

# BD

## Bases de Datos

### UD 7

#### Scripts con T-SQL

### ÍNDICE

1. Introducción a la programación de bases de datos
2. Elementos del lenguaje
  - 2.1 Tipos de datos
  - 2.2 Variables
  - 2.3 Operadores
3. Estructuras de control
4. Control de errores y transacciones

## 1. Introducción

En la actualidad, los sistemas gestores de bases de datos relacionales han incorporado mecanismos para desarrollar programas dentro de la propia base de datos.

Destacamos entre los más importantes en la actualidad los siguientes SGBD junto con los lenguajes de programación de bases de datos que utilizan:

SGBD	Lenguaje	Uso habitual
SQL Server	<i>Transact-SQL</i>	Propósito general y de pago (grandes empresas)
Oracle	<i>PL/SQL</i>	Propósito general y de pago (grandes empresas)
MariaDB (MySQL)	<i>PL-SQL (diferente y menos potente que el de Oracle)</i>	MariaDB libre y gratuito. Enfocado a portales web.
PostgreSQL	<i>PL/pgSQL</i>	Propósito general y libre/grat.

Cada vez más, las empresas están apostando por incluir una parte importante de su lógica de negocio dentro de la propia base de datos, la ventaja para tomar dicha decisión es muy clara: *“todo lo que se ejecute dentro de la base de datos será siempre más rápido que si lo extraemos a otro programa, lo ejecutamos con otro lenguaje de programación y se lo devolvemos a la base de datos para que lo procese”*.

Por ello, para este curso se ha optado por estudiar las estructuras de programación que ofrece SQL Server, las cuales tienen un paralelismo con Oracle y su PL/SQL (el cambio de un SGBD a otro es como cambiar de Java a C# o viceversa, lleva algo de tiempo, pero la curva de aprendizaje no es demasiado alta).

Conviene recordar que el lenguaje estándar SQL se divide en tres partes diferenciadas:

Lenguaje	Características
<b>DCL</b> o Data Control Language	<i>Control de permisos sobre tablas con sentencias como GRANT y REVOKE</i>
<b>DDL</b> o Data Definition Language	<i>Creación de la estructura de tablas</i>
<b>DML</b> o Data Manipulation Language	<i>Manipulación de datos con las sentencias INSERT, DELETE, UPDATE y la posterior consulta con SELECT</i>

Estos tres lenguajes, al ser estándares de SQL están implementados en todos los SGBD por lo que son comunes a todos ellos. La parte que cambia es la que cada SGBD ha implementado cómo se programa una sentencia de decisión IF-ELSE, un bucle FOR, WHILE, DO-WHILE, los tipos de datos de cada sistema, cómo se declaran las variables, etc.

## 2. Elementos del lenguaje

A continuación, conoceremos los principales elementos del lenguaje Transact-SQL.

### 2.1 Variables

Una variable es un elemento que contiene un valor de un tipo de dato específico. Este valor puede cambiar durante el proceso o programa en el que se utiliza la variable. SQL Server tiene dos tipos de variables: locales y globales.

Las variables locales las define el/la programador/a, mientras que las variables globales las suministra el sistema y están predefinidas. Son ejemplos de variables:

- Contadores: sirven para contar el número de veces que se realiza un bucle o controlar cuántas veces debe ejecutarse.
- Variables del mismo tipo que una columna de una tabla: se utilizan para almacenar el valor de esa columna para un registro concreto o bien para actualizar el valor de un registro de una tabla con ese dato.

#### Variables locales

Las variables locales deben ser declaradas por el/la programador/a antes de poder ser utilizadas y para ello utilizamos la sentencia **DECLARE**. Una vez declaradas podremos utilizarla y asignar un valor a la misma. El valor puede ser predefinido (introduciéndolo directamente nosotros mismos) o puede obtenerse a partir de la instrucción **SELECT** o **SET**. A continuación, tienes un ejemplo de declaración y uso de variables locales:

```
--DECLARA UNA VARIABLE
DECLARE @nomVar1 TIPO_DATO1;
DECLARE @nomVar2 TIPO_DATO2 = VALOR2; --Podemos asignar el valor

-- ASIGNA VALOR A UNA VARIABLE
SET @nombrevariable1 = VALOR1;
```

**Consejo:** Es una buena práctica declarar todas las variables en la parte de arriba del código, salvo que sea una variable cuyo uso esté muy localizado más abajo.

#### ***Ejemplo 1 – Declaración, asignación y uso de una variable en una SELECT***

Crea una variable llamada “varPrecio” a la que le asignes el valor 75 y la utilices para obtener los productos que tengan un precio mayor o igual a esta variable.

```
USE JARDINERIA;

DECLARE @varPrecio DECIMAL(9,2) = 75;

SELECT *
FROM PRODUCTOS
WHERE precio_venta >= @varPrecio;
```

### Ejemplo 2 – Declaración, asignación y uso de una variable en una SELECT

Crea una variable llamada “gama” y asígnale el valor Frutales. Crea otra variable llamada “nombre” y asígnale el valor “Pera”. Debes utilizar una consulta que muestre los productos de la gama de la variable que has creado y que el nombre de los productos empiece por la variable “nombre”.

**NOTA:** Los tipos de datos de las variables deben ser iguales que las columnas de las tablas

```
USE JARDINERIA;
```

```
DECLARE @gama VARCHAR(50) = 'Frutales', @nombre VARCHAR(70) = 'Pera'
```

```
SELECT *  
FROM PRODUCTOS  
WHERE gama = @gama  
AND nombre LIKE CONCAT(@nombre, '%');
```

**Truco:** Para evitar tener que acudir a la vista diseño de cada tabla podemos ejecutar el siguiente comando: `EXEC sp_help NOMBRETABLA`

De esta forma obtendremos el diseño de la tabla de un modo más rápido:

Results Messages

	Name	Owner	Type	Created_datetime						
1	PRODUCTOS	dbo	user table	2023-01-22 11:59:30.343						
	Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInSource	Collation
1	codProducto	varchar	no	15			no	no	no	SQL_Latin1_General_CP1_CI_AS
2	nombre	varchar	no	70			no	no	no	SQL_Latin1_General_CP1_CI_AS
3	gama	varchar	no	50			no	no	no	SQL_Latin1_General_CP1_CI_AS
4	dimensiones	varchar	no	25			yes	no	yes	SQL_Latin1_General_CP1_CI_AS
5	proveedor	varchar	no	50			yes	no	yes	SQL_Latin1_General_CP1_CI_AS
6	descripcion	varchar	no	-1			yes	no	yes	SQL_Latin1_General_CP1_CI_AS
7	cantidad_en_stock	smallint	no	2	5	0	no	(n/a)	(n/a)	NULL
8	precio_venta	numeric	no	9	15	2	no	(n/a)	(n/a)	NULL
Identity			Seed	Increment	Not For Replication					
1	No identity column defined.		NULL	NULL	NULL					

### Ejemplo 3 – Declaración y asignación de una variable a partir de una SELECT

Crea una variable llamada minPrecio que obtenga el precio más pequeño de todos los productos de la gama Frutales.

```
USE JARDINERIA;
```

```
DECLARE @gama VARCHAR(50) = 'Frutales', @minPrecio DECIMAL(9,2)
```

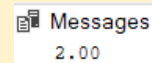
```
SELECT @minPrecio = MIN(precio_venta)  
FROM PRODUCTOS  
WHERE gama = @gama;
```

Si ejecutamos el código veremos... **¡Nada en absoluto! ¿Por qué?**

Porque simplemente estamos almacenando el precio mínimo de venta en la variable, pero sin mostrarla. Si queremos mostrar el valor, podemos utilizar la instrucción PRINT del siguiente modo: `PRINT @minPrecio`

**Nota:** Si solo ejecutamos la instrucción PRINT @minPrecio no se mostrará nada en absoluto. Recuerda debes seleccionar y ejecutar TODO el código al completo.

Si lo hemos hecho correctamente, mostrará el valor: 2.00



Messages  
2.00

#### **Ejemplo 4 – Declaración y asignación de dos o más variables a partir de una SELECT**

Crea una variable llamada minPrecio y otra maxPrecio que obtenga el precio más pequeño y más grande de todos los productos de la gama Frutales. Muestra el resultado utilizando PRINT

```
USE JARDINERIA;
```

```
DECLARE @gama VARCHAR(50) = 'Frutales'
```

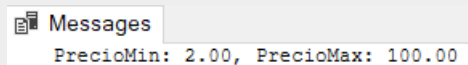
```
DECLARE @minPrecio DECIMAL(9,2), @maxPrecio DECIMAL(9,2)
```

```
SELECT @minPrecio = MIN(precio_venta),  
       @maxPrecio = MAX(precio_venta)
```

```
FROM PRODUCTOS
```

```
WHERE gama = @gama;
```

```
PRINT CONCAT('PrecioMin: ', @minPrecio, ', PrecioMax: ', @maxPrecio);
```



Messages  
PrecioMin: 2.00, PrecioMax: 100.00

Recuerda que podemos utilizar el resultado de una SELECT para almacenar un ÚNICO VALOR devuelto por la SELECT. No obstante, si ejecutamos el siguiente código:

#### **Ejemplo 5 – Asignación a partir de una SELECT que devuelva más de un valor**

Crea una variable “precio” el precio de todos los productos de la gama Frutales. Muestra el resultado utilizando PRINT

```
DECLARE @gama VARCHAR(50) = 'Frutales'
```

```
DECLARE @precio DECIMAL(9,2)
```

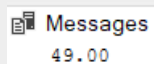
```
SELECT @precio = precio_venta
```

```
FROM PRODUCTOS
```

```
WHERE gama = @gama;
```

```
PRINT @precio;
```

El código devolverá:



Messages  
49.00

**¿Es correcto el resultado?** Evidentemente NO. Lo que ha ocurrido es que ha obtenido el primer valor que devuelve la SELECT y lo ha asignado a la variable, lo cual **no es correcto y depende totalmente del azar**.

Por lo tanto, debemos tener precaución cuando la consulta devuelva más de un valor y controlar esta situación.

Si una consulta SELECT devuelve un único valor podemos utilizar también SET para obtener el valor, pero solo cuando devuelva un valor (el ejemplo de maxPrecio y minPrecio no se podría implementar salvo que hiciéramos dos consultas diferentes).

### ***Ejemplo 6 – Asignación con SET del resultado de una SELECT***

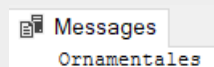
Crea una variable “gama” que devuelva el nombre de la gama que más productos tiene.

```
USE JARDINERIA
```

```
DECLARE @gama VARCHAR(50)
```

```
SET @gama = (SELECT TOP(1) gama
             FROM PRODUCTOS
             GROUP BY gama
             ORDER BY COUNT(1) DESC);
```

```
PRINT @gama
```



Messages  
Ornamentales

El código devolverá “Ornamentales”.

Es importante darse cuenta que es necesario incluir los paréntesis entre el = y el ; de la instrucción SET.

## **2.2 Tipos de datos**

En SQL Server, igual que cada columna tiene un tipo de datos relacionado también podemos definir tipos de datos a variables que puedan contener valores de estas columnas. Pueden ser de diferentes tipos, como, por ejemplo: números enteros, números decimales, cadenas de caracteres, importes, fechas, fecha/hora, etc.

Recordemos los tipos de datos más utilizados:

<b>ENTEROS</b>	
BIT	Acepta los valores 0, 1 o NULL
TINYINT	Acepta los valores del 0 al 255 (no permite valores negativos)
SMALLINT	Acepta los valores del -32.768 al 32.767 (permite negativos)
INT	Acepta los valores del -2.147.483.648 a 2.147.483.647
<b>DECIMALES Y MONETARIOS</b> <i>(todos incluyen valores negativos)</i>	
DECIMAL	Precisión hasta 9 dígitos (con decimales)  <b>Ejemplo.</b> DECIMAL (5,2): 5 dígitos de los cuales 2 son decimales. 123,00
NUMERIC	Precisión hasta 19 dígitos (con decimales y mayor precisión).  <b>Ejemplo.</b> Numeric (10,5): 10 dígitos de los cuales 5 son decimales. 12345,12000

## Cadenas de caracteres

Almacenamiento de nombres, apellidos, direcciones, observaciones, etc. En definitiva, campo que contenga texto, números o ambos con los que no tengamos intención de operar.

<i>Tipo de cadena</i>	<i>ASCII</i>	<i>Unicode</i>
Longitud FIJA	CHAR	NCHAR
Longitud VARIABLE	VARCHAR	NVARCHAR

Para ilustrar los tipos de datos de una forma práctica, utilizaremos el siguiente código:

### Ejemplo 1 – ***CHAR vs NCHAR vs VARCHAR vs NVARCHAR***

Visualizar el tamaño de una cadena de longitud fija/variable.

```
DECLARE @cadena CHAR(7) = 'hola'
```

```
SELECT @cadena as texto,  
       LEN(@cadena) as longitud,  
       DATALENGTH(@cadena) as espacioMemoria;
```

texto	longitud	espacioMemoria
hola	4	7

--

```
DECLARE @cadena NCHAR(7) = 'hola'
```

```
SELECT @cadena as texto,  
       LEN(@cadena) as longitud,  
       DATALENGTH(@cadena) as espacioMemoria;
```

texto	longitud	espacioMemoria
hola	4	14

--

```
DECLARE @cadena VARCHAR(7) = 'hola'
```

```
SELECT @cadena as texto,  
       LEN(@cadena) as longitud,  
       DATALENGTH(@cadena) as espacioMemoria;
```

texto	longitud	espacioMemoria
hola	4	4

--

```
DECLARE @cadena NVARCHAR(7) = 'hola'
```

```
SELECT @cadena as texto,  
       LEN(@cadena) as longitud,  
       DATALENGTH(@cadena) as espacioMemoria;
```

texto	longitud	espacioMemoria
hola	4	8

## Fecha/hora

<b>DATE</b>	Campo sólo fecha (si sólo necesitamos guardar la fecha)
<b>DATETIME</b>	Campo fecha/hora con precisión de segundos con decimales
<b>SMALLDATETIME</b>	Campo fecha/hora con precisión de segundos SIN decimales



## 2.3 Operadores

Un operador es un símbolo que especifica una acción que se realiza en una o más expresiones. A continuación, estudiaremos cada tipo de operador:

### Operadores aritméticos

Operador	Función
+	Suma
+=	Suma y asignación. Ejemplo. @id += 1;
-	Resta
-=	Resta y asignación. Ejemplo. @id -= 1;
*	Producto
*=	Producto y asignación. Ejemplo. @id *= 2;
/	División.
/=	División y asignación. Ejemplo. @id /= 3;
%	Resto de la división entera.

### Operadores lógicos

A continuación, veremos los principales operadores lógicos:

Operador	Función
AND	TRUE si las expresiones booleanas son TRUE.
OR	TRUE si cualquiera de las expresiones booleanas es TRUE.
BETWEEN	TRUE si el operando está dentro de un intervalo.
IN	TRUE si el operando es igual a uno de la lista de expresiones.
LIKE	TRUE si el operando coincide con un patrón.
NOT	Invierte el valor de cualquier otro operador booleano.
=	Igual a
<	Menor que
>	Mayor que
<=	Menor o igual
>=	Mayor o igual
<>	Distinto
IS NULL	Es nulo
IS NOT NULL	No es nulo

## Prioridad de los operadores

Nivel	Operador
1	* (multiplicación), / (división), % (módulo)
2	+ (positivo), - (negativo), + (suma), - (resta)
3	=, >, <, >=, <=, <> (operadores de comparación)
4	NOT
5	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
6	Asignación

### *Ejemplo – Prioridad de operadores*

*¿En qué orden se ejecutan las siguientes instrucciones?*

```
DECLARE @variable VARCHAR(100)

SET @variable = (SELECT TOP(1) nombre
                 FROM PRODUCTOS
                 WHERE precio_venta NOT BETWEEN 2*3+2 AND (10-3)*5
                 ORDER BY precio_venta ASC)

PRINT @variable
```

### 3. Estructuras de control

Las estructuras de control son los elementos que permiten realizar bucles, toma de decisiones en base a ciertas condiciones, imprimir por pantalla la salida de una instrucción o variable, etc. A continuación, se mostrará un resumen de las instrucciones más utilizadas a la hora de programar utilizando el lenguaje Transact-SQL:

PALABRA RESERVADA	UTILIZACIÓN
<b>/* SOY UN COMENTARIO MULTILINEA */</b> <b>-- COMENTARIO LINEA</b>	Inserta un comentario que sirve para <u>documentar</u> aquellas partes del código que lo requieran, por su complejidad o por otros motivos. Son imprescindibles para garantizar la calidad del código fuente.
<b>IF ... ELSE</b>	Permite controlar la ejecución de una instrucción o grupo de instrucciones en función del valor de una condición.
<b>WHILE</b>	Estructura repetitiva que ejecuta un bloque de instrucciones mientras la condición sea cierta.
<b>BEGIN ... END</b>	Reciben el nombre de <b>bloques</b> y permiten definir un bloque de instrucciones. Útiles el control de excepciones e <u>imprescindibles cuando cualquier sentencia dentro de un IF o ELSE ocupe más de una línea.</u>
<b>BREAK</b>	Provoca la salida del bucle WHILE incondicionalmente.
<b>CONTINUE</b>	Vuelve a ejecutar la condición del bucle WHILE (se utiliza si no queremos ejecutar las instrucciones dentro del WHILE y para reevaluar la/s condición/es).
<b>RETURN n</b>	Sale de forma incondicional de un bucle. Suele utilizarse en procedimientos, funciones o disparadores. <b>Consejo:</b> Se considera una buena práctica <u>definir un numero entero como estado devuelto</u> , que puede asignarse al ejecutar el procedimiento almacenado. De esta manera es más sencillo identificar qué parte del programa ha fallado y se facilita su depuración.
<b>PRINT</b>	Imprime un mensaje o variables definidas por pantalla. Utilizar CONCAT para formatear el mensaje de salida.
<b>CASE</b>	Funciona como un “conversor”. Se utiliza generalmente para “traducir” una clave a otro valor, bien asignándolo a una variable o bien mostrándolo directamente en una SELECT. <b>NO se utiliza como un switch, puesto que NO permite la ejecución de instrucciones.</b>

### 3.1 IF...ELSE

La instrucción IF se utiliza para definir una condición que determinará si se ejecutarán las instrucciones contenidas dentro de la sentencia **IF** o, si por el contrario, se ejecutarán las del **ELSE**. Es posible anidar otro IF dentro de un IF o de un ELSE (*la única regla es que para poner un ELSE, deberá haber antes definido un IF*).

#### Ejemplo 1 – Estructura IF-ELSE

Implementar un script que dado un idCliente muestre el número de pagos que tenga. En función del número de pagos que haya realizado deberá mostrar:

- Si ha realizado al menos un pago: “El cliente ha realizado N pagos”, siendo N el n° pagos
- Si no ha realizado ningún pago: “El cliente NO ha realizado ningún pago”.

```
USE JARDINERIA
```

```
DECLARE @codCliente INT, @numPagos INT
```

```
-- Obtenemos UN cliente al azar
```

```
SET @codCliente = (SELECT TOP(1) codCliente  
                  FROM CLIENTES  
                  ORDER BY NEWID())
```

```
-- Obtenemos el número de pagos realizados por el cliente
```

```
SET @numPagos = (SELECT COUNT(1)  
                FROM PAGOS  
                WHERE codCliente = @codCliente)
```

```
-- Evaluamos el resultado
```

```
IF @numPagos > 0
```

```
BEGIN  
    PRINT CONCAT('El cliente ', @codCliente, ' ha realizado ', @numPagos, ' pago/s.')
```

```
END
```

```
ELSE
```

```
BEGIN
```

```
    PRINT CONCAT('El cliente ', @codCliente, ' NO ha realizado ningún pago.')
```

```
END
```

Ejemplos de salida.

El cliente 21 NO ha realizado ningún pago.

El cliente 4 ha realizado 5 pago/s.

El cliente 26 ha realizado 1 pago/s.

El cliente 8 NO ha realizado ningún pago.

...

## Ejemplo 2 – Estructura IF-ELSE anidado

Implementar un script que dado un idCliente muestre el número de pagos que tenga. En función del número de pagos que haya realizado deberá mostrar:

- Si ha realizado al menos un pago: “El cliente ha realizado N pagos”, siendo N el n° pagos
- Si no ha realizado ningún pago: Se deberá obtener el número de pedidos que ha realizado.
  - o Si al menos ha realizado un pedido: “El cliente ha realizado N pedidos”
  - o Si no ha realizado ningún pedido: “El cliente no ha realizado ningún pedido”

USE JARDINERIA

DECLARE @codCliente INT, @numPagos INT

-- Obtenemos UN cliente al azar

```
SET @codCliente = (SELECT TOP(1) codCliente
                  FROM CLIENTES
                  ORDER BY NEWID())
```

-- Obtenemos el número de pagos realizados por el cliente

```
SET @numPagos = (SELECT COUNT(1)
                FROM PAGOS
                WHERE codCliente = @codCliente)
```

-- Evaluamos el resultado

IF @numPagos > 0

BEGIN

```
    PRINT CONCAT('El cliente ', @codCliente, ' ha realizado ', @numPagos, ' pago/s.')
```

END

ELSE

BEGIN

-- Obtenemos el número de pedidos realizados por el cliente

DECLARE @numPedidos INT

```
SET @numPedidos = (SELECT COUNT(1)
                  FROM PEDIDOS
                  WHERE codCliente = @codCliente)
```

IF @numPedidos > 0

BEGIN

```
    PRINT CONCAT('El cliente ', @codCliente, ' ha realizado ', @numPedidos,
                  ' pedido/s.')
```

END

ELSE

BEGIN

```
    PRINT CONCAT('El cliente ', @codCliente, ' NO ha realizado ningún pedido.')
```

END

END

Ejemplos de salida.

El cliente 21 NO ha realizado ningún pedido.

El cliente 4 ha realizado 5 pago/s.

El cliente 36 ha realizado 5 pedido/s.

El cliente 8 NO ha realizado ningún pedido.

...

### 3.2 ITERACIONES (bucles)

Las instrucciones de bucle son muy interesantes cuando necesitamos repetir dos o más veces una misma instrucción o conjunto de instrucciones y deseamos mantener un código limpio y eficiente (*no es admisible copiar y pegar el mismo código n veces*).

En SQL Server solo contamos con la instrucción WHILE para ejecutar bucles, por lo que no contamos con el famoso FOR-LOOP de Oracle ni con el DO-WHILE. No obstante, *utilizando la instrucción WHILE podemos “emular” el comportamiento de los otros dos bucles.*

La sintaxis para realizar bucles es la siguiente:

```
WHILE (condición/es)
BEGIN
    Instrucción 1
    Instrucción 2
    Instrucción 3
    Instrucción 4
    ...
END;
```

Como se ha visto anteriormente, contamos con las instrucciones BREAK y CONTINUE. La primera realiza la salida forzada del bucle (saltaría a la instrucción posterior al END) y la segunda fuerza la ejecución del bucle desde el principio.

#### Ejemplos de uso.

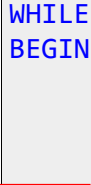
##### **Bucles: Uso de CONTINUE**

En el siguiente ejemplo, cuando termine de ejecutarse la instrucción 3 se volverá a pasar a evaluar la condición del bucle y si corresponde que vuelva a entrar en el bucle, se ejecutará de nuevo la 1, 2 y 3 y volverá a saltar al principio mientras se cumpla la condición del bucle.

NOTA: Normalmente, el CONTINUE va dentro de una instrucción IF que corresponde a un caso del que no deseamos que se ejecute el código que venga después (para saltar iteraciones).

```
USE JARDINERIA

WHILE (condición/es)
BEGIN
    Instrucción 1
    Instrucción 2
    Instrucción 3
    CONTINUE
    Instrucción 4
    ...
END
```



## **Bucles: Uso de BREAK**

No obstante, en este ejemplo, cuando se ejecute la **instrucción 3** se saltará al final del bucle (BREAK) y por lo tanto solo habrá ejecutado las sentencias 1, 2 y 3 una sola vez.

```
USE JARDINERIA
```

```
WHILE (condición/es)
```

```
BEGIN
```

```
    Instrucción 1
```

```
    Instrucción 2
```

```
    Instrucción 3
```

```
    BREAK
```

```
    Instrucción 4
```

```
    ...
```

```
END
```



A continuación, se verán diferentes ejemplos de uso de bucles.

### **Ejemplo 1 – Bucle (simulación FOR-LOOP)**

Implementar un bucle que cuente desde el número 1 al 30 y muestre los números divisibles entre 5.

```
DECLARE @contador INT = 1
```

```
DECLARE @valorMaximo INT = 30
```

```
WHILE @contador <= @valorMaximo
```

```
BEGIN
```

```
    -- ¿El número es divisible entre 5?
```

```
    IF @contador % 5 = 0
```

```
    BEGIN
```

```
        -- Mostramos su valor
```

```
        PRINT @contador
```

```
    END
```

```
    -- Incrementamos el contador en una unidad
```

```
    -- (si no se indica, se producirá un bucle infinito)
```

```
    SET @contador = @contador + 1
```

```
END
```

### **Ejemplo 2 – Bucle (simulación FOR-LOOP) con CONTINUE/BREAK**

Bucle que cuente desde el número 1 al 200 y muestre el primer número divisible entre 7, pero sin contar el propio 7.

```
DECLARE @contador INT = 1
```

```
DECLARE @valorMaximo INT = 200
```

```
WHILE @contador <= @valorMaximo
```

```
BEGIN
```

```
    -- Si el número es divisible entre 7
```

```
    IF @contador % 7 = 0
```

```

BEGIN
    -- Si el número es el 7, lo saltamos
    IF @contador = 7
    BEGIN
        SET @contador += 1 -- Si no incrementamos... ¡bucle infinito!
        CONTINUE
    END

    ELSE
    BEGIN
        -- Mostramos el número
        PRINT @contador
        BREAK
    END

    END

    -- Incrementamos el contador en una unidad
    SET @contador = @contador + 1
END

```

### ***Ejemplo 3 – Bucle (iterar por los registros de una tabla)***

*Muestra los nombres de todos los clientes que tengamos almacenados.*

*NOTA: Esta técnica es útil si la PK de la tabla es de tipo INT y se incrementa de modo 1, 2, 3, etc.*

```
USE JARDINERIA
```

```

DECLARE @id INT = 1, @maximo INT, @nombre VARCHAR(50)

-- Obtenemos el valor máximo (no confundir con COUNT!)
SET @maximo = (SELECT MAX(codCliente)
               FROM CLIENTES)

WHILE @id <= @maximo
BEGIN
    SET @nombre = (SELECT nombre_cliente
                  FROM CLIENTES
                  WHERE codCliente = @id)

    PRINT @nombre

    -- Incrementamos el siguiente id de la tabla CLIENTES
    SET @id = @id + 1
END

```



## 4. Control de errores y transacciones

Es un hecho. Cualquier código que escribamos, no importa todo el tiempo que estemos revisándolo y probándolo, es susceptible de fallar. Por ello, todos los lenguajes de programación proporcionan control de errores para gestionar este tipo de excepciones y definir cómo debe comportarse el código ante un error.

En Transact-SQL, la gestión de errores se realiza a través de las instrucciones TRY y CATCH. Veamos cómo se utilizan:

```
BEGIN TRY
    -- Código que puede fallar
    Sentencia 1
    Sentencia 2
    Sentencia 3
    ...
END TRY

BEGIN CATCH
    -- Código que se ejecutará si se produce error en alguna sentencia
    incluida dentro del bloque TRY
    Sentencia catch 1
    Sentencia catch 2
    ...
END CATCH
```

### ***Ejemplo 1 – Gestión de errores (división entre cero)***

*Realiza un script que compruebe que no se pueda dividir un número entero entre 0.*


```
DECLARE @dividendo INT, @divisor INT, @resultado INT

-- Prefijamos el dividendo a 50
SET @dividendo = 50

-- El divisor es un número aleatorio entre 0 y 9
SET @divisor = FLOOR(RAND()*10)

BEGIN TRY
    -- Si el número aleatorio es 0, daría error
    SET @resultado = @dividendo / @divisor
    PRINT @resultado
END TRY

BEGIN CATCH
    -- Si el número generado es 0, devuelve este mensaje
    PRINT 'No se puede dividir entre 0'
END CATCH
```

A red arrow originates from the line 'SET @resultado = @dividendo / @divisor' in the TRY block and points down to the CATCH block, indicating the execution path when an error occurs.

Normalmente, no tenemos claro cuál va a ser el error de un bloque de código como en el ejemplo anterior, por lo que necesitamos conocer cuál es la incidencia que se ha producido. Para ello, contamos con tres funciones:

ERROR	DESCRIPCIÓN
<code>ERROR_NUMBER()</code>	Número de error (estándar de SQL Server)
<code>ERROR_MESSAGE()</code>	Mensaje descriptivo del error
<code>ERROR_LINE()</code>	Número de línea en la que se ha producido el error
<code>ERROR_PROCEDURE()</code>	Nombre del procedimiento que ha causado el error (si se ejecuta desde un script devolverá siempre <i>NULL</i> )

### Ejemplo 2 – Gestión de errores mostrando la excepción por pantalla

Realiza un script que compruebe que no se pueda dividir un número entero entre 0.  
Muestra por pantalla todos los datos del error.

```
DECLARE @dividendo INT, @divisor INT, @resultado INT
```

```
-- Prefijamos el dividendo a 50
```

```
SET @dividendo = 50
```

```
SET @divisor = 0
```

```
BEGIN TRY
```

```
    SET @resultado = @dividendo / @divisor
```

```
    PRINT @resultado
```

```
END TRY
```

```
BEGIN CATCH
```

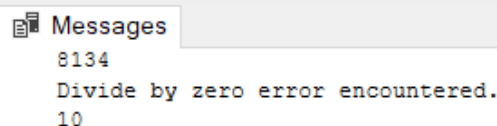
```
    PRINT ERROR_NUMBER()
```

```
    PRINT ERROR_MESSAGE()
```

```
    PRINT ERROR_LINE()
```

```
    PRINT ERROR_PROCEDURE()
```

```
END CATCH
```



Messages  
8134  
Divide by zero error encountered.  
10

Para que el mensaje quede un poco mejor organizado podemos utilizar la función `CONCAT` del siguiente modo:

```
...  
BEGIN CATCH  
    PRINT CONCAT ('CODERROR: ', ERROR_NUMBER(),  
                  ', DESCRIPCION: ', ERROR_MESSAGE(),  
                  ', LINEA: ', ERROR_LINE());  
END CATCH
```

Si ejecutamos el código anterior con esta modificación tendremos:

```
CODERROR: 8134, DESCRIPCION: Divide by zero error encountered., LINEA: 10
```

Alternativamente, y como mejora, además de mostrar el mensaje de error, podemos utilizar un mecanismo para **almacenar en una tabla los errores** que se produzcan en los programas de la empresa.

### Ejemplo 3 – Almacenar en una tabla de errores las excepciones que se produzcan

Creamos una tabla para almacenar la información de los errores que se produzcan.

```
--Creación de la tabla
CREATE TABLE SYS_ERRORES_PROGRAMAS (
    idError          INT IDENTITY (1,1),
    codError         INT,
    descError        VARCHAR(MAX),
    linea           INT,
    procedimiento    VARCHAR(MAX),
    fechaHora        SMALLDATETIME
    CONSTRAINT PK_SYS_ERRORES_PROGRAMAS PRIMARY KEY (idError)
);

-----
-- SCRIPT ANTERIOR ADAPTADO
-----
DECLARE @dividendo INT, @divisor INT, @resultado INT

-- Prefijamos el dividendo a 50
SET @dividendo = 50
SET @divisor = 0

BEGIN TRY
    SET @resultado = @dividendo / @divisor
    PRINT @resultado
END TRY

BEGIN CATCH
    INSERT INTO SYS_ERRORES_PROGRAMAS
        VALUES (ERROR_NUMBER(), ERROR_MESSAGE(), ERROR_LINE(),
                ERROR_PROCEDURE(), GETDATE());
END CATCH
```

Lo ejecutamos y comprobamos que tenemos el registro del error insertado en la tabla:

idError	codError	descError	linea	procedimiento	fechaHora
1	8134	Divide by zero error encountered.	10	NULL	2023-01-30 19:35:00

**Consejo:** En lugar de ejecutar la misma sentencia INSERT INTO en cada bloque de código de nuestros programas, la solución correcta pasa por **implementar un procedimiento almacenado** que ejecute ese código. Imagina que la estructura de la tabla cambia... tendríamos que modificar todas y cada una de las sentencias INSERT INTO que hubiera en nuestro código, mientras que del otro modo tan solo tendríamos que modificar el procedimiento.

## TRANSACCIONES

Una transacción es un conjunto de operaciones que se ejecutan como un único bloque, es decir, si falla una operación T-SQL, fallan todas.

Por tanto, si una transacción finaliza correctamente, los cambios realizados se confirman (**COMMIT**) permanentemente en la base de datos, no pudiendo volver al estado anterior salvo mediante copias de seguridad. Si una transacción encuentra un error en medio del proceso, deberán devolverse los datos (**ROLLBACK**) a como estaban antes de empezar la transacción.

### *Ejemplo.*

Imagina una aplicación informática con acceso a BD de un banco. Cuando se realiza una transferencia de una cuenta a otra debe restarse la cantidad ingresada de la cuenta origen y sumarla a la cuenta destino. Esto son dos operaciones.

Si minoramos el importe de la cuenta origen y antes de sumar el importe a la cuenta destino se produce un error de conexión y la operación se deja sin realizar, se produciría un descuadre. Por este motivo, hasta que no se finaliza completamente el bloque de operaciones (la sustracción del saldo y la adición del saldo) no se confirma por completo la transacción.

Por tanto... es **muy importante** recordar que las transacciones controlan la integridad de los datos, es decir, si no aseguramos que una operación se realice completamente o sino, que deshaga los cambios realizados, tendremos como consecuencia descuadres en la BD, es decir, ***tablas en las que no ha llegado a insertarse un registro y que pueden hacer fallar programas posteriores en cadena, llegando a paralizar la actividad de una empresa en el peor de los casos.***

Este tipo de incidencias suelen ser complicadas de gestionar porque hay que comprobar tabla por tabla el rastro de la operación, validando si se ha realizado bien o mal. Si solo son dos tablas puede ser asumible, pero por lo general, en empresas grandes hablamos de un mínimo de 10 tablas interrelacionadas.

Contamos con dos tipos de transacciones en SQL Server: las **implícitas** y las **explícitas**.

### **Transacciones implícitas**

SQL Server funciona por defecto con transacciones de confirmación automática, es decir, cada instrucción individual es una transacción y se confirma automáticamente después de su ejecución.

Podemos activar o desactivar el modo de transacciones implícitas ejecutando la siguiente instrucción:

```
-- Activamos el modo de transacciones implícitas (activo por defecto)
SET IMPLICIT_TRANSACTIONS ON

-- Desactivamos el modo de transacciones implícitas
SET IMPLICIT_TRANSACTIONS OFF
```

A partir de ahora, es conveniente que desactivemos el modo de transacciones implícitas para controlarlo nosotros/as mismos/as.

## IMPORTANTE

Si estamos accediendo a una BD multiusuario (la habitual con la que trabajan todas las empresas), si desactivamos las transacciones implícitas los cambios que hagamos los veremos en nuestra sesión, pero el resto de los/as compañeros/as no verán los cambios en las tablas hasta que confirmemos la transacción con COMMIT.

## Transacciones explícitas

Las transacciones explícitas son aquellas que debe delimitar el/la desarrollador/a utilizando la sentencia **BEGIN TRAN** o **BEGIN TRANSACTION**. Para finalizarla correctamente utilizaremos la instrucción **COMMIT** y para retroceder los cambios **ROLLBACK**.

### Ejemplo 1 – Creación de un pedido completo sin transacciones

Realiza un script que cree un pedido de dos productos para un cliente dado.

```
USE JARDINERIA
GO
-- Activamos transacciones automáticas
SET IMPLICIT_TRANSACTIONS ON

-- Cliente al que se le crearán los pedidos
DECLARE @codCliente INT = 8, @nuevoPedido INT, @error INT

-- Seleccionamos dos productos cualesquiera
DECLARE @codProd1 VARCHAR(15) = 'FR-19', @codProd2 VARCHAR(15) = 'OR-001'
DECLARE @precioProd1 NUMERIC(15,2), @precioProd2 NUMERIC(15,2)

BEGIN TRY
    -- Obtenemos el pedido a insertar a partir del último insertado
    SET @nuevoPedido = (SELECT MAX(codPedido) + 1
                        FROM PEDIDOS)

    -- Creamos el nuevo pedido sumando uno al último insertado
    INSERT INTO PEDIDOS
    VALUES (@nuevoPedido, '2023-01-30', '2023-02-04', NULL, 'Pendiente',
            NULL, @codCliente);
```

```

-- Obtenemos el precio de cada producto actualmente
SELECT @precioProd1 = precio_venta
FROM PRODUCTOS
WHERE codProducto = @codProd1;

SELECT @precioProd2 = precio_venta
FROM PRODUCTOS
WHERE codProducto = @codProd2;

-- Insertamos los productos incluidos en dicho pedido
INSERT INTO DETALLE_PEDIDOS
VALUES (@nuevoPedido, @codProd1, 1, @precioProd1, 1);

-- Provocamos un error
SELECT @error = 1/0

-- Insertamos los productos incluidos en dicho pedido
INSERT INTO DETALLE_PEDIDOS
VALUES (@nuevoPedido, @codProd2, 1, @precioProd2, 2);

END TRY

BEGIN CATCH
    -- Se han producido errores, deshacemos los cambios y mostramos el error
    PRINT CONCAT ('CODERROR: ', ERROR_NUMBER(),
                  ', DESCRIPCION: ', ERROR_MESSAGE(),
                  ', LINEA: ', ERROR_LINE());
END CATCH

```

Ejecutamos el código y obtenemos el siguiente resultado:

Messages

```

(1 row affected)

(1 row affected)
CODERROR: 8134, DESCRIPCION: Divide by zero error encountered., LINEA: 36

```

Ha habido un error, vamos a comprobar si se han realizado los cambios correctamente:

#### PEDIDOS

codPedido	fecha_pedido	fecha_esperada	fecha_entrega	estado	comentarios	codCliente
125	2009-02-14	2009-02-20	NULL	Rechazado	el producto ha sido rechazado por la pesima cal...	30
126	2009-05-13	2009-05-15	2009-05-20	Pendiente	NULL	30
127	2009-04-06	2009-04-10	2009-04-10	Entregado	NULL	30
128	2008-11-10	2008-12-10	2008-12-29	Rechazado	El pedido ha sido rechazado por el cliente por el...	38
129	2023-01-30	2023-02-04	NULL	Pendiente	NULL	8

#### DETALLE\_PEDIDOS

codPedido	codProducto	cantidad	precio unidad	numero linea
129	FR-19	1	4.00	1

Vemos que falta uno de los productos y en concreto la línea 2. No obstante, la base de datos se queda en este estado inconsistente. La solución pasaría por eliminar los registros creados erróneamente y pidiendo al usuario que vuelva a hacer el pedido. **¿Y si detectamos este error tras... una campaña de rebajas?**

## Ejemplo 2 – Creación de un pedido completo utilizando transacciones correctamente

Realiza un script que cree un pedido de dos productos para un cliente dado.

```
USE JARDINERIA
GO
-- Activamos transacciones explícitas
SET IMPLICIT_TRANSACTIONS OFF

-- Cliente al que se le crearán los pedidos
DECLARE @codCliente INT = 9, @nuevoPedido INT

-- Seleccionamos dos productos cualesquiera
DECLARE @codProd1 VARCHAR(15) = 'FR-19', @codProd2 VARCHAR(15) = 'OR-001'
DECLARE @precioProd1 NUMERIC(15,2), @precioProd2 NUMERIC(15,2)

BEGIN TRY
    BEGIN TRAN
        -- Obtenemos el pedido a insertar a partir del último insertado
        SET @nuevoPedido = (SELECT MAX(codPedido) + 1
                           FROM PEDIDOS)

        -- Creamos el nuevo pedido sumando uno al último insertado
        INSERT INTO PEDIDOS
        VALUES (@nuevoPedido, '2023-01-30', '2023-02-04', NULL, 'Pendiente',
                NULL, @codCliente);

        -- Obtenemos el precio de cada producto actualmente
        SELECT @precioProd1 = precio_venta
        FROM PRODUCTOS
        WHERE codProducto = @codProd1;

        SELECT @precioProd2 = precio_venta
        FROM PRODUCTOS
        WHERE codProducto = @codProd2;

        -- Insertamos los productos incluidos en dicho pedido
        INSERT INTO DETALLE_PEDIDOS
        VALUES (@nuevoPedido, @codProd1, 1, @precioProd1, 1);

        -- Insertamos los productos incluidos en dicho pedido
        INSERT INTO DETALLE_PEDIDOS
        VALUES (@nuevoPedido, @codProd2, 1, @precioProd2, 2);

        -- Si hemos llegado al final, confirmamos la transacción
        COMMIT
    END TRY

    BEGIN CATCH
        -- Se han producido errores, deshacemos los cambios y mostramos el error
        ROLLBACK
        PRINT CONCAT ('CODERROR: ', ERROR_NUMBER(),
```

```

', DESCRIPCION: ', ERROR_MESSAGE(),
', LINEA: ', ERROR_LINE());

```

END CATCH

Comprobamos si se han realizado los cambios correctamente:

#### PEDIDOS

codPedido	fecha_pedido	fecha_esperada	fecha_entrega	estado	comentarios	codCliente
120	2009-03-07	2009-03-27	NULL	Rechazado	El pedido fue rechazado por el cliente	16
121	2008-12-28	2009-01-08	2009-01-08	Entregado	Pago pendiente	16
122	2009-04-09	2009-04-15	2009-04-15	Entregado	NULL	16
123	2009-01-15	2009-01-20	2009-01-24	Pendiente	NULL	30
124	2009-03-02	2009-03-06	2009-03-06	Entregado	NULL	30
125	2009-02-14	2009-02-20	NULL	Rechazado	el producto ha sido rechazado por la pesima cal...	30
126	2009-05-13	2009-05-15	2009-05-20	Pendiente	NULL	30
127	2009-04-06	2009-04-10	2009-04-10	Entregado	NULL	30
128	2008-11-10	2008-12-10	2008-12-29	Rechazado	El pedido ha sido rechazado por el cliente por el...	38
129	2023-01-30	2023-02-04	NULL	Pendiente	NULL	9

#### DETALLE\_PEDIDOS

codPedido	codProducto	cantidad	precio_unidad	numero_linea
129	FR-19	1	4.00	1
129	OR-001	1	5.00	2

**¡Todo correcto!** Comprueba tú mismo/a que si introduces un error igual que en el ejemplo 1 los cambios se deshacen sin confirmar.

### IMPORTANTE

Después de un “BEGIN TRAN”, todas las sentencias de actualización, borrado, inserción, etc. (DML) se quedan pendientes de confirmar hasta que se encuentre un COMMIT o un ROLLBACK.

Las sentencias DDL (CREATE TABLE, ALTER TABLE, etc.) no tienen efectos en las transacciones. Se hace COMMIT automáticamente y un ROLLBACK no conseguiría dejar la tabla como estaba con anterioridad.



## Generar errores personalizados con RAISERROR

En ocasiones, es necesario provocar voluntariamente un error, por ejemplo, nos puede interesar que se genere un error cuando los datos incumplen una regla de negocio.

Esto nos permite generar errores personalizados que podemos controlar y tratar en consecuencia.

Anteriormente, cuando se producía la excepción de división entre cero, nos devolvía el código 8134. Existe una tabla de sistema en la que se almacenan todos estos códigos y la descripción asociada a cada idioma. La tabla se denomina **SYS.MESSAGES**

Si consultamos utilizando el código 8134 tenemos que:

```
SELECT *  
FROM SYS.MESSAGES  
WHERE message_id = 8134;
```

message_id	language_id	severity	is_event_logged	text
8134	1033	16	0	Divide by zero error encountered.
8134	1031	16	0	Fehler aufgrund einer Division durch Null.
8134	1036	16	0	Division par zéro.
8134	1041	16	0	0 除算エラーが発生しました。
8134	1030	16	0	Der opstod en Division med nul-fejl.
8134	3082	16	0	Error de división entre cero.
8134	1040	16	0	Errore di divisione per zero.
8134	1043	16	0	Fout wegens delen door nul.
8134	2070	16	0	Foi encontrado um erro de divisão por z...
8134	1035	16	0	On tapahtunut nollalla jakamisen virhe.
8134	1053	16	0	Division med noll-fel har påträffats.

La función **RAISERROR** recibe tres parámetros:

- **código de error:** de la tabla **sys.messages**, por ejemplo, 8134) o bien la descripción del error utilizando comillas simples (es preferible indicar el código, a continuación se verá cómo crear uno personalizado).
- **la severidad:** si indicamos una severidad de 0 a 10, la excepción NO se capturará en el bloque TRY, deberemos indicar 11 o más para que salte y no indicar nunca más de 17 porque se consideran errores del sistema.
- **el estado:** número entero de 0 a 255. Si hay varios puntos en el código con RAISERROR para la misma excepción, utilizar el estado identifica cuál es el que está fallando. Por ejemplo, en el primero ponemos un 0, el siguiente un 1, el siguiente un 2, y así sucesivamente.

Para crear un código de error personalizado utilizaremos el procedimiento **sp\_addmessage** del siguiente modo:

```
EXEC sp_addmessage @msgnum = 50001, -- Siempre >= 50001 para los personalizados  
                  @severity = 11, -- Si queremos que salte en el TRY >= 11  
                  @msgtext = 'El usuario no existe';
```

Al ejecutarlo deberá devolver: `Commands completed successfully.`

Y a partir de ese momento ya podemos utilizar el código 50001 para generar nuestros propios errores personalizados.

### **Ejemplo 3 – Utilización de RAISERROR con errores personalizados**

*Realiza un script que cree un pedido de dos productos para un cliente dado.*

```
USE JARDINERIA
GO
-- Activamos transacciones explícitas
SET IMPLICIT_TRANSACTIONS OFF

-- Cliente al que se le crearán los pedidos
DECLARE @codCliente INT = 9, @nuevoPedido INT

-- Seleccionamos dos productos cualesquiera
DECLARE @codProd1 VARCHAR(15) = 'FR-19', @codProd2 VARCHAR(15) = 'OR-001'
DECLARE @precioProd1 NUMERIC(15,2), @precioProd2 NUMERIC(15,2)

BEGIN TRY
    BEGIN TRAN
        -- Obtenemos el pedido a insertar a partir del último insertado
        SET @nuevoPedido = (SELECT MAX(codPedido) + 1
                           FROM PEDIDOS)

        -- Creamos el nuevo pedido sumando uno al último insertado
        INSERT INTO PEDIDOS
        VALUES (@nuevoPedido, '2023-01-30', '2023-02-04', NULL, 'Pendiente',
                NULL, @codCliente);

        -- Obtenemos el precio de cada producto actualmente
        SELECT @precioProd1 = precio_venta
        FROM PRODUCTOS
        WHERE codProducto = @codProd1;

        SELECT @precioProd2 = precio_venta
        FROM PRODUCTOS
        WHERE codProducto = @codProd2;

        -- Insertamos los productos incluidos en dicho pedido
        INSERT INTO DETALLE_PEDIDOS
        VALUES (@nuevoPedido, @codProd1, 1, @precioProd1, 1);

        -- Provocamos el error personalizado
        RAISERROR (50001, 11, 1);

        -- Insertamos los productos incluidos en dicho pedido
        INSERT INTO DETALLE_PEDIDOS
        VALUES (@nuevoPedido, @codProd2, 1, @precioProd2, 2);

        -- Si hemos llegado al final, confirmamos la transacción
```

**COMMIT**

END TRY

BEGIN CATCH


-- Se han producido errores, deshacemos los cambios y mostramos el error

**ROLLBACK**

PRINT CONCAT ('CODERROR: ', ERROR\_NUMBER(),  
                  ', DESCRIPCION: ', ERROR\_MESSAGE(),  
                  ', LINEA: ', ERROR\_LINE());

END CATCH

**Si ejecutamos el código veremos que se devuelve el error personalizado:**

 Messages

(1 row affected)  
CODERROR: 50001, DESCRIPCION: Error personalizado, LINEA: 31