



UNIVERSIDAD
DE GRANADA



Diseño y Desarrollo de Sistemas de Información

Grado en Ingeniería Informática

Implementación

©I. J. Blanco, F. J. Cabrerizo, C. Cruz, J.A. García, M. J. Martín, M.J. Rodríguez, D. Sánchez

Este documento está protegido por la Ley de Propiedad Intelectual (Real Decreto Ley 1/1996 de 12 de abril).

Queda expresamente prohibido su uso o distribución sin autorización del autor.

Departamento de Ciencias de la
Computación e Inteligencia Artificial
<http://decsai.ugr.es>

- ❑ Saber distinguir entre funcionalidad a implementar en el SGBD y el resto de funcionalidad del SI.
- ❑ Conocer los fundamentos del lenguaje PL/SQL de Oracle.
- ❑ Seleccionar SGBD y lenguaje de programación para la implementación del SI en una arquitectura cliente/servidor de dos capas.
- ❑ Crear la BD e implementar código en la misma por medio de disparadores.
- ❑ Realizar la implementación completa del SI.

- Los SGBD relacionales de cierta entidad permiten la **inclusión de código en la BD asociado a datos**. Para ello proporcionan soporte para algún lenguaje de programación.
- A la hora de implementar un SI, hay que determinar qué parte de la funcionalidad se implementa en el SGBD. Para ello, hay que tener en cuenta que los datos en el SGBD suelen ser compartidos por múltiples aplicaciones. Por tanto:

Hay que programar en el SGBD aquellas funcionalidades que estén fuertemente asociadas a los datos, y que sean independientes de las aplicaciones concretas que los puedan usar.

- La forma de incluir este tipo de funcionalidades en los datos es mediante el uso de **disparadores** (“triggers”).

- Un **disparador** es un procedimiento que **se ejecuta en respuesta a determinados eventos que puedan ocurrir sobre una tabla** en la BD, tales como una inserción, borrado o actualización, y contienen acciones a realizar **independientemente de qué aplicación** haya activado el evento.
- Los disparadores suelen utilizarse para tareas como, entre otras:
 - Generar automáticamente valores de columnas calculados a partir de otros
 - Almacenar tablas “log” identificando quién y cuándo ha realizado una operación sobre determinadas tablas
 - Replicación síncrona de tablas
 - Realizar comprobaciones de seguridad y requerir autorizaciones
 - Comprobar restricciones de integridad que no vienen controladas por el diseño lógico (ejemplo: dependencias funcionales o de otro tipo).
 - Control de transacciones en casos como los dos anteriores
 - Almacenar en tablas históricas tuplas conforme se borran de tablas operativas

- Basados en el estándar **ISO SQL/PSM** (“SQL/Persistent Stored Modules”):
 - **PL/SQL** (“Procedural Language for SQL”): extensión procedural de SQL propuesta por **Oracle**, también soportado por **IBM DB2**.
 - **MySQL/MariaDB** usan su propia implementación de SQL/PSM.
 - **T-SQL** (“Transactional SQL”) de **Sybase** y **Microsoft SQL Server**
 - **PL/pgSQL** (“Procedural Language/PostgreSQL”) de PostgreSQL.
 - **SQL PL** de **IBM DB2**
- SQL/PSM (y por tanto PL/SQL y otros) son **también** los lenguajes **utilizados para la definición de métodos en el modelo objeto-relacional** siguiendo los estándares SQL:1999 y siguientes.
- Otra especificación para la definición de métodos se basa en Java a través del estándar SQL/JRT (“SQL Routines and Types for the Java Programming Language”), que permite invocar “Java Stored Procedures”, así como usar clases de Java como tipos en SQL. Usado por Oracle a través de Oracle JVM, entre otros.

- PostgreSQL permite utilizar otros lenguajes en el SGBD a través de “extensions”.
 - En la version actual, PostgreSQL incluye extensions basadas en Tcl, Perl y Python (PL/Tcl, PL/Perl y PL/Python respectivamente).
 - También permite programación en C, pero no es segura, ya que se salta el control de integridad del Sistema, y solo puede ejecutarse como superusuario.
 - Se han planteado proyectos externos que incluyen soporte en PostgreSQL para Java (PL/Java), JavaScript (PL/v8), R (PL/R), Lua (PL/Lua) y otros.
- Hay más, como siempre.
- Nosotros vamos a estudiar aquí PL/SQL de Oracle, que se utiliza también como lenguaje de programación de aplicaciones Web en el Web Framework Oracle APEX.

1. Características de PL/SQL
2. Arquitectura cliente-servidor en PL/SQL
3. El concepto central en PL/SQL: el bloque
4. Estructuras de datos en PL/SQL: tipos, variables, estructuras y colecciones
5. Estructuras de control: selección e iteración
6. Subprogramas: procedimientos, funciones y paquetes
7. Disparadores
8. El resultado de las consultas paso a paso: cursores
9. Excepciones

PL/SQL combina la potencia y flexibilidad de un lenguaje de cuarta generación (SQL) con las estructuras procedimentales de una lenguaje de tercera generación

PL/SQL añade a SQL:

- Variables y tipos

- Estructuras de control (bucles, decisión)

- Procedimientos y funciones

- Objetos y métodos

Estructura de bloques:

DECLARE

<Lista de declaraciones>

BEGIN

<Lista de sentencias>

EXCEPTION

WHEN OTHERS THEN

<Lista de sentencias>

END;



Manejo de errores:

DECLARE

<Lista de declaraciones>

BEGIN

<sentencia 1>

<sentencia 2>

Error

...

<sentencia n>

EXCEPTION

WHEN OTHERS THEN

<Lista de sentencias>

END;



```

DECLARE
vprecio inventario.precio%TYPE;
BEGIN
  [Otras sentencias]
  BEGIN
    SELECT precio FROM inventario
    WHERE cantidad >= 100;
    INTO vprecio;
    EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR (-20001, 'Sin Datos');
    WHEN TOO_MANY_ROWS THEN
      RAISE_APPLICATION_ERROR (-20002, 'Demasiados Datos');
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE("Error, abortando ejecución.");
      RAISE;
    END;
  [Otras sentencias]
END;
  
```

Variables y tipos:

DECLARE

clienteID VARCHAR2(20);

fechactual DATE;

saldo NUMBER;

TYPE fichaCliente IS RECORD (

 clienteID VARCHAR2(20),

 nombre VARCHAR2(30),

 apellidos VARCHAR2(60));

BEGIN

...

END;



Estructuras de bucle:

DECLARE

contador BINARY_INTEGER := 1;

...

BEGIN

...

WHILE (contador >= 10) **LOOP**

INSERT INTO temporal (contador);

contador := contador + 1;

END LOOP;

...



Cursores:

DECLARE

CURSOR clientes **IS** SELECT nombre, apellidos FROM CLIENTES;

nombre VARCHAR2(30);

apellidos VARCHAR2(60);

...

BEGIN

OPEN clientes;

FETCH clientes INTO nombre, apellidos;

WHILE (clientes%FOUND) **LOOP**

...

FETCH clientes INTO nombre, apellidos;

END LOOP;

CLOSE clientes; ...



Procedimientos y funciones:

CREATE OR REPLACE PROCEDURE obtenerClientes

(patron IN VARCHAR2) **AS**

DECLARE

CURSOR clientes IS SELECT nombre, apellidos FROM CLIENTES WHERE
nombre LIKE patron;

...

BEGIN

OPEN clientes;

FETCH clientes INTO nombre, apellidos;

WHILE (clientes%FOUND) LOOP

...

FETCH clientes INTO nombre, apellidos;

END LOOP;

END;



Paquetes:

CREATE OR REPLACE PACKAGE BODY clientesPKG **AS**

 PROCEDURE obtenerClientes (patron IN VARCHAR2) AS

 CURSOR clientes IS SELECT nombre, apellidos FROM CLIENTES WHERE
 nombre LIKE patron;

 ...

BEGIN

 ...

END;

FUNCTION ...

...

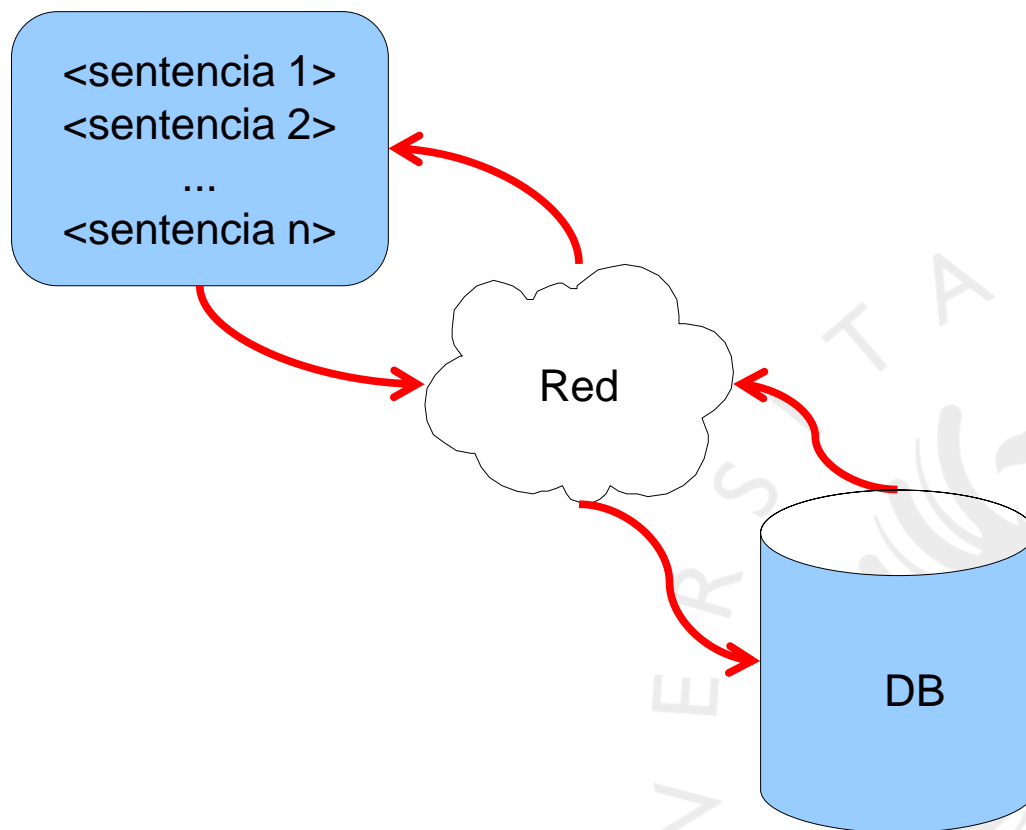
END clientesPKG;



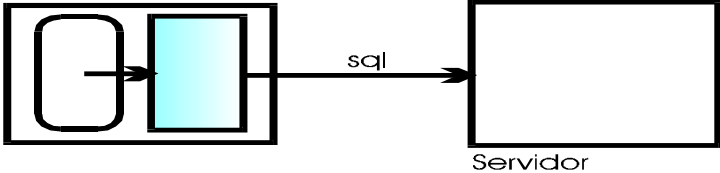
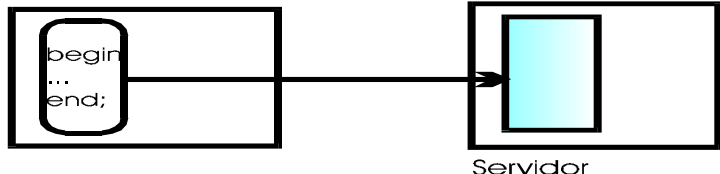
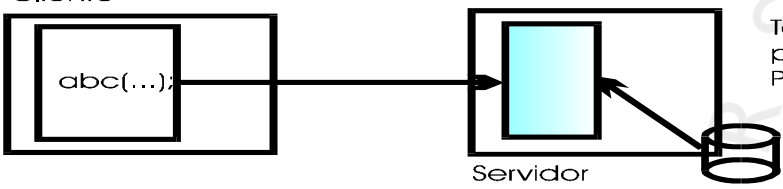
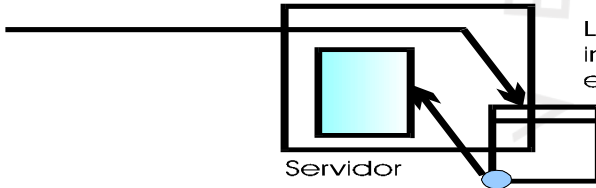
SQL dinámico:

Permite la compilación y ejecución de sentencias SQL (tanto DDL como DML) durante la ejecución de un módulo PL/SQL (procedimiento, función o bloque PL/SQL anónimo).

1. Características de PL/SQL
2. Arquitectura cliente-servidor en PL/SQL
3. El concepto central en PL/SQL: el bloque
4. Estructuras de datos en PL/SQL: tipos, variables, estructuras y colecciones
5. Estructuras de control: selección e iteración
6. Subprogramas: procedimientos, funciones y paquetes
7. Disparadores
8. El resultado de las consultas paso a paso: cursores
9. Excepciones



Dentro del mismo ordenador

Tipo programa	Localización de la máquina PL/SQL	Comentarios
PL/SQL en rutinas 4GL	<p>Cliente</p>  <p>Servidor</p>	<p>Máquina cliente de PL/SQL presente en:</p> <ul style="list-style-type: none"> - Oracle Forms - Oracle Menu - Oracle Reports - Oracle Graphics
Bloques PL/SQL anónimos	<p>Cliente</p>  <p>Servidor</p>	<ul style="list-style-type: none"> - SQL*Plus - SQL*DBA - Precompiladores - SQL*Module
Programas PL/SQL almacenados	<p>Cliente</p>  <p>Servidor</p>	<p>Todas las aplicaciones de usuario pueden llamar programas PL/SQL almacenados</p>
Disparadores definidos a nivel de BD	 <p>Servidor</p>	<p>Los disparadores de BD se ejecutan implícitamente sobre una tabla si se efectúa una operación DML sobre ella</p>

1. Características de PL/SQL
2. Arquitectura cliente-servidor en PL/SQL
3. El concepto central en PL/SQL: el bloque
4. Estructuras de datos en PL/SQL: tipos, variables, estructuras y colecciones
5. Estructuras de control: selección e iteración
6. Subprogramas: procedimientos, funciones y paquetes
7. Disparadores
8. El resultado de las consultas paso a paso: cursores
9. Excepciones

Tipos de bloques:

Bloques anónimos

Bloques con nombre

Subprogramas (funciones,
procedimientos y paquetes)

Disparadores

Bloques anónimos:

DECLARE

<Lista de declaraciones>

BEGIN

<Lista de sentencias>

EXCEPTION

WHEN OTHERS THEN

<Lista de sentencias>

END;



Bloques con nombre:

<<nombreBloque>>

DECLARE

<Lista de declaraciones>

BEGIN

<Lista de sentencias>

EXCEPTION

WHEN OTHERS THEN

<Lista de sentencias>

END nombreBloque;



Bloques en módulo:

CREATE OR REPLACE PROCEDURE

nombreProcedimiento (<lista parámetros>) AS

<Lista de declaraciones>

BEGIN

<Lista de sentencias>

EXCEPTION

WHEN OTHERS THEN

<Lista de sentencias>

END;



Disparadores:

```
CREATE OR REPLACE TRIGGER nombreDisparador
  BEFORE INSERT ON <nombreTabla> FOR EACH
  ROW AS
```

```
  <Lista de declaraciones>
```

```
BEGIN
```

```
  <Lista de sentencias>
```

```
EXCEPTION
```

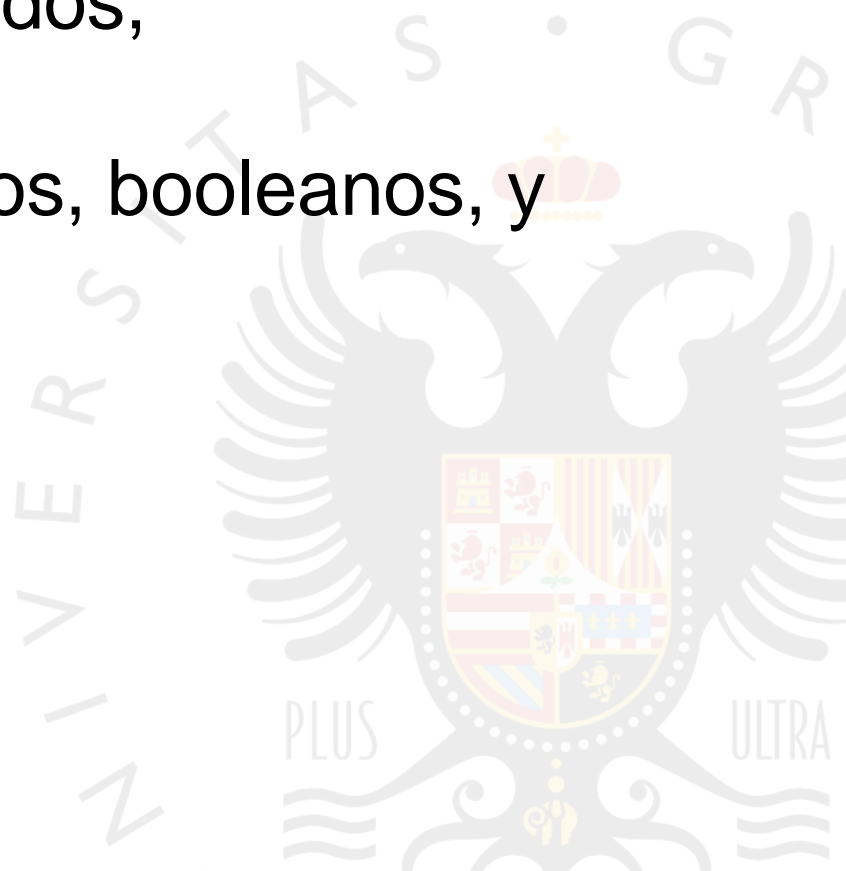
```
  WHEN OTHERS THEN
```

```
    <Lista de sentencias>
```

```
END;
```



Identificadores,
palabras reservadas,
identificadores entrecomillados,
delimitadores,
literales: cadenas, numéricos, booleanos, y
comentarios: --, /* */



Identificador entrecomillado

```
DECLARE
```

```
    V_Exception VARCHAR2(10)
```

```
BEGIN
```

```
    SELECT "EXCEPTION"
```

```
    INTO v_Exception FROM exception_table;
```

```
END;
```



Asignación por operador:

`a := <expresion>`

Asignación por SELECT:

`SELECT A INTO a FROM TABLA;`



Aritméticos: +, -, *, /

Caracteres: ||

Lógicos: AND, OR, NOT, <, <=, >, >=, !=, =,
LIKE



1. Características de PL/SQL
2. Arquitectura cliente-servidor en PL/SQL
3. El concepto central en PL/SQL: el bloque
4. Estructuras de datos en PL/SQL: tipos, variables, estructuras y colecciones
5. Estructuras de control: selección e iteración
6. Subprogramas: procedimientos, funciones y paquetes
7. Disparadores
8. El resultado de las consultas paso a paso: cursores
9. Excepciones

Escalares:

Numéricos: NUMBER [(P[,S])] y sus subtipos DECIMAL, DOUBLE PRECISION, INTEGER, NUMERIC, REAL y SMALLINT

BINARY_INTEGER (operaciones rápidas)

Caracter: VARCHAR2, CHAR, LONG

Raw: RAW, LONG RAW

Fecha: DATE

ROWID

Lógicos: BOOLEAN



Subtipo: tipo definido por el usuario basado en un tipo base

```
SUBTYPE <nombre_tipo> IS <tipo_base>;
```

Puede usarse para renombrar tipos para acomodarlos a las necesidades del usuario

```
SUBTYPE numerico IS NUMBER;
```

Pueden usarse como tipos sin restricciones y ser restringidos posteriormente durante una declaración de variable

```
Numero NUMERICO(3);
```

Para declarar una variable se le proporciona un nombre siguiendo las normas de los identificadores, seguida de un tipo:

`<nombre_variable> <tipo>;`

También puede inicializarse durante la creación:

`<nombre_variable> <tipo> := <valor>;`

También se puede declarar una variable de un tipo igual al de una columna de la base de datos:

```
<nombre_variable> <columna>%TYPE;
```

La referencia a la columna debe ser completa dependiendo del ámbito de la columna.

Ejemplos:

contador INTEGER;

contador INTEGER := 1;

id TAB1.COL1%TYPE;



Supongamos que disponemos de la siguiente tabla que recoge la escala de gravamen para declaraciones individuales para el cálculo de la cuota íntegra en el IRPF en la tabla adjunta.

El cálculo de la cuota íntegra de una base liquidable K se realiza sumando a la cuota correspondiente a la “base inmediatamente inferior (BK)” el resultado de aplicar el tipo correspondiente (TB) a la diferencia $K-BK$. Ejemplo: si $K=18030,36$, la base inmediatamente inferior es $17441,37$, su cuota (BK) es $3466,34$ y la diferencia $K-BK$ es $588,99$ a la que hay que aplicar el tipo (TB) 30% . Por tanto la cuota íntegra sería:

$$3466,34 + (18030,36 - 17441,37) \cdot 30 / 100 = 3643,037$$

Rellenar los datos de la tabla con el [fichero SQL](#).

Base hasta	Cuota	Resto hasta	Tipo aplicable (%)
0	0	2584.35	0.0
2584.35	0	3858.5	20.0
6442.85	771.7	3666.17	22.0
10109.02	1578.26	3666.17	24.5
13775.2	2476.47	3666.17	27.0
17441.37	3466.34	3666.17	30.0
21107.55	4566.19	3666.17	32.0
24773.72	5739.37	3666.17	34.0
28439.89	6985.86	3666.17	36.0
32106.07	8305.69	3666.17	38.0
35772.24	9698.83	3666.17	40.0
39438.41	11165.3	3666.17	42.5
43104.59	12723.43	3666.17	45.0
46770.76	14373.2	3666.17	47.0
50436.94	16096.31	3666.17	49.0
54103.11	17892.73	3666.17	51.0
57769.28	19762.48	3666.17	53.5
61435.46	21723.88	0	56.0

Construir un bloque anónimo (usando únicamente lo visto hasta ahora) que calcule la cuota integral correspondiente a la cantidad de 34558.2, utilizando la mencionada tabla, y visualice el resultado por la salida estándar (usando la función DBMS_OUTPUT.PUT_LINE)

Resultado:

[ejer1_v1.sql](#)

Se tienen datos que no son independientes:

<nombre 1> <tipo>;

...

<nombre n> <tipo>;

Se agrupan en una sola estructura con varios campos:

TYPE <tipo_registro> IS RECORD (

<nombre_campo_1> <tipo>,

...

<nombre_campo_n> <tipo>);



Una vez creado el tipo para la estructura es necesario declarar una variable para cada estructura que tenga que almacenarse:

`<nombre_variable> <tipo_registros>;`

Al igual que se pueden declarar variable con un tipo igual al de una columna de base de datos, se pueden crear registros del mismo tipo que una fila de tabla de base de datos sin necesidad de declarar el tipo:

`<nombre_variable> <tabla>%ROWTYPE;`

Ejemplos:

```
TYPE Cliente IS RECORD (  
    IdCliente NUMBER,  
    NombreCompañia VARCHAR2(50),  
    Ciudad          VARCHAR2(20)  
);
```



El acceso a un campo de un registro se realiza usando el operador “.”:

`<nombre_registro>.<nombre_campo>`

Un registro puede asignarse completamente:

```
registro1 := registro2;
SELECT ... INTO registro1 FROM ...;
```

DECLARE

TYPE Cliente IS RECORD (

 IdCliente NUMBER,

 NombreCompañía VARCHAR2(50),

 Ciudad VARCHAR2(20)

);

miCliente1 Cliente;

miCliente2 Cliente;

BEGIN

 miCliente1.IdCliente := 1;

 miCliente1.NombreCompañía:= 'compañía SA';

 miCliente1.Ciudad := 'Granada';

 SELECT idCliente, nombre, ciudad INTO miCliente2 FROM Clientes;

END;

Modificar el bloque desarrollado en el [ejercicio anterior](#) de modo que se use una estructura para agrupar los datos recuperados de la tabla impositiva

Resultado:

[ejer1_v2.sql](#)



Las tablas PL/SQL son similares a las matrices de C en tratamiento.

Es necesario declarar un tipo para la tabla:

```
TYPE <tipo_tabla> IS TABLE OF <tipo> INDEX BY  
    BINARY_INTEGER;
```

Una vez creado el tipo, se declara la variable:

```
<nombre_tabla> <tipo_tabla>;
```

Para acceder a una tupla de la tabla se usa la sintaxis:

`tabla(<numero fila>)`

Si se trata de una tabla de registros, se accede con la sintaxis:

`tabla(<numero fila>).campo`




```
DECLARE
```

```
TYPE Cliente IS RECORD (
```

```
    IdCliente NUMBER,
```

```
    NombreCompañia VARCHAR2(50),
```

```
    Ciudad    VARCHAR2(20)
```

```
);
```

```
TYPE Clientes IS TABLE OF Cliente INDEX BY BINARY_INTEGER ;
```

```
tClientes Clientes;
```

```
BEGIN
```

```
    tClientes(1).IdCliente := 1;
```

```
    tClientes(1).NombreCompañia:= 'compañía SA';
```

```
    tClientes(1).Ciudad := 'Granada';
```

```
END;
```

Una asignación a un elemento con índice <numero fila> que no exista crea esa entrada en la tabla.

Un acceso a un elemento con índice <numero fila> que no exista devuelve una excepción del tipo ORA-1403: No data found.



La tabla tiene campos internos.

Se acceden con:

`<nombre_tabla>.<nombre_campo>`

Son:

COUNT

DELETE[(i[,j])]

EXISTS(i)

FIRST

LAST

NEXT(i)

PRIOR(i)



Crear una tabla capaz de contener números enteros y meter en las entradas de índice 1, 2, 3, 5, 7 y 11 los número 1, 2, 3, 5, 7 y 11, respectivamente. Posteriormente, mostrar en la salida el contenido de las entradas de la tabla para esas posiciones.

Resultado:

[ejer2_v1.sql](#)

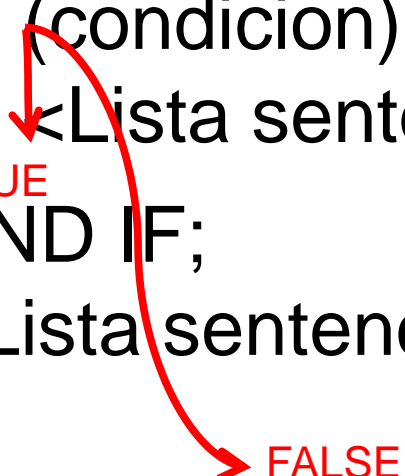
1. Características de PL/SQL
2. Arquitectura cliente-servidor en PL/SQL
3. El concepto central en PL/SQL: el bloque
4. Estructuras de datos en PL/SQL: tipos, variables, estructuras y colecciones
5. Estructuras de control: selección e iteración
6. Subprogramas: procedimientos, funciones y paquetes
7. Disparadores
8. El resultado de las consultas paso a paso: cursores
9. Excepciones

```
IF (condicion) THEN
    <Lista sentencias>
```

```
END IF;
<Lista sentencias>
```



```
IF (condicion) THEN
  <Lista sentencias>
TRUE
END IF;
<Lista sentencias>
FALSE
```



```

IF (condicion) THEN
    <Lista sentencias>
ELSE
    <Lista sentencias>
END IF;
    
```




```

IF (condicion) THEN
  <Lista sentencias>
ELSE
  <Lista sentencias>
END IF;
  
```

Diagram illustrating the execution flow of an IF statement:

- A red arrow points from the condition `(condicion)` to the word **TRUE**.
- Another red arrow points from the word **FALSE** to the `ELSE` branch.



```

IF (condicion) THEN
  <Lista sentencias>
ELSIF (condicion) THEN
  <Lista sentencias>
ELSE
  <Lista sentencias>
END IF;
  
```

Diagram illustrating the flow of a selection structure (IF-ELSIF-ELSE) based on the truth value of the condition:

- IF (condicion) THEN:** If the condition is **TRUE**, the flow proceeds to the first list of statements (**<Lista sentencias>**). If the condition is **FALSE**, the flow proceeds to the **ELSIF** block.
- ELSIF (condicion) THEN:** If the condition is **TRUE**, the flow proceeds to the second list of statements (**<Lista sentencias>**). If the condition is **FALSE**, the flow proceeds to the **ELSE** block.
- ELSE:** If the condition is **FALSE**, the flow proceeds to the third list of statements (**<Lista sentencias>**).
- END IF;** The flow ends after the selected block of statements.

Modificar el bloque desarrollado en el [ejercicio anterior](#) de modo que se calcule la cuota integra correspondiente a la cantidad de 120202,42.

Resultado:

[ejer1_v3.sql](#)



Dado que provoca un problema, controlar previamente si existe una tupla para calcular dichos datos y si no, devolver el mensaje (“No existen datos”).

Resultado:

[ejer1_v4.sql](#)



```

LOOP
  <Lista sentencias>
END LOOP;
```



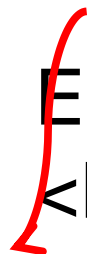

LOOP

<Lista sentencias>

EXIT;

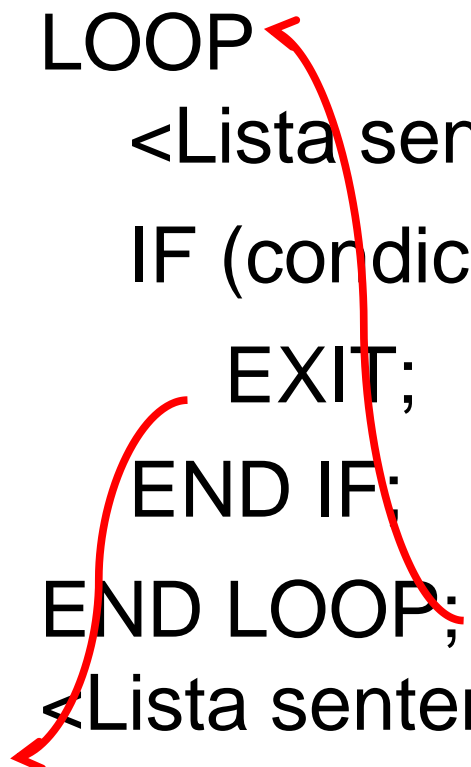
END LOOP;

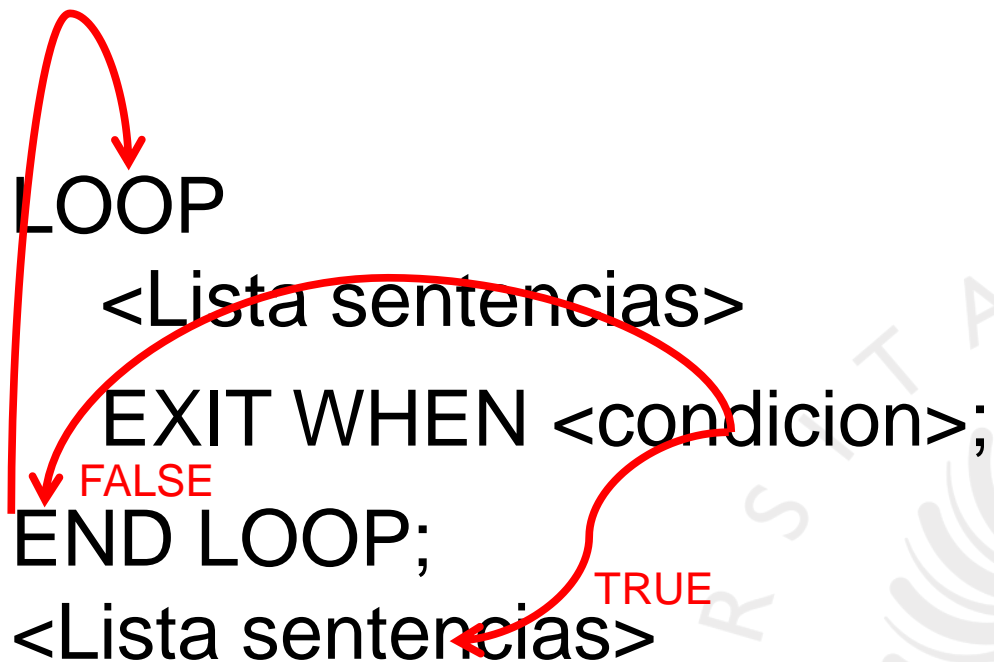
<Lista sentencias>



```

LOOP
  <Lista sentencias>
  IF (condicion) THEN
    EXIT;
  END IF;
END LOOP;
<Lista sentencias>
  
```



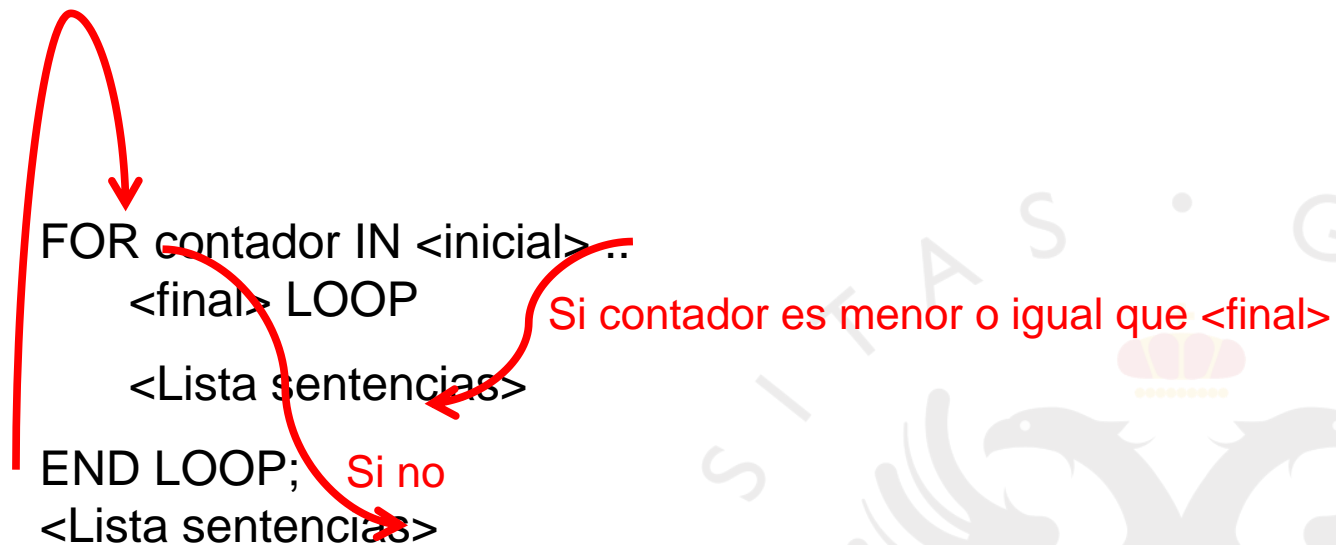




```

WHILE (condicion) LOOP
  <Lista sentencias>
END LOOP;
<Lista sentencias>
  
```

Diagram illustrating the execution flow of a WHILE loop:

- The loop starts at the beginning of the `WHILE (condicion) LOOP` block.
- If the condition is **TRUE**, the flow proceeds to the `<Lista sentencias>` block.
- After executing the `<Lista sentencias>` block, the flow loops back to the start of the `WHILE (condicion) LOOP` block.
- If the condition is **FALSE**, the flow exits the loop and proceeds to the `<Lista sentencias>` block located below the `END LOOP;` statement.



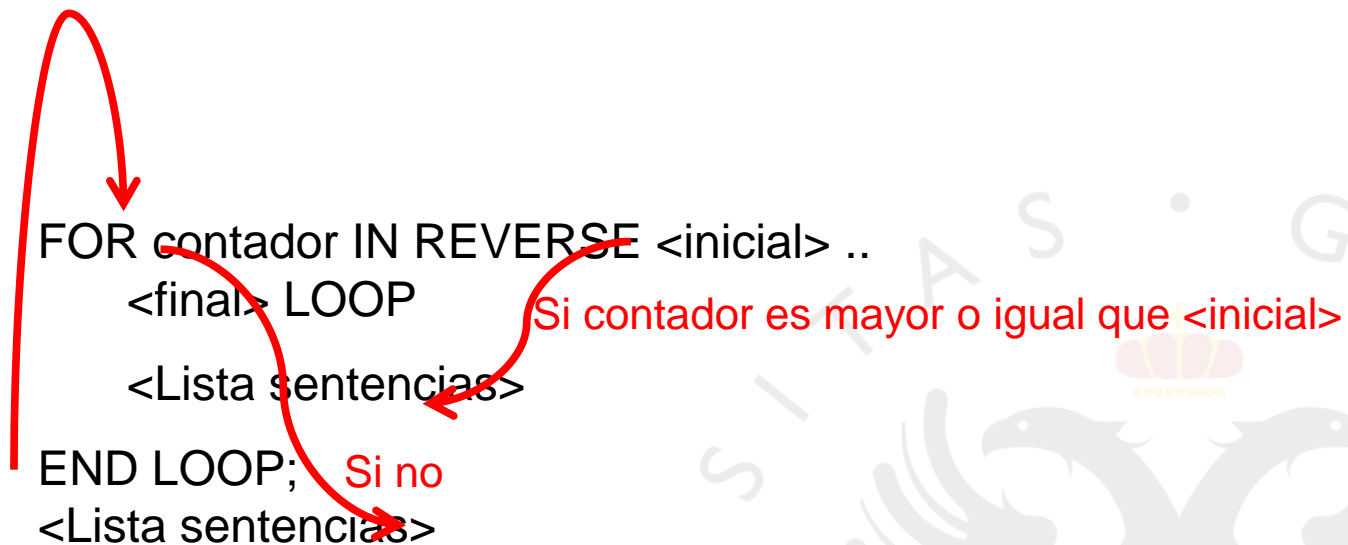
```

FOR contador IN REVERSE <inicial> ..
    <final> LOOP
        <Lista sentencias>
    END LOOP;
    <Lista sentencias>

```

Si contador es mayor o igual que <inicial>

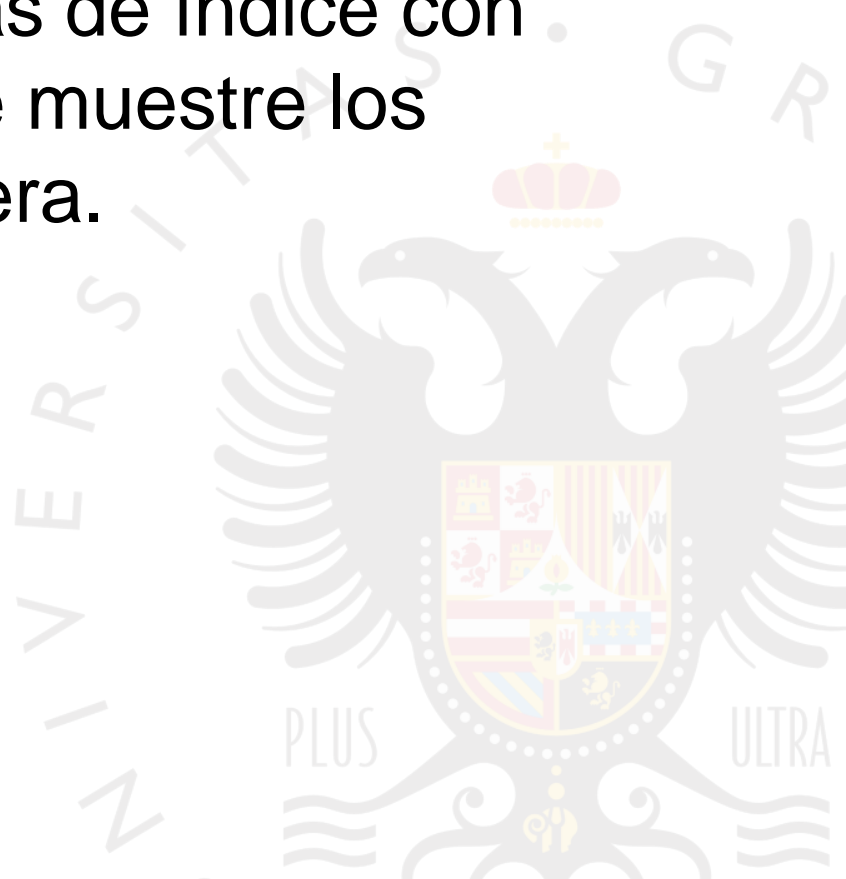
Si no



Modificar el bloque del [ejercicio 2](#) para que rellene la tabla con valores entre los valores 1 y 20, en entradas de índice con el mismo valor. Hacer que muestre los valores de la misma manera.

Resultado:

[ejer2_v2.sql](#)



1. Características de PL/SQL
2. Arquitectura cliente-servidor en PL/SQL
3. El concepto central en PL/SQL: el bloque
4. Estructuras de datos en PL/SQL: tipos, variables, estructuras y colecciones
5. Estructuras de control: selección e iteración
6. Subprogramas: procedimientos, funciones y paquetes
7. Disparadores
8. El resultado de las consultas paso a paso: cursores
9. Excepciones

Un bloque que necesita reutilizarse en el futuro se encapsula dentro de un módulo (función o procedimiento).

Esto permite:

- aislar funcionalidad,
- corregir errores fácilmente, y
- reutilizar en el futuro.

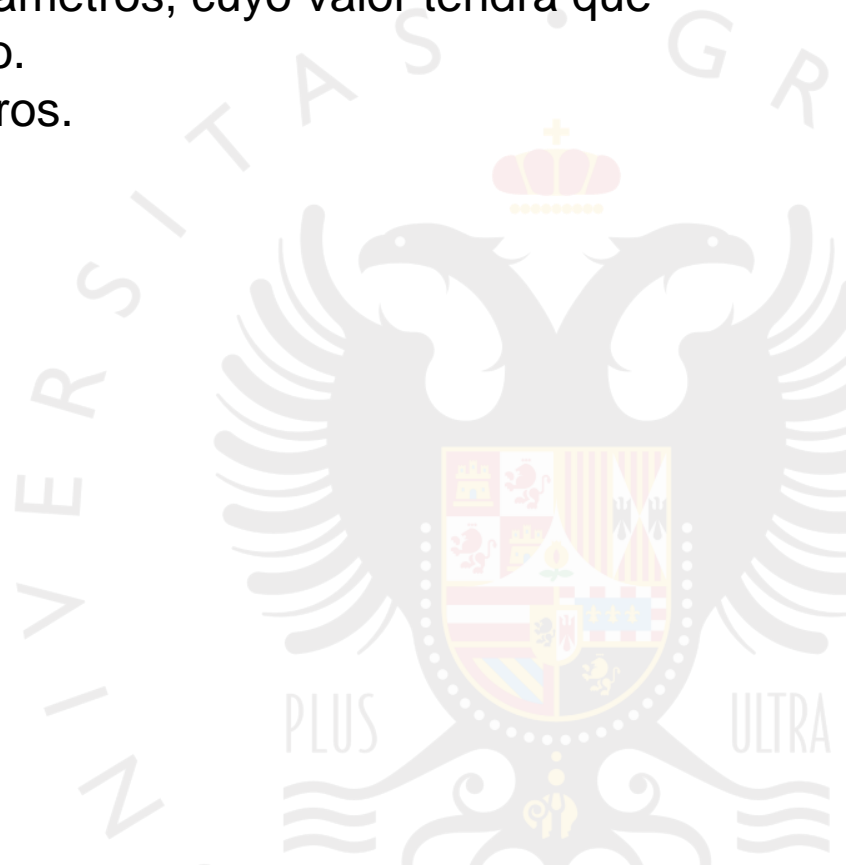


El primer paso: convertir constantes en variables (pasos [2](#) y [3](#) del ejercicio primero):

los valores se modifican rápidamente.

El segundo paso: convertir variables en parámetros, cuyo valor tendrá que comunicarse cuando se ejecute el módulo.

El tercer paso: la interfaz o lista de parámetros.



Convertir constantes en variables

```

DECLARE
TYPE Cliente IS RECORD (
    IdCliente NUMBER,
    NombreCompañía VARCHAR2(50),
    Ciudad      VARCHAR2(20)
);
miCliente1 Cliente;
idCliente := 1;
nombreCompañía := 'compañía SA';
ciudad := 'Granada';
BEGIN
    miCliente1.IdCliente := idCliente;
    miCliente1.NombreCompañía:= nombreCompañía;
    miCliente1.Ciudad := ciudad;
END;
```



Variables a parámetros

```

DECLARE
TYPE Cliente IS RECORD (
    IdCliente NUMBER,
    NombreCompañía VARCHAR2(50),
    Ciudad      VARCHAR2(20)
);
miCliente1 Cliente;
idCliente := 1;
nombreCompañía := 'compañía SA';
ciudad := 'Granada';
BEGIN
    miCliente1.IdCliente := idCliente;
    miCliente1.NombreCompañía:= nombreCompañía;
    miCliente1.Ciudad := ciudad;
END;
```



Parámetros de entrada: son los que hay que comunicar a un módulo.

Palabra reservada IN

Parámetros de salida: son los que un módulo comunica al módulo o bloque que le llama.

Palabra reservada OUT

Parámetros de entrada/salida: son los que se comunican a un módulo pero pueden ser modificados por este.

Palabra reservada IN OUT

Función: módulo que recibe varios parámetros de entrada y devuelve un único valor de salida (valor, que no parámetro):

```
FUNCTION <nombre> (<Lista parametros funcion>)  
    RETURN <tipo> [AS | IS] <bloque sin DECLARE>
```



Una lista de parámetros de una función es una serie de parámetros separados por comas, donde cada parámetro tiene la siguiente forma:

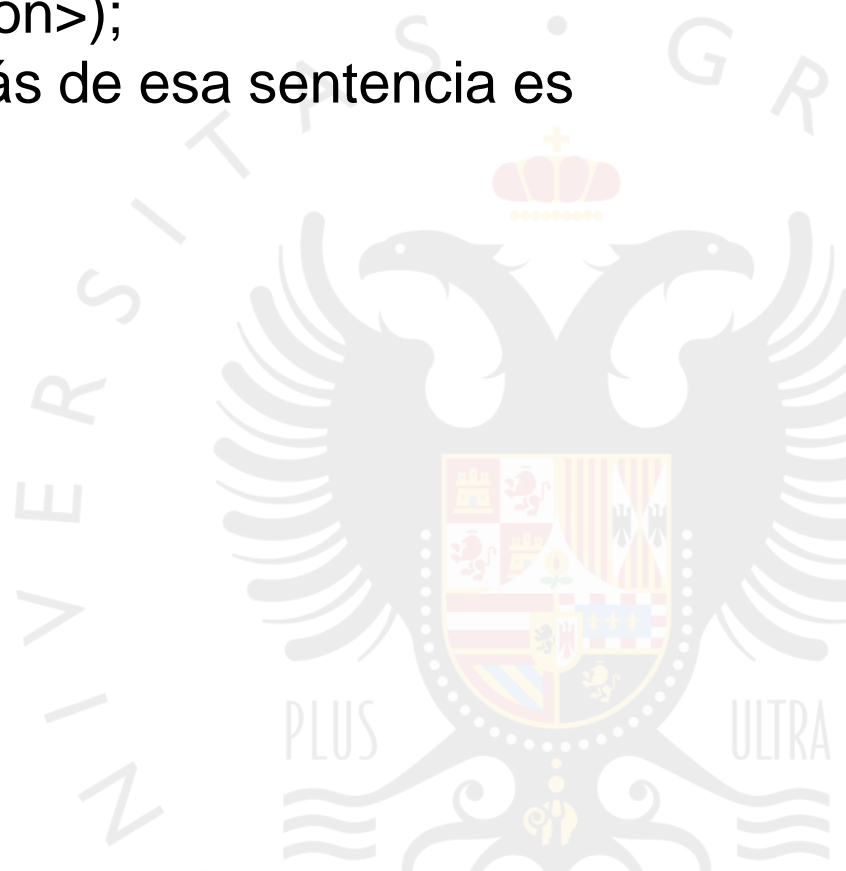
<nombre parametro> <tipo>



Toda función termina devolviendo un valor del mismo tipo que el especificado en su cabecera, mediante la cláusula:

RETURN (<expresion>);

Cualquier código que aparezca detrás de esa sentencia es inútil.



Ejemplo:

```
FUNCTION suma (a NUMBER, b NUMBER) RETURN
    NUMBER AS
```

```
BEGIN
```

```
    RETURN (a + b);
```

```
END;
```

La llamada a una función implica siempre el uso del valor devuelto:

```
sumados := suma (a, b);
```

La sentencia SQL para compilar y almacenar una función PL/SQL en la base de datos es:

```
CREATE [OR REPLACE] FUNCTION <nombre> (<Lista parametros
funcion>) RETURN <tipo> [AS | IS] <bloque sin DECLARE>
```

Si el código fuente de un módulo tiene fallos, la sentencia SQL para mostrar dichos fallos es:

show errors



```
CREATE OR REPLACE FUNCTION ObtenerDatosCliente
    (idCliente NUMBER) RETURN VARCHAR2 IS
    output VARCHAR2;

BEGIN
    SELECT nombreCompañia INTO output FROM Clientes      c WHERE
    idCliente = c.idCliente;
    return (output);
END ;

BEGIN
    DBMS_OUTPUT.PUT_LINE ('El nombre de la compañía es ' ||
        ObtenerDatosCliente (1));
END;
```



Crear una función, a partir de la [última versión del bloque del ejercicio 1](#), que calcule la cuota íntegra para una cantidad que se le pasa mediante un parámetro y devuelva dicha cantidad (en vez de mostrarla como salida por mensaje).

Resultado:

[ejer1_v5.sql](#)

Probar la correcta ejecución de la función
con dos valores distintos (34558.2 y
120202.42)

Resultado:

[ejer1_v6.sql](#)



Procedimiento: módulo que recibe ninguno, uno o varios parámetros de entrada (o de entrada/salida) y modifica ninguno, uno o varios parámetros de salida (o de entrada salida):

PROCEDURE <nombre> (<Lista parametros procedimiento>) AS <bloque sin DECLARE>



Una lista de parámetros de un procedimiento es una serie de parámetros separados por comas, donde cada parámetro tiene la siguiente forma:

<nombre parametro> <modo> <tipo>

Los modos posibles son IN, OUT e IN OUT



La sentencia SQL para compilar y almacenar un procedimiento PL/SQL en la base de datos es:

```
CREATE [OR REPLACE] PROCEDURE <nombre> (<Lista parametros procedimiento>) AS <bloque sin DECLARE>
```

Si el código fuente de un módulo tiene fallos, la sentencia SQL para mostrar dichos fallos es:

```
show errors
```

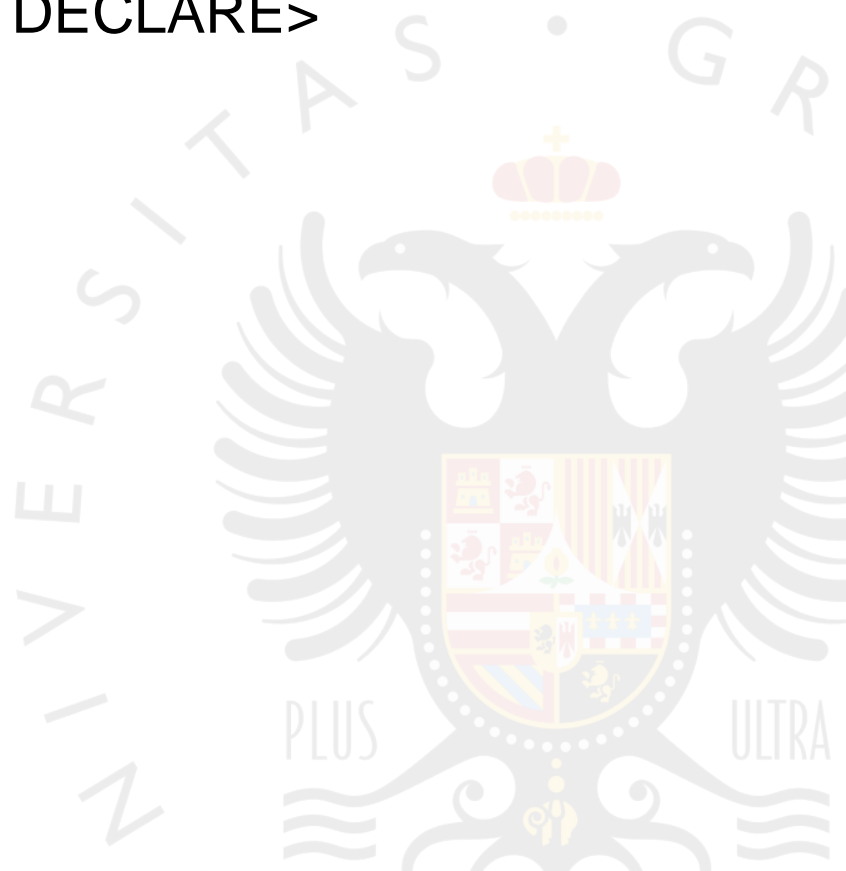
```
CREATE OR REPLACE
PROCEDURE ActualizaCiudad(idCliente NUMBER,
                          newCiudad VARCHAR2)

IS
BEGIN
UPDATE Clientes c SET ciudad = newCiudad
WHERE idCliente = c.idCliente;
END ActualizaCiudad;
```



Paquete: conjunto de declaraciones de tipos, de variables, de funciones y de procedimientos agrupados siguiendo un criterio lógico:

PACKAGE <nombre> **AS** <bloque sin DECLARE>



La declaración de un paquete tiene dos pasos:

- declaración de la especificación: contiene las declaraciones de tipos, de variables, interfaces de funciones e interfaces de procedimientos que serán accesibles desde el exterior del paquete, y
- declaración del cuerpo: contiene las declaraciones de tipos y de variables internas al paquete, y las declaraciones de todas las funciones y procedimientos.

Veamos un ejemplo para la creación de un paquete `p` con las siguientes premisas:

- cinco variables (`a`, `b`, `c`, `d`, `e`),

- dos funciones (`f1` y `f2`) y

- tres procedimientos (`p1`, `p2`, `p3`)

- las variables `d` y `e`, la función `f2` y el procedimiento `p2`

- son útiles para el funcionamiento interno del

- paquete, pero no son necesarios para el exterior

- (nadie de fuera los necesita).

La especificación del paquete debería incluir:
 las declaraciones de las variables a, b y c,
 la interfaz (o cabecera) de la función f1 y
 las interfaces (o cabeceras) de los procedimientos p1 y
 p3.



La sentencia PL/SQL para la creación de la especificación de un paquete es:

```
CREATE [OR REPLACE] PACKAGE p [AS | IS]
```

```
  a <tipo>;
```

```
  b <tipo>;
```

```
  c <tipo>;
```

```
  FUNCTION f1 (...) RETURN <tipo>;
```

```
  PROCEDURE p1 (...);
```

```
  PROCEDURE p3 (...);
```

```
END [p];
```



La sentencia PL/SQL para la creación del cuerpo de un paquete es:

```
CREATE [OR REPLACE] PACKAGE BODY p AS
    d <tipo>;
    e <tipo>;
    FUNCTION f1 (...) RETURN <tipo>
        AS ...;
    FUNCTION f2 (...) RETURN <tipo>
        AS ...;
    PROCEDURE p1 (...) AS ...;
    PROCEDURE p2 (...) AS ...;
    PROCEDURE p3 (...) AS ...;
END [p];
```



Para acceder a una variable, función o procedimiento contenidos en un paquete se usa la sintaxis:

<nombre paquete>.<nombre variable>

<nombre paquete>.<nombre modulo> (<lista parametros reales>)

Por ejemplo, serían accesos válidos:

p.a

p.f1 (...)

p.p3 (...)



Sin embargo, no serían accesos válidos:

p.d

p.f2 (...)



```
CREATE OR REPLACE PACKAGE PKG_CLIENTE
IS
```

```
-- Declaraciones de tipos y variables públicas
```

```
TYPE Cliente IS RECORD (
```

```
    IdCliente NUMBER,
```

```
    NombreCompañía VARCHAR2(50),
```

```
    Ciudad      VARCHAR2(20)
```

```
);
```

```
miCliente Cliente;
```

```
-- Declaraciones de procedimientos y funciones pública
```

```
PROCEDURE ActualizaCiudad(idCliente NUMBER,
```

```
    newCiudad VARCHAR2);
```

```
FUNCTION ObtenerDatosCliente (idCliente NUMBER) RETURN VARCHAR2;
```

```
END PKG_CLIENTE ;
```

1. Características de PL/SQL
2. Arquitectura cliente-servidor en PL/SQL
3. El concepto central en PL/SQL: el bloque
4. Estructuras de datos en PL/SQL: tipos, variables, estructuras y colecciones
5. Estructuras de control: selección e iteración
6. Subprogramas: procedimientos, funciones y paquetes
7. Disparadores
8. El resultado de las consultas paso a paso: cursores
9. Excepciones

Disparador: bloque PL/SQL asociado a un evento ocurrido dentro de la base de datos.

Los eventos a los que puede asociarse un disparador son:

INSERT,

DELETE, o

UPDATE (de una, varias o todas las columnas de la tabla).

Para cada evento, pueden definirse dos momentos de disparo:

BEFORE o

AFTER

Si la operación asociada afecta a más de una tupla, puede elegirse si se ejecuta el bloque para una o para todas las tuplas de la operación.

Cláusula FOR EACH ROW



```
CREATE OR REPLACE TRIGGER <nombre>
{BEFORE | AFTER}
{INSERT | [OR] DELETE | [OR] UPDATE [OF <lista
  nombres columna>]}
ON <nombre tabla>
[FOR EACH ROW]
<bloque>
```



Disparadores: sintaxis

Un disparador puede asociarse a varios eventos, pero todos en el mismo momento.:

AFTER INSERT OR DELETE OR UPDATE



Disparadores: utilidad

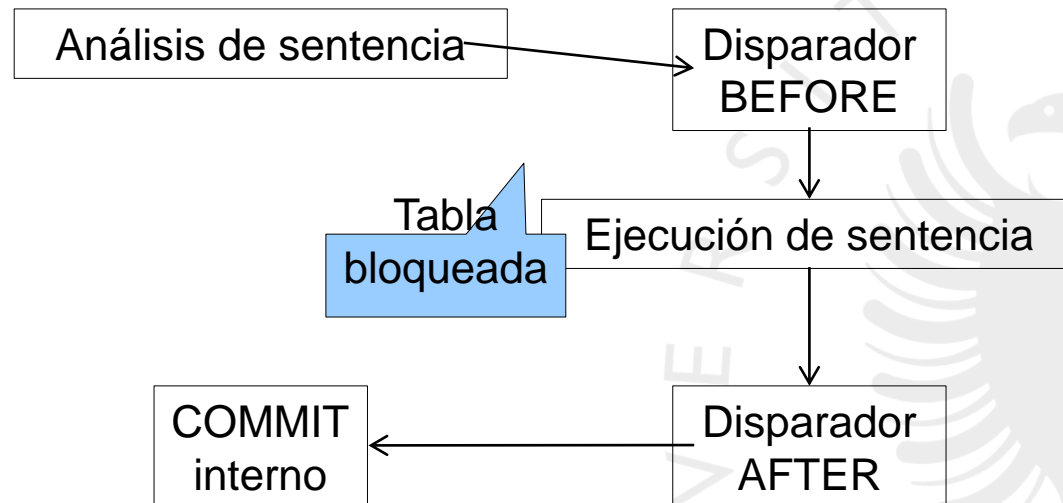
Un disparador BEFORE suele usarse:

- En el INSERT, para comprobar restricciones, y
- En el DELETE, para garantizar consistencia antes de borrar.

Un disparador AFTER suele emplearse:

- En el INSERT, para garantizar consistencia.

Los disparadores son invocados por el propio ejecutor de consultas siguiendo el siguiente esquema:



Un disparador puede tener dos variables internas de tipo registro, que son :new y :old.

Ambas son registros del tipo de la fila de la tabla a la que se asocia el disparador.

:new contiene los valores de la nueva tupla (en la inserción) o los nuevos valores para la tupla ya existente (en la modificación).

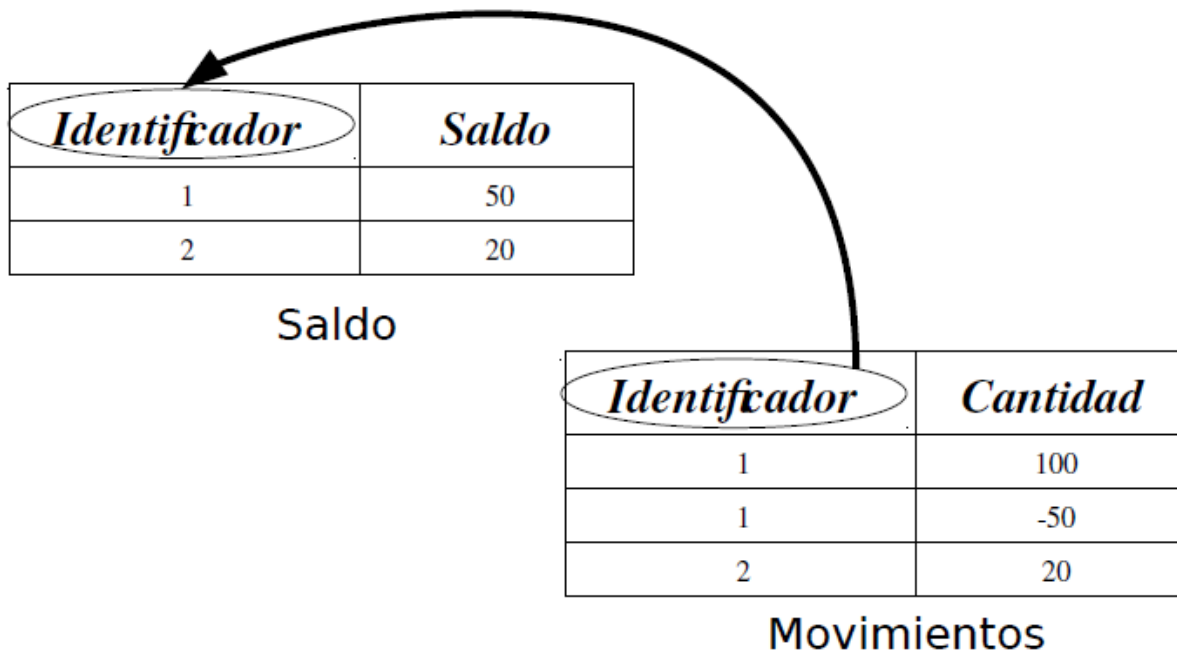
:old contiene los valores de la tupla para borrado o los antiguos valores de una tupla modificada.

Un disparador de inserción no tiene registro :old.

Un disparador de borrado no tiene registro :new.



Disparadores: ejemplo de uso



Disparadores: ejemplo de uso

La inserción de un nuevo movimiento para la cuenta 2 debería desencadenar una actualización del saldo de la cuenta en la tabla de saldos.

<i>Identificador</i>	<i>Cantidad</i>
1	100
1	-50
2	20
2	-5

Movimientos

<i>Identificador</i>	<i>Saldo</i>
1	50
2	15

Saldo

Disparadores: ejemplo de uso

Solución (después de [crear las tablas](#)):

```
CREATE OR REPLACE TRIGGER nuevoSaldo  
AFTER INSERT ON movimiento  
FOR EACH ROW  
BEGIN
```

```
    UPDATE saldo SET saldo = saldo + :new.cantidad WHERE  
        identificador = :new.identificador;
```

```
END;
```

Comprobar el funcionamiento del disparador con la inserción de (2,-5) en *movimiento*.

Para controlar el borrado de movimientos:

```
CREATE OR REPLACE TRIGGER nuevoSaldo
AFTER INSERT OR DELETE ON movimiento
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        UPDATE saldo SET saldo = saldo + :new.cantidad WHERE identificador =
            :new.identificador;
    ELSE
        UPDATE saldo SET saldo = saldo - :old.cantidad WHERE identificador =
            :old.identificador;
    END IF;
END;
```

Comprobar el resultado de insertar un movimiento para una cuenta que no exista.



Solucione el fallo sin tener que meter
“manualmente” una tupla para la cuenta
inexistente.

Resultado:

[ejer3_v2.sql](#)



Modificar los disparadores para observar todos los casos posibles sobre la tabla *movimiento*.

Resultado:

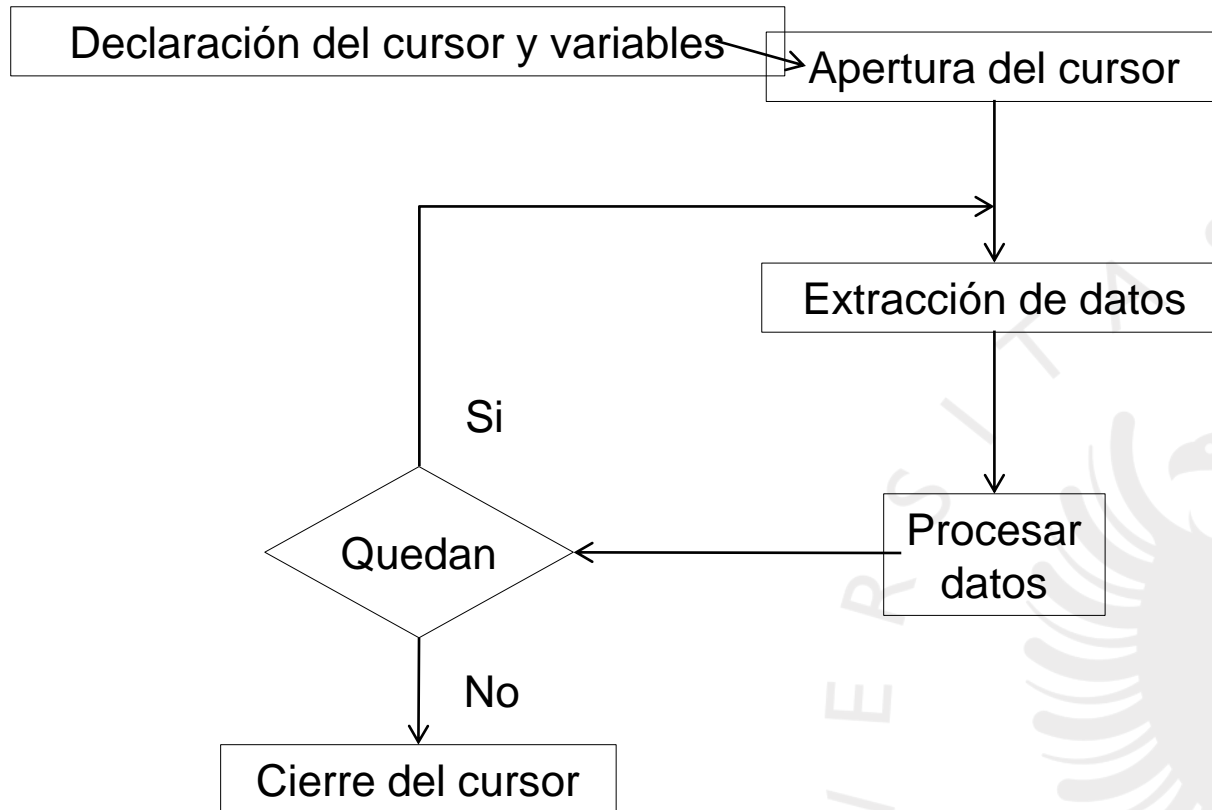
[ejer3_v3.sql](#)



1. Características de PL/SQL
2. Arquitectura cliente-servidor en PL/SQL
3. El concepto central en PL/SQL: el bloque
4. Estructuras de datos en PL/SQL: tipos, variables, estructuras y colecciones
5. Estructuras de control: selección e iteración
6. Subprogramas: procedimientos, funciones y paquetes
7. Disparadores
8. El resultado de las consultas paso a paso: cursores
9. Excepciones

Cursor: estructura de datos que almacena información acerca de una sentencia analizada (incluyendo número de filas procesadas, la versión de la sentencia analizada y las filas de la relación resultante en el caso de consultas).

Cursores: vida de un cursor



Cursores: declaración

Todo cursor se asocia a una consulta y tiene que ser declarado en la sección de declaraciones de un bloque:

```
CURSOR <nombre> IS <sentencia>;
```

Por ejemplo:

```
CURSOR saldos IS SELECT * FROM saldo;
```



La consulta asociada a un cursor se ejecuta cuando se abre dicho cursor, mediante la sentencia:

```
OPEN <nombre cursor>;
```

Por ejemplo:

```
OPEN saldos;
```



Cursores: extracción de datos

Cada fila almacenada dentro del cursor debe ser transferida para su tratamiento a una variable del mismo tipo que el de una fila del cursor.

Esa variable tiene que declararse explícitamente o mediante la sentencia:

```
<nombre variable> <nombre cursor>%ROWTYPE;
```

Para cargar una fila dentro de una variable, se usa la sentencia:

```
FETCH <nombre cursor> INTO <nombre variable>;
```

Las consideraciones de la cláusula INTO son las mismas que las de la sentencia SELECT ...
INTO ...

Cursores: variables

Un cursor tiene varias variables internas:

%ISOPEN: especifica si el cursor está abierto.

%ROWCOUNT: especifica el número de filas almacenadas dentro de un cursor abierto.

%FOUND: especifica si la última operación FETCH ha devuelto datos.

%NOTFOUND: especifica si la última operación FETCH no ha devuelto datos.

Cursores: recorrido

Para recorrer un cursor es necesario (después de abierto) usar bucles con centinela del tipo:

OPEN saldos;

FETCH saldos INTO registro;

WHILE (saldos%FOUND) LOOP

 <procesamiento de registro>

 FETCH saldos INTO registro;

END LOOP;

CLOSE saldos;



Cursores: recorrido

```
DECLARE
  CURSOR cCliente IS SELECT * FROM Clientes;
  IdCliente NUMBER;
  NombreCompañia VARCHAR2(50);
  Ciudad    VARCHAR2(20);
BEGIN
  OPEN cCliente;
  FETCH cCliente INTO IdCliente, NombreCompañia, Ciudad;
  WHILE cCliente%found LOOP
    dbms_output.put_line('El id = ' || IdCliente || ' corresponde a ' ||
      NombreCompañia);
    FETCH cCliente INTO IdCliente, NombreCompañia, Ciudad;
  END LOOP;
  CLOSE cCliente;
END;
```



Cursores: recorrido

También es posible una estructura del tipo:

FOR registro IN saldos LOOP

<procesamiento de registro>

END LOOP;

Esta operación lleva implícita la apertura y cierre del cursor, así como el FETCH en cada iteración.

Cursores: recorrido

```
DECLARE
  CURSOR cCliente IS SELECT * FROM Clientes;
BEGIN
  FOR regCliente IN cCliente LOOP
    dbms_output.put_line('El id = ' || regCliente.IdCliente ||
      ' corresponde a ' || regCliente.NombreCompañía);
  END LOOP;
END;
```

Cursores: recorrido

```

DECLARE
  CURSOR cCliente IS SELECT * FROM Clientes;
  registroCliente cCliente%ROWTYPE;
BEGIN
  OPEN cCliente;
  LOOP
    FETCH cCliente INTO registroCliente;
    EXIT WHEN cCliente%NOTFOUND;
    dbms_output.put_line('El id = ' || registroCliente.IdCliente || ' corresponde a ' ||
      registroCliente.NombreCompañia);
  END LOOP;
  CLOSE cCliente;
END;

```



Se puede insertar una tupla en *saldo* que no tenga asociada ninguna tupla en *movimiento*. Crear un procedimiento que compruebe que existe al menos una tupla en *movimiento* para cada tupla de *saldo* y, si no es así, que elimine la entrada correspondiente en la tabla *saldo*.

Resultado:

[ejer4_v1.sql](#)

1. Características de PL/SQL
2. Arquitectura cliente-servidor en PL/SQL
3. El concepto central en PL/SQL: el bloque
4. Estructuras de datos en PL/SQL: tipos, variables, estructuras y colecciones
5. Estructuras de control: selección e iteración
6. Subprogramas: procedimientos, funciones y paquetes
7. Disparadores
8. El resultado de las consultas paso a paso: cursores
9. Excepciones

Excepción: error que se produce durante la ejecución de una sentencia o de un bloque PL/SQL.

Interrumpe la ejecución normal del bloque o sentencia.

Cada excepción puede ser tratada por un manejador de excepción.

Hay excepciones predefinidas del sistema.

Pueden ser declaradas por el programador.

Excepciones: Declaración

Existe un tipo de dato EXCEPTION:
DECLARE

e_movimientosSinCuenta EXCEPTION;

Cada excepción lleva asociado un código de error (negativo) y un mensaje de error.

Existen excepciones predefinidas del sistema que se muestran en la siguiente tabla.

Excepciones predefinidas

Exception	ORA Error	SQLCODE	Raise When ...
ACCESS_INTO_NULL	06530	-6530	A program attempts to assign values to the attributes of an uninitialized object
CASE_NOT_FOUND	06592	-6592	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	A program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	06511	-6511	A program attempts to open an already open cursor. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers, so your program cannot open that cursor inside the loop.
DUP_VAL_ON_INDEX	00001	-1	A program attempts to store duplicate values in a column that is constrained by a unique index.
INVALID_CURSOR	01001	-1001	A program attempts a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	In a SQL statement, the conversion of a character string into a number fails because the string does not represent a valid number. (In procedural statements, VALUE_ERROR is raised.) This exception is also raised when the LIMIT -clause expression in a bulk FETCH statement does not evaluate to a positive number.
LOGIN_DENIED	01017	-1017	A program attempts to log on to Oracle with an invalid username or password.

Excepciones predefinidas

Exception	ORA Error	SQLCODE	Raise When ...
NO_DATA_FOUND	01403	+100	<p>A SELECT INTO statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an index-by table.</p> <p>Because this exception is used internally by some SQL functions to signal completion, you should not rely on this exception being propagated if you raise it within a function that is called as part of a query.</p>
NOT_LOGGED_ON	01012	-1012	A program issues a database call without being connected to Oracle.
PROGRAM_ERROR	06501	-6501	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. When an open host cursor variable is passed to a stored subprogram, the return types of the actual and formal parameters must be compatible.
SELF_IS_NULL	30625	-30625	A program attempts to call a MEMBER method, but the instance of the object type has not been initialized. The built-in parameter SELF points to the object, and is always the first parameter passed to a MEMBER method.
STORAGE_ERROR	06500	-6500	PL/SQL runs out of memory or memory has been corrupted.
SUBSCRIPT_BEYOND_COUNT	06533	-6533	A program references a nested table or varray element using an index number larger than the number of elements in the collection.

Excepciones predefinidas

Exception	ORA Error	SQLCODE	Raise When ...
SUBSCRIPT_OUTSIDE_LIMIT	06532	-6532	A program references a nested table or varray element using an index number (-1 for example) that is outside the legal range.
SYS_INVALID_ROWID	01410	-1410	The conversion of a character string into a universal rowid fails because the character string does not represent a valid rowid.
TIMEOUT_ON_RESOURCE	00051	-51	A time out occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	01422	-1422	A SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR. In procedural statements, VALUE_ERROR is raised if the conversion of a character string into a number fails. (In SQL statements, INVALID_NUMBER is raised.)
ZERO_DIVIDE	01476	-1476	A program attempts to divide a number by zero.

Supongamos que, en nuestro bloque, el hecho de que una operación `SELECT` sobre la cuenta *saldo* produzca una excepción *no_data_found* quiera ser capturado como una excepción específica *e_movimientosSinCuenta* que pueda ser tratada como tal.

La directiva (PRAGMA) `EXCEPTION_INIT` permite asociar el código de error asociado a *no_data_found* a una excepción definida por el usuario:

```
PRAGMA EXCEPTION_INIT (e_movimientosSinCuenta,  
-1403);
```

Esta operación consigue que, cuando se produzca la excepción *no_data_found* en el ámbito de la variable *e_cuentaSinMovimientos*, el manejador de excepción que se dispara es el asociado a esta última:

DECLARE

e_movimientosSinCuenta EXCEPTION;

PRAGMA EXCEPTION_INIT (e_movimientosSinCuenta, -
1403);

TYPE t_cuenta IS saldo%ROWTYPE;

r_cuenta t_cuenta;

...

```
BEGIN
```

```
...
```

```
SELECT * INTO r_cuenta FROM saldo WHERE  
    identificador = id;
```

```
...
```

```
EXCEPTION
```

```
    WHEN e_movimientosSinCuenta THEN  
        RAISE;
```

```
END;
```



El error que se produce en ambas es -1403 (no se encuentran datos) pero, en nuestro caso, generamos una excepción particular y controlada.



Los manejadores de excepciones se declaran en la sección EXCEPTION del bloque (antes del END del bloque).

Cada manejador se describe con:

WHEN <excepcion> THEN

<Cuerpo de bloque sin DECLARE, BEGIN ni
END>

<Fin de excepcion>

Una excepción puede acabar con una sentencia RAISE, en cuyo caso se produce la misma excepción en el bloque que contiene a aquel en el que produjo inicialmente:

RAISE;

Un manejador de excepción puede informar de otra excepción distinta a la producida en el bloque:

RAISE <excepcion>;

Un manejador puede interrumpir la ejecución de todos los bloques devolviendo un mensaje de error distinta de las predefinidas:

`RAISE_APPLICATION_ERROR` (<codigo de error>, <Mensaje de error>);

Los códigos de error son negativos y entre -20000 y -20999.

El mensaje tiene que ser menor de 512 caracteres.

Un manejador puede realizar una serie de acciones en el cuerpo de bloque del manejador sin promocionar excepción alguna (`RAISE` o `RAISE_APPLICATION_ERROR`). En este caso, el bloque que llamó al que produce la excepción, o en el que está insertado, sigue su ejecución normal.

```
DECLARE
```

```
    compañía VARCHAR(20);
```

```
BEGIN
```

```
[Otras sentencias]
```

```
BEGIN
```

```
    SELECT compañía INTO compañía FROM Clientes WHERE IdCliente = 1;
```

```
    EXCEPTION
```

```
    WHEN NO_DATA_FOUND THEN
```

```
        DBMS_OUTPUT.PUT_LINE("No hay ningún cliente con ese identificador.");
```

```
    WHEN TOO_MANY_ROWS THEN
```

```
        DBMS_OUTPUT.PUT_LINE("Hay más de un cliente con ese identificador.");
```

```
    WHEN OTHERS THEN
```

```
        DBMS_OUTPUT.PUT_LINE("Error, abortando ejecución.");
```

```
        RAISE;
```

```
    END;
```

```
[Otras sentencias]
```

```
END;
```

Crear un procedimiento que borre un movimiento indicado por los parámetros de entrada. Tratar las excepciones

- para los casos de que no se tengan movimientos asociados a los parámetros de entrada
- para cuando el saldo de la cuenta se quede a 0.

Resultado:

[ejercicioExcepciones.sql](#)

- A la hora de realizar la implementación del SI, podéis utilizar cualquier lenguaje de programación que dominéis que proporcione recursos para conectarse a algún SGBD relacional.
- Se recomienda utilizar el servidor Oracle de la Escuela y el lenguaje que hayáis empleado en el Seminario 1.
- En el desarrollo de SI, especialmente (pero no solo) en el ámbito de aplicaciones Web, suelen utilizarse también “Software Stack”, formados por la combinación de distintos tipos de Software que proporcionan una plataforma para todos los aspectos de la implementación del SI.
- Los Stacks tienen la ventaja de que tienden a ser mantenidos y evolucionados, proporcionando plataformas bien probadas a través de muchas aplicaciones, maduras, robustas y con mejores perspectivas en lo que se refiere a mantenimiento de compatibilidad conforme aparecen nuevas versiones de los productos Software que las conforman.
- No se recomienda su uso en esta práctica, pero si queréis usarlos, consultad previamente al profesorado.

- Ejemplos (con SGBD relacionales):
 - **ORACLE Stack**. Oracle proporciona múltiples herramientas.
 - **WAMP**: **Windows** (SO), **Apache** (servidor Web), **MySQL** ó **MariaDB** (SGBD), y **Perl, PHP o Python** (lenguajes de scripting).
 - **MAMP y LAMP**: como **WAMP** reemplazando **Windows** por **MacOS** y **Linux**, respectivamente.
 - **LAPP**: **Linux** (SO), **Apache** (servidor Web), **PostgreSQL** (SGBD), y **Perl, PHP o Python** (lenguajes de scripting).
 - **NMP**: **Nginx** (servidor Web), **PostgreSQL** (SGBD), y **Perl, PHP o Python** (lenguajes de scripting).
- Otros Stacks muy usados (con SGBD no relacionales – no válidos para las prácticas):
 - **MEAN**: **MongoDB** (SGBD), **Express.js** (Controlador), **AngularJS** (presentación), **Node.js** (javascript en servidor Web).
 - **MERN y MEVN**: como **MEAN** reemplazando **AngularJS** por **React** y **Vue.js**, respectivamente.

Más en https://en.wikipedia.org/wiki/Solution_stack

- **Seleccionar e instalar el software** necesario para la implementación (si usáis el mismo que en el Seminario 2, ya lo tenéis).
- **Escribir las sentencias de creación de tablas** con las claves y restricciones que correspondan, y crear las tablas en el SGBD.
- **Identificar las transacciones** que hay que crear y controlar en el SI.
- Implementar **al menos un disparador por subsistema**, a acordar con el profesorado. **Será obligatorio controlar mediante disparadores las dependencias y otras restricciones que no puedan controlarse por otros medios**, como el diseño lógico de la BD o mecanismos de control establecidos en la creación de tablas.
- **Realizar la implementación del SI** externa a la BD, incluyendo:
 - **Conexión al SGBD**, inicio y cierre de sesión.
 - Control de todas las **transacciones**.
 - **Interfaz de usuario**, incluyendo menú (puede ser tan simple como se quiera, al igual que en el Seminario 2) y entrada/salida de datos.
 - **Control de excepciones** del SGBD.

Un único documento .zip que contendrá:

- Un fichero .pdf que extienda (y en su caso corrija) al entregado en la Práctica 2. Deberá incluir adicionalmente:
 - Sentencias de creación de tablas, con claves y restricciones.
 - Descripción de las transacciones identificadas (secuencia de operaciones lógicas).
 - Código de los disparadores implementados en el SGBD.
 - Breve motivación de la elección de software (menos de ½ página).
- El código fuente del SI implementado.

La entrega y defensa podrá hacerse en cualquier momento previa cita acordada por correo electrónico, teniendo como fechas límite las especificadas para cada grupo.