

- ▶ **Introducción a Transact SQL**
- ▶ **Programación con Transact SQL**
- ▶ **Fundamentos de Transact SQL**
  - Primeros pasos con Transact SQL
  - Scripts y lotes.
- ▶ **Tipos de datos en Transact SQL**
  - Tipos de datos numéricos.
  - Tipos de datos de carácter.
  - Tipos de datos de fecha.
  - Tipos de datos binarios.
  - Tipo de datos XML.
  - Otros tipos de datos.
  - Tipos de datos personalizados.
- ▶ **Variables en Transact SQL**
  - Declarar variables en Transact SQL
  - Asignar variables en Transact SQL
- ▶ **Equivalencia de datos de SQL Server y .NET**
- ▶ **Operadores en Transact SQL**
- ▶ **Estructuras de control en Transact SQL**
  - Estructura IF
  - Estructura CASE
  - Bucle WHILE
  - Estructura GOTO
- ▶ **Control de errores en Transact SQL**
  - Uso de TRY CATCH
  - Funciones especiales de Error
  - La variable de sistema @@ERROR
  - Generar un error con RAISERROR
- ▶ **Consultar datos en Transact SQL**
  - La sentencia SELECT
  - La cláusula WHERE
  - La cláusula ORDER BY
- ▶ **Consultas agregadas**
  - La cláusula GROUP BY
  - La cláusula HAVING
  - AVG
  - Count
  - Max, Min
  - Sum
  - Uso de Select TOP con consultas agregadas
- ▶ **Select FOR XML**
  - Cláusula FOR XML.
  - Campos y variables XML.
- ▶ **Operaciones con conjuntos.**
  - UNION
  - EXCEPT
  - INTERSECT
- ▶ **Insertar datos en Transact SQL**

- Inserción individual de filas.
- Insertción múltiple de filas.
- Insertción de valores por defecto.
- Clausula OUTPUT

- ▶ **Actualizar datos en Transact SQL**

- Update
- Update INNER JOIN
- Clausula OUTPUT

- ▶ **Borrar datos en Transact SQL**

- Delete
- Clausula OUTPUT
- Truncate Table

- ▶ **Transacciones en Transact SQL**

- Concepto de transaccion
- Transacciones implícitas y explícitas
- Transacciones anidadas.
- Puntos de recuperacion

- ▶ **Procedimientos almacenados en Transact SQL**

- ▶ **Funciones en Transact SQL**

- Funciones escalares
- Funciones en línea
- Funciones en línea de múltiples sentencias

- ▶ **Funciones integradas de Transact SQL (I)**

- Cast y Convert
- IsNull
- COALESCE
- GetDate y GetUTCDate

- ▶ **Triggers en Transact SQL**

- Trigger DML
- Trigger DDL

- ▶ **Cursor en Transact SQL**

- ▶ **SQL dinámico en Transact SQL**

- La instrucción comando EXECUTE
- El procedimiento almacenado sp\_executesql

## Introducción a Transact SQL

**SQL** es un lenguaje de consulta para los sistemas de bases de datos relacionales, pero que no posee la potencia de los lenguajes de programación.

Para abordar el presente tutorial con mínimo de garantías es necesario conocer previamente **SQL**.

Podemos acceder a un completo tutorial de SQL desde [AQUI](#).

**Transact SQL** es el lenguaje de programación que proporciona SQL Server para ampliar SQL con los elementos característicos de los lenguajes de programación: variables, sentencias de control de flujo, bucles ...

Cuando se desea realizar una aplicación completa para el manejo de una base de datos relacional, resulta necesario utilizar alguna herramienta que soporte la capacidad de consulta del SQL y la versatilidad de los lenguajes de programación tradicionales. **Transact SQL** es el lenguaje de programación que proporciona **SQL Server** para extender el SQL estándar con otro tipo de instrucciones.

**Transact SQL** existe desde las primeras versiones de SQL Server, si bien a lo largo de este tutorial nos centraremos en la versión **SQL Server 2005**.

### ¿Que vamos a necesitar?

Para poder seguir este tutorial correctamente necesitaremos tener los siguientes elementos:

- Un servidor **SQL Server 2005**. Podemos descargar gratuitamente la versión SQL Server Express desde el siguiente enlace. [SQL Server 2005 Express](#).
- Herramientas cliente de SQL Server. Recomendamos:
  - [Microsoft SQL Server Management Studio](#)
  - [Toad para SQL Server](#)

La instalación y configuración de SQL Server está fuera del alcance de este tutorial. Si bien podemos acceder a una [guía básica de instalación desde este enlace](#).

---

# Programación con Transact SQL

## Introducción

SQL es un lenguaje de consulta para los sistemas de bases de datos relacionales, pero que no posee la potencia de los lenguajes de programación. No permite el uso de variables, estructuras de control de flujo, bucles ... y demás elementos característicos de la programación. No es de extrañar, **SQL es un lenguaje de consulta, no un lenguaje de programación.**

Sin embargo, SQL es la herramienta ideal para trabajar con bases de datos. Cuando se desea realizar una aplicación completa para el manejo de una base de datos relacional, resulta necesario utilizar alguna herramienta que soporte la capacidad de consulta del SQL y la versatilidad de los lenguajes de programación tradicionales. Transact SQL es el lenguaje de programación que proporciona Microsoft SQL Server para extender el SQL estándar con otro tipo de instrucciones y elementos propios de los lenguajes de programación .

Con **Transact SQL** vamos a poder programar las unidades de programa de la base de datos **SQL Server**, estas son:

- Procedimientos almacenados
- Funciones
- Triggers
- Scripts

Pero además Transact SQL nos permite realizar programas sobre las siguientes herramientas de SQL Server:

- Service Broker
-

# Fundamentos de Transact SQL

## Primeros pasos con Transact SQL

Para programar en **Transact SQL** es necesario conocer sus fundamentos.

Como introducción vamos a ver algunos elementos y conceptos básicos del lenguaje.

- **Transact SQL** no es CASE-SENSITIVE, es decir, no diferencia mayúsculas de minúsculas como otros lenguajes de programación como C o Java.
- Un comentario es una aclaración que el programador incluye en el código. Son soportados 2 estilos de comentarios, el de línea simple y de multilínea, para lo cual son empleados ciertos caracteres especiales como son:
  - -- Para un comentario de línea simple
  - /\* ... \*/ Para un comentario de varias líneas
- Un literal es un valor fijo de tipo numérico, carácter, cadena o lógico no representado por un identificador (es un valor explícito).
- Una variable es un valor identificado por un nombre (identificador) sobre el que podemos realizar modificaciones. En **Transact SQL** los identificadores de variables deben comenzar por el carácter @, es decir, el nombre de una variable debe comenzar por @. Para declarar variables en **Transact SQL** debemos utilizar la palabra clave **declare**, seguido del identificador y tipo de datos de la variable.

Veamos algunos ejemplos:

```
-- Esto es un comentario de línea simple

/*
Este es un comentario con varias líneas.
Conjunto de Líneas.
*/
declare @nombre varchar(50)-- declare declara una variable
-- @nombre es el identificador de la
-- variable de tipo varchar
set @nombre = 'www.devjoker.com' -- El signo = es un operador
-- www.devjoker.com es un literal
print @Nombre -- Imprime por pantalla el valor de @nombre.
-- No diferencia mayúsculas ni minúsculas
```

## Scripts y lotes.

Un script de **Transact SQL** es un conjunto de sentencias de **Transact SQL** en formato de texto plano que se ejecutan en un servidor de **SQL Server**.

Un script está compuesto por uno o varios lotes. Un lote delimita el alcance de las variables y sentencias del script. Dentro de un mismo script se diferencian los diferentes lotes a través de la instrucción **GO**.

```
-- Este es el primer lote del script
SELECT * FROM COMENTARIOS
GO -- GO es el separador de lotes
-- Este es el segundo lote del script
SELECT getdate() -- getdate() es una función integrada que devuelve
-- la fecha
```

En ocasiones es necesario separar las sentencias en varios lotes, porque **Transact SQL** no permite la ejecución de ciertos comandos en el mismo lote, si bien normalmente también se utilizan los lotes para realizar separaciones lógicas dentro del script.

## Tipos de datos en Transact SQL

Cuando definimos una tabla, variable o constante debemos asignar un tipo de dato que indica los posibles valores. El tipo de datos define el formato de almacenamiento, espacio de disco-memoria que va a ocupar un campo o variable, restricciones y rango de valores validos.

Transact SQL proporciona una variedad predefinida de tipos de datos. Casi todos los tipos de datos manejados por Transact SQL son similares a los soportados por SQL.

### Tipos de datos numéricos.

SQL Server dispone de varios tipos de datos numéricos. Cuanto mayor sea el número que puedan almacenar mayor será en consecuencia el espacio utilizado para almacenarlo. Como regla general se recomienda usar el tipo de dato mínimo posible. Todos los dato numéricos admiten el valor NULL.

**Bit.** Una columna o variable de tipo bit puede almacenar el rango de valores de 1 a 0.

**Tinyint.** Una columna o variable de tipo tinyint puede almacenar el rango de valores de 0 a 255.

**SmallInt.** Una columna o variable de tipo smallint puede almacenar el rango de valores -32768 a 32767.

**Int.** Una columna o variable de tipo int puede almacenar el rango de valores  $-2^{31}$  a  $2^{31}-1$ .

**BigInt.** Una columna o variable de tipo bigint puede almacenar el rango de valores  $-2^{63}$  a  $2^{63}-1$ .

**Decimal(p,s).** Una columna de tipo decimal puede almacenar datos numéricos decimales sin redondear. Donde p es la precision (número total del dígitos) y s la escala (número de valores decimales)

**Float.** Una columna de datos float puede almacenar el rango de valores  $-1,79 \times 10^{308}$  a  $1,79 \times 10^{308}$ , si la definimos con el valor máximo de precisión. La precisión puede variar entre 1 y 53.

**Real.** Sinónimo de float(24). Puede almacenar el rango de valores  $-3,4 \times 10^{38}$  a  $3,4 \times 10^{38}$ .

**Money.** Almacena valores numéricos monetarios de  $-2^{63}$  a  $2^{63}-1$ , con una precisión de hasta diez milesimas de la unidad monetaria.

**SmallMoney.** Almacena valores numéricos monetarios de -214.748,3647 a 214.748,3647, con una precisión de hasta diez milesimas de la unidad monetaria.

**Todos los tipos de datos enteros pueden marcarse con la propiedad identity para**

### hacerlos autonuméricos.

```
DECLARE @bit bit,  
        @tinyint tinyint,  
        @smallint smallint,  
        @int int,  
        @bigint bigint,  
        @decimal decimal(10,3), -- 10 dígitos, 3 enteros y  
                                -- 3 decimales  
        @real real,  
        @double float(53),  
        @money money  
set @bit = 1  
print @bit  
set @tinyint = 255  
print @tinyint  
set @smallint = 32767  
print @smallint  
set @int = 642325  
print @int  
set @decimal = 56565.234 -- Punto como separador decimal  
print @decimal  
set @money = 12.34  
print @money
```

### Tipos de datos de carácter.

**Char(n).** Almacena n caracteres en formato ASCII, un byte por cada letra. Cuando almacenamos datos en el tipo char, siempre se utilizan los n caracteres indicados, incluso si la entrada de datos es inferior. Por ejemplo, si en un char(5), guardamos el valor 'A', se almacena 'A ', ocupando los cinco bytes.

**Varchar(n).** Almacena n caracteres en formato ASCII, un byte por cada letra. Cuando almacenamos datos en el tipo varchar, únicamente se utilizan los caracteres necesarios. Por ejemplo, si en un varchar(255), guardamos el valor 'A', se almacena 'A', ocupando solo un byte.

**Varchar(max).** Igual que varchar, pero al declararse como max puede almacenar  $2^{31}-1$  bytes.

**Nchar(n).** Almacena n caracteres en formato UNICODE, dos bytes por cada letra. Es recomendable utilizar este tipo de datos cuando los valores que vayamos a almacenar puedan pertenecer a diferentes idiomas.

**Nvarchar(n).** Almacena n caracteres en formato UNICODE, dos bytes por cada letra. Es recomendable utilizar este tipo de datos cuando los valores que vayamos a almacenar puedan pertenecer a diferentes idiomas.

**Nvarchar(max).** Igual que varchar, pero al declararse como max puede almacenar  $2^{31}-1$  bytes.

### **Tipos de datos de fecha.**

**Datetime.** Almacena fechas con una precisión de milisegundo. Debe usarse para fechas muy específicas.

**SmallDatetime.** Almacena fechas con una precisión de minuto, por lo que ocupa la mitad de espacio de que el tipo datetime, para tablas que puedan llegar a tener muchos datos es un factor a tener muy en cuenta.

**TimeStamp.** Se utiliza para marcar un registro con la fecha de inserción - actualización. El tipo timestamp se actualiza automáticamente cada vez que insertamos o modificamos los datos.

### **Tipos de datos binarios.**

**Binary.** Se utiliza para almacenar datos binarios de longitud fija, con una longitud máxima de 8000 bytes.

**Varbinary.** Se utiliza para almacenar datos binarios de longitud variable, con una longitud máxima de 8000 bytes. Es muy similar a binary, salvo que varbinary utiliza menos espacio en disco.

**Varbinary(max).** Igual que varbinary, pero puede almacenar  $2^{31}-1$  bytes

### **Tipo de datos XML.**

**XML.** Una de las grandes mejoras que incorpora SQL Server 2005 es el soporte nativo para XML. Como podemos deducir, este tipo de datos se utiliza para almacenar XML.

```
DECLARE @myxml XML
set @myxml = (SELECT @@SERVERNAME NOMBRE FOR XML RAW, TYPE)
print cast(@myxml as varchar(max))
```

Obtendremos la siguiente salida: <row nombre="SVR01"/>

### **Otros tipos de datos.**

**UniqueIdentifier.** Se utiliza para identificadores únicos. Para generar identificadores únicos debemos utilizar la función NEWID().

```
DECLARE @myuniqueid UNIQUEIDENTIFIER
set @myuniqueid = NEWID()
print cast(@myuniqueid as varchar(36))
```

Obtendremos la siguiente salida: 46141D79-102C-4C29-A620-792EA0208637



**Sql\_Variant.** Permite almacenar valores de diferentes tipos de datos. No puede almacenar varchar(max), xml, timestamp y tipos de datos definidos por el usuario.

### **Tipos de datos personalizados.**

Transact SQL **permite la creación de tipos de datos personalizados, a través de la instrucción CREATE TYPE. Personalmente, NO aconsejo el uso de tipos de datos personalizados.**

```
CREATE TYPE MD5 FROM CHAR(32) NULL
GO
DECLARE @miMD5 MD5
set @miMD5 = '0000000000000000000000000000000A'
print @miMD5
```

## Variables en Transact SQL

### Declarar variables en Transact SQL

Una variable es un valor identificado por un nombre (identificador) sobre el que podemos realizar modificaciones.

En **Transact SQL** los identificadores de variables deben comenzar por el caracter @, es decir, el nombre de una variable debe comenzar por @. Para declarar variables en **Transact SQL** debemos utilizar la palabra clave **declare**, seguido del identificador y tipo de datos de la variable.

```
-- Esto es un comentario de linea simple

/*
Este es un comentario con varias líneas.
Conjunto de Lineas.
*/
declare @nombre varchar(50)-- declare declara una variable
-- @nombre es el identificador de la
-- variable de tipo varchar
set @nombre = 'www.devjoker.com' -- El signo = es un operador
-- www.devjoker.com es un literal
print @Nombre -- Imprime por pantalla el valor de @nombre.
-- No diferencia mayúsculas ni minúsculas
```

### Asignar variables en Transact SQL

En **Transact SQL** podemos asignar valores a una variable de varias formas:

- A través de la instrucción **set**.
- Utilizando una sentencia **SELECT**.
- Realizando un **FETCH** de un cursor.

El siguiente ejemplo muestra como asignar una variable utilizando la instrucción **SET**.

```
DECLARE @nombre VARCHAR(100)
-- La consulta debe devolver un único registro
SET @nombre = (SELECT nombre
FROM CLIENTES
WHERE ID = 1)
PRINT @nombre
```

El siguiente ejemplo muestra como asignar variables utilizando una sentencia **SELECT**.

```
DECLARE @nombre VARCHAR(100),
@apellido1 VARCHAR(100),
@apellido2 VARCHAR(100)

SELECT @nombre=nombre ,
@apellido1=Apellido1,
@apellido2=Apellido2
FROM CLIENTES
WHERE ID = 1
```

```
PRINT @nombre  
PRINT @apellido1  
PRINT @apellido2
```

Un punto a tener en cuenta cuando asignamos variables de este modo, es que si la consulta SELECT devuelve más de un registro, las variables quedarán asignadas con los valores de la última fila devuelta.

Por último veamos como asignar variables a través de un cursor.

```
DECLARE @nombre VARCHAR(100),  
          @apellido1 VARCHAR(100),  
          @apellido2 VARCHAR(100)  
  
DECLARE CDATOS CURSOR  
FOR  
SELECT nombre , Apellido1, Apellido2  
FROM CLIENTES  
  
OPEN CDATOS  
FETCH CDATOS INTO @nombre, @apellido1, @apellido2  
  
WHILE (@@FETCH_STATUS = 0)  
BEGIN  
    PRINT @nombre  
    PRINT @apellido1  
    PRINT @apellido2  
    FETCH CDATOS INTO @nombre, @apellido1, @apellido2  
END  
  
CLOSE CDATOS  
DEALLOCATE CDATOS
```

Veremos los cursores con más detalle más adelante en este tutorial.

---

## Equivalencia de datos de SQL Server y .NET

La siguiente lista muestra los tipos de datos de SQL Server 2005 y sus equivalentes con CRL, para el namespace **System.Data.SqlTypes** y los tipos nativos de CRL .NET

### FrameWork

SQL Server	CLR data type (SQL Server)	CLR data type (.NET Framework)
<b>varbinary</b>	<b>SqlBytes, SqlBinary</b>	<b>Byte[]</b>
<b>Binary</b>	<b>SqlBytes, SqlBinary</b>	<b>Byte[]</b>
<b>varbinary(1), binary(1)</b>	<b>SqlBytes, SqlBinary</b>	<b>byte, Byte[]</b>
<b>Image</b>	Ninguno	ninguno
<b>varchar</b>	Ninguno	ninguno
<b>char</b>	ninguno	ninguno
<b>nvarchar(1), nchar(1)</b>	<b>SqlChars, SqlString</b>	<b>Char, String, Char[]</b>
	<b>SqlChars, SqlString</b>	
<b>nvarchar</b>	<b>SQLChars</b> es mejor para la transferencia de datos y <b>SQLString</b> obtiene mejor rendimiento para operaciones con Strings.	<b>String, Char[]</b>
<b>Nchar</b>	<b>SqlChars, SqlString</b>	<b>String, Char[]</b>
<b>text</b>	ninguno	ninguno
<b>ntext</b>	ninguno	ninguno
<b>uniqueidentifier</b>	<b>SqlGuid</b>	<b>Guid</b>
<b>rowversion</b>	ninguno	<b>Byte[]</b>
<b>bit</b>	<b>SqlBoolean</b>	<b>Boolean</b>
<b>tinyint</b>	<b>SqlByte</b>	<b>Byte</b>
<b>smallint</b>	<b>SqlInt16</b>	<b>Int16</b>
<b>int</b>	<b>SqlInt32</b>	<b>Int32</b>
<b>Bigint</b>	<b>SqlInt64</b>	<b>Int64</b>
<b>smallmoney</b>	<b>SqlMoney</b>	<b>Decimal</b>
<b>money</b>	<b>SqlMoney</b>	<b>Decimal</b>
<b>numeric</b>	<b>SqlDecimal</b>	<b>Decimal</b>
<b>decimal</b>	<b>SqlDecimal</b>	<b>Decimal</b>
<b>real</b>	<b>SqlSingle</b>	<b>Single</b>
<b>float</b>	<b>SqlDouble</b>	<b>Double</b>
<b>smalldatetime</b>	<b>SqlDateTime</b>	<b>DateTime</b>
<b>datetime</b>	<b>SqlDateTime</b>	<b>DateTime</b>
<b>sql_variant</b>	ninguno	<b>Object</b>
<b>User-defined type(UDT)</b>	ninguno	Misma clase que la definida en el asamblea.
<b>table</b>	ninguno	ninguno
<b>Cursor</b>	ninguno	ninguno
<b>timestamp</b>	ninguno	ninguno
<b>xml</b>	<b>SqlXml</b>	ninguno

## Operadores en Transact SQL

La siguiente tabla ilustra los operadores de Transact SQL .

Tipo de operador	Operadores
Operador de asignación	=
Operadores aritméticos	+ (suma) - (resta) * (multiplicación) / (división) ** (exponente) % (modulo)
Operadores relacionales o de comparación	= (igual a) <> (distinto de) != (distinto de) < (menor que) > (mayor que) >= (mayor o igual a) <= (menor o igual a) !> (no mayor a) !< (no menor a)
Operadores lógicos	AND (y lógico) NOT (negacion) OR (o lógico) & (AND a nivel de bit)   (OR a nivel de bit) ^ (OR exclusivo a nivel de bit)
Operador de concatenación	+
Otros	ALL (Devuelve TRUE si el conjunto completo de comparaciones es TRUE) ANY (Devuelve TRUE si cualquier elemento del conjunto de comparaciones es TRUE) BETWEEN (Devuelve TRUE si el operando está dentro del intervalo) EXISTS (TRUE si una subconsulta contiene filas) IN (TRUE si el operando está en la lista) LIKE (TRUE si el operando coincide con un patron) NOT (Invierte el valor de cualquier operador booleano) SOME (Devuelve TRUE si alguna de las comparaciones de un conjunto es TRUE)

## Estructuras de control en Transact SQL

### Estructura condicional IF

La estructura condicional **IF** permite evaluar una expresion booleana (resultado SI - NO), y ejecutar las operaciones contenidas en el bloque formado por **BEGIN END**.

```
IF (<expresion>)  
  BEGIN  
    ...  
  END  
ELSE IF (<expresion>)  
  BEGIN  
    ...  
  END  
ELSE  
  BEGIN  
    ...  
  END
```

Ejemplo de la estructura condicional **IF**.

```
DECLARE @Web varchar(100),  
        @diminutivo varchar(3)  
  
SET @diminutivo = 'DJK'  
  
IF @diminutivo = 'DJK'  
  BEGIN  
    PRINT 'www.devjoker.com'  
  END  
ELSE  
  BEGIN  
    PRINT 'Otra Web (peor!)'  
  END
```

La estructura **IF** admite el uso de subconsultas:

```
DECLARE @coPais int,  
        @descripcion varchar(255)  
set @coPais = 5  
set @descripcion = 'España'  
IF EXISTS(SELECT * FROM PAISES  
          WHERE CO_PAIS = @coPais)  
  BEGIN  
    UPDATE PAISES  
    SET DESCRIPCION = @descripcion  
    WHERE CO_PAIS = @coPais  
  END  
  
ELSE  
  BEGIN  
    INSERT INTO PAISES  
    (CO_PAIS, DESCRIPCION) VALUES  
    (@coPais, @descripcion)  
  END
```

## Estructura condicional CASE


La estructura condicional **CASE** permite evaluar una expresion y devolver un valor u otro.  
La sintaxis general de case es:

```
CASE <expression>
  WHEN <valor_expression> THEN <valor_devuelto>
  WHEN <valor_expression> THEN <valor_devuelto>
  ELSE <valor_devuelto> -- Valor por defecto
END
```

### Ejemplo de **CASE**.

```
DECLARE @Web varchar(100),
          @diminutivo varchar(3)
SET @diminutivo = 'DJK'
SET @Web = (CASE @diminutivo
            WHEN 'DJK' THEN 'www.devjoker.com'
            WHEN 'ALM' THEN 'www.aleamedia.com'
            ELSE 'www.devjoker.com'
            END)
PRINT @Web
```

Otra sintaxis de **CASE** nos permite evaluar diferentes expresiones:

```
CASE
  WHEN <expresion>  <valor_expresion> THEN <valor_devuelto>
  WHEN <expresion> = <valor_expresion> THEN <valor_devuelto>
  ELSE <valor_devuelto> -- Valor por defecto
END
```

El mismo ejemplo aplicando esta sintaxis:

```
DECLARE @Web varchar(100),
          @diminutivo varchar(3)
SET @diminutivo = 'DJK'

SET @Web = (CASE
             WHEN @diminutivo = 'DJK' THEN 'www.devjoker.com'
             WHEN @diminutivo = 'ALM' THEN 'www.aleamedia.com'
             ELSE 'www.devjoker.com'
             END)
PRINT @Web
```

Otro aspecto muy interesante de **CASE** es que permite el uso de subconsultas.

```
DECLARE @Web varchar(100),
          @diminutivo varchar(3)
SET @diminutivo = 'DJK'

SET @Web = (CASE
            WHEN @diminutivo = 'DJK' THEN (SELECT web
            FROM WEBS
            WHERE id=1)
```

```

        WHEN @diminutivo = 'ALM' THEN (SELECT web
                                      FROM WEBS
                                      WHERE id=2)
        ELSE 'www.devjoker.com'
    END)
PRINT @Web

```

## Bucle WHILE

El bucle **WHILE** se repite mientras expresion se evalúe como verdadero.  
Es el único tipo de bucle del que dispone **Transact SQL**.

```

WHILE <expresion>
BEGIN
...
END

```

Un ejemplo del bucle **WHILE**.

```

DECLARE @contador int
SET @contador = 0
WHILE (@contador < 100)
BEGIN
    SET @contador = @contador + 1

    PRINT 'Iteracion del bucle ' + cast(@contador AS varchar)
END

```

Podemos pasar a la siguiente iteración del bucle utilizando **CONTINUE**.

```

DECLARE @contador int
SET @contador = 0
WHILE (@contador < 100)
BEGIN
    SET @contador = @contador + 1
    IF (@contador % 2 = 0)
        CONTINUE
    PRINT 'Iteracion del bucle ' + cast(@contador AS varchar)
END

```

El bucle se dejará de repetir con la instrucción **BREAK**.

```

DECLARE @contador int
SET @contador = 0
WHILE (1 = 1)
BEGIN
    SET @contador = @contador + 1
    IF (@contador % 50 = 0)
        BREAK
    PRINT 'Iteracion del bucle ' + cast(@contador AS varchar)
END

```

También podemos utilizar el bucle **WHILE** conjuntamente con subconsultas.

```

DECLARE @coRecibo int
WHILE EXISTS (SELECT *
             FROM RECIBOS
             WHERE PENDIENTE = 'S') -- Ojo, la subconsulta se ejecuta
                                   -- una vez por cada iteracion
                                   -- del bucle!

```



```

BEGIN
    SET @coRecibo = (SELECT TOP 1 CO_RECIBO
                     FROM RECIBOS WHERE PENDIENTE = 'S')

    UPDATE RECIBOS
    SET PENDIENTE = 'N'
    WHERE CO_RECIBO = @coRecibo
END

```

## Estructura GOTO

La sentencia goto nos permite desviar el flujo de ejecución hacia una etiqueta. Fué muy utilizada en versiones anteriores de SQL Server conjuntamente con la variable de sistema @@ERROR para el control de errores.

Actualmente, se desaconseja el uso GOTO, recomendándose el uso de TRY - CATCH para la gestion de errores.

```

DECLARE @divisor int,
        @dividendo int,
        @resultado int
SET @dividendo = 100
SET @divisor = 0
SET @resultado = @dividendo/@divisor

IF @@ERROR > 0
    GOTO error

PRINT 'No hay error'
RETURN

```

error:

```

PRINT 'Se ha producido una division por cero'

```

## Control de errores en Transact SQL

### Uso de TRY CATCH

A partir de la versión 2005, **SQL Server** proporciona el control de errores a través de las instrucciones **TRY** y **CATCH**.

Estas nuevas instrucciones suponen un gran paso adelante en el control de errores en **SQL Server**, un tanto precario en las versiones anteriores.

La sintaxis de **TRY CATCH** es la siguiente:

```
BEGIN TRY
...
END TRY
BEGIN CATCH
...
END CATCH
```

El siguiente ejemplo ilustra el uso de **TRY - CATCH**.

```
BEGIN TRY

    DECLARE @divisor int ,

            @dividendo int,

            @resultado int

    SET @dividendo = 100

    SET @divisor = 0

    -- Esta linea provoca un error de division por 0
    SET @resultado = @dividendo/@divisor
    PRINT 'No hay error'
END TRY
BEGIN CATCH
    PRINT 'Se ha producido un error'
END CATCH
```

### Funciones especiales de Error

Las funciones especiales de error, están disponibles únicamente en el bloque **CATCH** para la obtención de información detallada del error.

Son:

- **ERROR\_NUMBER()**, devuelve el número de error.
- **ERROR\_SEVERITY()**, devuelve la severidad del error.
- **ERROR\_STATE()**, devuelve el estado del error.
- **ERROR\_PROCEDURE()**, devuelve el nombre del procedimiento almacenado que ha provocado el error.
- **ERROR\_LINE()**, devuelve el número de línea en el que se ha producido el error.
- **ERROR\_MESSAGE()**, devuelve el mensaje de error.

Son extremadamente útiles para realizar una auditoría de errores.

```
BEGIN TRY

    DECLARE @divisor int ,

            @dividendo int,

            @resultado int

    SET @dividendo = 100

    SET @divisor = 0

    -- Esta linea provoca un error de division por 0
    SET @resultado = @dividendo/@divisor
    PRINT 'No hay error'
END TRY
BEGIN CATCH
    PRINT ERROR_NUMBER()
    PRINT ERROR_SEVERITY()
    PRINT ERROR_STATE()
    PRINT ERROR_PROCEDURE()
    PRINT ERROR_LINE()
    PRINT ERROR_MESSAGE()
END CATCH
```

Lógicamente, podemos utilizar estas funciones para almacenar esta información en una tabla de la base de datos y registrar todos los errores que se produzcan.

### La variable de sistema @@ERROR

En versiones anteriores a **SQL Server 2005**, no estaban disponibles las instrucciones **TRY CATCH**. En estas versiones se controlaban los errores utilizando la variable global de sistema @@ERROR, que almacena el número de error producido por la última sentencia **Transact SQL** ejecutada.

```
DECLARE @divisor int ,
        @dividendo int ,
        @resultado int

SET @dividendo = 100
SET @divisor = 0
-- Esta linea provoca un error de division por 0
SET @resultado = @dividendo/@divisor

IF @@ERROR = 0
    BEGIN
        PRINT 'No hay error'
    END
ELSE
    BEGIN
        PRINT 'Hay error'
    END
```

El uso de @@ERROR para controlar errores puede provocar multitud de problemas. Uno de los más habituales es sin duda, incluir una nueva sentencia **Transact SQL** entre la línea que provoco el error y la que lo controla. Esa nueva instrucción restaura el valor de @@ERROR y

no controlaremos el error.

El siguiente ejemplo ilustra esta situación:

```
DECLARE @divisor int ,
        @dividendo int ,
        @resultado int

SET @dividendo = 100
SET @divisor = 0
-- Esta linea provoca un error de division por 0
SET @resultado = @dividendo/@divisor
PRINT 'Controlando el error ...' -- Esta linea estable @@ERROR a cero
IF @@ERROR = 0
    BEGIN
        -- Se ejecuta esta parte!
        PRINT 'No hay error'
    END
ELSE
    BEGIN
        PRINT 'Hay error'
    END
```

## Generar un error con RAISERROR

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

Podemos provocar un error en tiempo de ejecución a través de la función RAISERROR.

```
DECLARE @tipo int,
        @clasificacion int

SET @tipo = 1
SET @clasificacion = 3
IF (@tipo = 1 AND @clasificacion = 3)
    BEGIN
        RAISERROR ('El tipo no puede valer uno y la clasificacion 3',
                    16, -- Severidad
                    1 -- Estado
                    )
    END
```

La función RAISERROR recibe tres parámetros, el mensaje del error (o código de error predefinido), la severidad y el estado.

La severidad indica el grado de criticidad del error. Admite valores de 0 al 25, pero solo podemos asignar valores del 0 al 18. Los errores el 20 al 25 son considerados fatales por el sistema, y cerraran la conexion que ejecuta el comando **RAISERROR**. Para asignar valores del 19 al 25 necesitaras ser miembros de la función de **SQL Server** sysadmin.

El estado es un valor para permitir que el programador identifique el mismo error desde diferentes partes del código. Admite valores entre 1 y 127.

---