Programación y Procedimientos Almacenados

Contenido

1. Introducción

Hasta ahora hemos estudiado sentencias SQL orientadas a realizar una determinada tarea sobre la base de datos como definir tablas, obtener información de las tablas, actualizarlas.

Las sentencias que ya conocemos son las que en principio forman parte de cualquier lenguaje SQL. Ahora veremos que TRANSACT-SQL va más allá de un lenguaje SQL cualquiera ya que aunque no permita:

- Crear interfaces de usuario.
- Crear aplicaciones ejecutables, sino elementos que en algún momento llegarán al servidor de datos y serán ejecutados.

Incluye características propias de cualquier lenguaje de programación, características que nos permiten definir la lógica necesaria para el tratamiento de la información:

- Tipos de datos.
- Definición de variables.
- Estructuras de control de flujo.
- Gestión de excepciones.
- Funciones predefinidas.

Estas características nos van a permitir crear bloques de código orientados a realizar operaciones más complejas. Estos bloques no son programas sino procedimientos o funciones que podrán ser llamados en cualquier momento.

En SQL Server 2005 podemos definir tres tipos de bloques de código, los procedimientos almacenados *STORED PROCEDURES*, los desencadenadores (o triggers) y funciones definidas por el usuario.

Nosotros estudiaremos los dos primeros, pero lo que estudiemos nos serviría también para el tercer tipo.

Existe un cuarto tipo de bloques de código que además hemos utilizado ya, los scripts. Hasta ahora para probar y practicar con las instrucciones SQL las hemos ejecutado desde el editor de consultas del MSSMS creando una nueva consulta.

Cuando guardamos esta consulta esta se almacena en un fichero de texto plano que lleva la extensión .sql (la extensión la podemos cambiar pero se recomienda conservarla), ese conjunto de sentencias Transact SQL en formato de texto plano que se ejecutan en un servidor de SQL Server se denomina **script** o script de SQL.

Un script está compuesto por uno o varios lotes. Un lote es un conjunto de instrucciones que se envía a una instancia de SQL Server para su ejecución. En ese momento, las instrucciones del lote se compilan en un único plan de ejecución, se utiliza el comando GO para indicar el final de un lote.

En ocasiones es necesario separar las sentencias en varios lotes, por ejemplo porque Transact SQL no permite la ejecución de ciertos comandos en un mismo lote, o porque para que se pueda ejecutar una determinada instrucción se tiene que haber completado una acción anterior que se encuentra en el mismo script, también se utilizan los lotes para realizar separaciones lógicas dentro del script.

Hay que tener en cuenta que un lote delimita el alcance de las variables y sentencias del script, cuando definimos una variable local en un lote su alcance será el lote, fuera del lote no existirá.

En este tema y el siguiente redactaremos scripts con uno o varios lotes de instrucciones según el caso y veremos mejor cuándo es necesario definir lotes concretos.

Para volver a los demás bloques de código que son unidades de ejecución, empezaremos por los procedimientos almacenados, veremos primero las sentencias para crear y eliminar procedimientos almacenados, luego estudiaremos las instrucciones Transact-SQL más propias de un lenguaje de programación como son por ejemplo los bucles y estructuras condicionales. Estas instrucciones las utilizaremos dentro de los procedimientos almacenados pero veremos que también nos servirán en los scripts.

En la siguiente unidad didáctica estudiaremos los disparadores o Triggers muy similares a los procedimientos almacenados que difieren básicamente en la forma en que entran en funcionamiento y en algunas tablas especiales que pueden utilizar.

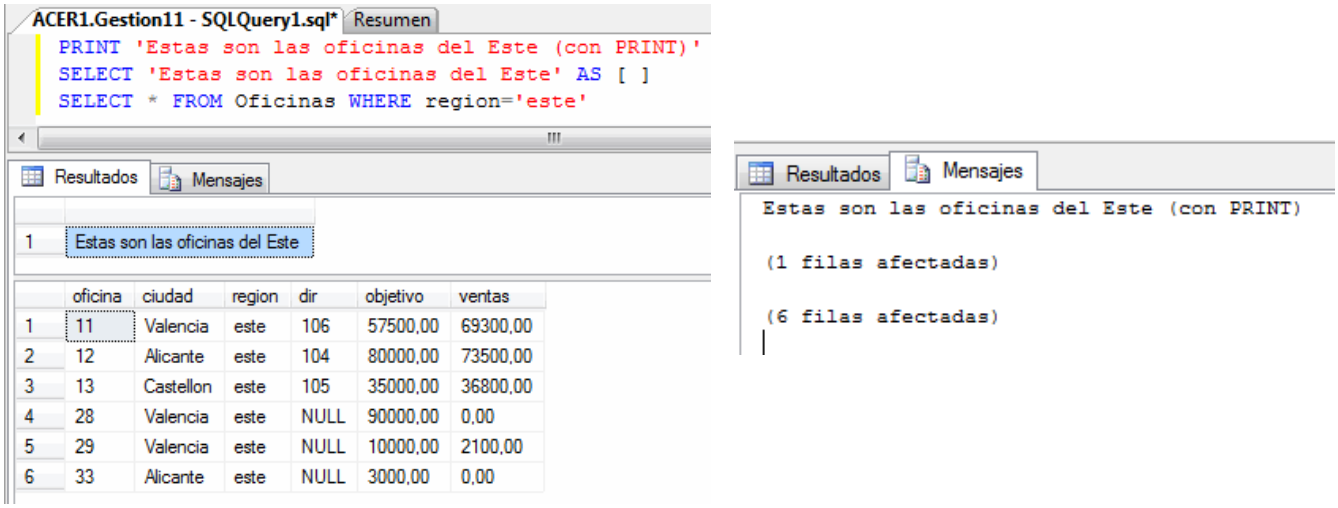
2. Mensajes

Antes de empezar a definir procedimientos almacenados introduciremos otros elementos del lenguaje que nos serán útiles posteriormente. Empezaremos por los mensajes, aquí los mensajes no serán cuadros de mensajes como podemos encontrar en otros lenguajes, nos limitaremos a enviar al usuario una cadena de texto o los valores que queramos.

Podemos utilizar dos instrucciones, PRINT y SELECT, la diferencia más determinante a la hora de utilizar una u otra es el lugar donde se produce la salida del mensaje.

Con PRINT la salida se produce en la pestaña Mensajes mientras que con SELECT la salida se produce en la pestaña Resultados, así que si queremos que el mensaje acompañe al resultado de una consulta será mejor utilizar SELECT.

Veamos este ejemplo :



En el ejemplo se aprecia que la cadena indicada en el PRINT sale en la pestaña Mensajes mientras que la cadena indicada con el SELECT sale en la pestaña Resultados, así el mensaje queda junto con la SELECT siguiente.

Cuando utilizamos SELECT estamos ejecutando una SELECT como las que conocemos pero sin FROM pero la lista de selección la podemos formar de la misma que hemos formado todas las listas de selección de los ejercicios de los temas anteriores. Aquí, para que no aparezca nada en la cabecera de la columna hemos añadido un alias de columna con un espacio en blanco.

Con el PRINT sólo podemos poner una cadena de texto o una expresión que dé como resultado una cadena:

```
PRINT 'Estos son los empleados contratados el año pasado(en '+STR(YEAR(GETDATE())-1,4) + ')'
```

El resultado será (el año cambiará según el año actual):

```
Estos son los empleados contratados el año pasado(en 2011)
```

3. Variables

Una variable de TRANSACT-SQL, como en cualquier lenguaje de programación es una estructura que permite almacenar un valor, de un tipo determinado. Para poder utilizar una variable en TRANSACT-SQL debemos previamente declararla (definirla).

3.1. Declaración de variables

La declaración de una variable consiste en definirla indicando su nombre e indicando el tipo de datos de los valores que albergará. Para los tipos de datos podemos utilizar los propios de la base de datos SQL-SERVER, pero también podemos utilizar tipos propios del lenguaje que no pueden ser utilizados en DDL. El tipo Cursor y el tipo Table son dos de estos tipos aunque nosotros no los utilizaremos, utilizaremos los vistos en el tema del DDL.

Las variables se definen utilizando la instrucción DECLARE con el siguiente formato:

```
DECLARE {@nbvariable [AS] tipo} [,...n]
```

El nombre de la variable debe empezar por el símbolo @, este símbolo hace que SQL interprete el nombre como un nombre de variable y no un nombre de objeto de la base de datos.

Ejemplo:

```
DECLARE @empleados INT
```

Con esto hemos definido la variable @empleados de tipo entero. La palabra AS no añade funcionalidad.

Hay que repetir el tipo de datos para cada variable, no se puede definir por ejemplo DECLARE @var1, var2 AS INT, hay que definirlas con dos DECLARE o con uno sólo pero repitiendo el tipo de datos:

```
DECLARE @var1 INT, @var2 INT
```

En la declaración de las variables locales no les podemos asignar directamente un valor, como se puede en otros lenguajes, para ello hay que incluir después de definirla una asignación.

3.2. Asignación

La asignación consiste en rellenar la variable con algún valor, se dice que el valor se asigna a la variable. Para asignar un valor a una variable podemos utilizar la instrucción SET o la SELECT y el signo igual con el formato:

```
{SET | SELECT} @nbvariable = valor
```

El valor puede ser cualquier valor constante, otro nombre de variable, una expresión válida o algo más potente, como una consulta escalar o parte de una SELECT (en este caso hay que utilizar obligatoriamente SELECT, no se puede utilizar SET.

Por ejemplo:

```
DECLARE @empleados AS INT, @otra AS INT
SET @otra = 20                  -- valor constante
SELECT @empleados = 0;
PRINT @empleados
SET @empleados = @otra * 100;   -- expresión
PRINT @empleados
SELECT @empleados = @otra * 100;
PRINT @empleados
SELECT @empleados = (SELECT COUNT(numemp) FROM empleados);    -- consulta escalar
PRINT @empleados
SET @empleados = (SELECT COUNT(numemp) FROM empleados);
PRINT @empleados
SELECT @empleados = COUNT(numemp) FROM empleados;             -- parte de la SELECT
PRINT @empleados
SELECT @empleados = COUNT(numemp), @otra= MAX(oficina) FROM empleados;
PRINT @empleados
PRINT @otra
```

En el script anterior se puede apreciar las diferentes maneras de asignar valores a variables locales, los PRINT son para ver el resultado de cada asignación.

En las dos primeras asignaciones hemos utilizado indistintamente SET y SELECT para asignar un valor fijo a una variable, lo mismo pasa para asignar el resultado de una expresión o el resultado de una consulta escalar (la consulta SELECT completa encerrada entre paréntesis).

Si utilizamos SELECT para la asignación, podemos utilizar una sintaxis diferente con parte de la consulta escalar porque ahora incluimos la asignación dentro de la SELECT que recupera los valores:

```
SELECT @empleados = COUNT(numemp) FROM empleados;
PRINT @empleados
```

Y además podemos aprovechar para realizar varias asignaciones con una sola SELECT:

```
SELECT @empleados = COUNT(numemp), @otra= MAX(oficina) FROM empleados;
PRINT @empleados
PRINT @otra
```

Esto sólo lo podemos hacer si la consulta devuelve una sola fila.

4. Introducción a los procedimientos almacenados

Un procedimiento almacenado *STORED PROCEDURE* está formado por un conjunto de instrucciones Transact-SQL que definen un determinado proceso, puede aceptar parámetros de entrada y devolver un valor o conjunto de resultados. Este procedimiento se guarda en el servidor y puede ser ejecutado en cualquier momento.

Los procedimientos almacenados se diferencian de las instrucciones SQL ordinarias y de los scripts de SQL que hemos ejecutado hasta ahora en los ejercicios, en que están precompilados. La primera vez que se ejecuta un procedimiento, el procesador de consultas de SQL Server lo analiza y prepara un plan de ejecución que se almacena en una tabla del sistema. Posteriormente, el procedimiento se ejecuta según el plan almacenado. Puesto que ya se ha realizado la mayor parte del trabajo de procesamiento de consultas, los procedimientos almacenados se ejecutan casi de forma instantánea por lo que el uso de procedimientos almacenados mejora notablemente la potencia y eficacia del SQL.

SQL Server incorpora procedimientos almacenados del sistema, se encuentran en la base de datos master y se reconocen por su nombre, todos tienen un nombre que empieza por sp_. Permiten recuperar información de las tablas del sistema y pueden ejecutarse en cualquier base de datos del servidor.

También están los procedimientos de usuario, los crea cualquier usuario que tenga los permisos oportunos.

Se pueden crear también procedimiento temporales locales y globales. Un procedimiento temporal local se crea por un usuario en una conexión determinada y sólo se puede utilizar en esa sesión, un procedimiento temporal global lo pueden utilizar todos los usuarios, cualquier conexión puede ejecutar un procedimiento almacenado temporal global. Éste existe hasta que se cierra la conexión que el usuario utilizó para crearlo, y hasta que se completan todas las versiones del procedimiento que se estuvieran ejecutando mediante otras conexiones. Una vez cerrada la conexión que se utilizó para crear el procedimiento, éste ya no se puede volver a ejecutar, sólo podrán finalizar las conexiones que hayan empezado a ejecutar el procedimiento.

Tanto los procedimientos temporales como los no temporales se crean y ejecutan de la misma forma, el nombre que le pongamos indicará de qué tipo es el procedimiento.

Los procedimientos almacenados **se crean** mediante la sentencia **CREATE PROCEDURE** y **se ejecutan** con **EXEC** (o EXECUTE). Para ejecutarlo también se puede utilizar el nombre del procedimiento almacenado sólo, siempre que sea la primera palabra del lote. Para eliminar un procedimiento almacenado utilizamos la sentencia DROP PROCEDURE.

5. Eliminar procedimientos almacenados.

Aunque no sabemos todavía crear un procedimiento comentaremos aquí la instrucción para eliminar procedimientos porque es muy sencilla y así podremos utilizarla en los demás ejercicios.

```
DROP {PROC|PROCEDURE} [nombreEsquema.]nombreProcedimiento [,...n ].
```

Transact-SQL permite abreviar la palabra reservada PROCEDURE por PROC sin que ello afecte a la funcionalidad de la instrucción.

Ejemplos:

```
DROP PROCEDURE Dice_Hola;
```

Elimina el procedimiento llamado Dice_Hola.

```
DROP PROC Dice_Hola;
```

Es equivalente, PROC y PROCEDURE indican lo mismo.

Para eliminar varios procedimientos de golpe, indicamos sus nombres separados por comas:

```
DROP PROCEDURE Dice_Hola, Ventas_anuales;
```

Elimina los procedimientos Dice_Hola y Ventas_anuales.

6. Ejecutar un procedimiento

6.1. El comando EXEC

Como ya hemos comentado, un procedimiento se ejecuta mediante el comando EXECUTE (o EXEC). Veamos un poco más con detalle el comando. Tiene la sintaxis siguiente:

```
    [{EXEC | EXECUTE}]
       {nb_proc | @nb_proc_var }
          [{[@parametro = ] {valor
                        | @variable [ OUTPUT ]
                        | [ DEFAULT ]
                      }
           }[,...n ]
          ]
       [;]
```

EXEC y EXECUTE son equivalentes, tienen el mismo efecto, el que se ejecute el procedimiento cuyo nombre se indica a continuación.

Se puede indicar un nombre de procedimiento almacenado o un nombre de variable que contiene el nombre del procedimiento, se reconoce el nombre de variable por llevar una @ como primer carácter.

La palabra EXEC (o EXECUTE) es opcional, se puede obviar si el nombre del procedimiento es la primera palabra de un lote de instrucciones. Para asegurarse que sea así debe ser la primera palabra del script que contiene las instrucciones por lotes o debe ir detrás de la palabra GO que indica el inicio de un nuevo bloque. Se recomienda escribir siempre la palabra EXEC/EXECUTE para evitar futuros errores.

Después del nombre del procedimiento se puede indicar una lista de valores para los parámetros del procedimiento. Esta parte de la instrucción la describiremos mejor en la CREATE PROCEDURE cuando estudiemos los tipos de parámetros que podemos tener.

El ejemplo básico sería

```
EXEC Proc1
```

Hace que el procedimiento Proc1 se ejecute. En este caso el procedimiento no tiene parámetros. Los parámetros los veremos con más detalle en apartados posteriores.

6.2. Ejecutar una cadena SQL

Otra aplicación de la palabra EXEC es la de ejecutar una cadena SQL. En este caso la sintaxis es la siguiente:

{EXEC|EXECUTE}('cadenaSQL')

Después de la palabra EXEC indicamos entre paréntesis una cadena que representa código SQL.

Por ejemplo:

```
EXEC ('SELECT * FROM Clientes')
EXEC ('SELECT * FROM CLIENTES; SELECT * FROM Productos')
```

Con los conocimientos adquiridos hasta ahora, esta variante del EXEC podría parecer inútil ya que esto mismo se podría ejecutar escribiendo directamente las SELECT sin EXEC, pero hay casos en que sí es útil, por ejemplo, cuando dentro de la instrucción SQL queremos utilizar un valor que no conocemos cuando redactamos la instrucción pero que sabemos que tendremos en una variable cuando se ejecute la instrucción.

Por ejemplo:

```
SELECT * FROM Oficinas WHERE region = @regionsolicitada
```

En este caso la región solicitada no la conocemos en el momento de redactar la sentencia sino que el usuario la introducirá cuando se ejecute la instrucción. Para guardar el valor introducido y poder utilizarlo después, se utiliza una variable, en este caso la hemos llamado @regionsolicitada. En TRANSACT-SQL los nombres de variables o parámetros deben empezar por @ siempre.

Cuando se ejecute la SELECT, en @regionsolicitada habrá un valor, por ejemplo, 'Norte', y entonces el nombre del parámetro se sustituirá por el valor que contiene y se ejecutará la instrucción:

```
 SELECT * FROM Oficinas WHERE region = 'Norte'
```

Esta forma de redactar sentencias SQL mezclando texto fijo y variables se utiliza a menudo, sobre todo cuando la sentencia SQL está embebida en otro lenguaje (por ejemplo en php, Visual Basic, Java, etc… Por lo que es imprescindible saber utilizarla.

Cuando tenemos que formar una cadena SQL (por ejemplo para poner dentro de un EXEC() recomiendo un truco, escribe primero la instrucción normal con un valor ejemplo en vez de la variable:

SELECT * FROM Oficinas WHERE ventas > 10000

Después encierra entre comillas (recuerda que en TRANSACT-SQL es la simple ') todo lo que va a ser fijo y el valor que has puesto como ejemplo lo sustituyes por + @var (He puesto en rojo lo que tienes que añadir para que se vea mejor):

'SELECT * FROM Oficinas WHERE ventas > ' + @valor

O bien por + @var + ' si después la sentencia continua, en este caso hay que acordarse de cerrar la última cadena:

SELECT * FROM Oficinas WHERE ventas > 10000 AND dir IS NULL se cambia por:

'SELECT * FROM Oficinas WHERE ventas > ' + @valor + ' AND dir IS NULL'

La cosa se complica un poco cuando el valor ejemplo es un valor de cadena encerrado entre comillas:

SELECT * FROM Oficinas WHERE region = 'Norte'

En este caso hacemos lo mismo pero hay que doblar la comilla simple:

'SELECT * FROM Oficinas WHERE region = ''' + @valor + '''

La ' que se escribe delante de la que hay en negro actúa de carácter de escape para que la comilla que viene detrás no se considere como comilla (que cierra o abre una cadena) sino como un carácter normal (como si fuese una letra).

Otra cosa a tener en cuenta son los espacios en blanco a dejar delante o detrás de la '.

Por ejemplo en ' AND dir IS NULL' hemos dejado un espacio en blanco delante de la palabra AND para que cuando se sustituya el valor de la variable, la sentencia quede bien, si no ponemos ese blanco el resultado sería:

SELECT * FROM Oficinas WHERE ventas > 1000AND dir IS NULL

Y daría error.

Lo mismo pasa con los valores de cadena, la siguiente sentencia estaría mal:

'SELECT * FROM Oficinas WHERE region = '' ' + @valor + ''''

Porque al sustituir, quedaría:

SELECT * FROM Oficinas WHERE region = ' Norte'

Habría un espacio en blanco delante de Norte y buscaría las oficinas cuya región sea espacioNorte, no saldría ninguna probablemente.

En otros lenguajes en que podemos utilizar las comillas dobles para delimitar cadenas de texto es más fácil, no hay que utilizar carácter de escape:
"SELECT * FROM Oficinas WHERE region = '" + @valor + "'"
¡Ojo! Esta cadena no sería válida en TRANSACT-SQL.

7. Crear un procedimiento - CREATE PROCEDURE

Para crear un procedimiento almacenado como hemos dicho se emplea la instrucción CREATE PROCEDURE:

```
CREATE {PROC|PROCEDURE} [NombreEsquema.]NombreProcedimiento
    [{@parametro tipo} [= valorPredet] [OUT|OUTPUT] ] [,...n]
    AS { <bloque_instrucciones> [ ...n] }[;]
<bloque_instrucciones> ::=
{[BEGIN] instrucciones [END] }
```

Las instrucciones CREATE PROCEDURE no se pueden combinar con otras instrucciones SQL en el mismo lote por lo que debe ser la única sentencia de un script o bien, si está en medio de más instrucciones deberá llevar delante y/o detrás el comando GO.

Después del verbo CREATE PROCEDURE indicamos el nombre del procedimiento, opcionalmente podemos incluir el nombre del esquema donde queremos que se cree el procedimiento, por defecto se creará en dbo.
Ya que Sqlserver utiliza el prefijo sp_ para nombrar los procedimientos del sistema, se recomienda no utilizar nombres que empiecen por sp_.

Como se puede deducir de la sintaxis (no podemos indicar un nombre de base de datos asociado al nombre del procedimiento), sólo se puede crear el procedimiento almacenado en la base de datos actual, no se puede crear en otra base de datos.

Si queremos definir un procedimiento temporal local el nombre deberá empezar por una almohadilla (#) y si el procedimiento es temporal global el nombre debe de empezar por ##.
El nombre completo de un procedimiento almacenado o un procedimiento almacenado temporal global, incluidas ##, no puede superar los 128 caracteres. El nombre completo de un procedimiento almacenado temporal local, incluidas #, no puede superar los 116 caracteres.

Transact-SQL permite abreviar la palabra reservada PROCEDURE por PROC sin que ello afecte a la funcionalidad de la instrucción.

```
CREATE PROC Calcula_comision AS…
```

Es equivalente a

```
CREATE PROCEDURE calcula_comision AS…
```

@parametro: representa el nombre de un parámetro. Se pueden declarar uno o más parámetros indicando para cada uno su nombre (debe de empezar por arroba) y su tipo de datos, y opcionalmente un valor por defecto (=valorPredet) este valor será el asumido si en la llamada el usuario no pasa ningún valor para el parámetro. Un procedimiento almacenado puede tener un máximo de 2.100 parámetros.

Los parámetros son locales para el procedimiento; los mismos nombres de parámetro se pueden utilizar en otros procedimientos. De manera predeterminada, los parámetros sólo pueden ocupar el lugar de expresiones constantes; no se pueden utilizar en lugar de nombres de tabla, nombres de columna o nombres de otros objetos de base de datos.

OUTPUT | OUT (son equivalentes)

Indica que se trata de un parámetro de salida. El valor de esta opción puede devolverse a la instrucción EXECUTE que realiza la llamada.

El parámetro variable OUTPUT debe definirse al crear el procedimiento y también se indicará en la llamada junto a la variable que recogerá el valor devuelto del parámetro. El nombre del parámetro y de la variable no tienen por qué coincidir; sin embargo, el tipo de datos y la posición de los parámetros deben coincidir a menos que se indique el nombre del parámetro en la llamada de la forma @parametro=valor.

A continuación veremos con ejemplos los diferentes parámetros que podemos tener en un procedimiento. Para los ejemplos hemos escrito scripts que se pueden probar creando una nueva consulta en el Management Studio y ejecutándola.

Para que se vea tanto la instrucción que crea el procedimiento como la instrucción que permite ejecutarlo, en el mismo script hemos incluido el CREATE PROCEDURE y el EXEC, pero una vez ejecutado el CREATE PROCEDURE el procedimiento estará creado (como cualquier objeto de la base de datos) y ya no habrá que volver a ejecutarlo si se ejecutase otra vez daría error, como cuando intentamos crear una tabla con un nombre que ya existe. Mientras que la instrucción EXEC se podrá ejecutar cada vez que queramos ejecutar el procedimiento.

7.1. Procedimiento básico

```
CREATE PROCEDURE  Dice_Hola      -- Empieza la definición del procedimiento
AS
PRINT 'Hola';                    -- Aquí termina
GO              -- Indicamos GO para cerrar el lote que crea el procedimiento
                   y empezar otro lote.
EXEC Dice_Hola;  -- Con esto llamamos al procedimiento para probarlo (se ejecuta).
```

El procedimiento que hemos creado no tiene parámetros por lo que la llamada incluye únicamente el nombre del procedimiento.

En este caso, como la llamada es la primera del lote (va detrás del GO) podíamos haber obviado la palabra EXEC y haber escrito directamente:

```
Dice_Hola
```

7.2. Con un parámetro de entrada

Ahora vamos a incluir un parámetro de entrada que indique la palabra que queremos que se escriba.

```
CREATE PROCEDURE Dice_Palabra @palabra CHAR(30)
AS
PRINT @palabra;
GO
-- Ahora vamos a crear el procedimiento creado con dos palabras diferentes:
EXEC Dice_Palabra 'Lo que quiera';
EXEC Dice_Palabra 'Otra cosa';
```

Después del nombre del procedimiento ponemos la lista de parámetros, observa que el parámetro se llama @palabra y tenemos que indicar su tipo de datos, en este caso será una cadena de 30 caracteres.

Después de la palabra AS empiezan las instrucciones que se ejecutarán, en este caso escribir (PRINT) el contenido del parámetro @palabra. El parámetro se rellenará con el valor indicado en la llamada.

Con el GO cerramos el lote para poder escribir más instrucciones en el script. Recuerda que CREATE PROCEDURE debe ser la única del lote.

En la llamada EXEC después del nombre del procedimiento se indica el valor con el que se quiere que se rellene el parámetro de entrada.

En el script hemos realizado dos llamadas, una con el valor 'Lo que quiera' y otra con el valor 'Otra cosa'. Luego el procedimiento después de ser creado se ejecutará dos veces, la primera escribirá *Lo que quiera* y la segunda *Otra cosa*.

¡Ojo! Si vuelves a ejecutar el script te va a dar error en el CREATE PROCEDURE porque el procedimiento ya está creado.

7.3. Con varios parámetros de entrada

Si queremos indicar varios parámetros los separamos por comas.

Antes del CREATE PROCEDURE hemos añadido la instrucción USE para asegurarnos que estemos trabajando con la base de datos Biblio, y la DROP PROCEDURE para borrar el procedimiento, de esta forma se puede ejecutar el script completo las veces que queramos. La primera vez (el procedimiento todavía no está creado) se comentará la DROP PROCEDURE para que no dé error (no se puede eliminar un procedimiento que no existe), pero las siguientes veces, antes de crearlo nos aseguramos que no existe eliminándolo previamente. Más adelante veremos otra forma más cómoda para poder ejecutar el script completo varias veces sin que dé error.

```
USE Biblio;
--DROP PROC VerUsuariosPoblacion; --La comentamos la primera vez que se
                                  ejecuta el script, después quitamos el comentario
GO
CREATE PROCEDURE VerUsuariosPoblacion @pob CHAR(30),@pro CHAR(30)
AS
SELECT * FROM usuarios WHERE poblacion=@pob AND provincia = @pro;
GO
EXEC VerUsuariosPoblacion 'Silla','Valencia'
```

Ahora el procedimiento creado tiene dos parámetros de entrada @pob y @pro, los dos cadenas de 30 caracteres.

El procedimiento lo que hace es obtener la lista de los usuarios de la población y provincia indicados en los parámetros.

En la llamada podemos indicar los valores de los parámetros de varias formas, indicando sólo el valor de los parámetros (en este caso los tenemos que indicar en el mismo orden en que están definidos) como en el ejemplo anterior, o bien indicando el nombre del parámetro así:

```
EXEC VerUsuariosPoblacion @pob='Silla', @pro='Valencia'
```

Indicar el nombre del parámetro en la llamada también nos permite indicar los valores en cualquier orden, la siguiente llamada es equivalente a la anterior, hemos invertido el orden de los parámetros en la llamada y se ejecuta igual:

```
EXEC VerUsuariosPoblacion @pro='Valencia', @pob='Silla'
```

En este procedimiento todos los parámetros son obligatorios, no deja llamar con un solo parámetro, por ejemplo la siguiente llamada da error:

```
EXEC VerUsuariosPoblacion 'Silla'
```

7.4. Con parámetros opcionales

Para definir un parámetro opcional tenemos que asignarle un valor por defecto en la definición del procedimiento.

```
DROP PROC VerUsuariosPoblacion2;
GO
CREATE PROCEDURE VerUsuariosPoblacion2 @pob CHAR(30),@pro CHAR(30)='Valencia'
AS
SELECT * FROM usuarios WHERE poblacion=@pob AND provincia = @pro;
GO
EXEC VerUsuariosPoblacion2 'Silla'
```

En este caso, el procedimiento tiene un parámetro obligatorio @pob y uno opcional @pro, es opcional porque después de su tipo de datos indicamos = *valor*, lo que significa que este parámetro será igual a este valor por defecto, luego en la llamada se podrá obviar o no el valor del parámetro, si la llamada no incluye un valor para ese parámetro, se rellenará con el valor por defecto.

En el ejemplo, en la llamada sólo hemos indicado un valor, para el primer parámetro, el parámetro opcional no lo hemos indicado y se rellenará con el valor *Valencia*.

Lo hemos podido hacer así porque el parámetro opcional es el último de la lista de parámetros. Cuando el parámetro opcional no es el último la llamada tiene que ser diferente, tenemos que nombrar los parámetros en la llamada, al menos a partir del parámetro opcional.

Por ejemplo:

```
DROP PROC VerUsuariosPoblacion3;
GO
CREATE PROCEDURE VerUsuariosPoblacion3 @a CHAR(2),@pob CHAR(30)='Pinto',@pro
CHAR(30)
AS
SELECT * FROM usuarios WHERE poblacion=@pob AND provincia = @pro;
GO
EXEC VerUsuariosPoblacion3  'ab',@pro='Madrid'
```

En este caso el parámetro opcional es el segundo de la lista de parámetros, cuando hacemos la llamada no podemos hacer como en otros lenguajes de dejar un hueco:

```
EXEC VerUsuariosPoblacion3  'ab', ,'Madrid'  Esto da error
```

La llamada anterior da error, debemos utilizar la palabra DEFAULT o indicar el nombre de todos los parámetros que van detrás del opcional (como está en el ejemplo propuesto).

```
EXEC VerUsuariosPoblacion3  'ab',DEFAULT,'Madrid'
```

7.5. Con parámetros de salida

Un procedimiento almacenado puede también devolver resultados, definiendo el parámetro como OUTPUT o bien utilizando la instrucción RETURN que veremos en el siguiente apartado.

Para poder recoger el valor devuelto por el procedimiento, en la llamada se tiene que indicar una variable donde guardar ese valor.

Ejemplo:

Definimos el procedimiento ultimo_contrato que nos devuelve la fecha en que se firmó el último contrato en una determinada oficina. El procedimiento tendrá pues dos parámetros, uno de entrada para indicar el número de la oficina a considerar y uno de salida que devolverá la fecha del contrato más reciente de entre los empleados de esa oficina.

```
USE Gestion
GO
CREATE PROC ultimo_contrato @ofi INT, @fecha DATETIME OUTPUT
AS
SET @fecha=(SELECT MAX(contrato) FROM empleados WHERE oficina=@ofi)
GO
```

Con @fecha DATETIME OUTPUT indicamos que el parámetro @fecha es de tipo fecha y de salida, el proceso que realice la llamada podrá recoger su valor después de ejecutar el procedimiento.

La asignación del valor al parámetro de salida se realiza mediante la asignación

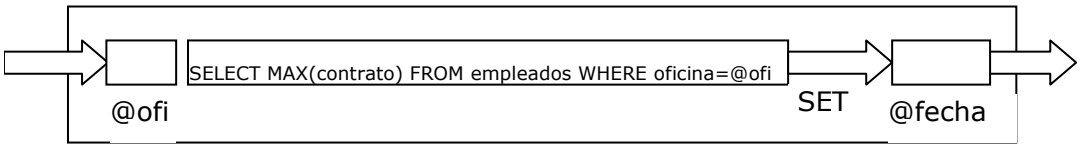
SET @fecha = valor y el valor será el devuelto por la consulta escalar que aparece entre paréntesis.

En la llamada, para los parámetros de salida, en vez de indicar un valor de entrada se indica un nombre de variable, variable que recogerá el valor devuelto por el procedimiento sin olvidar la palabra OUTPUT:

```
DECLARE @ultima AS DATETIME;
EXEC ultimo_contrato 12,@ultima OUTPUT;
PRINT @ultima;
```

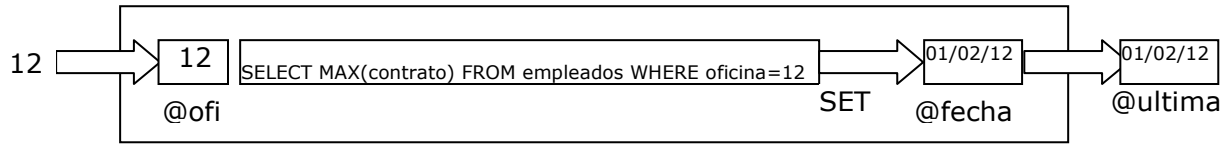
En este script primero declaramos una variable @ultima de tipo fecha, luego llamamos al procedimiento ultimo_contrato con el valor de entrada 12 y el valor devuelto por el procedimiento se guardará en la variable @ultima (ojo no olvides escribir OUTPUT).

Cuando definimos (creamos) el procedimiento, se crea con los dos parámetros vacíos:



Al declarar @ultima y ejecutar el procedimiento, el valor 12 que se indica en la llamada es recogido por @ofi, se ejecuta la SELECT con el valor sustituido y SET deja el resultado en @fecha, como @fecha es un parámetro de salida su contenido pasa a la variable indicada en la llamada (@ultima) y así el valor puede ser utilizado para por ejemplo visualizarlo con PRINT.

No se podría hacer un PRINT @fecha porque tanto @ofi como @fecha son internos al procedimiento, su alcance se limita a este, fuera no existen.



7.6. RETURN

RETURN ordena salir incondicionalmente de una consulta o procedimiento, se puede utilizar en cualquier punto para salir del procedimiento y las instrucciones que siguen a RETURN no se ejecutan. Además los procedimientos almacenados pueden devolver un valor entero mediante esta orden.

```
RETURN [expresion_entera]
```

expresion_entera es el valor entero que se devuelve.

A menos que se especifique lo contrario, todos los procedimientos almacenados del sistema devuelven el valor 0. Esto indica que son correctos y un valor distinto de cero indica que se ha producido un error.

Cuando se utiliza con un procedimiento almacenado, RETURN no puede devolver un valor NULL. Si un procedimiento intenta devolver un valor NULL (por ejemplo, al utilizar RETURN **@var** si **@var** es NULL), se genera un mensaje de advertencia y se devuelve el valor 0.

Si queremos recoger el valor de estado devuelto por el procedimiento la llamada debe ser distinta, y seguir el siguiente modelo:

```
EXECUTE @variable_donde_recogemos_estado = nombre_procedimiento @par, ...
```

Con el procedimiento del punto anterior no se puede utilizar esta forma de devolver un resultado porque lo que se devuelve es una fecha, no es un valor entero, pero si quisiéramos definir un procedimiento que nos devuelva el número de empleados de una oficina podríamos hacerlo de dos formas:

De la forma normal con un parámetro de salida:

```
USE Gestion
GO
CREATE PROC trabajadores @ofi INT, @num INT OUTPUT
AS
SET @num=(SELECT COUNT(*) FROM empleados WHERE oficina=@ofi)
GO
```

Para obtener el nº de empleados de la oficina 12 y recoger el resultado en la variable @var la llamada sería:

```
DECLARE @var INT;
EXEC trabajadores 12, @var OUTPUT
```

Utilizando return en vez de un parámetro de salida el procedimiento sería:

```
CREATE PROC trabajadores2 @ofi INT
AS
RETURN (SELECT COUNT(*) FROM empleados WHERE oficina=@ofi)
GO
```

Y la llamada:

```
DECLARE @var INT;
EXEC @var= trabajadores2 12
PRINT @var
```

Con PRINT @var nos visualiza el número de trabajadores de la oficina número 12.

Con esto hemos visto todos los tipos de parámetros que podemos tener en un procedimiento.

A continuación veremos otras instrucciones TRANSACT-SQL que se utilizan para desarrollar el código de los procedimientos y que también se pueden utilizar en cualquier script.

8. Instrucciones de control de flujo

8.1. IF… ELSE

Proporciona una ejecución condicional, permite ejecutar o no ciertas instrucciones dependiendo de si se cumple o no una determinada condición.

```
IF condicion
     { sentencia_sql | bloque_sql }
[ ELSE
     { sentencia_sql | bloque_sql } ]
```

Si la condición se cumple (da como resultado TRUE) se ejecuta la instrucción SQL o bloque de instrucciones que aparecen a continuación de la condición, si la condición no se cumple se ejecutan las sentencias que aparecen después de la palabra ELSE. El bloque ELSE es opcional.

Ejemplo: Si nos queremos guardar en un script todos los ejemplos para probarlos en cualquier momento, hemos visto que era conveniente antes de los CREATE PROCEDURE colocar un DROP PROCEDURE para que la instrucción CREATE no dé error si el procedimiento ya existe, pero la primera vez la instrucción DROP PROC nos dará error porque el procedimiento todavía no existe, así que lo mejor es ejecutar el DROP sólo si el procedimiento existe, utilizando la función object_id('nombre_de_objeto','tipo de objeto') que nos devuelve el id del objeto y NULL si el objeto no existe.

```
IF object_id('trabajadores', 'P') IS NOT NULL
            DROP PROCEDURE trabajadores;
GO
CREATE PROC trabajadores….
```

La función object_id(nb_objeto, [tipo_objeto]) nos devuelve el identificativo del objeto indicado mediante su nombre y tipo, para expresar el tipo de objeto utilizamos 'U' para tablas, 'P' para procedimientos almacenados, 'TR' para triggers, 'V' para vistas.

Para consultar más tipos :

http://technet.microsoft.com/en-us/library/ms190324(SQL.90).aspx

Por ejemplo object_id('clientes', 'U') nos devuelve el id de la tabla *clientes*, si la tabla no existe la función devuelve NULL.

Otro ejemplo. Ahora queremos eliminar el procedimiento trabajadores y si no existe que nos aparezca un mensaje indicándolo:

```
IF object_id('trabajadores', 'P') IS NOT NULL
      BEGIN
       PRINT 'Borramos el procedimiento'
            DROP PROC trabajadores
      END
ELSE PRINT 'El procedimiento no existe'
```

Cuando queremos definir un bloque de instrucciones utilizamos los delimitadores BEGIN..END. En este ejemplo si se cumple la condición queremos hacer dos cosas (PRINT y DROP) por lo que debemos definir un bloque con BEGIN…END.

Se pueden anidar varias sentencias IF (poner un IF dentro de otro) hasta el límite que permita la memoria.

Con la instrucción IF podemos utilizar la función EXISTS (sentencia_select) que funciona igual que el operador EXISTS que estudiamos en el tema de SUBCONSULTAS con la diferencia que aquí la sentencia_select no es una subconsulta por lo que no tenemos referencias externas, pero podemos utilizar variables.

Por ejemplo:

```
USE Gestion8
DECLARE @ofi as int
SET @ofi = 33
IF EXISTS(SELECT * FROM empleados WHERE oficina = @ofi)
  BEGIN
     SELECT 'los empleados de la oficina ' + STR(@ofi,2) + ' son:'
     SELECT numemp, nombre FROM empleados WHERE oficina = @ofi
  END
ELSE
  PRINT 'La oficina ' + STR(@ofi,2) + ' no tiene empleados'
```

Declaramos una variable @ofi para almacenar el nº de la oficina a recuperar.

Nos guardamos en la variable el valor 33.

La función EXISTS devuelve verdadero si la SELECT devuelve al menos una fila (si existe un empleado de la oficina indicada en @ofi), y en tal caso se escribirá el mensaje *Los empleados...son:* y a continuación la lista de empleados de esa oficina.

Si la función EXISTS devuelve false (ELSE) se escribirá el mensaje *La oficina… no tiene empleados*.

Para el mensaje, en este caso hemos utilizado SELECT para que aparezca el mensaje en la misma pestaña que los empleados de la oficina.

Observa que para poder concatenar el nº de la oficina con el resto del texto del mensaje hemos tenido que convertir el valor entero a cadena mediante la función STR().

8.2. TRY... CATCH

Definición

La estructura TRY…CATCH implementa un mecanismo de control de errores para Transact-SQL, permite incluir un grupo de instrucciones Transact-SQL en un bloque TRY. Si se produce un error en el bloque TRY, el control se transfiere a otro grupo de instrucciones que está incluido en un bloque CATCH.

```
BEGIN TRY
     {sentencia_sql|bloque_sql}
END TRY
BEGIN CATCH
      [{sentencia_sql|bloque_sql}]
END CATCH [ ; ]
```

A la hora de utilizar esta estructura existen ciertas reglas a considerar:

* Cada construcción TRY…CATCH debe encontrarse en un solo lote.

Por ejemplo este script no funciona porque el GO termina el lote que contiene el TRY antes del bloque catch:

```
BEGIN TRY
     PRINT 'entra en try'
     GO
END TRY
BEGIN CATCH
      PRINT 'Entra en catch'
END CATCH  ;
```

* Un bloque CATCH debe seguir inmediatamente a un bloque TRY.

Por ejemplo el siguiente código da error al llegar al segundo PRINT:

```
BEGIN TRY
     PRINT 'entra en try'
END TRY
PRINT 'sale de try'
BEGIN CATCH
      PRINT 'Entra en catch, error'
END CATCH  ;
```

* Si no hay errores en el código incluido en un bloque TRY, cuando la última instrucción de este bloque ha terminado de ejecutarse, el control se transfiere a la instrucción inmediatamente posterior a la instrucción END CATCH asociada.

Por ejemplo:

```
BEGIN TRY
     PRINT 'entra en try'
     PRINT 12/3
     PRINT 'correcto'
END TRY
BEGIN CATCH
      PRINT 'Entra en catch, error'
END CATCH  ;
```

El siguiente código entra en el TRY se ejecuta normalmente y termina sin entrar en el bloque catch.

* Si hay un error en el código incluido en el bloque TRY, el control se transfiere a la primera instrucción del bloque CATCH asociado.

```
BEGIN TRY
     PRINT 'entra en try'
     PRINT 12/0
     PRINT 'correcto'
END TRY
BEGIN CATCH
      PRINT 'Entra en catch, error'
END CATCH  ;
```

En este caso la división por cero va a dar error y se pasará el control al bloque catch, el PRINT 'correcto' no se ejecutará.

* Si la instrucción END CATCH es la última instrucción de un procedimiento almacenado o desencadenador, y se ha entrado en el bloque catch, el control se devuelve a la instrucción que llamó al procedimiento almacenado o activó el desencadenador.

* Las construcciones TRY…CATCH se pueden anidar.

Las construcciones TRY…CATCH capturan los errores no controlados de los procedimientos almacenados o desencadenadores ejecutados por el código del bloque TRY. Alternativamente, los procedimientos almacenados o desencadenadores pueden contener sus propias construcciones TRY…CATCH para controlar los errores generados por su código. Por ejemplo, cuando un bloque TRY ejecuta un procedimiento almacenado y se produce un error en éste, el error se puede controlar de las formas siguientes:

- Si el procedimiento almacenado no contiene su propia construcción TRY…CATCH, el error devuelve el control al bloque CATCH asociado al bloque TRY que contiene la instrucción EXECUTE.
- Si el procedimiento almacenado contiene una construcción TRY…CATCH, el error transfiere el control al bloque CATCH del procedimiento almacenado. Cuando finaliza el código del bloque CATCH, el control se devuelve a la instrucción inmediatamente posterior a la instrucción EXECUTE que llamó al procedimiento almacenado.

* No se puede utilizar un instrucción GOTO para entrar en un bloque TRY o CATCH pero se puede utilizar para saltar a una etiqueta dentro del mismo bloque TRY o CATCH, o bien para salir de un bloque TRY o CATCH.

* TRY…CATCH no se puede utilizar en una función definida por el usuario.

Recuperar información sobre errores

En el ámbito de un bloque CATCH, se pueden utilizar las siguientes funciones del sistema para obtener información acerca del error que provocó la ejecución del bloque CATCH:

- ERROR_NUMBER() devuelve el número del error.
- ERROR_SEVERITY() devuelve la gravedad.
- ERROR_STATE() devuelve el número de estado del error.
- ERROR_PROCEDURE() devuelve el nombre del procedimiento almacenado o desencadenador donde se produjo el error.
- ERROR_LINE() devuelve el número de línea de la rutina que provocó el error.
- ERROR_MESSAGE() devuelve el texto completo del mensaje de error. Este texto incluye los valores suministrados para los parámetros reemplazables, como longitudes, nombres de objetos u horas.

Estas funciones devuelven NULL si se las llama desde fuera del ámbito del bloque CATCH. Con ellas se puede recuperar información sobre los errores desde cualquier lugar dentro del ámbito del bloque CATCH. Por ejemplo, en la siguiente secuencia de comandos se muestra un procedimiento almacenado que contiene funciones de control de errores. Se llama al procedimiento almacenado en el bloque CATCH de una construcción TRY…CATCH y se devuelve información sobre el error.

Ejemplo, vamos a definir un procedimiento para calcular el precio unitario a partir de un importe y una cantidad y a continuación probaremos si funciona:

```
IF OBJECT_ID ('CalculaPrecio', 'P' ) IS NOT NULL
    DROP PROCEDURE CalculaPrecio;
GO
-- Creamos el procedimiento
CREATE PROCEDURE CalculaPrecio @importe MONEY,@cant INT,@precio MONEY OUTPUT
AS
BEGIN TRY
    SET @precio=@importe/@cant;
END TRY
BEGIN CATCH
      IF ERROR_NUMBER() = 8134 SET @precio=0
      ELSE SELECT ERROR_NUMBER() as ErrorNumero,ERROR_MESSAGE() as MensajeDeError;
END CATCH;
GO       -- FIN del procedimiento
-- Lo utilizamos
DECLARE @resultado MONEY
EXEC CalculaPrecio 1000, 10, @resultado OUTPUT
SELECT 'resul', @resultado
EXEC CalculaPrecio 1000, 0, @resultado OUTPUT
SELECT 'resul', @resultado
```

Si al llamar al procedimiento le pasamos una cantidad igual a cero, la división provocará un error, se pasará el control al bloque CATCH, en este bloque se evalúa si el error corresponde al error de división por cero (8134), si es así el procedimiento devuelve un precio igual a cero, si no se envía un mensaje para avisar del error.

Errores controlados por TRY...CATCH

TRY…CATCH detecta todos los errores de ejecución que tienen una gravedad mayor de 10 y que no cierran la conexión de la base de datos.

TRY…CATCH no detecta:

- Advertencias o mensajes informativos que tienen gravedad 10 o inferior.
- Errores que tienen gravedad 20 o superior que detienen el procesamiento de las tareas de SQL Server Database Engine en la sesión. Si se produce un error con una gravedad 20 o superior y no se interrumpe la conexión con la base de datos, TRY…CATCH controlará el error.
- Atenciones, como solicitudes de interrupción de clientes o conexiones de cliente interrumpidas.
- Cuando el administrador del sistema finaliza la sesión mediante la instrucción KILL.
- Un bloque CATCH no controla los siguientes tipos de errores cuando se producen en el mismo nivel de ejecución que la construcción TRY…CATCH:
    - Errores de compilación, como errores de sintaxis, que impiden la ejecución de un lote.
    - Errores que se producen durante la precompilación de instrucciones, como errores de resolución de nombres de objeto que se producen después de la compilación debido a una resolución de nombres diferida.

Estos errores se devuelven al nivel de ejecución del lote, procedimiento almacenado o desencadenador.

En el ejemplo siguiente se muestra cómo la construcción TRY…CATCH no captura un error de resolución de nombre de objeto generado por una instrucción SELECT, sino que es el bloque CATCH el que lo captura cuando la misma instrucción SELECT se ejecuta dentro de un procedimiento almacenado (a un nivel inferior).

```
USE Gestion;
GO
BEGIN TRY
    SELECT * FROM TablaQueNoExiste;
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() as ErrorNumero,ERROR_MESSAGE() as MensajeDeError;
END CATCH
```

Intentamos ejecutar una SELECT de una tabla que no existe en la base de datos. El error no se captura y el control se transfiere fuera de la construcción TRY…CATCH, al siguiente nivel superior, en este caso salta el mensaje de error del sistema.

Al ejecutar la misma instrucción SELECT dentro de un procedimiento almacenado, el error se producirá en un nivel inferior al bloque TRY y la construcción TRY…CATCH controlará el error.

```
-- Creamos el procedimiento.
CREATE PROCEDURE TablaInexistente
AS
    SELECT * FROM TablaQueNoExiste;
GO
-- Utilizamos el procedimiento
BEGIN TRY
    EXECUTE TablaInexistente
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() as ErrorNumero,ERROR_MESSAGE() as MensajeDeError;
END CATCH;
```

En este caso como el error se produce dentro del procedimiento, el control del error se devuelve al nivel superior (el que ha realizado el EXECUTE) por lo que es capturado por el TRY y entra en el bloque CATCH, en vez de que salte el mensaje de error del sistema saldrá el nuestro.

8.3. WHILE – BREAK- CONTINUE

Esta instrucción permite definir un bucle que repite una sentencia o bloque de sentencias mientras se cumpla una determinada condición.

```
WHILE condicion
     { sentencia_sql | bloque_sql }
     | BREAK
     | CONTINUE
```

Podemos anidar bucles, colocar un bucle WHILE dentro de otro

BREAK Produce la salida del bucle WHILE más interno. La instrucción BREAK interna sale al siguiente bucle más externo. Todas las instrucciones que se encuentren después del final del bucle interno se ejecutan primero y después se reinicia el siguiente bucle más externo.

CONTINUE
Hace que se reinicie el bucle WHILE y omite las instrucciones que haya después de la palabra clave CONTINUE.

Por ejemplo, tenemos los siguientes bucles anidados:

```
WHILE condicion1              -- Bucle 1
      Instrucciones_1_1
      WHILE condicion2              -- Bucle 2
            Intrucciones2_1
            If condicion3 BREAK
            Instrucciones2_2
            IF condicion4 CONTINUE
            Instrucciones2_3
      Instrucciones1_2
```

Las instrucciones se ejecutarían en este orden:
Preguntamos por condicion1:
Si condicion1 se cumple:
Se ejecuta el bloque Instrucciones1_1
Preguntamos por condicon2:
Si se cumple condicion2:
Se ejecuta el bloque de instrucciones Instrucciones_2_1.
Si condición3 se cumple:
Salimos del bucle 2,
Se ejecuta el bloque Instrucciones1_2
Y se vuelve a preguntar por condicion1.
Si condición3 no se cumple:
Se ejecuta el bloque instrucciones2_2
Si se cumple condicion4:
Saltamos el bloque Instrucciones2_3
Y se vuelve a preguntar por condicion2
Si no se cumple condicion4:
Se ejecuta el bloque Instrucciones2_3
Y se vuelve a preguntar por condicion2
Si condicion2 no se cumple:
Se ejecuta el bloque Instrucciones 1_2
Se vuelve a preguntar por condicion1
Si condicion1 no se cumple:
Hemos terminado

Cada bloque, si contiene más de una instrucción debe estar encerrado entre BEGIN ..END.

8.4. WAITFOR

Bloquea la ejecución de un lote, un procedimiento almacenado o una transacción hasta alcanzar la hora o el intervalo de tiempo especificado, o hasta que una instrucción especificada modifique o devuelva al menos una fila. Nosotros estudiaremos los dos primeros casos.

```
WAITFOR {DELAY 'tiempo_a_transcurrir'
       |TIME 'fechaHora_de_ejecucion'}
```

DELAY permite indicar un período de tiempo especificado (hasta un máximo de 24 horas) que debe transcurrir antes de la ejecución de un lote, un procedimiento almacenado o una transacción.

'tiempo_a_transcurrir' Es el período de tiempo que hay que esperar, se puede especificar en uno de los formatos aceptados para el tipo de datos datetime o como una variable local. No se pueden especificar fechas; por lo tanto, no se permite la parte de fecha del valor datetime.

TIME permite indicar la hora especifica a la que se ejecuta el lote, el procedimiento almacenado o la transacción.

'fechaHora_de_ejecucion' Es la hora a la que termina la instrucción WAITFOR por tanto a la que empieza la ejecución de las instrucciones siguientes a WAITFOR. Se puede especificar en uno de los formatos aceptados para el tipo de datos datetime o como una variable local. No se pueden especificar fechas; por lo tanto, no se permite la parte de fecha del valor datetime.

Cada instrucción WAITFOR tiene un subproceso asociado. Si se especifica un gran número de instrucciones WAITFOR en el mismo servidor, se pueden acumular muchos subprocesos a la espera de que se ejecuten estas instrucciones. SQL Server supervisa el número de subprocesos asociados con las instrucciones WAITFOR y selecciona aleatoriamente algunos de estos subprocesos para salir si el servidor empieza a experimentar la falta de subprocesos.

Ejemplo:

```
PRINT CONVERT(CHAR(8),Getdate(),108);
WAITFOR DELAY '00:00:03'
PRINT CONVERT(CHAR(8),Getdate(), 108);
WAITFOR DELAY '00:03'
PRINT CONVERT(CHAR(8),Getdate(), 108);
```

Visualiza la hora actual, espera 3 segundos y vuelve a visualizar la hora actual, después espera 3 minutos y vuelve a visualizar la hora actual. Se ha utilizado la función CONVERT con el estilo 108 para que aparezca sólo la hora y con segundos.

8.5. GOTO

Altera el flujo de ejecución y lo dirige a una etiqueta.
Las etiquetas se indican mediante un nombre seguido del carácter:

```
GOTO NombreEtiqueta
```

Ejemplo:

```
IF Condicion GOTO etiq
----
Etiq:
----
----
```

Es una instrucción a evitar porque puede llevar a redactar programas no estructurados.

9. CURSORES

9.1. Introducción

Las aplicaciones, especialmente las aplicaciones interactivas en línea, no siempre trabajan de forma eficaz con el conjunto de resultados devuelto por una SELECT si lo toman como una unidad. Estas aplicaciones necesitan un mecanismo que trabaje con una fila o un pequeño bloque de filas cada vez. Los cursores proporcionan dicho mecanismo.

Un cursor no es una instrucción propiamente dicho sino una estructura que me permite recuperar filas de una consulta y procesarlas una a una.

Se puede asignar un cursor a una variable o parámetro con un tipo de datos **cursor**.

Los cursores amplían el procesamiento de los resultados porque:

- Permiten situarse en filas específicas del conjunto de resultados.
- Recuperan una fila o un bloque de filas de la posición actual en el conjunto de resultados.
- Aceptan modificaciones de los datos de las filas en la posición actual del conjunto de resultados.
- Aceptan diferentes grados de visibilidad para los cambios que realizan otros usuarios en la información de la base de datos que se presenta en el conjunto de resultados.
- Proporcionan instrucciones Transact-SQL en secuencias de comandos, procedimientos almacenados y acceso de desencadenadores a los datos de un conjunto de resultados.

9.2. Operativa

En SQL Server se pueden crear cursores desde instrucciones Transact-SQL o desde diferentes APIs de acceso a datos. Nosotros limitaremos nuestro estudio a los primeros (y a la sintaxis SQL-92) pero en cualquier caso se utiliza el mismo esquema de proceso común a todos los cursores de SQL Server:

1. Declarar el cursor. Asignar el cursor al conjunto de resultados de una instrucción Transact-SQL definiendo características del cursor como, por ejemplo, si sus filas se pueden actualizar.
2. Ejecutar la instrucción Transact-SQL para llenar el cursor.
3. Recuperar las filas del cursor que se quiera. La operación de recuperación de una fila o un bloque de filas de un cursor recibe el nombre de recuperación. La realización de una serie de recuperaciones para obtener filas, ya sea hacia adelante o hacia atrás, recibe el nombre de desplazamiento.
4. Existe la opción de realizar operaciones de modificación (actualización o eliminación) en la fila de la posición actual del cursor.
5. Cerrar el cursor.

9.3. Declarar el cursor - DECLARE...CURSOR

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
     FOR select_statement
     [ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ] } ]
[;]
```

cursor_name Es el nombre del cursor.

INSENSITIVE Define un cursor que hace una copia temporal de los datos que utiliza. Todas las solicitudes que se realizan al cursor se responden desde esta tabla temporal de **tempdb**; por tanto, las modificaciones realizadas por otros usuarios en las tablas base no se reflejan en los datos devueltos por las operaciones de recuperación realizadas en el cursor y además este cursor no admite modificaciones. Cuando se utiliza la sintaxis de SQL-92, si se omite INSENSITIVE, las eliminaciones y actualizaciones confirmadas realizadas en las tablas subyacentes (por cualquier usuario) se reflejan en recuperaciones posteriores.

SCROLL Especifica que están disponibles todas las opciones de recuperación (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE). Si no se especifica SCROLL en una instrucción DECLARE CURSOR de SQL-92, la única opción de recuperación que se admite es NEXT.

select_statement Es una instrucción SELECT estándar que define el conjunto de resultados del cursor. Las palabras clave COMPUTE, COMPUTE BY, FOR BROWSE e INTO no están permitidas en el argumento *select_statement* de una declaración de cursor.

READ ONLY Evita que se efectúen actualizaciones a través de este cursor.

UPDATE [OF *column_name* [,...*n*]] Define las columnas que se pueden actualizar en el cursor. Si se especifica OF *column_name* [**,**...*n*], sólo las columnas enumeradas admiten modificaciones. Si se especifica UPDATE sin indicar una lista de columnas, se pueden actualizar todas las columnas.

Ejemplos:

```
DECLARE vend_cursor CURSOR FOR SELECT * FROM Vendors;
```

Declara el cursor *vend_cursor* sobre la SELECT. Como no especifica nada más el cursor sólo se podrá recorrer fila tras fila y siempre hacia adelante (no incluye la cláusula SCROLL). La recuperación de filas se realiza sobre las filas originales (no incluye INSENSITIVE) por lo que los datos serán más actualizados pero también será un cursor más lento.

```
DECLARE vend_cursor INSENSITIVE CURSOR FOR SELECT * FROM Vendors FOR READ ONLY;
```

Declara el cursor vend_cursor sobre la SELECT. Como incluye INSENSITIVE se realiza sobre una copia temporal de los datos, será más rápido. Como incluye FOR READ ONLY los datos no se podrán actualizar.

```
DECLARE vend_cursor CURSOR FOR SELECT * FROM Vendors
   FOR UPDATE  OF name, birth_date;
```

Declara el cursor vend_cursor sobre la SELECT. El cursor será como el del primer ejemplo pero sólo se podrán actualizar las columnas name y birth_date.

9.4. El tipo de datos CURSOR

SQL Server permite variables con el tipo de datos **CURSOR**.

Para asignar un cursor a una variable podemos usar cualquier de estos métodos:

```
DECLARE @MiVariable CURSOR;
DECLARE MiCursor CURSOR FOR SELECT nombre FROM Clientes;
SET @MiVariable = MiCursor;
```

O bien

```
DECLARE @MiVariable  CURSOR;
SET @MiVariable = CURSOR
            FOR SELECT nombre  FROM  Clientes;
```

Tras asociar un cursor a una variable **cursor**, se puede utilizar la variable **cursor** en lugar del nombre del cursor en las instrucciones de cursor de Transact-SQL.

9.5. Rellenar el cursor - OPEN

```
OPEN {  cursor_name  | cursor_variable_name }
```

Abre el cursor y lo llena ejecutando la instrucción Transact-SQL especificada en la instrucción DECLARE CURSOR o SET variablecursor.

Ejemplos:

```
OPEN Micursor;
```

```
OPEN @MiVariable;
```

9.6. Cerrar el cursor - CLOSE

Para cerrar un cursor tenemos la instrucción CLOSE.

```
CLOSE { cursor_name | cursor_variable_name }
```

Cierra un cursor abierto mediante la liberación del conjunto actual de resultados y todos los bloqueos de cursor mantenidos en las filas en las que está colocado. CLOSE deja las estructuras de datos accesibles para que se puedan volver a abrir, pero las recuperaciones y las actualizaciones posicionadas no se permiten hasta que se vuelva a abrir el cursor. CLOSE debe ejecutarse en un cursor abierto, por lo que no se permite en cursores que sólo están declarados o que ya están cerrados.

9.7. Liberar el cursor - DEALLOCATE

Para liberar el cursor y los recursos asociados al mismo tenemos la instrucción DEALLOCATE.

```
DEALLOCATE { cursor_name | @cursor_variable_name }
```

Quita una referencia a un cursor. Cuando se ha quitado la última referencia al cursor, Microsoft SQL Server libera las estructuras de datos que componen el cursor.

Las instrucciones que realizan operaciones sobre cursores utilizan un nombre de cursor o una variable de cursor para hacer referencia al cursor. DEALLOCATE quita la asociación existente entre un cursor y el nombre del cursor o la variable de cursor. Si un nombre o variable es el último que hace referencia a un cursor, se quita el cursor y se liberan los recursos que utiliza.

9.8. Recorrer el cursor - FETCH

Para recorrer y recuperar las filas de un cursor tenemos la instrucción FETCH.

```
FETCH [ [NEXT |PRIOR |FIRST |LAST |ABSOLUTE { n | @nvar } |RELATIVE { n | @nvar}]
        FROM ]
      { cursor_name  | cursor_variable_name }
      [ INTO @variable_name [ ,...n ] ] [;]
```

NEXT Devuelve la fila de resultados inmediatamente posterior a la fila actual, y aumenta la fila actual a la fila devuelta. Si FETCH NEXT es la primera operación de recuperación en un cursor (después del OPEN), se devuelve la primera fila del conjunto de resultados. NEXT es la opción predeterminada para la recuperación de cursores.

PRIOR Devuelve la fila de resultados inmediatamente anterior a la fila actual, y reduce la fila actual a la fila devuelta. Si FETCH PRIOR es la primera operación de recuperación en un cursor, no se devuelve ninguna fila y el cursor queda posicionado delante de la primera fila.

FIRST Devuelve la primera fila del cursor y la convierte en la fila actual.

LAST Devuelve la última fila del cursor y la convierte en la fila actual.

ABSOLUTE { *n* | @*nvar*} Si *n* o *@nvar* es positivo, se devuelve la fila *n* desde el principio del cursor y se convierte en la nueva fila actual. Si *n* o *@nvar* es negativo, se devuelve la fila *n* anterior al final del cursor y se convierte en la nueva fila actual. Si *n* o *@nvar* es 0, no se devuelve ninguna fila. *n* debe ser una constante entera y *@nvar* debe ser **smallint**, **tinyint** o **int**.

RELATIVE { *n* | @*nvar*} Si *n* o *@nvar* es positivo, se devuelve la fila *n* posterior a la fila actual y se convierte en la nueva fila actual. Si *n* o *@nvar* es negativo, se devuelve la fila *n* anterior a la fila actual y se convierte en la nueva fila actual. Si *n* o *@nvar* es 0, se devuelve la fila actual. Si se especifica FETCH RELATIVE con *n* o *@nvar* establecido en un número negativo o en 0 en la primera operación de recuperación que se realiza en un cursor, no se devuelven filas. *n* debe ser una constante entera y *@nvar* debe ser **smallint**, **tinyint** o **int**.

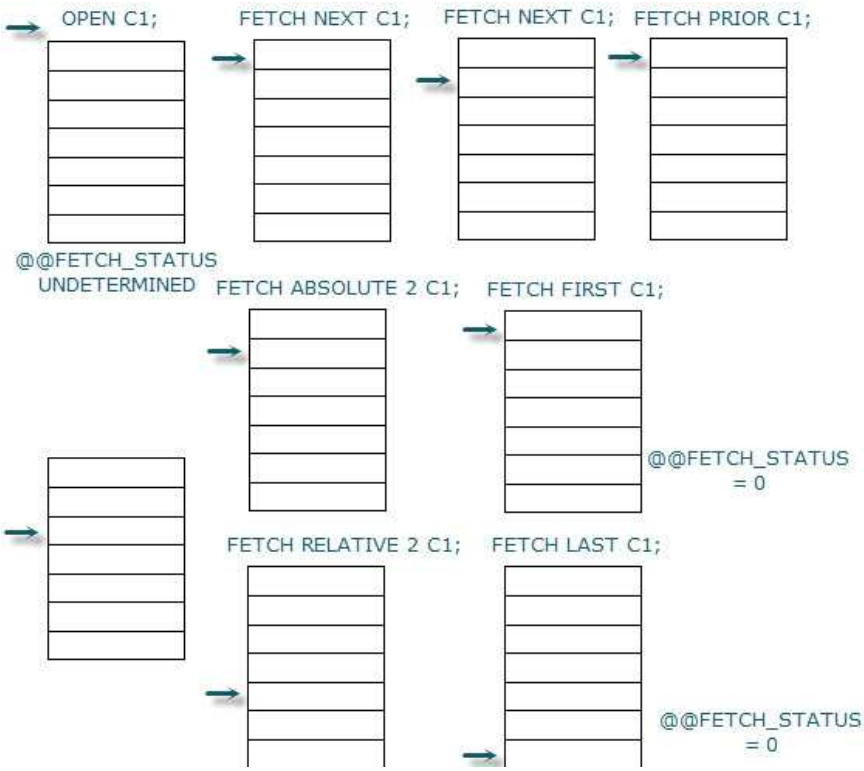
Nota. Si no se especifica la opción SCROLL en una instrucción DECLARE CURSOR de SQL-92, NEXT es la única opción admitida para FETCH. Si se especifica SCROLL en una instrucción DECLARE CURSOR de SQL-92, se admiten todas las opciones de FETCH.

cursor_name Es el nombre del cursor abierto desde el que se debe realizar la recuperación.

@cursor_variable_name Es el nombre de una variable de cursor que hace referencia al cursor abierto desde el que se va efectuar la recuperación.

INTO @*variable_name*[,*...n*] Permite colocar en variables locales los datos de las columnas de una recuperación. Todas las variables de la lista, de izquierda a derecha, están asociadas a las columnas correspondientes del conjunto de resultados del cursor. El tipo de datos de cada variable tiene que coincidir o ser compatible con la conversión implícita del tipo de datos de la columna correspondiente del conjunto de resultados. El número de variables debe coincidir con el número de columnas de la lista de selección del cursor.

En la siguiente imagen tienes un ejemplo de cómo se va moviendo el puntero del cursor que señala la fila actual según la instrucción ejecutada.



Para poder controlar el final del cursor tenemos la función @@FETCH_STATUS que informa del estado de la última instrucción FETCH ejecutada.

Esta información de estado se debe utilizar para determinar la validez de los datos devueltos por una instrucción FETCH antes de iniciar cualquier operación con esos datos.

Si la instrucción FETCH se ejecutó correctamente y se recuperó una fila de resultados, el valor devuelto es 0.

Ejemplos:

```
DECLARE Employee_Cursor CURSOR FOR
SELECT EmployeeID, Title FROM Employees;
OPEN Employee_Cursor;
FETCH NEXT FROM Employee_Cursor;
WHILE @@FETCH_STATUS = 0
   BEGIN
      FETCH NEXT FROM Employee_Cursor;
   END;
CLOSE Employee_Cursor;
DEALLOCATE Employee_Cursor;
GO
```

Este ejemplo es similar al ejemplo anterior, con la diferencia de que la salida de las instrucciones FETCH se almacena en variables locales en lugar de devolverse directamente al cliente. La instrucción PRINT combina las variables en una misma cadena y la devuelve al cliente.

```
DECLARE @LastName varchar(50), @FirstName varchar(50)

DECLARE contact_cursor CURSOR
   FOR SELECT LastName, FirstName FROM Contacts WHERE LastName LIKE 'B%'
            ORDER BY LastName, FirstName
OPEN contact_cursor
-- Perform the first fetch and store the values in variables.
-- Note: The variables are in the same order as the columns
-- in the SELECT statement.

FETCH NEXT FROM contact_cursor INTO @LastName, @FirstName
-- Check @@FETCH_STATUS to see if there are any more rows to fetch.
WHILE @@FETCH_STATUS = 0
BEGIN
   -- Concatenate and display the current values in the variables.
   PRINT 'Contact Name: ' + @FirstName + ' ' +  @LastName
   -- This is executed as long as the previous fetch succeeds.
   FETCH NEXT FROM contact_cursor INTO @LastName, @FirstName
END
CLOSE contact_cursor
DEALLOCATE contact_cursor
GO
```

Referencias.

Cursors http://technet.microsoft.com/en-gb/library/ms191179(v=sql.90).aspx

DECLARE CURSOR http://technet.microsoft.com/en-gb/library/ms180169(v=sql.90).aspx

OPEN http://technet.microsoft.com/en-gb/library/ms190500(v=sql.90).aspx

FETCH http://technet.microsoft.com/en-gb/library/ms180152(v=sql.90).aspx