Unit 10

Programming in Transact-SQL – TC1

Objetives

- Use programming statements
- Write Stored Procedures
- Introduce cursors

Introduction

Transact-SQL supports:

- Datatypes.
- Variable Definitions.
- IF and Loop structures.
- Exceptions management.
- Standard functions.

All these features will enable us to create code blocks to run complex operations.

Introduction

In SQL Server we have three types of code blocks stored as database objects:

- Stored Procedures
- Triggers
- User defined Functions.

We will study stored procedures and triggers.

Introduction

We have already used code blocks, SQL scripts.

- SQL Script : set of statements in plain text format that are executed in a SQL server.
- Batch (lote): set of statements that are sent to an instance of SQLServer to be executed.
- GO: Signals the end of a batch of Transact-SQL statements.

Language Elements

Messages

PRINT → Output in the tab *Messages*

SELECT → Output in the tab *Results*

ACER1.Gestion11 - SQLQuery1.sql* | Resumen

```
PRINT 'Estas son las oficinas del Este (con PRINT)'
SELECT 'Estas son las oficinas del Este' AS [ ]
SELECT * FROM Oficinas WHERE region='este'
```

Resultados | Mensajes

1	Estas son las oficinas del Este

	oficina	ciudad	region	dir	objetivo	ventas
1	11	Valencia	este	106	57500,00	69300,00
2	12	Alicante	este	104	80000,00	73500,00
3	13	Castellon	este	105	35000,00	36800,00
4	28	Valencia	este	NULL	90000,00	0,00
5	29	Valencia	este	NULL	10000,00	2100,00
6	33	Alicante	este	NULL	3000,00	0,00

Resultados | Mensajes

```
Estas son las oficinas del Este (con PRINT)

(1 filas afectadas)

(6 filas afectadas)
```

Messages

PRINT String [;]

Example:

PRINT ‘These are the employees hired last year (in '+STR(YEAR(GETDATE())-1,4) + ')‘;

SELECT select-list [;]

Example:

SELECT ‘These are the employees hired last year in :’ AS [ ], YEAR(GETDATE()) -1 AS [ ];

Variables - Declaration

DECLARE {@variableNa [AS] datatype} [,...n] [;]

- variableNa begins with @
- datatype
- Examples:

DECLARE @employees INT;

DECLARE @var1 INT, @var2 INT;

Variables - Setting

To fill the variable with some value.

{SET | SELECT} @variableNa = value [;]

value can be:

- A constant,
- Another variable name or expression,
- A scalar query,
- Part of a SELECT (in this case the use of SELECT is compulsory).

We will use SET as much as possible.

Variables - Setting

DECLARE @employee AS INT, @var AS INT;

SET @var = 20 ;

SELECT @employee = 0;

SET @employee = @var * 100;

SELECT @employee = @var * 100;

SET @employee =(SELECT COUNT(numemp) FROM mployees);
SELECT @employee =(SELECT COUNT(numemp) FROM employees);

SELECT @employee = COUNT(numemp) FROM employees;

SELECT @employee = COUNT(numemp), @var= MAX(office) FROM employees;

STORED PROCEDURES

A STORED PROCEDURE

- Is a group of Transact-SQL statements compiled into a single execution plan.
- It can have input parameters and can return one or several values to the process that executed it.
- It is stored in the database and it can be executed whenever we want.

STORED PROCEDURES

- Benefits of using Stored procedures:
  - The first time the stored procedure is executed, an execution plan is created so that performance is increased.
  - They allow modular programming.
  - They can enhance the security of applications.
  - They can reduce network traffic.

- See more: http://technet.microsoft.com/en-US/library/ms191436(v=sql.90).aspx

STORED PROCEDURES

- Many administrative activities in the SQL Server are performed through a special kind of procedure known as a **system stored procedure**. They are stored in the database **master** and their name begins with **sp_**. They allow us to retrieve data from the system tables.
- Then we have the user-defined stored procedures created by any user.
- A stored procedure can also be **temporary** and **local** or **global**.

STORED PROCEDURES

- Only the connection that created a **local temporary** stored procedure can execute it, and the procedure is automatically deleted when the connection is closed. Prefixed with #.

- A **global temporary** stored procedure can be executed by any connection and it exists until the connection used by the user who created the procedure is closed, and any currently executing versions of the procedure by any other connections are completed. Prefixed with ##.

STORED PROCEDURES

- To define it: CREATE PROCEDURE SPName ...
- To call it:
  - EXEC SPName ...
  - EXECUTE SPName ...
  - SPName ... Only if it is the first statement of the batch. Not recommended.
- To eliminate it : DROP PROCEDURE SPName;
- To change its name: DROP and CREATE.

DROP PROCEDURE

Removes one or more stored procedures from the current database.

Syntax:

```
DROP {PROC|PROCEDURE}
      {[SchemaNa.]SPname }[,...n ][;]
```

Examples:

```
DROP PROCEDURE Say_Hello;

DROP PROC Say_Hello, Count_employees;
```

EXEC - EXECUTE

Syntax:

```
[{EXEC | EXECUTE}]  SPname
   [{[@parameter = ] {value
                      | @variable [ OUTPUT ]
                      | [ DEFAULT ]
                      }
     }[,...n ]
   ] [;]
```

EXEC - EXECUTE

EXEC can be used to execute a SQL string:

{EXEC | EXECUTE}(' SQL string ') [;]

Useful for example when the statement does not accept a variable name:

CREATE DATABASE @var; Not admitted

→

EXEC(' CREATE DATABASE ' + @var);

EXEC(' USE Gestion; CREATE TABLE …);

EXEC - EXECUTE

Combining SQL text with variables is a frequently used action, so knowing how to do it is essential.

TIP: Write the sentence without the variable but with a fixed value:

SELECT * FROM Offices WHERE sales > **10000**

Enclose the fixed text in ' ' , replace the fixed value with the variable name, and concatenate them:

'SELECT * FROM OfficesWHERE sales > ' + **@value**

EXEC - EXECUTE

SELECT * FROM Offices WHERE sales > **10000** AND dir IS NULL ;

'SELECT * FROM Offices WHERE sales > ' + **@limit** + ' AND dir IS NULL; '

SELECT * FROM Offices WHERE zone = '**North**'

'SELECT * FROM Offices WHERE zone = ''' + **@value** + ''''

Be careful of inverted commas and spaces!

CREATE PROCEDURE

```
CREATE [PROC|PROCEDURE]  [SchemaNa.]SPname
    [{@parameter datatype} [= defaultvalue]
                                [OUT|OUTPUT] ]
    ][,...n ]
AS <sql_statement> [;]

< sql_statement >  ::=
 { [ BEGIN ]
       statement  [..n]
   [ END ] }
```

CREATE PROCEDURE

- It is the only statement in a single batch.
- It can be created only in the current database.
- After the procedure name we can specify a maximum of 2,100 parameters:
  - @parameter: parameter name.
  - datatype
  - Optionally a default value (=defaultvalue).
- Parameters are local to the procedure.
- By default, parameters can take the place only of constant expressions; they cannot be used instead of table names, column names, or the names of other database objects.
- OUTPUT | OUT Indicates that the parameter is an output parameter. .

CREATE PROCEDURE

```
CREATE PROCEDURE  Say_Hello  AS PRINT 'Hello';
GO        -- end of procedure
-- Now we test the procedure:
EXEC Say_Hello;
GO
-- Then we create a new one:
CREATE PROCEDURE Say_Word @word CHAR(30)
AS
        PRINT @word;
GO    -- end of procedure. Then we test it:
EXEC Say_Word 'first';
EXEC Say_Word 'second';
GO
```

CREATE PROCEDURE with several parameters

```
USE Biblio;
GO
CREATE PROCEDURE ClientsPlace @place CHAR(30),@state
   CHAR(30)  AS
         SELECT * FROM clients WHERE
         place=@place AND state = @state;
GO
EXEC ClientsPlace 'Silla', 'Valencia'
EXEC ClientsPlace @place =' Silla ', @state ='Valencia'
EXEC ClientsPlace @state ='Valencia', @place =' Silla '
EXEC ClientsPlace 'Madrid' raise an ERROR
```

CREATE PROCEDURE optional parameters

```
CREATE PROCEDURE ClientsPlace2 @place CHAR(30),@state
   CHAR(30)='Madrid'
AS
   SELECT * FROM clients WHERE place = @place
                                        AND state = @state;
GO
EXEC ClientsPlace2 'Pinto'
```

This EXEC is valid because @state is optional and the last parameter.

CREATE PROCEDURE optional parameters

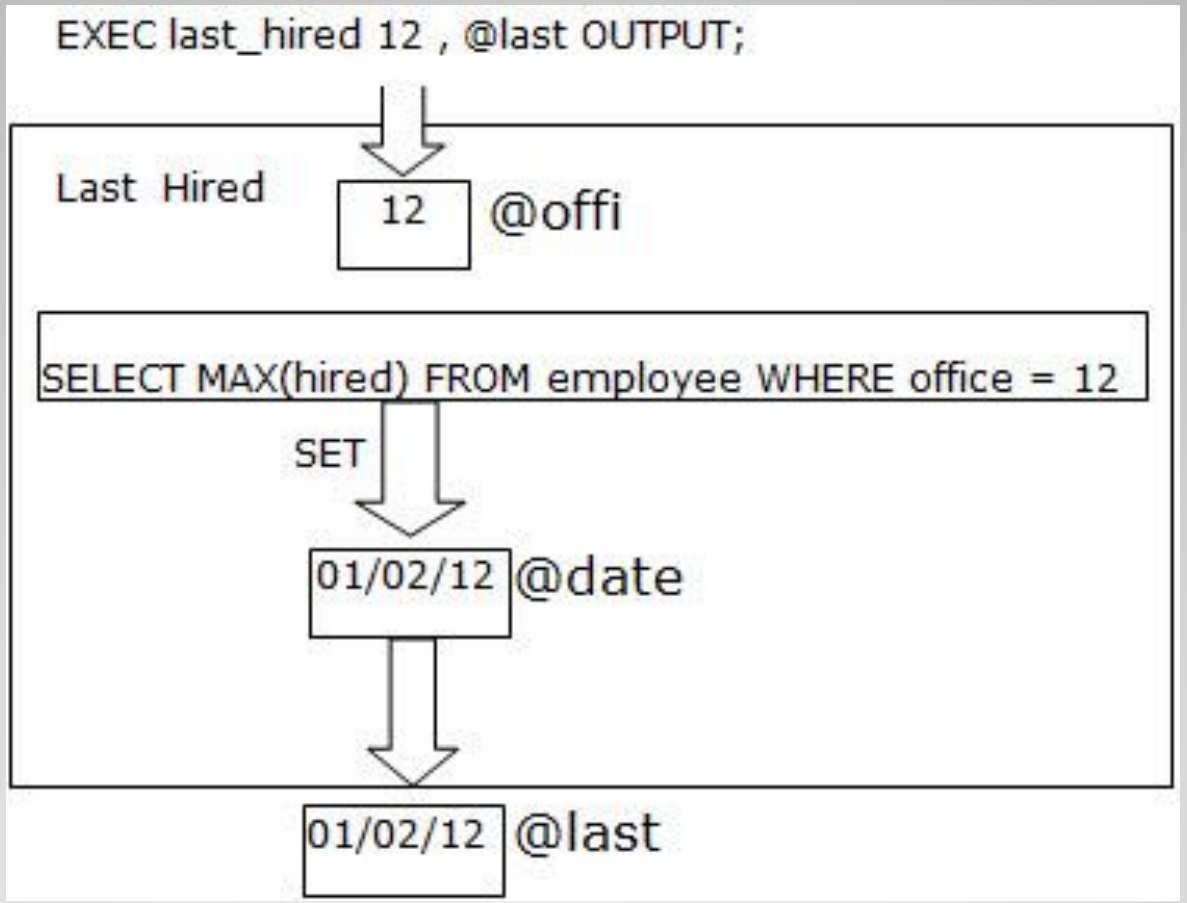
```
CREATE PROCEDURE ClientsPlace3 @a CHAR(2),@place
   CHAR(30)='Madrid',@state CHAR(30)  AS
         SELECT * FROM clients WHERE place =@place AND
   state = @state;
GO
EXEC ClientsPlace3  a, @state='Madrid'
EXEC ClientsPlace3  a, DEFAULT, 'Madrid'

EXEC ClientsPlace3  a, ,'Madrid' raises an ERROR
```

CREATE PROCEDURE output parameters

```
USE Gestion
GO
CREATE PROC last_hired @offi INT, @date DATE
                                          OUTPUT
AS
   SET @date=(SELECT MAX(hired) FROM employee WHERE
   office=@offi)
GO
DECLARE @last AS DATE;
EXEC last_hired 12,@last OUTPUT;
PRINT @last;
```

CREATE PROCEDURE output parameters



CREATE PROCEDURE with RETURN

RETURN [integer_expression]

- Written inside the procedure.
- Exits unconditionally from the procedure.
- Returns an integer value.

To use the returned value in the batch that executed the procedure:

EXECUTE @collector = SPname @par, ...

CREATE PROCEDURE using RETURN

```
CREATE PROC workersn @offi INT, @num INT OUTPUT AS
        SET @num=(SELECT COUNT(*) FROM employee
                WHERE office=@offi)
GO
DECLARE @var INT;
EXEC workersn 12, @var OUTPUT
```

Using an output parameter

```
CREATE PROC workersn2 @offi INT  AS
        RETURN (SELECT COUNT(*) FROM  employee WHERE
                                office=@offi)
GO
DECLARE @var INT;
EXEC @var= workersn2 12
```

Using RETURN

More statements

More statements

In order to fill the procedure body with code, let's see some useful Transact-SQL statements.

Most of them are familiar.

IF… ELSE

```
IF condition
  { sql_statement | sql_block }
[ELSE
  { sql_statement | sql_block } ]
```

Example:

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

IF... ELSE

```
USE GEstion8
DECLARE @offi as int
SET @offi = 33
IF EXISTS(SELECT * FROM employee WHERE office =@offi)
  BEGIN
     SELECT 'the workers from the office ' + STR(@offi,2) +
   'are:'
     SELECT numemp, name FROM employee WHERE office =
   @offi
  END
ELSE
  PRINT 'The office ' + STR(@offi,2) + ' has no employees'
```

TRY… CATCH

Implements error handling, it catches all execution errors that have a severity higher than 10 that do not close the database connection.

```
BEGIN TRY
    {sql_statement|sql_block}
END TRY
BEGIN CATCH
     [{sql_statement|sql_block}]
END CATCH [ ; ]
```

The statements enclosed in the TRY block are executed. If an error occurs, control is passed to the CATCH block.

TRY… CATCH

A TRY block must be immediately followed by an associated CATCH block.

Nesting TRY..CATCH is possible.

A TRY…CATCH cannot span multiple batches, nor multiple blocks of statements.

GOTO statements cannot be used to enter a TRY or CATCH block. But they can be used inside the TRY or CATCH block to jump to a label inside the same block or to leave the TRY or CATCH block.

TRY… CATCH

When a TRY block executes a stored procedure and an error occurs in the stored procedure, the error can be handled in the following ways:

- If the stored procedure does not contain its own TRY…CATCH, the error returns control to the CATCH block associated with the TRY block that contains the EXECUTE statement.
- If the stored procedure contains a TRY…CATCH, the error transfers control to the CATCH block in the stored procedure. When the CATCH block code finishes, control is passed back to the statement immediately after the EXECUTE statement that called the stored procedure.

Retrieving Error Information

In the scope of a CATCH block, the following system functions are available:

ERROR_NUMBER() → the number of the error.

ERROR_SEVERITY() → the severity.

ERROR_STATE() → the error state number.

ERROR_PROCEDURE() → the name of the stored procedure where the error occurred.

ERROR_LINE() → the line number inside the routine that caused the error.

ERROR_MESSAGE() → the complete text of the error message.

Retrieving Error Information

```
CREATE PROCEDURE GetPrice @amount MONEY,@quantity
   INT,@price MONEY OUTPUT AS
BEGIN TRY
  SET @price=@amount/@ quantity;
END TRY
BEGIN CATCH
   IF ERROR_NUMBER() = 8134 SET @price=0
   ELSE SELECT ERROR_NUMBER() as
   ErrorNumero,ERROR_MESSAGE() as ErrorMessage;
END CATCH;
GO
DECLARE @result MONEY;
EXEC GetPrice 1000, 0, @result OUTPUT;
SELECT 'The price is: ', @result;
```

Errors unaffected by a TRY…CATCH

TRY…CATCH constructs do not trap the following conditions :

- Warnings or informational messages that have a severity <=10.
- Errors severity >= 20 that stop the SQL Server Database Engine task processing for the session.
- Attentions, such as client-interrupt requests or broken connections.
- When the session is ended by using the KILL statement.

Errors unaffected by a TRY…CATCH

- The following types of errors are not handled by a CATCH block when they occur at the same level of execution as the TRY…CATCH construct :
  - Compile errors, such as syntax errors, that prevent a batch from running.
  - Errors such as object name resolution errors that occur after compilation because of deferred name resolution.
- These errors are returned to the level that ran the batch.

Errors unaffected by a TRY...CATCH

```
USE Gestion;
GO
BEGIN TRY
    SELECT * FROM  NonExistingTable;
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() as ErrorNumber,
          ERROR_MESSAGE() as ErrorMessage;
END CATCH
```

If the table does not exist, the error is not handled by the CATCH block.

In the next example, the error will be handled by the CATCH asociated with the TRY block that executed the procedure.

Errors unaffected by a TRY…CATCH

```
IF OBJECT_ID (SelectTable','P') IS NOT NULL
    DROP PROCEDURE SelectTable;
GO
CREATE PROCEDURE SelectTable AS
    SELECT * FROM NonExistingTable;
GO
BEGIN TRY
    EXECUTE SelectTable
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() as ErrorNumber,
          ERROR_MESSAGE() as ErrorMessage;
END CATCH;
```

WHILE–BREAK-CONTINUE

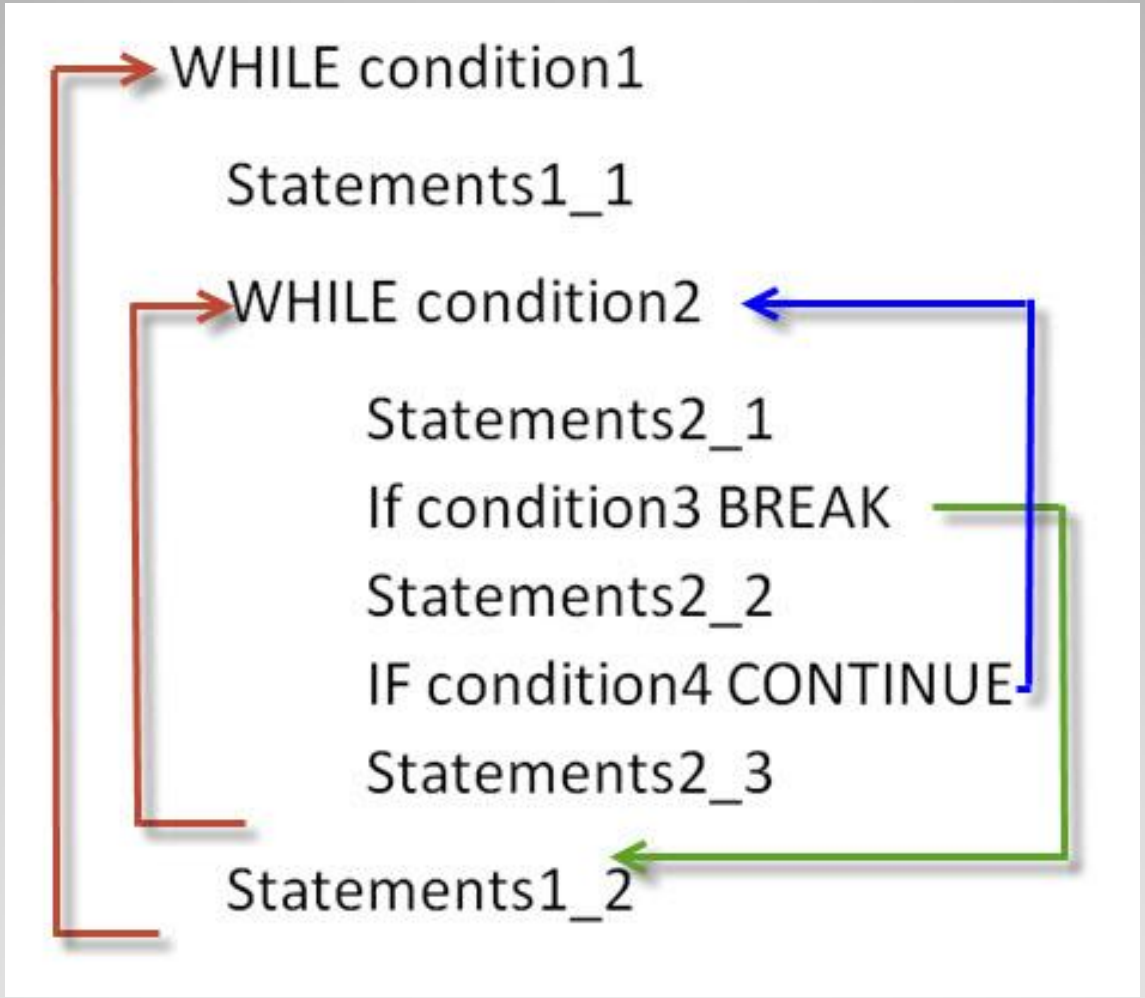
```
WHILE condition
    { sql_statement | sql_block }
    | BREAK
    | CONTINUE
```

The statements are executed repeatedly as long as the specified condition is true.

BREAK Causes an exit from the innermost WHILE loop.

CONTINUE jump to the beginning of the WHILE loop.

WHILE–BREAK-CONTINUE



WAITFOR

Blocks the execution of a batch, stored procedure, or transaction until a specified time or time interval is reached.

```
WAITFOR  {  DELAY 'time_to_pass’
                |TIME 'run_datetime'}
```

```
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);
```

GOTO

Alters the flow of execution to a label. The statements that follow GOTO are skipped and processing continues at the label.

Syntax:

```
GOTO LabelName
```

Example:

```
IF condition GOTO LabelName
---
LabelName:
---
```

Cannot go to a label outside the batch.