A2. Guiones con estructuras de control de flujo.

Índice.

1. Variables de PL/SQL....2
   - 1.1. Variables....3
   - 1.2. Tipos de datos....6
   - 1.3. Declaración de variables....7
   - 1.4. Asignación de valores....8
   - 1.5. Comentarios....9
   - 1.6. Literales....10

2. Parámetros....11

3. Bloques de instrucciones....13

4. Instrucciones condicionales....16

5. Instrucciones repetitivas o bucles....19

6. Bucles anidados....23

7. Manejo de errores....24

A2. Guiones con estructuras de control de flujo.

1. Variables de PL/SQL.

A2. Guiones con estructuras de control de flujo.

1.1. Variables.

Los datos que maneja un programa se almacenan en variables y en constantes que han de ser de cualquier tipo de dato soportado por cada lenguaje de programación:

- **Variables** → elementos de datos dotados de nombre y cuyo valor puede ir cambiando a lo largo de la ejecución del programa.
- **Constantes** → elementos de datos dotados de nombre y cuyo valor permanece inalterado durante la ejecución del programa.

Las reglas para nombrar las variables son bastante flexibles, porque permiten:

- Nombres largos de más de 255 caracteres.
- Uso de caracteres especiales.
- Posibilidad de empezar con caracteres numéricos.

TODAS las variables que se pueden utilizar SERÁN escalares → van a tener un solo valor.

A2. Guiones con estructuras de control de flujo.

1.1. Variables.

Un tipo especial de variables son las de usuario, porque pueden ser manipuladas desde dentro y fuera de los programas almacenados.

Las variables de usuario representan una característica de MySQL desde la versión 3.

Las variables de usuario NO requieren declaración y van precedidas por el carácter **@**.

Son un tipo de datos variant que pueden almacenar texto, fechas y números.

Su alcance es de una sesión → accesibles desde cualquier programa que se ejecuta durante esa sesión.

A2. Guiones con estructuras de control de flujo.

1.1. Variables.

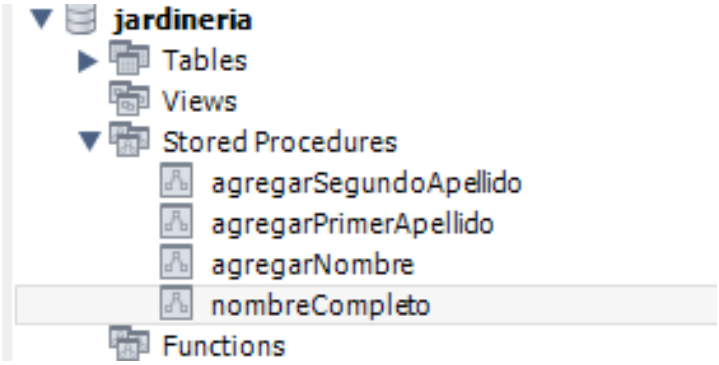
```
use Jardineria;

DELIMITER //
    DROP PROCEDURE IF EXISTS nombreCompleto //
    CREATE PROCEDURE nombreCompleto()
        BEGIN
            CALL agregarNombre();
            CALL agregarPrimerApellido();
            CALL agregarSegundoApellido();
        END
// DELIMITER ;

DELIMITER //
    DROP PROCEDURE IF EXISTS agregarNombre //
    CREATE PROCEDURE agregarNombre()
        BEGIN
            SET @nombre = 'Fulgencio';
        END
// DELIMITER ;

DELIMITER //
    DROP PROCEDURE IF EXISTS agregarPrimerApellido //
    CREATE PROCEDURE agregarPrimerApellido()
        BEGIN
            SET @nombre = CONCAT( @nombre, ' ', 'Guillermino' );
        END
// DELIMITER ;

DELIMITER //
    DROP PROCEDURE IF EXISTS agregarSegundoApellido //
    CREATE PROCEDURE agregarSegundoApellido()
        BEGIN
            SET @nombre = CONCAT( @nombre, ' ', 'Pancracio' );
        END
// DELIMITER ;
```




```
MySQL 8.0 Command Line Client
mysql> use Jardineria;
Database changed
mysql>
mysql> call nombreCompleto();
Query OK, 0 rows affected (0.00 sec)

mysql> select @nombre;
+---------------------------------+
| @nombre                         |
+---------------------------------+
| Fulgencio Guillermino Pancracio |
+---------------------------------+
1 row in set (0.00 sec)

mysql>
```

1.2. Tipos de datos.

La elección de un tipo de dato para una variable (o constante) condiciona su forma de almacenamiento, sus restricciones y su rango de valores válidos.

PL/SQL proporciona una variedad predefinida de tipos de datos que coinciden con los soportados por SQL:

TIPO DE DATO NUMÉRICO		
TINYINT	8 bits	256
SMALLINT	16 bits	65.535
MEDIUMINT	24 bits	16.777.215
INT, INTEGER	32 bits	2.147.483.647
BIGINT	64 bits	18.446.744.073.709.551.615
FLOAT	Real de precisión simple	3,4E38
DOUBLE	Real de precisión doble	1,7E308
DECIMAL(precisión, escala)		
NUMERIC(precisión, escala)		

TIPO DE DATO DE TEXTO	
CHAR(longitud)	longitud fija de hasta 255 caracteres
TINYTEXT	longitud variable de hasta 255 caracteres
TEXT	longitud variable de hasta 65.535 caracteres
MEDIUMTEXT	longitud variable de hasta 2.097.152 caracteres
LONGTEST	longitud variable de hasta 4 Gbits de caracteres
VARCHAR(longitud)	longitud variable 65.535 caracteres

TIPO DE DATO FECHA		
DATE	Fecha	AAAA-MM-DD
DATETIME	Fecha y hora	AAAA-MM-DD hh:mm:ss

TIPO DE DATO ENUMERACIÓN
ENUM(valor1, ..., valorN)
SET(valor)

TIPO DE DATO BINARIO	
BLOB	hasta 64 KB de datos binarios
LONGBLOB	hasta 4 GB de datos binarios

1.3. Declaración de variables.

La declaración de las variables se debe realizar antes del comienzo de las instrucciones.

- Se pueden declarar varias variables **del mismo tipo** seguidas y separadas por comas.
- Se pueden asignar un valor inicial mediante la cláusula DEFAULT.
- En la declaración se debe indicar el tipo de dato de la variable.

La declaración de las variables se puede realizar con la cláusula DECLARE:

```
DECLARE nombre_variable1 [,nombre_variable2...] tipo [DEFAULT valor];
```

```
DECLARE fecha DATE;
DECLARE Saldo DECIMAL(8,2) DEFAULT 0,00;
DECLARE Nombre VARCHAR2(15) DEFAULT '';
```

1.4. Asignación de valores a las variables.

La asignación de valores a las variables se puede realizar de las siguientes formas:

- En la propia declaración de la variable → valor por defecto.

```
DECLARE nombre_variable1 [,nombre_variable2...] tipo [DEFAULT valor];
```

```
DECLARE fecha DATE DEFAULT '2021-05-05';
DECLARE Saldo DECIMAL(8,2) DEFAULT 0,00;
DECLARE Nombre VARCHAR2(15) DEFAULT '';
```

- Con la sentencia SET → asigna un valor a la variable y puede unir varias asignaciones.

```
SET nombre_variable1 = expresión1 [,nombre_variable2 = expresión2 ...]
```

```
SET fecha = '2021-05-05';
SET Saldo = 0,00;
SET Nombre = '';
```

- A través de una operación o de una función.

```
SET fecha = fechaInicial + 40;
SET Saldo = Ingresos - Gastos;
SET Nombre = leerNombre();
```

- A partir de una consulta.

```
SET fecha = SELECT( fecha FROM Operaciones where ID = 200002;
SET Saldo = SELECT( sum(Importe) FROM Ingresos) - SELECT( sum(Importe) FROM Gastos);
SET Nombre = SELECT( nombre FROM Clientes where NIF = '12345678A');
```

1.5. Comentarios.

Los comentarios pueden ser de dos tipos:

- Comentarios de una sola línea → precedidos de –
- Comentarios de varias líneas → entre */ y /*

```
/* variable de control de la fecha */
DECLARE fecha DATE;
-- Dinero del cliente
DECLARE Saldo DECIMAL(8,2) DEFAULT 0,00;
/* Nombre del cliente */
DECLARE Nombre VARCHAR2(15) DEFAULT '';
```

1.6. Literales.

Los literales se utilizan en las comparaciones de valores o para asignar valores concretos a los identificadores, que actúan como variables o constantes.

Para expresar estos literales hay que tener en cuenta los siguiente:

- Los literales numéricos se expresan por medio de notación decimal o exponencial: 123, +123, 1.23E2, -123, -1.23E2
- Los literales de tipo carácter y de tipo cadena se deben delimitar por comillas simples: nombre = 'Fulgencio'
- Los literales lógicos son TRUE y FALSE.
- El literal NULL expresa que la variable NO tiene ningún valor asignado.

2. Parámetros.

Los parámetros son variables que se envían y reciben de los programas a los que se llama.

Los parámetros se definen en la cláusula CREATE de la creación de los procedimientos de la siguiente forma:

```
CREATE PROCEDURE([[IN |OUT |INOUT] nombre_parámetro tipo de datos...])
```

Los tipos de parámetros, según el modo en que se pasan al procedimiento invocado son:

IN → opción por defecto. En otros lenguajes representa el paso de parámetros por valor, es decir, el procedimiento almacenado trabaja con una copia del parámetro que recibe y, por tanto, NO modifica en NADA el valor del parámetro pasado.

OUT → es una forma de paso de parámetros por variable, es decir, las modificaciones que se realicen dentro del programa modifican DIRECTAMENTE el parámetro pasado como argumento. Hasta que no se le asigne un valor, será NULO. Se suele emplear como flag (bandera o indicador) de cómo ha ido la ejecución de un programa.

INOUT → otra forma de paso de parámetros por variable, pero con la característica de que se le puede pasar un valor inicial que el programa invocado puede tener en cuenta.

2. Parámetros.

Algunos ejemplos del uso de parámetros son los siguientes:

```
DELIMITER //
    DROP PROCEDURE IF EXISTS incrementarValorIN //
    CREATE PROCEDURE incrementarValorIN( IN valor INTEGER )
        BEGIN
            SET valor = valor + 1;
        END
// DELIMITER ;

SET @valor = 10;
CALL incrementarValorIN( @valor );
SELECT @valor;
```

	@valor
▶	10

```
DELIMITER //
    DROP PROCEDURE IF EXISTS incrementarValorINOUT //
    CREATE PROCEDURE incrementarValorINOUT( INOUT valor INTEGER )
        BEGIN
            SET valor = valor + 1;
        END
// DELIMITER ;

SET @valor = 10;
CALL incrementarValorINOUT( @valor );
SELECT @valor;
```

	@valor
▶	11

```
DELIMITER //
    DROP PROCEDURE IF EXISTS incrementarValorOUT //
    CREATE PROCEDURE incrementarValorOUT( OUT valor INTEGER )
        BEGIN
            SET valor = valor + 1;
        END
// DELIMITER ;

SET @valor = 10;
CALL incrementarValorOUT( @valor );
SELECT @valor;
```

	@valor
▶	NULL

3. Bloques de instrucciones.

Una **estructura de bloques** es la reunión de una serie de instrucciones en agrupaciones lógicas para realizar una determinada función (como son los bloques de los manejadores de errores), bajo el fin de delimitar el ámbito de las variables, declarándolas y definiéndolas dentro de un bloque interno para que no sean visibles fuera de él.

Una variable externa al bloque interno será accesible SIEMPRE desde dicho bloque interno.

Si la variable interna y la externa tuviesen el mismo nombre → se accederá a la variable INTERNA.

El caso más sencillo de bloques de instrucciones (bloque único) es el asociado a un procedimiento o función, cuyas instrucciones se hallan entre las sentencias BEGIN y END.

```
CREATE {PROCEDURE | FUNCTION |TRIGGER} nombre_del_programa
  BEGIN
        Instrucciones
  END;
```

3. Bloques de instrucciones.

Un **bloque de instrucciones** no sólo agrupa instrucciones SINO también otros elementos como:

- Declaraciones de variables y condiciones.
- Declaraciones de cursores.
- Declaraciones de manejadores de error.
- Código de programa.

A la hora de declararse debe seguirse el orden anterior para evitar mensajes de error.

Los bloques, además, pueden etiquetarse para facilitar la lectura del procedimiento (y su mantenimiento)

```
[etiqueta:] BEGIN
    Declaración de variables y condiciones.
    Declaración de cursores.
    Declaración de manejadores de error.
    Código de programa.
END [etiqueta];
```

3. Bloques de instrucciones.

```
DELIMITER $$

DROP PROCEDURE IF EXISTS bloque1 $$
CREATE PROCEDURE bloque1 ()
  BEGIN
    DECLARE v_externa VARCHAR(45) DEFAULT 'Variable visible en todo el procedimiento';
    BEGIN
       DECLARE v_interna VARCHAR(45);
       SET v_interna='Sólo soy accesible en las líneas 7 a 11';
       SELECT v_externa; -- Correcto
    END;
    SELECT v_interna; -- Error
  END $$
DELIMITER ;
```

```
DELIMITER $$

DROP PROCEDURE IF EXISTS bloque2 $$
CREATE PROCEDURE bloque2 ()
  BEGIN
    DECLARE v VARCHAR(65) DEFAULT 'Variable visible en todo el procedimiento';
        BEGIN
            SET v='Cambio del valor de la variable en el bloque interno';
        END;
    SELECT v; --'Cambio del valor de la variable en el bloque interno'
  END $$
DELIMITER ;
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS bloque3 $$
CREATE PROCEDURE bloque3 ()
  BEGIN
     DECLARE v INT DEFAULT 500;
     BEGIN
        DECLARE v INT;
            SET v = 200;
            SELECT  v; -- 200
        END;
     SELECT v;   -- 500
  END $$
DELIMITER ;
```

4. Instrucciones condicionales.

Las instrucciones condicionales pueden ser de los siguientes tipos:

- IF-THEN-ELSE

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list] ...
    [ELSE statement_list]
END IF
```

- CASE

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

```
CASE
    WHEN search_condition THEN statement_list
    [WHEN search_condition THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

4. Instrucciones condicionales.

Las instrucciones condicionales pueden ser de los siguientes tipos:

- IF-THEN-ELSE

```
DELIMITER //

CREATE FUNCTION SimpleCompare(n INT, m INT)
  RETURNS VARCHAR(20)

  BEGIN
    DECLARE s VARCHAR(20);

    IF n > m THEN SET s = '>';
    ELSEIF n = m THEN SET s = '=';
    ELSE SET s = '<';
    END IF;

    SET s = CONCAT(n, ' ', s, ' ', m);

    RETURN s;
  END //

DELIMITER ;
```

```
DELIMITER //

CREATE FUNCTION VerboseCompare (n INT, m INT)
  RETURNS VARCHAR(50)

  BEGIN
    DECLARE s VARCHAR(50);

    IF n = m THEN SET s = 'equals';
    ELSE
      IF n > m THEN SET s = 'greater';
      ELSE SET s = 'less';
      END IF;

      SET s = CONCAT('is ', s, ' than');
    END IF;

    SET s = CONCAT(n, ' ', s, ' ', m, '.');

    RETURN s;
  END //

DELIMITER ;
```

A2. Guiones con estructuras de control de flujo.

4. Instrucciones condicionales.

Las instrucciones condicionales pueden ser de los siguientes tipos:

- CASE

```
DELIMITER |

CREATE PROCEDURE p()
  BEGIN
    DECLARE v INT DEFAULT 1;

    CASE v
      WHEN 2 THEN SELECT v;
      WHEN 3 THEN SELECT 0;
      ELSE
        BEGIN
        END;
    END CASE;
  END;
  |
```

A2. Guiones con estructuras de control de flujo.

5. Instrucciones repetitivas o bucles.

Las estructuras repetitivas o bucles pueden ser de las siguientes formas:

- LOOP

```
[begin_label:] LOOP
    statement_list
END LOOP [end_label]
```

- REPEAT

```
[begin_label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [end_label]
```

- WHILE

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

5. Instrucciones repetitivas o bucles.

Las estructuras repetitivas o bucles pueden ser de las siguientes formas:

- LOOP

```
[begin_label:] LOOP
    statement_list
END LOOP [end_label]
```

```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 < 10 THEN
      ITERATE label1;
    END IF;
    LEAVE label1;
  END LOOP label1;
  SET @x = p1;
END;
```

5. Instrucciones repetitivas o bucles.

Las estructuras repetitivas o bucles pueden ser de las siguientes formas:

- REPEAT

```
[begin_label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [end_label]
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS ejemplo_bucle_repeat$$
CREATE PROCEDURE ejemplo_bucle_repeat(IN tope INT, OUT suma INT)
BEGIN
  DECLARE contador INT;

    SET contador = 1;
    SET suma = 0;

    REPEAT
    SET suma = suma + contador;
    SET contador = contador + 1;
  UNTIL contador > tope
  END REPEAT;
END
$$

DELIMITER ;
CALL ejemplo_bucle_repeat(10, @resultado);
SELECT @resultado;
```

5. Instrucciones repetitivas o bucles.

Las estructuras repetitivas o bucles pueden ser de las siguientes formas:

- WHILE

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS ejemplo_bucle_while$$
CREATE PROCEDURE ejemplo_bucle_while(IN tope INT, OUT suma INT)
BEGIN
  DECLARE contador INT;

  SET contador = 1;
  SET suma = 0;

  WHILE contador <= tope DO
    SET suma = suma + contador;
    SET contador = contador + 1;
  END WHILE;
END
$$

DELIMITER ;
CALL ejemplo_bucle_while(10, @resultado);
SELECT @resultado;
```

6. Bucles anidados.

Un **bucle anidado** consiste en la utilización comandos de repetición dentro de otros.

```
DELIMITER $$

DROP PROCEDURE IF EXISTS bucle5 $$
CREATE PROCEDURE bucle5 ()
BEGIN
   DECLARE i,j TINYINT UNSIGNED DEFAULT 1;

  bucle_externo: LOOP
     SET j=1;
     bucle_interno: LOOP
        SELECT  CONCAT("Valor de i y j:  ",i, "-", j) AS i_j;
        SET j=j+1;
        IF j>2 THEN
            LEAVE bucle_interno;
        END IF;
     END LOOP bucle_interno;
     SET i=i+1;
     IF i>2 THEN
        LEAVE bucle_externo;
     END IF;
  END LOOP bucle_externo;

END $$

DELIMITER ;
```

Resulta importante etiquetar el comienzo y final del bucle no sólo con la instrucción LEAVE o ITERATE, sino por claridad en el código.

7. Manejo de errores.

El manejo de errores se realiza a través del bloque DECLARE … HANDLER de la siguiente forma:

```
DECLARE handler_action HANDLER
    FOR condition_value [, condition_value] ...
    statement

handler_action:
    CONTINUE
  | EXIT
  | UNDO

condition_value:
    mysql_error_code
  | SQLSTATE [VALUE] sqlstate_value
  | condition_name
  | SQLWARNING
  | NOT FOUND
  | SQLEXCEPTION
```

Las acciones posibles que se pueden seleccionar como *handler_action* son:

- CONTINUE → sigue la ejecución del programa.
- EXIT → finaliza la ejecución del programa.
- UNDO → NO está soportado en MySQL.

7. Manejo de errores.

```
DECLARE CONTINUE HANDLER FOR 1051
BEGIN
  -- body of handler
END;
```

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'
BEGIN
  -- body of handler
END;
```

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
  -- body of handler
END;
```

```
DECLARE CONTINUE HANDLER FOR SQLWARNING
BEGIN
  -- body of handler
END;
```

```
DECLARE CONTINUE HANDLER FOR NOT FOUND
BEGIN
  -- body of handler
END;
```

```
DELIMITER $$
CREATE PROCEDURE handlerdemo ()
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x = 1;
  SET @x = 1;
  INSERT INTO test.t VALUES (1);
  SET @x = 2;
  INSERT INTO test.t VALUES (1);
  SET @x = 3;
END
$$
```

```
DELIMITER $$
CREATE PROCEDURE handlerdemo ()
BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE '23000' SET @x = 1;
  SET @x = 1;
  INSERT INTO test.t VALUES (1);
  SET @x = 2;
  INSERT INTO test.t VALUES (1);
  SET @x = 3;
END
$$
```