

BD

Bases de Datos

Procedimientos y funciones

ÍNDICE

1. Procedimientos almacenados
2. Funciones

ANEXOS

Anexo I: Uso de tablas temporales como variables a modo de vectores

Anexo II: Uso de funciones de conversión

Anexo III: Uso de la función REPLICATE

1. Procedimientos almacenados

Para el desarrollo de software es muy importante evitar la repetición de código a lo largo del programa, puesto que con ello se conseguirá aumentar exponencialmente su complejidad. Esto puede conllevar que se disparen los costes del mantenimiento y resolución de incidencias y llegar incluso a pasar gran parte de tiempo refactorizando código o incluso rehaciéndolo por completo, con el perjuicio económico que puede conllevar para una empresa.

SQL Server disponemos de procedimientos almacenados y funciones, los cuales son elementos que se almacenan en la propia base de datos y contienen un conjunto de instrucciones. En resumen, son scripts que reciben un nombre concreto (por ejemplo. *obtenerDatosCliente*, *crearPedido*, etc.) pueden recibir parámetros de entrada/salida necesarios para su ejecución o no recibir ninguno si no fueran necesarios. Estos procedimientos realizan operaciones sobre las tablas de la base de datos, controlando la gestión de excepciones y devolviendo un valor que indicará si el programa ha finalizado correctamente o con errores.

IMPORTANTE: Los procedimientos se crean sobre una determinada base de datos, por lo que antes de crearlos o utilizarlos debemos asegurar que estamos conectados en la base de datos correcta.

Ejemplo 1: Creación de procedimiento sin parámetros

En lugar de **CREATE PROCEDURE** utilizamos **CREATE OR ALTER PROCEDURE** para que se pueda ejecutar cuando todavía no existe el procedimiento o, si ya existiera, para sobrescribirlo con la nueva implementación. Este procedimiento no recibe ningún parámetro y lo que realiza tan solo es imprimir la cadena: ¡Hola mundo!

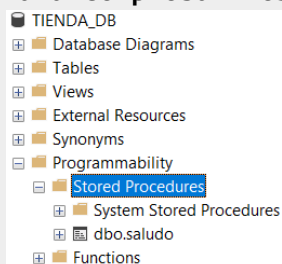
```
-- Creamos el procedimiento seleccionando desde CREATE hasta END + tecla F5
CREATE OR ALTER PROCEDURE saludo
AS
BEGIN
    PRINT '¡Hola mundo!'
END

-- Para ejecutarlo seleccionamos lo siguiente + tecla F5
EXEC saludo
```

Resultado

Messages
¡Hola mundo!

Para comprobar los procedimientos creados vamos a



Ejemplo 2: Creación de procedimiento con un parámetro de entrada

En este ejemplo el procedimiento recibirá parámetros de entrada, por lo que cambiará un poco. Debemos indicar la lista de parámetros entre paréntesis separados por comas si hubiera más de uno e indicando el tipo de cada parámetro. Los parámetros deben llevar @ igual que las variables, aunque no es necesario crearlos previamente. Recuerda que en la ejecución los parámetros NO van envueltos entre paréntesis.

IMPORTANTE: Si indicamos CHAR o VARCHAR sin poner la longitud entre paréntesis, asumirá que la longitud es UN carácter. Es decir, CHAR = CHAR(1) y VARCHAR = VARCHAR(1)

-- Creamos el procedimiento seleccionando desde CREATE hasta END + tecla F5

```
CREATE OR ALTER PROCEDURE dimeNumero (@numero INT)
```

```
AS
```

```
BEGIN
```

```
    PRINT CONCAT('El número es el: ', @numero)
```

```
END
```

-- Para ejecutarlo seleccionamos lo siguiente + tecla F5

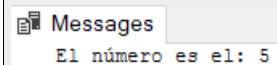
```
EXEC dimeNumero 5
```

-- También, podemos utilizar una variable que contenga el valor (es preferible):

```
DECLARE @variable INT = 5
```

```
EXEC dimeNumero @variable
```

Resultado



Messages
El número es el: 5

Ejemplo 3: Creación de procedimiento con varios parámetros de entrada

Indicamos los parámetros con @ y separados por comas. Llamada al procedimiento SIN paréntesis.

-- Creamos el procedimiento seleccionando desde CREATE hasta END + tecla F5

```
CREATE OR ALTER PROCEDURE dimeNumeroSaluda (@numero INT, @saludo VARCHAR(20))
```

```
AS
```

```
BEGIN
```

```
    PRINT CONCAT('El número es el: ', @numero)
```

```
    PRINT @saludo
```

```
END
```

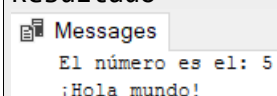
-- Para ejecutarlo seleccionamos lo siguiente + tecla F5

```
DECLARE @variable INT = 5
```

```
DECLARE @cadena VARCHAR(20) = '¡Hola mundo!'
```

```
EXEC dimeNumeroSaluda @variable, @cadena
```

Resultado



Messages
El número es el: 5
¡Hola mundo!

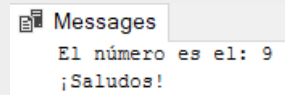
Ejemplo 4: Creación de procedimiento con parámetros de entrada por defecto

En lugar de `CREATE PROCEDURE` utilizamos `CREATE OR ALTER PROCEDURE` para que se pueda ejecutar cuando todavía no existe el procedimiento o, si ya existiera, para sobrescribirlo con la nueva implementación. Este procedimiento no recibe ningún parámetro y lo que realiza tan solo es imprimir la cadena: ¡Hola mundo!

```
-- Creamos el procedimiento seleccionando desde CREATE hasta END + tecla F5
CREATE OR ALTER PROCEDURE dimeNumeroSaluda (@numero INT = 9,
                                             @saludo VARCHAR(20) = '¡Saludos!')
AS
BEGIN
    PRINT CONCAT('El número es el: ', @numero)
    PRINT @saludo
END
```

-- Podremos llamar al procedimiento SIN parámetros y los cogerá por defecto
EXEC dimeNumeroSaluda

Resultado

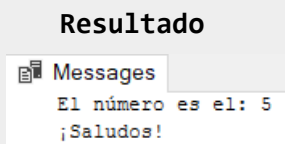


Messages
El número es el: 9
¡Saludos!

-- Si en la llamada ponemos el primero
DECLARE @variable INT = 5

EXEC dimeNumeroSaluda @variable

Resultado

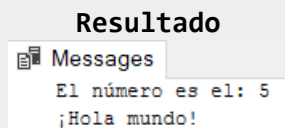


Messages
El número es el: 5
¡Saludos!

-- Si ponemos los dos
DECLARE @variable INT = 5
DECLARE @cadena VARCHAR(20) = '¡Hola mundo!'

EXEC dimeNumeroSaluda @variable, @cadena

Resultado

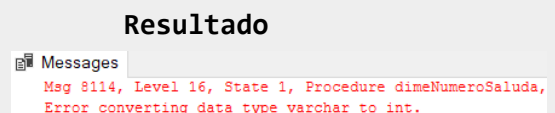


Messages
El número es el: 5
¡Hola mundo!

-- PERO LO QUE NO PODEMOS HACER ES...
DECLARE @variable INT = 5
DECLARE @cadena VARCHAR(20) = '¡Hola mundo!'

EXEC dimeNumeroSaluda @cadena

Resultado



Messages
Msg 8114, Level 16, State 1, Procedure dimeNumeroSaluda,
Error converting data type varchar to int.

¿Por qué ha fallado el código?

Respuesta: Los parámetros deben ir en orden, por lo que, si indicamos el primero, el tipo de dato debe coincidir con el que espera el procedimiento (como le hemos pasado un VARCHAR y espera un INT) devuelve el error.

Cuidado, si los tipos de datos coinciden por casualidad en ambos parámetros, el procedimiento se ejecutaría sin errores pero podría causar un efecto no deseado y además es complicado de detectar porque no se avisa del error.

Ejemplo 5: Creación de procedimiento con parámetros de entrada y de salida

Para marcar que un parámetro es de salida deberemos indicarlo con **OUTPUT**.

NOTA: Para crear dos procedimientos seguidos necesitamos indicar **GO** entre uno y otro.

```
-- Creamos ambos procedimientos
CREATE OR ALTER PROCEDURE multiplica1 (@numero1 INT,
                                       @numero2 INT)
AS
BEGIN
    PRINT CONCAT('Resultado: ', @numero1 * @numero2)
END

GO

CREATE OR ALTER PROCEDURE multiplica2 (@numero1 INT,
                                       @numero2 INT,
                                       @resultado INT OUTPUT)
AS
BEGIN
    SET @resultado = @numero1 * @numero2
END
```

¿Hacen lo mismo estas dos implementaciones del procedimiento “multiplica”? Razona tu respuesta.

```
-- Ejecución del primero
DECLARE @num1 INT = 4, @num2 INT = 2

EXEC multiplica1 @num1, @num2

-- Ejecución del segundo
DECLARE @num1 INT = 4, @num2 INT = 2, @resultado INT

EXEC multiplica2 @num1, @num2, @resultado OUTPUT

PRINT @resultado
```

Messages
Resultado: 8

Messages
8

¡IMPORTANTE! Si no indicamos **OUTPUT** en la llamada también, el parámetro será de entrada y su valor de salida será **NULL**

Respuesta

Aparentemente, los dos procedimientos hacen lo mismo, pero en realidad no es así. El primero de ellos simplemente imprime el resultado desde DENTRO del procedimiento, pero no lo saca de él, por lo que NO podremos operar, ni llamar a otro procedimiento ni nada (aunque nosotros veamos la salida por pantalla).

Por otro lado, el segundo, permite la utilización de la variable resultado para realizar operaciones adicionales, llamar a otro procedimiento con el parámetro, etc. La opción 2 es la que siempre debemos utilizar.

Ejemplo 6: Valor de retorno de un procedimiento

Cuando finaliza la ejecución de un procedimiento se devuelve un valor de finalización.

Si se ha ejecutado sin errores: devolverá el valor 0.

Si ha habido algún error: devolverá cualquier valor distinto de 0.

A partir de este momento, cuando ejecutemos cualquier procedimiento debemos recoger el valor de retorno y evaluar si ha finalizado correctamente. Si no ha finalizado correctamente no seguiremos con la ejecución y podemos mostrar un mensaje de error indicándolo.

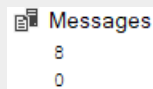
```
CREATE OR ALTER PROCEDURE multiplica2 (@numero1 INT,
                                       @numero2 INT,
                                       @resultado INT OUTPUT)

AS
BEGIN
    SET @resultado = @numero1 * @numero2
END

-- Seleccionamos y F5 para ejecutarlo
DECLARE @num1 INT = 4, @num2 INT = 2, @resultado INT, @valorRetorno INT

EXEC @valorRetorno = multiplica2 @num1, @num2, @resultado OUTPUT

PRINT @resultado
PRINT @valorRetorno
```



Messages

8
0

-- Si modificamos el procedimiento anterior, introduciendo un error:

```
CREATE OR ALTER PROCEDURE multiplica2 (@numero1 INT,
                                       @numero2 INT,
                                       @resultado INT OUTPUT)

AS
BEGIN
    SET @resultado = @numero1 / 0
END

-- Seleccionamos y F5 para ejecutarlo
DECLARE @num1 INT = 4, @num2 INT = 2, @resultado INT, @valorRetorno INT

EXEC @valorRetorno = multiplica2 @num1, @num2, @resultado OUTPUT

-- Comprobamos finalización OK y mostramos el resultado
IF @valorRetorno = 0
BEGIN
    PRINT @resultado
END

-- Ha finalizado con errores
ELSE
BEGIN
    PRINT @valorRetorno
    PRINT 'ERROR en el procedimiento multiplica2'
END
```

El **retorno ha sido -6**, lo cual indica que el proceso no ha finalizado correctamente y que por tanto **NO PODEMOS ASUMIR COMO CORRECTO EL VALOR DEVUELTO**.

```
Messages
Msg 8134, Level 16, State 1, Procedure multiplica2,
Divide by zero error encountered.
-6
ERROR en el procedimiento multiplica2
```

Ejemplo 7: Validación de parámetros en procedimientos

Antes de ejecutar el código de un procedimiento, es muy importante que tengamos informados todos los parámetros necesarios para ello. Si no fuera así, el procedimiento no debería ejecutarse y se debería informar del error.

Cuando haya parámetros obligatorios a NULL podemos devolver -1 para detectarlo

```
-- Seleccionamos y F5 para ejecutarlo
CREATE OR ALTER PROCEDURE multiplica2 (@numero1 INT,
                                       @numero2 INT,
                                       @resultado INT OUTPUT)
AS
BEGIN
    -- Validamos los parámetros
    IF @numero1 IS NULL OR @numero2 IS NULL
    BEGIN
        RETURN -1
    END

    SET @resultado = @numero1 / @numero2
END

-- Llamada al procedimiento
DECLARE @num1 INT = 4, @num2 INT, @resultado INT, @valorRetorno INT

EXEC @valorRetorno = multiplica2 @num1, @num2, @resultado OUTPUT

IF @valorRetorno = 0
BEGIN
    PRINT @resultado
END

ELSE
BEGIN
    PRINT @valorRetorno
    PRINT 'ERROR en el procedimiento multiplica2'
END
```

Resultado

```
Messages
-1
ERROR en el procedimiento multiplica2
```

Ejemplo 8: Borrado de procedimientos

Borrado de un procedimiento. A partir de ese momento ya no se podrá llamar más.


```
-- Seleccionamos y F5 para ejecutarlo
DROP PROCEDURE multiplica1

-- Llamada a un procedimiento inexistente
EXEC multiplica1
```

Messages

```
Msg 2812, Level 16, State 62, Line 27
Could not find stored procedure 'multiplica1'.
```

En los ejemplos que se han visto hasta el momento, se ha realizado simplemente la impresión de los parámetros de entrada, pero la funcionalidad real que se le da a los procedimientos es para combinar:

- Obtención de datos de tablas con SELECT
- Inserción de registros con INSERT
- Actualización de datos con UPDATE
- Borrado de registros con DELETE
- Uso de instrucciones de programación IF, ELSE, WHILE, etc.
- Manejo de excepciones con TRY/CATCH
- Transacciones para evitar que los cambios queden inconsistentes ante un error

Consejo: Es una buena práctica **validar que los parámetros de entrada** introducidos por el usuario o por otros procedimientos **son correctos**. **Si no lo fueran**, el procedimiento **no debe ejecutarse** y se parará la ejecución informando del error.

Ejemplo 9: Procedimiento con sentencia DML con PK IDENTITY

Procedimiento que inserte un cliente en la base de datos con gestión de excepciones y transacción.

```
USE TIENDA_DB
GO
-- Seleccionamos y F5 para ejecutarlo
CREATE OR ALTER PROCEDURE crearCliente (@NIFCIF CHAR(10),
                                         @nombre VARCHAR(50),
                                         @apellidos VARCHAR(200),
                                         @tipoVia CHAR(2),
                                         @calle VARCHAR(200),
                                         @numeroVia VARCHAR(5),
                                         @pisoEscPuerta VARCHAR(30),
                                         @telefono CHAR(9),
                                         @email VARCHAR(80),
                                         @codProv TINYINT,
                                         @codMuni SMALLINT)
AS
BEGIN
    BEGIN TRY

        -- Validación de parámetros (todos los obligatorios según la tabla)
```

```

IF @NIFCIF IS NULL OR @nombre IS NULL OR @apellidos IS NULL OR
@tipoVia IS NULL OR @calle IS NULL OR @numeroVia IS NULL OR
@pisoEscPuerta IS NULL OR @telefono IS NULL OR @email IS NULL OR
@codProv IS NULL OR @codMuni IS NULL
BEGIN
    RETURN -1
END

BEGIN TRANSACTION

    -- Sentencia de inserción
    INSERT INTO CLIENTE (NIFCIF, nombre, apellidos,
                        tipoVia, calle, numeroVia, pisoEscPuerta,
                        telefono, email, codProv, codMuni)
    VALUES (@NIFCIF, @nombre, @apellidos,
            @tipoVia, @calle, @numeroVia, @pisoEscPuerta,
            @telefono, @email, @codProv, @codMuni);

    -- Confirmamos los cambios
    COMMIT

END TRY

BEGIN CATCH
    -- Retrocedemos los cambios y mostramos el error
    ROLLBACK
    PRINT CONCAT('ERROR en crearCliente, codError=', ERROR_NUMBER(),
                ', Descripcion=', ERROR_MESSAGE(),
                ', linea=', ERROR_LINE());
END CATCH

END

-- Llamada al procedimiento
DECLARE @NIFCIF CHAR(10), @nombre VARCHAR(50), @apellidos VARCHAR(200)
DECLARE @tipoVia CHAR(2), @calle VARCHAR(200), @numeroVia VARCHAR(5)
DECLARE @pisoEscPuerta VARCHAR(30), @telefono CHAR(9), @email VARCHAR(80)
DECLARE @codProv TINYINT, @codMuni SMALLINT, @retorno INT

SET @NIFCIF = '11111111A'
SET @nombre = 'Cliente 1'
SET @apellidos = 'Apellidos cliente 1'
SET @tipoVia = 'CL'
SET @calle = 'MAYOR'
SET @numeroVia = '15'
SET @pisoEscPuerta = '3C'
SET @telefono = '961111111'
SET @email = 'cliente1@gmail.com'
SET @codProv = 03      -- Provincia: Alicante
SET @codMuni = 149     -- Municipio: Alicante

EXEC @retorno = crearCliente @NIFCIF, @nombre, @apellidos,
                             @tipoVia, @calle, @numeroVia,
                             @pisoEscPuerta, @telefono, @email,

```

```
@codProv, @codMuni
```

```
IF @retorno = 0
BEGIN
    PRINT CONCAT('Cliente ', @NIFCIF, ' creado correctamente')
END
ELSE
BEGIN
    PRINT CONCAT('ERROR en procedimiento crearCliente, retorno=', @retorno)
END
```

Resultado

Messages

```
(1 row affected)
Cliente 11111111A creado correctamente
```

Si pusiéramos algún parámetro de entrada a NULL (por ejemplo el campo NIFCIF) el resultado sería el siguiente:

Messages

```
ERROR en procedimiento crearCliente, retorno=-1
```

Ejemplo 10: Procedimiento con sentencia DML (sin clave primaria IDENTITY)

Procedimiento que inserte un cliente en la base de datos con gestión de excepciones y transacción. En este caso debemos buscar el siguiente identificador libre de la tabla CLIENTE con MAX()+1

```
USE TIENDA_DB
GO
-- Seleccionamos y F5 para ejecutarlo
CREATE OR ALTER PROCEDURE crearCliente (@NIFCIF CHAR(10),
                                         @nombre VARCHAR(50),
                                         @apellidos VARCHAR(200),
                                         @tipoVia CHAR(2),
                                         @calle VARCHAR(200),
                                         @numeroVia VARCHAR(5),
                                         @pisoEscPuerta VARCHAR(30),
                                         @telefono CHAR(9),
                                         @email VARCHAR(80),
                                         @codProv TINYINT,
                                         @codMuni SMALLINT)
AS
BEGIN
    BEGIN TRY

        -- Obtenemos el idCliente a insertar (NO es un parámetro)
        -- sino que debe de obtener DENTRO
        DECLARE @idCliente INT
        SET @idCliente = (SELECT MAX(idCliente)+1
                        FROM CLIENTE)

        -- Validación de parámetros (todos los obligatorios según la tabla)
        IF @NIFCIF IS NULL OR @nombre IS NULL OR @apellidos IS NULL OR
```

```

        @tipoVia IS NULL OR @calle IS NULL OR @numeroVia IS NULL OR
        @pisoEscPuerta IS NULL OR @telefono IS NULL OR @email IS NULL OR
        @codProv IS NULL OR @codMuni IS NULL
BEGIN
    RETURN -1
END

BEGIN TRANSACTION

-- Sentencia de inserción
INSERT INTO CLIENTE (NIFCIF, nombre, apellidos,
                    tipoVia, calle, numeroVia, pisoEscPuerta,
                    telefono, email, codProv, codMuni)
VALUES (@NIFCIF, @nombre, @apellidos,
        @tipoVia, @calle, @numeroVia, @pisoEscPuerta,
        @telefono, @email, @codProv, @codMuni);

-- Confirmamos los cambios
COMMIT

END TRY

BEGIN CATCH
    -- Retrocedemos los cambios y mostramos el error
    ROLLBACK
    PRINT CONCAT('ERROR en crearCliente, codError=', ERROR_NUMBER(),
                ', Descripcion=', ERROR_MESSAGE(),
                ', linea=', ERROR_LINE());
END CATCH
END

```

Ejemplo 10: Procedimiento que llama a otro procedimiento

Crea un procedimiento que cree un pedido para un cliente.

Crea otro procedimiento que a partir del pedido anterior y recibiendo el idProducto y la cantidad cree el detalle del pedido (productos que ha comprado).

Recuerda que debemos utilizar TRY/CATCH y transacciones. Fíjate que en este caso ya no tenemos una transacción por procedimiento, sino que al ser un proceso que involucra más acciones (si creamos el pedido, pero falla la creación de algún producto en el pedido tenemos que retrocederlo TODO), por lo que la transacción debe ser EXTERNA a los procedimientos que se llaman.

```

-- Seleccionamos y F5 para ejecutarlo
CREATE OR ALTER PROCEDURE crearPedido (@idCliente INT,
                                       @idVendedor INT,
                                       @idTransportista INT,
                                       @costeEnvio DECIMAL(4,2),
                                       @recogidaTiendaSN CHAR(1),
                                       @idTiendaRecogida INT,
                                       @idPedido INT OUTPUT)
AS
BEGIN
    BEGIN TRY

```

```

DECLARE @retorno INT

-- Validamos que todos tenemos los parámetros informados
IF @idCliente IS NULL OR @idVendedor IS NULL
BEGIN
    RETURN -1
END

-- Obtenemos el último idPedido insertado
SET @idPedido = (SELECT MAX(idPedido)+1
                 FROM PEDIDO)

-- Insertamos el nuevo pedido
INSERT INTO PEDIDO (idPedido, idCliente, fecHoraPedido,
                  fecPrevEntrega, fecEntrega,
                  idVendedor, idTransportista, costeEnvio,
                  recogidaTiendaSN, idTiendaRecogida)
VALUES (@idPedido, @idCliente, GETDATE(), NULL, NULL,
        @idVendedor, @idTransportista, @costeEnvio,
        @recogidaTiendaSN, @idTiendaRecogida)

END TRY

BEGIN CATCH
    -- Mostramos la información del error
    PRINT CONCAT ('CODERROR: ', ERROR_NUMBER(),
                  ', DESCRIPCION: ', ERROR_MESSAGE(),
                  ', LINEA: ', ERROR_LINE())
END CATCH

END

GO

CREATE OR ALTER PROCEDURE crearProductoPedido (@idPedido INT,
                                              @idProducto INT,
                                              @unidades TINYINT)
AS
BEGIN
    BEGIN TRY

        DECLARE @precioCompra DECIMAL(9,2)

        -- Validamos que todos tenemos los parámetros informados
        IF @idPedido IS NULL OR @idProducto IS NULL
            OR @unidades IS NULL OR @unidades <= 0
        BEGIN
            RETURN -1
        END

        -- Obtenemos el precio actual del producto
        SET @precioCompra = (SELECT precioUnitario
                            FROM PRODUCTO
                            WHERE idProducto = @idProducto)
    
```

```

        INSERT INTO LINEA_PEDIDO (idPedido, idProducto,
                                   precioCompra, unidades)
        VALUES (@idPedido, @idProducto,
                @precioCompra, @unidades)

    END TRY

    BEGIN CATCH
        -- Mostramos la información del error
        PRINT CONCAT ('CODERROR: ', ERROR_NUMBER(),
                    ', DESCRIPCION: ', ERROR_MESSAGE(),
                    ', LINEA: ', ERROR_LINE())
    END CATCH
END

-- Hacemos las llamadas a los procedimientos
DECLARE @idPedido INT, @idCliente INT, @idVendedor INT
DECLARE @idTransportista INT, @costeEnvio DECIMAL(4,2)
DECLARE @recogidaTiendaSN CHAR(1), @idTiendaRecogida INT
DECLARE @retorno INT
DECLARE @idProducto1 INT, @unidades1 TINYINT
DECLARE @idProducto2 INT, @unidades2 TINYINT
DECLARE @idProducto3 INT, @unidades3 TINYINT

-- Variables para la creación del pedido
SET @idCliente = 2
SET @idVendedor = 3
SET @idTransportista = NULL
SET @costeEnvio = NULL
SET @recogidaTiendaSN = 'S'
SET @idTiendaRecogida = 5

-- Iniciamos la transacción
BEGIN TRAN

-- Llamamos al procedimiento
EXEC @retorno = crearPedido @idCliente, @idVendedor,
                            @idTransportista, @costeEnvio,
                            @recogidaTiendaSN, @idTiendaRecogida,
                            @idPedido OUTPUT

-- Tratamiento de la respuesta a la llamada
-- Si ha habido error, paramos la ejecución con RETURN
IF @retorno <> 0
BEGIN
    ROLLBACK -- Importante: Deshacemos la transacción
    PRINT 'ERROR en procedimiento crearPedido'
    RETURN
END

-- Llamamos al siguiente procedimiento
SET @idProducto1 = (SELECT TOP(1) idProducto

```

```

        FROM PRODUCTO
        ORDER BY NEWID())

SET @unidades1 = FLOOR(RAND()*10)

EXEC @retorno = crearProductoPedido @idPedido, @idProducto1, @unidades1
IF @retorno <> 0
BEGIN
    ROLLBACK
    PRINT 'ERROR en procedimiento crearProductoPedido, producto 1'
    RETURN
END

-- Llamamos al siguiente procedimiento
SET @idProducto2 = (SELECT TOP(1) idProducto
                    FROM PRODUCTO
                    ORDER BY NEWID())
SET @unidades2 = FLOOR(RAND()*10)

EXEC @retorno = crearProductoPedido @idPedido, @idProducto2, @unidades2
IF @retorno <> 0
BEGIN
    ROLLBACK
    PRINT 'ERROR en procedimiento crearProductoPedido, producto 2'
    RETURN
END

-- Llamamos al siguiente procedimiento
SET @idProducto3 = (SELECT TOP(1) idProducto
                    FROM PRODUCTO
                    ORDER BY NEWID())
SET @unidades3 = FLOOR(RAND()*10)

EXEC @retorno = crearProductoPedido @idPedido, @idProducto3, @unidades3
IF @retorno <> 0
BEGIN
    ROLLBACK
    PRINT 'ERROR en procedimiento crearProductoPedido, producto 3'
    RETURN
END

-- Si hemos llegado hasta aquí, todo ha ido bien
-- Confirmamos la transacción
COMMIT

PRINT 'Procedimiento finalizado correctamente'
PRINT CONCAT('Pedido ', @idPedido, ' creado')

```

Resultado

Messages

```
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
Procedimiento finalizado correctamente
Pedido 361177 creado
```

Si queremos eliminar el mensaje que aparece cuando se inserta una fila deberemos agregar al principio del script que llama a los procedimientos la instrucción **SET NOCOUNT ON** del siguiente modo:

```
DECLARE @idPedido INT, @idCliente INT, @idVendedor INT
DECLARE @idTransportista INT, @costeEnvio DECIMAL(4,2)
DECLARE @recogidaTiendaSN CHAR(1), @idTiendaRecogida INT
DECLARE @retorno INT
DECLARE @idProducto1 INT, @unidades1 TINYINT
DECLARE @idProducto2 INT, @unidades2 TINYINT
DECLARE @idProducto3 INT, @unidades3 TINYINT
```

```
-- Variables para la creación del pedido
```

```
SET @idCliente = 2
SET @idVendedor = 3
SET @idTransportista = NULL
SET @costeEnvio = NULL
SET @recogidaTiendaSN = 'S'
SET @idTiendaRecogida = 5
```

```
-- Evitar que aparezca lo que se inserta
SET NOCOUNT ON
```

```
-- Iniciamos la transacción
BEGIN TRAN
```

```
-- Llamamos al procedimiento
```

```
EXEC @retorno = crearPedido @idCliente, @idVendedor,
```

```
...
```

Resultado

Messages

```
Procedimiento finalizado correctamente
Pedido 361178 creado
```


Ejemplo 11 (avanzado): Procedimiento que devuelve múltiples valores con JSON

Crea un procedimiento que devuelva la lista de provincias en formato JSON.

```
-- Seleccionamos y F5 para ejecutarlo
CREATE OR ALTER PROCEDURE getProvincias (@json VARCHAR(MAX) OUTPUT)
AS
BEGIN
    BEGIN TRY

        SET @json = (SELECT *
                     FROM PROVINCIA
                     FOR JSON AUTO, ROOT('PROVINCIAS'))

    END TRY

    BEGIN CATCH
        -- Mostramos la información del error
        PRINT CONCAT ('CODERROR: ', ERROR_NUMBER(),
                     ', DESCRIPCION: ', ERROR_MESSAGE(),
                     ', LINEA: ', ERROR_LINE())
    END CATCH
END

-- Llamar al procedimiento y mostrar JSON
DECLARE @jsonProv VARCHAR(MAX), @retorno INT
EXEC @retorno = getProvincias @jsonProv OUTPUT

IF @retorno <> 0
BEGIN
    PRINT 'Ha ocurrido un error'
    RETURN
END
ELSE
BEGIN
    PRINT @jsonProv
END
```

Resultado

Messages

```
{"PROVINCIAS":[{"codProv":1,"nombre":"Araba/Álava"}, {"codProv":2,"nombre":"Albacete"}, {"codProv":3,"nombre":"Alicante/Alacant"}]}
```

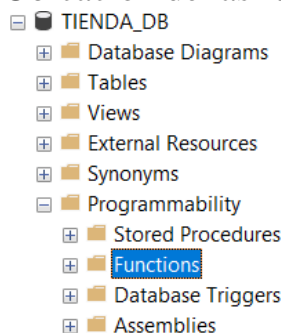
Con el código formateado se vería así:

```
1 {
2   "PROVINCIAS": [
3     {
4       "codProv": 1,
5       "nombre": "Araba/Álava"
6     },
7     {
8       "codProv": 2,
9       "nombre": "Albacete"
10    },
11    {
12      "codProv": 3,
13      "nombre": "Alicante/Alacant"
14    }
15  ]
16 }
```

2. Funciones

Siguiendo con el concepto de reutilización del código que se ha presentado con el uso de procedimientos en nuestro código de Transact-SQL, este lenguaje cuenta también con otro elemento muy similar a estos, pero que presenta ciertas diferencias que es necesario conocer para utilizarlos correctamente. Al igual que los procedimientos, las funciones se almacenan dentro de la base de datos en el servidor de SQL Server (justo debajo de los procedimientos almacenados).

Ubicación de las funciones dentro del servidor de bases de datos.



Con respecto a las diferencias con los procedimientos, a continuación, se muestra una tabla comparativa de ambos elementos:

PROCEDIMIENTOS	FUNCIONES
Pueden devolver múltiples valores a través de los parámetros de salida.	Sólo pueden devolver UN VALOR.
Parámetros de entrada y salida (E/S).	Solo parámetros de entrada (E).
Permite TRY/CATCH y transacciones	NO permite TRY/CATCH ni transacciones
Permite llamar tanto a procedimientos como a funciones	NO pueden llamar a procedimientos, pero sí a otras funciones
No se puede usar en una SELECT	SI se puede usar en una SELECT
Permite cualquier sentencia DML	Sólo permite SELECTs

Por tanto, una función es un bloque de código que permite la ejecución de SELECTs (tantas como sean necesarias) y que devolverá un valor de cualquier tipo de dato. **Podemos utilizar una función en las cláusulas SELECT, WHERE o HAVING** (recuerda que desde una función se puede llamar a otras funciones, siempre teniendo en cuenta que la salida de una de ellas va a ser la entrada de la siguiente, y así sucesivamente).

Que haya ciertas acciones que alguno de los dos elementos no pueda realizar no significa en ningún caso que los procedimientos sean mejores que las funciones o viceversa. Se debe conocer cuándo es mejor utilizar uno u otro dependiendo de cada situación o de las propias necesidades del programa.

Ejemplo 1: Función que recibe parámetros de entrada

Crea una función que calcule la suma de dos números enteros.

NOTA: Cuando creamos una función **debemos indicar el tipo de dato que se devuelve ANTES de AS.**

```
-- Seleccionamos y F5 para ejecutarlo
CREATE OR ALTER FUNCTION suma (@numero1 INT, @numero2 INT)
RETURNS INT
AS
BEGIN
    RETURN @numero1+@numero2
END
```

-- Llamamos a la función siempre con el prefijo dbo.

```
SELECT dbo.suma(2, 3)
```

Resultado

Results		Messages	
	(No column name)		
1	5		

Ejemplo 2: Función que recibe parámetros de entrada

Función que devuelva el coste total de un pedido que recibe como parámetro.

Utiliza dicha función para mostrar los pedidos de un cliente y el coste de cada uno de ellos.

```
-- Seleccionamos y F5 para ejecutarlo
CREATE OR ALTER FUNCTION costePedido (@idPedido INT)
RETURNS DECIMAL(9,2)
AS
BEGIN
    DECLARE @salida DECIMAL(9,2)

    SET @salida = (SELECT SUM(precioCompra*unidades)
                  FROM LINEA_PEDIDO
                  WHERE idPedido = @idPedido)

    RETURN @salida
END

-- Llamamos a la función siempre con el prefijo dbo.
SELECT idCliente, idPedido, dbo.costePedido(idPedido) AS costePedido
FROM PEDIDO
WHERE idCliente = 1;
```

Resultado

idCliente	idPedido	costePedido
1	26710	170.97
1	26711	407.93
1	26712	34.99
1	26713	371.94
1	26714	894.93
1	26715	387.94

Ejemplo 3: Función que devuelve la cuenta de elementos

Función que devuelva el número de productos pedidos por un cliente para cada pedido.
Utiliza dicha función para mostrar el coste anterior y el número de productos por pedido.

```
-- Seleccionamos y F5 para ejecutarlo
CREATE OR ALTER FUNCTION numProdPedido (@idPedido INT)
RETURNS INT
AS
BEGIN
    DECLARE @salida INT

    SET @salida = (SELECT COUNT(1)
                   FROM LINEA_PEDIDO
                   WHERE idPedido = @idPedido)

    RETURN @salida
END

-- Llamamos a la función siempre con el prefijo dbo.
SELECT idCliente, idPedido,
       dbo.costePedido(idPedido) AS costePedido,
       dbo.numProdPedido(idPedido) AS numProductos
FROM PEDIDO
WHERE idCliente = 1;
```

Resultado

idCliente	idPedido	costePedido	numProductos
1	26710	170.97	2
1	26711	407.93	4
1	26712	34.99	1
1	26713	371.94	2
1	26714	894.93	4
1	26715	387.94	4

Ejemplo 4: Función que llama a otra función

Función que devuelva el número de productos pedidos por un cliente para cada pedido.
Utiliza dicha función para mostrar el coste anterior y el número de productos por pedido.

```
-- Seleccionamos y F5 para ejecutarlo
CREATE OR ALTER FUNCTION calculaIVA_Pedido (@idPedido INT, @IVA DECIMAL(2,2))
RETURNS DECIMAL(9,2)
AS
BEGIN
    DECLARE @salida DECIMAL(9,2)

    -- Llamamos a la otra función
    SET @salida = dbo.costePedido(@idPedido) * @IVA

    RETURN @salida
END
```

```
-- Llamamos a la función siempre con el prefijo dbo.
SELECT idCliente, idPedido,
       dbo.costePedido(idPedido) AS costePedido,
       dbo.calculaIVA_Pedido(idPedido, 0.21) AS IVA,
       dbo.numProdPedido(idPedido) AS numProductos
FROM PEDIDO
WHERE idCliente = 1;
```

Resultado

idCliente	idPedido	costePedido	IVA	numProductos
1	26710	170.97	35.90	2
1	26711	407.93	85.67	4
1	26712	34.99	7.35	1
1	26713	371.94	78.11	2
1	26714	894.93	187.94	4
1	26715	387.94	81.47	4

Ejemplo 5: Función que devuelve S/N se cumple una condición

Función que devuelva S/N un cliente ha realizado algún pedido.

Después, modifica la función para hacer otra versión que devuelva 0/1 según tenga o no pedidos.

```
-- Seleccionamos y F5 para ejecutarlo
CREATE OR ALTER FUNCTION clientePedidos_SN (@idCliente INT)
RETURNS CHAR(1) AS
BEGIN
    DECLARE @salida CHAR(1), @numPedidos INT
    SET @numPedidos = (SELECT COUNT(1)
                      FROM PEDIDO
                      WHERE idCliente = @idCliente)

    IF @numPedidos > 0
    BEGIN
        SET @salida = 'S'
    END
    ELSE
    BEGIN
        SET @salida = 'N'
    END

    RETURN @salida
END

-- Llamamos a la función siempre con el prefijo dbo.
SELECT idCliente, dbo.clientePedidos_SN(idCliente) tienePedidos_SN
FROM CLIENTE;
```

Resultado

idCliente	tienePedidos_SN
92404	S
45368	N
110361	S
104238	S

Ejemplo 6: Función que devuelve una tabla (similar a las vistas)

Función que devuelva los pedidos realizados en un año concreto que se le pasa como parámetro.

NOTA: Este tipo de función NO tienen ni **BEGIN** ni **END**

```
-- Seleccionamos y F5 para ejecutarlo
CREATE OR ALTER FUNCTION pedidosAnyo (@anyo INT)
RETURNS TABLE
AS
    RETURN SELECT *
        FROM PEDIDO
        WHERE YEAR(fecHoraPedido) = @anyo;

-- Llamamos a la función siempre con el prefijo dbo.
SELECT *
FROM dbo.pedidosAnyo(2022);
```

Resultado

idPedido	idCliente	fecHoraPedido	fecPrevEntrega	fecEntrega
5	38349	2022-10-14 09:33:00	2022-10-21	NULL
31	91385	2022-01-07 11:29:00	2022-01-14	NULL
35	100273	2022-11-13 22:46:00	2022-11-20	2022-11-22
96	68622	2022-05-05 16:20:00	2022-05-12	2022-05-14
101	27216	2022-03-10 04:52:00	2022-03-17	2022-03-13
143	54169	2022-09-15 01:36:00	2022-09-22	2022-09-24
192	76570	2022-01-24 17:42:00	2022-01-31	2022-01-27

El uso de estas funciones que devuelven tablas es muy similar al de las vistas (VIEWS), pero con la diferencia de que podemos pasarle parámetros, por lo que sería como la versión 2.0 de las vistas.

ANEXO I: Uso de tablas temporales como @variables a modo de “vectores”

En SQL Server es posible la declaración como una @variable de tipo TABLE en la que deberemos especificar los diferentes campos que la componen.

Ejemplo.

```
-- 1º Creamos la tabla CLIENTES
CREATE TABLE CLIENTES (
    DNI CHAR (10),
    nombre VARCHAR (100) NOT NULL,
    CONSTRAINT PK_CLIENTE PRIMARY KEY (DNI)
);

-- 2º Insertamos dos registros de prueba en dicha tabla
INSERT INTO CLIENTES
VALUES ('11111111', 'Pepito'),
      ('22222222', 'Juanito');

/*3º SCRIPT DE USO
Declaramos una variable de tipo TABLE que contenga los mismos tipos de datos que la tabla
(no es necesario que el nombre de los campos coincida)
¡¡OJO!! ¡¡En las tablas temporales NO HAY INTEGRIDAD REFERENCIAL!! */
DECLARE @clientTable TABLE (NIF CHAR(10), nombre VARCHAR(100))

-- 4º A partir de aquí podemos utilizar la tabla como un “vector”
-- Por ejemplo, podemos insertar el contenido de una SELECT en la tabla

INSERT INTO @clientTable
SELECT *
FROM CLIENTES;

-- Y mostrar el resultado de la SELECT utilizando la variable TABLE

SELECT *
FROM @clientTable;
```

NIF	nombre
11111111	Pepito
22222222	Juanito

Conclusión: No debemos abusar del uso de las variables de tipo tabla porque afectan al rendimiento y por tanto su uso debe estar justificado y deberá ser puntual y controlado.

ANEXO III: Uso de funciones de conversión

Para evitar que muestre NULL cuando no haya registros y en su lugar aparezca 0 o cualquier otro valor podemos recurrir a la función integrada ISNULL.

ISNULL (valor, valor-siNulo)

- Si valor IS NOT NULL, mostrará valor
- Si valor IS NULL, mostrará valor-siNulo

La función **ISNUMERIC** permite identificar si una cadena es numérica o no lo es:

`SELECT ISNUMERIC('123');` → Devuelve 1 (true)

`SELECT ISNUMERIC('a2c');` → Devuelve 0 (false)

Conversión de tipos

En ocasiones, puede ser interesante conocer cómo convertir un número entero a una cadena de caracteres o convertir un número decimal a un número entero.

Para ello disponemos de dos funciones: **CAST** y **CONVERT**.

1) La función **CAST** devuelve el valor convertido al tipo que se le indique.

Formato → **CAST (valor AS tipo_dato)**

Ejemplo 1. Convertir un número entero/decimal a cadena de caracteres.

`SELECT CAST(33.98 AS VARCHAR);`

Resultados Mensajes	
(Sin nombre de columna)	
1	33.98

Ejemplo 2. Convertir un número decimal a un número entero.

`SELECT CAST(33.98 AS INT);`

Resultados Mensajes	
(Sin nombre de columna)	
1	33

Ejemplo 3. Convertir una cadena a tipo fecha

`SELECT CAST('20170825' AS DATE);`

Resultados Mensajes	
(Sin nombre de columna)	
1	2017-08-25

2) La función **CONVERT** convierte un valor de cualquier tipo al tipo especificado.

Formato → **CONVERT (tipo_dato, valor)**

Ejemplo 1. Convertir un número entero/decimal a cadena de caracteres.

```
SELECT CONVERT(VARCHAR, 33.98);
```

Ejemplo 2. Convertir un número decimal a un número entero.

```
SELECT CONVERT(INT, 35.89);
```

Ejemplo 3. Convertir una cadena a tipo fecha

```
SELECT CONVERT(DATE, '20170825');
```

ANEXO III: Uso de la función REPLICATE

La función **REPLICATE** puede resultar muy útil para generar espacios en blanco y conseguir tabular/encolumnar diferente información que saquemos por la consola.

Descripción de la situación:

Si ejecutamos la siguiente instrucción:

```
SELECT nombre_contacto, apellido_contacto, telefono, fax
FROM CLIENTES;
```

Tenemos que los datos aparecerán por columnas al haber utilizado una SELECT:

	nombre_contacto	apellido_contacto	telefono	fax
1	Carlos	GoldFish	5556901745	5556901746
2	Anne	Wright	5557410345	5557410346
3	Pepe	Flaute	5552323129	5552323128
4	Akane	Tendo	55591233210	55591233211
5	Antonio	Lasas	34916540145	34914851312
6	Jose	Bermejo	654987321	916549872

Sin embargo, si nos damos cuenta, cada campo tiene una longitud diferente, y si utilizamos un script en el que vamos imprimiendo registro a registro utilizando la función PRINT veremos que se genera algo parecido a lo siguiente:

```
Mensajes
Carlos GoldFish 5556901745 5556901746
Anne Wright 5557410345 5557410346
Pepe Flaute 5552323129 5552323128
Akane Tendo 55591233210 55591233211
Antonio Lasas 34916540145 34914851312
Jose Bermejo 654987321 916549872
Paco Lopez 62456810 919535678
Guillermo Rengifo 689234750 916428956
David Serrano 675598001 916421756
Jose Tacaño 655983045 916689215
Antonio Lasas 34916540145 34914851312
Pedro Camunas 34914873241 34914871541
Juan Rodriguez 34912453217 34912484764
Javier Villar 654865643 914538776
```

REPLICATE recibe dos parámetros: cadena a repetir y número veces a repetirla.

La clave está en saber cuántas veces tenemos que repetirla, pero existe un método infalible:

```
dbo.CLIENTES
Columnas
  codCliente (PK, int, No NULL)
  nombre_cliente (varchar(50), No NULL)
  nombre_contacto (varchar(30), NULL)
  apellido_contacto (varchar(30), NULL)
  telefono (varchar(15), No NULL)
  fax (varchar(15), No NULL)
  linea_direccion1 (varchar(50), No NULL)
  linea_direccion2 (varchar(50), NULL)
  ciudad (varchar(50), No NULL)
```

Vamos a la tabla y comprobamos el valor máximo del campo. En nuestro caso son 30 el nombre y los apellidos y 15 el teléfono y el fax.

```
PRINT CONCAT(@nombre, REPLICATE(' ', 30-LEN(@nombre)), ' ',
             @apellido, REPLICATE(' ', 30-LEN(@apellido)), ' ',
             @tlf, REPLICATE(' ', 15-LEN(@tlf)), ' ',
             @fax, REPLICATE(' ', 15-LEN(@fax)))
```

Longitud total – longitud ocupada = espacios en blanco de separación.

Si probamos el código, tenemos:

Mensajes			
Carlos	GoldFish	5556901745	5556901746
Anne	Wright	5557410345	5557410346
Pepe	Flaute	5552323129	5552323128
Akane	Tendo	55591233210	55591233211
Antonio	Lasas	34916540145	34914851312
Jose	Bermejo	654987321	916549872
Paco	Lopez	62456810	919535678
Guillermo	Rengifo	689234750	916428956
David	Serrano	675598001	916421756
Jose	Tacaño	655983045	916689215
Antonio	Lasas	34916540145	34914851312
Pedro	Camunas	34914873241	34914871541
Juan	Rodriguez	34912453217	34912484764
Javier	Villar	654865643	914538776
Maria	Rodriguez	666555444	912458657
Beatriz	Fernandez	698754159	978453216
Victoria	Cruz	612343529	916548735
Luis	Martinez	916458762	912354475
Mario	Suarez	964493072	964493063

Incluso en este caso nos facilitaría poder exportar los datos a un fichero e importarlo a otra base de datos del mismo SGBD (SQL Server) u otro diferente (Oracle, MariaDB, etc.), puesto que cada campo ocupa un tamaño concreto (30, 15, etc.) y sabemos donde empieza y donde acaba cada campo.

Esto sería la base para empezar con las migraciones de datos.