condición de error es llamada una *excepción*. Estas pueden ser definidas en forma interna (en tiempo de ejecución de un programa) o explícitamente por el usuario. Ejemplos de excepciones definidas en forma interna son la división por cero y la falta de memoria en tiempo de ejecución. Estas mismas condiciones excepcionales tienen sus nombres propios y pueden ser referenciadas con ellos: *zero\_divide* y *storage\_error*.

También se pueden definir excepciones a medida y nombrarlas de alguna forma, utilizando las reglas de construcción mencionadas en el capítulo 2.

Cuando ocurre un error se alcanza la excepción, esto quiere decir que se ejecuta la porción del programa donde ésta se encuentra implementada, transfiriéndose el control a ese bloque de sentencias. Las excepciones definidas por el usuario deben ser alcanzadas explícitamente utilizando la sentencia *raise*.

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

### **Excepciones predefinidas**

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

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

| Nombre Excepción               | Gatillada cuando...  | SQLCODE |
|--------------------------------|--|---------|
| <b>ACCESS_INTO_NULL</b>        | El programa intentó asignar valores a los atributos de un objeto no inicializado   | -6530   |
| <b>COLLECTION_IS_NULL</b>      | El programa intentó asignar valores a una tabla anidada aún no inicializada  | -6531   |
| <b>CURSOR_ALREADY_OPEN</b>     | El programa intentó abrir un cursor que ya se encontraba abierto. Recuerde que un cursor de ciclo FOR automáticamente lo abre y ello no se debe especificar con la sentencia OPEN                                  | -6511   |
| <b>DUP_VAL_ON_INDEX</b>        | El programa intentó almacenar valores duplicados en una columna que se mantiene con restricción de integridad de un índice único (unique index)  | -1      |
| <b>INVALID_CURSOR</b>          | El programa intentó efectuar una operación no válida sobre un cursor   | -1001   |
| <b>INVALID_NUMBER</b>          | En una sentencia SQL, la conversión de una cadena de caracteres hacia un número falla cuando esa cadena no representa un número válido   | -1722   |
| <b>LOGIN_DENIED</b>            | El programa intentó conectarse a Oracle con un nombre de usuario o password inválido   | -1017   |
| <b>NO_DATA_FOUND</b>           | Una sentencia SELECT INTO no devolvió valores o el programa referenció un elemento no inicializado en una tabla indexada   | +100    |
| <b>NOT_LOGGED_ON</b>           | El programa efectuó una llamada a Oracle sin estar conectado   | -1012   |
| <b>PROGRAM_ERROR</b>           | PL/SQL tiene un problema interno   | -6501   |
| <b>ROWTYPE_MISMATCH</b>        | Los elementos de una asignación (el valor a asignar y la variable que lo contendrá) tienen tipos incompatibles. También se presenta este error cuando un parámetro pasado a un subprograma no es del tipo esperado | -6504   |
| <b>SELF_IS_NULL</b>            | El parámetro SELF (el primero que es pasado a un método MEMBER) es nulo  | -30625  |
| <b>STORAGE_ERROR</b>           | La memoria se terminó o está corrupta  | -6500   |
| <b>SUBSCRIPT_BEYOND_COUNT</b>  | El programa está tratando de referenciar un elemento de un arreglo indexado que se encuentra en una posición más grande que el número real de elementos de la colección  | -6533   |
| <b>SUBSCRIPT_OUTSIDE_LIMIT</b> | El programa está referenciando un elemento de un arreglo utilizando un número fuera del rango permitido (por   | -6532   |

|                            |  |
|----------------------------|--|
|                            | ejemplo, el elemento “-1”)   |
| <b>SYS_INVALID_ROWID</b>   | La conversión de una cadena de -1410 caracteres hacia un tipo <i>rowid</i> falló porque la cadena no representa un número  |
| <b>TIMEOUT_ON_RESOURCE</b> | Se excedió el tiempo máximo de espera -51 por un recurso en Oracle   |
| <b>TOO_MANY_ROWS</b>       | Una sentencia SELECT INTO devuelve -1422 más de una fila   |
| <b>VALUE_ERROR</b>         | Ocurrió un error aritmético, de -6502 conversión o truncamiento. Por ejemplo, sucede cuando se intenta calzar un valor muy grande dentro de una variable más pequeña |
| <b>ZERO_DIVIDE</b>         | El programa intentó efectuar una división -1476 por cero   |

### Excepciones definidas por el usuario

PL/SQL permite al usuario definir sus propias excepciones, las que deberán ser declaradas y gatilladas explícitamente utilizando otros comandos del lenguaje.

#### *Declaración*

Las excepciones sólo pueden ser declaradas en el segmento “Declare” de un bloque, subprograma o paquete. Se declara una excepción escribiendo su nombre seguida de la palabra clave EXCEPTION. Las declaraciones son similares a las de variables, pero recuerde que una excepción es una condición de error, no un ítem de datos. Aun así, las mismas reglas de alcance aplican tanto sobre variables como sobre las excepciones.

#### Ejemplo:

DECLARE

error\_01 EXCEPTION;

#### *Reglas de Alcance*

Una excepción no puede ser declarada dos veces en un mismo bloque. Tal como las variables, una excepción declarada en un bloque es local a ese bloque y global a todos los sub-bloques que comprende.

*La sentencia "RAISE"*

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

Ejemplo:

```
DECLARE
    out_of_stock    EXCEPTION;      -- declaración de la excepción
    total           NUMBER(4);
BEGIN
    ...
    IF total < 1 THEN
        RAISE out_of_stock;         -- llamado a la excepción
    END IF;
EXCEPTION
    WHEN out_of_stock THEN
        -- manejar el error aquí
    WHEN OTHERS THEN
        ...
END;
```

Finalmente, cabe destacar la existencia de la excepción OTHERS, que simboliza cualquier condición de excepción que no ha sido declarada. Se utiliza comúnmente al final del bloque de excepciones para absorber cualquier tipo de error que no ha sido previsto por el programador. En ese caso, es común observar la sentencia ROLLBACK en el grupo de sentencias de la excepción o alguna de las funciones SQLCODE – SQLERRM, que se detallan en el próximo punto.



### **Uso de SQLCODE y SQLERRM**

Al manejar una excepción es posible apoyarse con las funciones predefinidas `SQLCODE` y `SQLERRM` para aclarar al usuario la situación de error acontecida.

*Sqrcode* siempre retornará el número del error de Oracle y un “0” (cero) en caso exitoso al ejecutarse una sentencia SQL.

Por otra parte, *Sqlermm* retornará el correspondiente mensaje de error para la situación ocurrida. También es posible entregarle a la función `SQLERRM` un número negativo que represente un error de Oracle y ésta devolverá el mensaje asociado.

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

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

#### **Ejemplo:**

```
DECLARE
    err_num    NUMBER;
    err_msg    VARCHAR2(100);
BEGIN
    ...
EXCEPTION
    WHEN OTHERS THEN
        err_num := SQLCODE;
        err_msg := SUBSTR(SQLERRM, 1, 100);
        INSERT INTO errores VALUES(err_num, err_msg);
END;
```

## CAPÍTULO 7: SUBPROGRAMAS

---

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

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

Ejemplo:

```
PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
    old_balance REAL;
    new_balance REAL;
    overdrown EXCEPTION;
BEGIN
    SELECT bal INTO old_balance FROM accts WHERE acct_no = acct_id;
    new_balance := old_balance - amount;
    IF new_balance < 0 THEN
        RAISE overdrown;
    ELSE
        UPDATE accts SET bal = new_balance WHERE acct_no = acct_id;
    END IF;
EXCEPTION
    WHEN overdrown THEN
        ...
END debit_account;
```

En el ejemplo, cuando el subprograma es invocado, recibe los parámetros *acct\_id* y *amount*. Con el primero de ellos selecciona el valor del campo “bal” y lo almacena en *old\_balance*. Luego almacena una diferencia en otra variable, *new\_balance*, la que de ser negativa gatillará una condición de excepción definida por el usuario (*overdrown*).

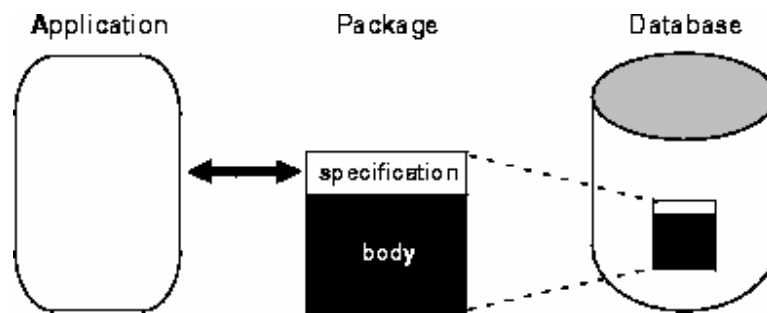


Figura 7-1: Estructura de un Paquete

### *Ventajas de los subprogramas*

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

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

Además, los subprogramas proveen las características de *reusabilidad* y *mantenibilidad*. Una vez construido, un subprograma puede ser utilizado en cualquier número de aplicaciones. Si la definición del tema que implementa es cambiada, entonces sólo se debe alterar el subprograma y no todos los lugares donde es referenciado.

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

## **Procedimientos**

Un procedimiento es un subprograma que ejecuta una acción específica. La sintaxis para construirlos es la siguiente:

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

Además, cada parámetro se escribe con la siguiente notación:

```
nombre_parámetro [IN | OUT [NOCOPY] | IN OUT [NOCOPY] tipo_de_dato  
[ { := | DEFAULT } expresión ]
```

En los pasajes de parámetros no se puede precisar el largo de alguno de ellos explícitamente, como en:

```
PROCEDURE xxx (param01 CHAR(5)) IS ...
```

Esta sentencia es inválida. Debería decir sólo ... param01 CHAR..., sin especificar el largo del carácter.

Sin embargo, si es absolutamente necesario restringir el largo de una cadena como la del ejemplo, se puede corregir la situación codificando la llamada al procedimiento **xxx** de la siguiente manera:

DECLARE

temp CHAR(5);

SUBTYPE Char5 IS temp%TYPE;

PROCEDURE xxx (param01 Char5) IS ...

Un procedimiento posee dos partes: una especificación y un cuerpo. La especificación es simple, comienza con la palabra `PROCEDURE` y termina (en la misma línea) con el nombre del procedimiento o la lista de parámetros (que es opcional).

El cuerpo del procedimiento comienza con la palabra reservada “IS” y termina con “END”, seguido opcionalmente por el nombre del procedimiento.

### **Funciones**

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

```
FUNCTION nombre [ (parámetro [, parámetro, ...] ) ] RETURN tipo_de_dato IS
BEGIN
    sentencias_ejecutables
[EXCEPTION
condiciones_de_excepción]
END [nombre] ;
```

Y la sintaxis de los parámetros es idéntica al caso de los procedimientos:

```
nombre_parámetro [IN | OUT [NOCOPY] | IN OUT [NOCOPY] tipo_de_dato
[ { := | DEFAULT } expresión ]
```

La función también posee una especificación y un cuerpo. El segmento de especificación comienza con la palabra `FUNCTION` y termina con la cláusula `RETURN`, la cual especifica el tipo de dato retornado por la función.

El cuerpo comienza con la palabra `IS` y termina con la palabra `END`, es decir, incluye las secciones de declaraciones, sentencias ejecutables y una parte opcional de manejo de excepciones.

Ejemplo:

```
FUNCTION revisa_salario (salario REAL, cargo CHAR(10)) RETURN BOOLEAN IS
    salario_minimo REAL;
    salario_maximo REAL;
BEGIN
    SELECT lowsal, highsal INTO salario_minimo, salario_maximo
        FROM salarios WHERE job = cargo ;
    RETURN (salario >= salario_minimo) AND (salario <= salario_maximo)
END revisa_salario ;
```

Esta misma función de ejemplo puede ser llamada desde una sentencia PL/SQL que reciba un valor booleano, como por ejemplo, en:

```
DECLARE
    renta_actual REAL;
    codcargo CHAR(10);
BEGIN
    ...
    IF revisa_salario (renta_actual, codcargo) THEN ...
```

La función *revisa\_salario* actúa como una variable de tipo booleano, cuyo valor depende de los parámetros recibidos.

### *La sentencia RETURN*

Esta sentencia termina inmediatamente la ejecución de un programa, retornando el control al bloque de programa que lo llamó. No se debe confundir con la cláusula *return* de las funciones, que especifica el tipo de dato devuelto por ella.

Un subprograma puede contener varias sentencias *Return*. Si se ejecuta cualquiera de ellas, el subprograma completo se termina.

La sintaxis para los procedimientos es simple, sólo se necesita la palabra RETURN. Sin embargo, en el caso de las funciones, esta sentencia debe contener un valor, que es aquel que se va a devolver al programa que la llamó. La expresión que sigue a la sentencia puede ser tan compleja como se desee pero siempre debe respetar el tipo de datos que está definido en la cabecera (especificación) de la función.

Una función debe contener como mínimo una sentencia RETURN, de otra manera, al no encontrarla, PL/SQL generará la excepción PROGRAM\_ERROR.

### **Uso de Parámetros**

Los subprogramas traspasan información utilizando parámetros. Las variables o expresiones referenciadas en la lista de parámetros de una llamada a un subprograma son llamados parámetros *actuales*. Las variables declaradas en la especificación de un subprograma y utilizadas en el cuerpo son llamados parámetros *formales*.

En este último caso (para los parámetros formales) se pueden explicitar tres modos diferentes para definir la conducta de los parámetros: IN, OUT e IN OUT.

Sin embargo, evite usar los tipos de parámetros OUT e IN OUT en las funciones, ya que por su naturaleza éstas devuelven un sólo dato y no a través de parámetros. Es una mala práctica hacer que una función devuelva muchos valores; para eso utilice los procedimientos.

*Parámetros de modo IN*

Estos parámetros son la entrada a las funciones y procedimientos y actúan dentro de ellas como constantes, es decir, sus valores no pueden ser alterados dentro de un subprograma.

Los parámetros actuales (que se convierten en formales dentro de un subprograma) pueden ser constantes, literales, variables inicializadas o expresiones.

*Parámetros de modo OUT*

Un parámetro de salida (OUT) permite comunicar valores al bloque de programa que invocó al subprograma que utiliza este parámetro. Esto quiere decir además, que es posible utilizar un parámetro de este tipo como si fuera una variable local al subprograma.

En el programa que invoca la función o procedimiento, el parámetro de modo OUT debe ser una variable, nunca una expresión o una constante.

*Parámetros de modo IN OUT*

Esta modalidad permite proporcionar valores iniciales a la rutina del subprograma y luego devolverlos actualizados. Al igual que el tipo anterior, un parámetro de este tipo debe corresponder siempre a una variable.

Si la salida de un subprograma es exitosa, entonces PL/SQL asignará los valores que corresponda a los parámetros actuales (los que reciben los valores de los parámetros formales en la rutina que llama al subprograma). Por el contrario, si la salida del subprograma se produce con un error no manejado en alguna excepción, PL/SQL no asignará ningún valor a los parámetros actuales.

*Resumen de las características de los parámetros*

| IN                            | OUT                              | IN OUT  |
|-------------------------------|----------------------------------|---|
| es el tipo por defecto        | debe ser especificado            | debe ser especificado   |
| pasa valores a un subprograma | retorna valores a quien lo llamó | pasa valores iniciales al subprograma y retorna un valor actualizado a quien lo llamó |
| un parámetro formal actúa     | parámetros formales actúan       | parámetros formales actúan  |



| como una constante  | como variables                                  | como una variable inicializada                    |
|---|---|---|
| a un parámetro formal no se le puede asignar un valor   | a un parámetro formal debe asignársele un valor | a un parámetro formal podría asignársele un valor |
| los parámetros actuales pueden ser constantes, variables inicializadas, literales o expresiones | los parámetros actuales deben ser variables     | los parámetros actuales deben ser variables       |

## **Recursividad**

La recursividad es una poderosa técnica que simplifica el diseño de algoritmos. Básicamente, la recursividad significa autoreferencia. Se aplica cuando un mismo algoritmo debe ser utilizado varias veces dentro de la solución a un problema determinado, cambiando cada vez las condiciones iniciales de cada ejecución (del algoritmo).

Un programa recursivo es aquel que se llama a si mismo. Piense en una llamada recursiva como una llamada a otro subprograma que hace lo mismo que el inicial. Cada llamada crea una nueva instancia de todos los ítems declarados en el subprograma, incluyendo parámetros, variables, cursores y excepciones.

Se recomienda ser muy cuidadoso con las llamadas recursivas. Entre otras cosas, existe un máximo de veces que un mismo cursor puede ser abierto y eso se define en una variable de Oracle llamada OPEN\_CURSORS. Al menos alguna vez la recursividad se debe revertir, es decir, las autoreferencias deben darse un número finito de veces.

### **Ejemplo:**

```

FUNCTION factorial (n POSITIVE) RETURN INTEGER IS    -- devuelve n!
BEGIN
  IF n = 1 THEN    -- condición de término
    RETURN 1
  ELSE
    RETURN n * factorial(n - 1);    -- esta es la llamada recursiva
  END IF;

```

END factorial;

### *Recursividad versus Iteración*

La recursividad no es una herramienta considerada fundamental en la programación PL/SQL. Cualquier problema que requiera su utilización también puede ser resuelto utilizando iteración. Una versión iterativa de un programa es usualmente más fácil de diseñar y de entender. Sin embargo, la versión recursiva es más simple, pequeña y más fácil de depurar. A modo de ejemplo, observe las siguientes dos versiones de cómo calcular el número n-ésimo de la serie de Fibonacci:

-- Versión recursiva

```
FUNCTION fib (n POSITIVE) RETURN INTEGER IS
BEGIN
    IF (n = 1) OR (n = 2) THEN
        RETURN 1;
    ELSE
        RETURN fib(n - 1) + fib(n - 2);
    END IF;
END fib;
```

-- Versión iterativa

```
FUNCTION fib (n POSITIVE) RETURN INTEGER IS
    pos1 INTEGER := 1;
    pos2 INTEGER := 0;
    cum INTEGER;
BEGIN
    IF (n = 1) OR (n = 2) THEN
        RETURN 1;
    ELSE
        cum := pos1 + pos2;
        FOR i IN 3..n LOOP
            pos2 := pos1;
```

```
        pos1 := cum;
        cum := pos1 + pos2;
    END LOOP;
    RETURN cum;
END IF;
END fib;
```

La versión recursiva de la función **fib** es mucho más elegante que la iterativa. Sin embargo, esta última es más eficiente; corre más rápido y utiliza menos memoria del computador. Si las llamadas son demasiadas se podrá advertir la diferencia en eficiencia. Considere esto para futuras implementaciones de una u otra alternativa.

### **Polimorfismo**

El polimorfismo es una característica del manejo de objetos. Significa que es posible definir más de un objeto con los mismos nombres, pero diferenciados únicamente por la cantidad o tipo de los parámetros que reciben o devuelven.

En el caso de los subprogramas, es posible declarar mas de uno con el mismo nombre, pero se deberá tener la precaución de diferenciarlos en cuanto al tipo de parámetros que utilizan.

#### Ejemplo:

```
PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
BEGIN
    FOR i IN 1..n LOOP
        tab(i) := SYSDATE;
    END LOOP;
END initialize;
```

```
PROCEDURE initialize (tab OUT RealTabTyp, n INTEGER) IS
BEGIN
    FOR i IN 1..n LOOP
        tab(i) := 0.0;
    END LOOP;
END initialize;
```

Estos procedimientos sólo difieren en el tipo de dato del primer parámetro. Para efectuar una llamada a cualquiera de ellos, se puede implementar lo siguiente:

```
DECLARE
    TYPE DateTabTyp IS TABLE OF DATE INDEX BY BINARY_INTEGER;
    TYPE RealTabTyp IS TABLE OF REAL INDEX BY BINARY_INTEGER;
    hiredate_tab      DateTabTyp;
    comm_tab          RealTabTyp;
    indx              BINARY_INTEGER;
    ...
BEGIN
    indx := 50;
    initialize(hiredate_tab, indx);    -- llama a la primera versión
    initialize(comm_tab, indx);       -- llama a la segunda versión
    ...
END;
```

## CAPÍTULO 8: PAQUETES

---

Un paquete es un esquema u objeto que agrupa tipos de PL/SQL relacionados, ítems y subprogramas. Los paquetes se constituyen de dos partes: la *especificación* y el *cuerpo*.

La *especificación* es la interfaz con las aplicaciones. En ella es posible declarar los tipos, variables, constantes, excepciones, cursores y subprogramas disponibles para su uso posterior.

El *cuerpo* define completamente a cursores y subprogramas e implementa lo que se declaró inicialmente en la especificación.

Es posible depurar y modificar cuantas veces se desee el cuerpo de un paquete sin necesidad de alterar por ello la especificación del mismo.

### Ejemplo de creación de paquetes:

```
CREATE OR REPLACE PACKAGE emp_actions AS    -- Especificación del paquete
    TYPE EmpRecTyp IS RECORD (emp_id INTEGER, salary REAL);
    CURSOR desc_salary RETURN EmpRecTyp;
    PROCEDURE hire_employee (
        ename VARCHAR2,
        job    VARCHAR2,
        mgr    NUMBER,
        sal    NUMBER,
        comm   NUMBER,
        deptno NUMBER);
    PROCEDURE fire_employee(emp_id NUMBER);
END emp_actions;
```

```
CREATE OR REPLACE PACKAGE BODY emp_actions AS    -- Cuerpo del paquete
  CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT empno, sal FROM emp ORDER BY sal DESC;
  PROCEDURE hire_employee (
    ename VARCHAR2,
    job    VARCHAR2,
    mgr    NUMBER,
    sal    NUMBER,
    comm   NUMBER,
    deptno NUMBER) IS
  BEGIN
    INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, job, mgr,
    SYSDATE, sal, comm, deptno);
  END hire_employee;
  PROCEDURE fire_employee (emp_id NUMBER) IS
  BEGIN
    DELETE FROM emp WHERE empno = emp_id;
  END fire_employee;
END emp_actions;
```

### **Ventajas de la utilización de Paquetes**

#### *Modularidad*

Los paquetes permiten encapsular tipos relacionados, ítems y subprogramas en lo que se denomina un módulo de PL/SQL. Cada paquete es diseñado de manera que sea fácil de entender y con interfaces simples, claras y bien definidas.

### *Diseño fácil de aplicaciones*

Cuando se diseña una aplicación, todo lo que se necesita para comenzar es la información declarativa que se escribe en una especificación de paquete. Usted puede escribirla e incluso compilarla sin la necesidad de haber creado previamente el cuerpo del mismo.

### *Ocultamiento de información*

Dentro de los paquetes se puede especificar que tipos, ítems y subprogramas serán *públicos* (visibles y accesibles) o *privados* (ocultos e inaccesibles). La idea fundamental es dar a conocer solamente aquellos componentes que pueden ser modificados en alguna instancia, manteniendo la integridad del paquete en todo momento.

### *Agregan Funcionalidad*

Variables y cursores que han sido empaquetados para su uso público persisten durante toda la duración de la sesión. Así, ellos pueden ser compartidos por todos los subprogramas que se ejecutan en el mismo ambiente. Con eso también se puede mantener la información a través de todas las transacciones sin tener que almacenarla en la base de datos.

### *Rendimiento*

Cuando se llama a un subprograma empaquetado por primera vez, éste es almacenado completamente en memoria. De esta manera, las sucesivas llamadas a los subprogramas del mismo paquete serán más rápidas. También es posible cambiar la definición de una función empaquetada sin tener que alterar los subprogramas que la llaman.

### **Especificación de Paquetes**

Una especificación de paquete contiene declaraciones públicas. El alcance de esas declaraciones es local al esquema de base de datos y global al paquete. Así, los ítems declarados son accesibles desde su aplicación y desde cualquier lugar dentro del mismo paquete.

La siguiente es una declaración de función que es posible encontrar en una especificación de paquete:

```
FUNCTION factorial (n INTEGER) RETURN INTEGER; -- devuelve el factorial de n
```

Esta es toda la información que usted requiere para llamar posteriormente a la función “factorial”.

El contenido de un paquete puede ser referenciado de esta manera incluso desde triggers de base de datos, procedimientos almacenados y otras herramientas de Oracle.

### **Cuerpo de un Paquete**

El cuerpo de un paquete implementa una especificación de paquete. Contiene la definición de cada cursor y subprograma declarado en la especificación. Recuerde que todos los subprogramas definidos en el cuerpo de un paquete estarán accesibles solamente si han sido especificados previamente.

En un cuerpo pueden encontrarse también declaraciones privadas (tipos y otros ítems) necesarias dentro del paquete y cuyo alcance se limita también al cuerpo del mismo y no está disponible para ser accedido desde otros paquetes.

Si en una declaración de paquete sólo se encuentran declaraciones de tipos, constantes, variables y excepciones el cuerpo del paquete se hace innecesario. Sin embargo, se reitera que cualquier definición de cursor o subprograma debe estar implementada en el cuerpo del mismo paquete.



*El paquete STANDARD*

Dentro de PL/SQL existe un paquete llamado *standard*, el cual provee el conjunto de tipos, excepciones y subprogramas que están disponibles automáticamente en el ambiente de trabajo de PL/SQL.

Por ejemplo, en él se define la función ABS, que retorna el valor absoluto de un argumento y que se define como sigue:

```
FUNCTION ABS (n NUMBER) RETURN NUMBER;
```

El contenido del paquete standard es visible desde todas las aplicaciones y puede ser accesado incluso desde triggers y procedimientos almacenados. Si usted decidiera redeclarar la función ABS puede hacerlo. Sin embargo, aún tendría la posibilidad de referenciar la función del mismo nombre que existe en el paquete standard, utilizando la notación “punto”, como en: ... STANDARD.ABS(x) ...

La mayoría de las funciones de conversión preconstruidas en PL/SQL son polimórficas. Por ejemplo, el paquete standard contiene las siguientes declaraciones:

```
FUNCTION TO_CHAR(right DATE) RETURN VARCHAR2 ;  
FUNCTION TO_CHAR(left NUMBER) RETURN VARCHAR2 ;  
FUNCTION TO_CHAR(left DATE, right VARCHAR2) RETURN VARCHAR2 ;  
FUNCTION TO_CHAR(left NUMBER, right VARCHAR2) RETURN VARCHAR2 ;
```

PL/SQL resuelve la llamada a la instancia de la función que corresponda observando los tipos de datos de los parámetros actuales y formales.

# ANEXOS

## TEMARIO CURSO PL/SQL

---

### 1. Conceptos de Modelamiento de Datos

*Objetivo:* El alumno conocerá los conceptos introductorios al tema de modelamiento de datos. Se revisará el modelo entidad - relación y sus principales componentes.

*Tópicos:*

- a) Las bases de datos y el modelo entidad - relación
- b) Entidades
- c) Relaciones
- d) Cardinalidad

### 2. Introducción

*Objetivo:* El alumno conocerá algunas características de la base de datos “Oracle” y del producto PL/SQL y cómo se inserta en la arquitectura de desarrollo de la base de datos. También se familiarizará con la herramienta “SQL Navigator”, con la que se desarrollarán los contenidos prácticos del curso.

*Tópicos:*

- a) Estructuras de Bloques
- b) Variables y Constantes
- c) Cursores
- d) Manejo de Errores
- e) Subprogramas
- f) Paquetes
- g) Ventajas de la utilización de PL/SQL

### **3. Fundamentos del Lenguaje**

*Objetivo:* El alumno conocerá los componentes básicos del lenguaje. Tal como en otros lenguajes de programación, PL/SQL tiene un set de caracteres, palabras reservadas, signos de puntuación y otras reglas fijas que el alumno deberá conocer antes de empezar a trabajar con el resto de funcionalidades.

*Tópicos:*

- a) Delimitadores e Identificadores
- b) Tipos de datos y conversiones
- c) Alcance y Visibilidad de las variables

### **4. Estructuras del Lenguaje**

*Objetivo:* Al finalizar este capítulo, el alumno conocerá las diferentes estructuras de control de flujo, iteración y otras que componen este lenguaje de programación. También se incluyen nociones de la interacción con la base de datos Oracle.

*Tópicos:*

- a) Estructuras de Control
- b) Sentencias SQL
- c) Procesamiento de Transacciones

### **5. Uso de Cursores**

*Objetivo:* En este capítulo, el alumno aprenderá a manejar cursores con la sintaxis de PL/SQL. Incluye la declaración, apertura, parsing, recuperación de datos y cierre de cursores.

*Tópicos:*

- a) Declaración y Utilización de cursores
- b) Atributos de los cursores explícitos e implícitos

## 6. Manejo de Errores

*Objetivo:* Al finalizar este capítulo, el alumno aprenderá a construir y manejar situaciones de excepción. Se enseñará a construir el bloque de “excepciones” de un programa construido con PL/SQL:

*Tópicos:*

- a) Cómo construir excepciones
- b) Tipos de errores
- c) Variables internas de error (sqlcode, sqlerrm)

## 7. Subprogramas

*Objetivo:* El alumno aprenderá como construir procedimientos almacenados y funciones en PL/SQL. También conocerá y aplicará el uso de los diferentes tipos de parámetros y llamados a rutinas externas. También conocerá las técnicas de recursividad y polimorfismo.

*Tópicos:*

- a) Procedimientos y Funciones
- b) Tipos de Parámetros
- c) Recursividad
- d) Polimorfismo

## 8. Paquetes

*Objetivo:* Al finalizar el capítulo, el alumno aprenderá la utilidad de empaquetar funciones y procedimientos. Creará “Package Specification” (especificación o declaración de paquetes) y también “Package Body” (cuerpo de los paquetes).

*Tópicos:*

- a) Principales características y ventajas
- b) Especificación de Paquetes
- c) Cuerpo de los Paquetes

## LABORATORIOS CURSO PL/SQL

---

### **Laboratorio #1**

#### *Objetivo:*

Al finalizar el primer laboratorio, el alumno se familiarizará con PL/SQL y el ambiente de desarrollo SQL Navigator. Manejo de variables y tipos de datos. También aprenderá los primeros pasos en la creación de programas.

#### *Procedimiento:*

1. Abrir el entorno de trabajo de SQL Navigator y conocer la interfaz de trabajo de este software, específicamente en el ambiente en que se desarrollan los programas con PL/SQL.

En una aplicación que se creará durante el mismo laboratorio, se mostrarán los componentes de un programa explicados en clases.

Los temas que se mostrarán en forma práctica por el mismo profesor serán los siguientes:

- Creación de un procedimiento
- Creación de una función
- Muestra de los bloques que se pueden construir en cada programa
- Cómo compilar, grabar, ejecutar un programa
- Cómo crear paquetes

A continuación se solicitará al alumno que cree un pequeño programa que realice una tarea simple sobre la base de datos de ejemplo.

2. Como requerimiento opcional, el alumno podrá investigar la herramienta SQL Navigator con el fin de soltar la mano en el manejo de ésta.

### **Fin del Laboratorio #1**

## **Laboratorio #2**

### *Objetivo:*

Al final del segundo laboratorio, el alumno aprenderá a construir aplicaciones utilizando todas las estructuras de control aprendidas hasta aquí. También ejercitará la construcción de sentencias SQL utilizándolas contra la base de datos de ejemplo.

### *Procedimiento:*

1. Abrir el entorno de trabajo de SQL Navigator y crear un procedimiento que realice diferentes tareas donde sea obligatorio el uso de las sentencias de control de flujo aprendidas (condicionales e iterativas).
2. Se deberán ejecutar consultas SQL contra la base de datos que efectúen las tareas indicadas en el mismo momento del laboratorio por su profesor.

**Fin del Laboratorio #2**

### **Laboratorio #3**

#### *Objetivo:*

En este laboratorio se efectuará un reforzamiento de todos los temas vistos en clases y se completará con la práctica del manejo de cursores.

#### *Procedimiento:*

1. Repasar las estructuras de control, bloques y demás componentes de un programa vistos en clases.
2. Se practicará con la declaración, apertura, utilización (recuperación de filas) y cierre de los cursores.
3. Se usarán los atributos de los cursores en programas creados en el mismo laboratorio.
4. De manera opcional se mostrará un programa real construido en un proyecto de DMR.

**Fin del Laboratorio #3**



## **Laboratorio #4**

### *Objetivo:*

Al final del laboratorio, el alumno habrá aprendido a manejar excepciones dentro de un programa. Conocerá la forma en que estas se gatillan y como manejar el error para evitar caídas fatales de los programas en tiempo de ejecución.

### *Procedimiento:*

1. Se provocarán de manera intencional varios errores para estudiar el comportamiento de un programa en esos casos.
2. Al final del laboratorio deberá existir un programa (puede ser alguno creado en un laboratorio anterior) totalmente estable, es decir, que acepte cualquier tipo de error y lo maneje de manera adecuada.

**Fin del Laboratorio #4**

## **Laboratorio #5**

### *Objetivo:*

El alumno aprenderá cómo crear procedimientos y funciones y cómo comunicarlos mediante el uso de parámetros de entrada, salida y de entrada/salida.

### *Procedimiento:*

1. El alumno creará un procedimiento nuevo que reciba cierta cantidad de parámetros.
2. Luego, creará otro procedimiento diferente que efectuará un llamado al primero y lo obligará a ejecutarse con ciertos valores en sus parámetros.
3. A continuación se creará una función que será llamada desde ambos procedimientos. Se deberán probar las características de recursividad de las funciones.
4. Finalmente y en forma opcional se deberán probar las características de polimorfismo de las funciones, implementando el ejemplo que aparece en el manual del curso.

**Fin del Laboratorio #5**

## **Laboratorio #6**

### *Objetivo:*

Al finalizar este laboratorio, el alumno estará en condiciones de crear especificaciones y cuerpos de paquetes.

### *Procedimiento:*

1. El alumno creará una especificación de paquete que comprenda todas las funciones y procedimientos creados en los laboratorios del curso. Sólo se creará un “package specification”.
2. En el cuerpo del paquete deberá copiar la definición de todos sus subprogramas que declaró en la sección de especificación del paquete que está creando.

**Fin del Laboratorio #6**

---

**Modelo de datos para el curso de PL/SQL**


---

**EMPLEADO**

| Atributo           | Valores     |               |               |               |
|--------------------|-------------|---------------|---------------|---------------|
| <b>Código</b>      | 100         | 101           | 102           | 103           |
| <b>Nombres</b>     | Sergio      | Marco Antonio | Luis Fernando | María Cecilia |
| <b>Ap. Paterno</b> | Contreras   | Fernández     | Cárcamo       | Poblete       |
| <b>Ap. Materno</b> | Ruiz        | Castro        | Vergara       | Romero        |
| <b>Sexo</b>        | M           | M             | M             | F             |
| <b>Fecha Ncto.</b> | 14/Jul/1965 | 18/Sep/1970   | 15/Oct/1973   | 03/Ene/1971   |
| <b>Renta</b>       | \$220.000   | \$185.000     | \$350.000     | \$510.000     |
| <b>Cargo</b>       | A1          | A2            | B1            | B1            |
| <b>Depto.</b>      | S01         | S01           | S02           | S03           |

**DEPARTAMENTO**

| Atributo      | Valores  |              |          |
|---------------|----------|--------------|----------|
| <b>Código</b> | S01      | S02          | S03      |
| <b>Nombre</b> | Finanzas | Contabilidad | Personal |

**CARGO**

| Atributo            | Valores   |           |              |
|---------------------|-----------|-----------|--------------|
| <b>Código</b>       | A1        | A2        | B1           |
| <b>Nombre</b>       | Contador  | Asistente | Jefe de Area |
| <b>Renta Mínima</b> | \$200.000 | \$120.000 | \$350.000    |
| <b>Renta Máxima</b> | \$410.000 | \$230.000 | \$500.000    |

**CARGAS**

| Atributo             | Valores        |                |              |                |
|----------------------|----------------|----------------|--------------|----------------|
| <b>Código</b>        | C1             | C2             | C3           | C4             |
| <b>Nombre</b>        | Claudia        | Marta          | Celia        | Francisco      |
| <b>Apellidos</b>     | Fernández Lara | Fernández Lara | Cárcamo Vera | Romero Poblete |
| <b>Fecha Ncto.</b>   | 01/Feb/1992    | 09/Jul/1998    | 17/Feb/1988  | 23/Nov/1990    |
| <b>Cod. Empleado</b> | 101            | 101            | 102          | 103            |

### Scripts de creación de tablas

---

```
create table EMPLEADO(  
  id_empl      number(3)    not null,  
  nomb_empl    varchar2(25) not null,  
  apepat       varchar2(15) not null,  
  apemat       varchar2(15) null,  
  sexo         char(1)      not null,  
  fechanac     date         null,  
  renta        number(10)   not null,  
  id_cargo     char(2)      not null,  
  id_depto     char(3)      not null,  
  CONSTRAINT pk_empleado PRIMARY KEY (id_empl)  
);
```

```
create table DEPARTAMENTO(  
  id_depto     char(3)      not null,  
  nomb_depto   varchar2(15) not null,  
  CONSTRAINT pk_departamento PRIMARY KEY (id_depto)  
);
```

```
create table CARGO(  
  id_cargo     char(2)      not null,  
  nomb_cargo   varchar2(15) not null,  
  rentamin     number(10)   not null,  
  rentamax     number(10)   not null,  
  CONSTRAINT pk_cargo PRIMARY KEY (id_cargo)  
);
```

```
create table CARGAS(  
  cod_carga    varchar2(3)  not null,  
  nomb_carga   varchar2(20) not null,  
  apell_carga  varchar2(20) not null,  
  fnac_carga   date         not null,  
  id_empl     number(3)    not null,  
  CONSTRAINT pk_cargas PRIMARY KEY (cod_carga)  
);
```

### Scripts de carga de datos en tablas

---

```
INSERT INTO empleado(id_empl, nomb_empl, apepat, apemat, sexo, fechanac, renta,  
id_cargo, id_depto)  
VALUES (100, 'Sergio', 'Contreras', 'Ruiz', 'M', TO_DATE('14/07/1965',  
'dd/mm/yyyy'), 220000, 'A1', 'S01');
```

```
INSERT INTO empleado(id_empl, nomb_empl, apepat, apemat, sexo, fechanac, renta,  
id_cargo, id_depto)  
VALUES (101, 'Marco Antonio', 'Fernández', 'Castro', 'M', TO_DATE('18/09/1970',  
'dd/mm/yyyy'), 185000, 'A2', 'S01');
```

```
INSERT INTO empleado(id_empl, nomb_empl, apepat, apemat, sexo, fechanac, renta,  
id_cargo, id_depto)  
VALUES (102, 'Luis Fernando', 'Cárcamo', 'Vergara', 'M', TO_DATE('15/10/1973',  
'dd/mm/yyyy'), 350000, 'B1', 'S02');
```

```
INSERT INTO empleado(id_empl, nomb_empl, apepat, apemat, sexo, fechanac, renta,  
id_cargo, id_depto)  
VALUES (103, 'María Cecilia', 'Poblete', 'Romero', 'F', TO_DATE('03/01/1971',  
'dd/mm/yyyy'), 510000, 'B1', 'S03');
```

```
INSERT INTO departamento(id_depto, nomb_depto) VALUES ('S01', 'Finanzas');  
INSERT INTO departamento(id_depto, nomb_depto) VALUES ('S02', 'Contabilidad');  
INSERT INTO departamento(id_depto, nomb_depto) VALUES ('S03', 'Personal');
```



```
INSERT INTO cargo(id_cargo, nomb_cargo, rentamin, rentamax) VALUES ('A1',  
'Contador', 200000, 410000);
```

```
INSERT INTO cargo(id_cargo, nomb_cargo, rentamin, rentamax) VALUES ('A2',  
'Asistente', 120000, 230000);
```

```
INSERT INTO cargo(id_cargo, nomb_cargo, rentamin, rentamax) VALUES ('B1', 'Jefe de  
Area', 350000, 500000);
```

```
INSERT INTO cargas(cod_carga, nomb_carga, apell_carga, fnac_carga, id_empl)  
VALUES ('C1', 'Claudia', 'Fernández Lara', TO_DATE('01/02/1992', 'dd/mm/yyyy'),  
101);
```

```
INSERT INTO cargas(cod_carga, nomb_carga, apell_carga, fnac_carga, id_empl)  
VALUES ('C2', 'Marta', 'Fernández Lara', TO_DATE('09/07/1998', 'dd/mm/yyyy'),  
101);
```

```
INSERT INTO cargas(cod_carga, nomb_carga, apell_carga, fnac_carga, id_empl)  
VALUES ('C3', 'Celia', 'Cárcamo Vera', TO_DATE('17/02/1988', 'dd/mm/yyyy'), 102);
```

```
INSERT INTO cargas(cod_carga, nomb_carga, apell_carga, fnac_carga, id_empl)  
VALUES ('C4', 'Francisco', 'Romero Poblete', TO_DATE('23/11/1990', 'dd/mm/yyyy'),  
103);
```