# CS108: Advanced Database

## - Database Programming

Lecture 08:

Programmable Objects

# Overview

- **Variables**
- **Flow Elements**
  - The IF . . . ELSE Flow Element
  - The WHILE Flow Element
- Temporary Tables
- Routines

# Variables

- Variables allow us to temporarily store data values for later use in the same batch in which they were declared.
- Use a DECLARE statement to declare one or more variables, and use a SET statement to assign a value to a single variable.
- For example,

```
DECLARE @i AS INT;
SET @i = 10;
```

- The declaration and initialization of variables in the same statement

```
DECLARE @i AS INT = 10;
```

# Variables

- When we are assigning a value to a scalar variable, the value must be the result of a scalar expression. The expression can be a scalar subquery.
- The SET statement can operate only on one variable at a time, so if we need to assign values to multiple attributes, we need to use multiple SET statements.
- SQL Server also supports a nonstandard assignment SELECT statement, which allows us to query data and assign multiple values obtained from the same row to multiple variables by using a single statement.

- The expression can be a scalar subquery.

```
DECLARE @FirstName AS NVARCHAR(10), @LastName AS NVARCHAR(20);

SET @FirstName = (SELECT FirstName FROM Person.Person
                  WHERE  BusinessEntityID = 3);
SET @LastName  = (SELECT LastName FROM Person.Person
                  WHERE  BusinessEntityID = 3);

SELECT @FirstName AS firstname, @LastName AS lastname;
```

- SQL Server also supports a nonstandard assignment SELECT statement.

```
DECLARE @FirstName AS NVARCHAR(10), @LastName AS NVARCHAR(20);

SELECT
   @FirstName = FirstName,
   @LastName  = LastName
FROM   Person.Person
WHERE  BusinessEntityID = 3;

SELECT @FirstName AS firstname, @LastName AS lastname;
```

# The assignment SELECT

- The assignment SELECT has predictable behavior when exactly one row qualifies.
  - if the query has more than one qualifying row, the code doesn't fail
  - the values from the current row overwrite the existing values in the variables.
- When the assignment SELECT finishes, the values in the variables are those from the last row that SQL Server happened to access.

# The SET statement

- The SET statement is safer than assignment SELECT because it requires we to use a scalar subquery to pull data from a table.

```
DECLARE @Name VARCHAR(20)

SELECT @Name = Firstname + ' ' + Lastname
FROM   Person.Person
WHERE  EmailPromotion = 2
```

```
SET @Name = (SELECT Firstname + ' ' + Lastname
             FROM   Person.Person
             WHERE  EmailPromotion = 2)
```

```
Subquery returned more than 1 value. This is not permitted when the
subquery follows =, !=, <, <= , >, >= or when the subquery is used
as an expression.
```

# Batches

- A batch is one or more T-SQL statements sent by a client application to SQL Server for execution as a single unit.
- The batch undergoes parsing (syntax checking), resolution (checking the existence of referenced objects and columns), permissions checking, and optimization as a unit.
- A transaction is an atomic unit of work. A batch can have multiple transactions, and a transaction can be submitted in parts as multiple batches.

# Batches

- Example, a Batch as a Unit of parsing

```
-- Valid batch
PRINT 'First batch';
USE AdventureWorks2014;
GO
-- Invalid batch
PRINT 'Second batch';
SELECT * FROM HumanResources.Employee;
SELECT * FOM  Person.Person;
GO
-- Valid batch
PRINT 'Third batch';
SELECT FirstName FROM Person.Person;
```

```
First batch
Msg 102, Level 15, State 1, Line 8
Incorrect syntax near 'FOM'.
Third batch
(19972 row(s) affected)
```

# Batches and Variables

- A variable is local to the batch in which it is defined.
- If we try to refer to a variable that was defined in another batch, we will get an error saying that the variable was not defined.

```
DECLARE @i AS INT;
SET @i = 10;
-- Succeeds
PRINT @i;
GO
-- Fails
PRINT @i;
```

```
10
Msg 137, Level 15, State 2, Line 3
Must declare the scalar variable "@i".
```

# The IF . . . ELSE Flow Element

- The IF . . . ELSE element allows us to control the flow of our code based on a predicate.
  - A statement or statement block that is executed if the predicate is TRUE, and optionally a statement
  - or statement block that is executed if the predicate is FALSE or UNKNOWN.
- For example,

```
IF YEAR(SYSDATETIME()) <> YEAR(DATEADD(day, 1, SYSDATETIME()))
    PRINT 'Today is the last day of the year.';
ELSE
    PRINT 'Today is not the last day of the year.';
```

# The IF . . . ELSE Flow Element

- T-SQL uses three-valued logic and that the ELSE block is activated when the predicate is either FALSE or UNKNOWN.
  - Both FALSE and UNKNOWN are possible outcomes of the predicate
- If we need to run more than one statement in the IF or ELSE sections, we need to use a statement block.
  - We mark the boundaries of a statement block with the BEGIN and END keywords.

```
IF <Boolean Expression>
    <SQL statement> | BEGIN <code series> END
[ELSE
    <SQL statement> | BEGIN <code series> END]
```

```
IF <Expression>
BEGIN --First block of code starts here -- executes only if
      --expression is TRUE
    Statement that executes if expression is TRUE
    Additional statements
    ...
    Still going with statements from TRUE expression
    IF <Expression> --Only executes if this block is active
    BEGIN
        Statement that executes if both outside and inside
        expressions are TRUE
        Additional statements
        ...
        Still statements from both TRUE expressions
    END
    Out of the condition from inner condition, but still
    part of first block
END --First block of code ends here
ELSE
BEGIN
    Statement that executes if expression is FALSE
    Additional statements
    ...
    Still going with statements from FALSE expression
END
```

# The WHILE Flow Element

- The WHILE element executes a statement or statement block repeatedly while the predicate we specify after the WHILE keyword is TRUE.
  - When the predicate is FALSE or UNKNOWN, the loop terminates.
- T-SQL doesn't provide a built-in looping element that executes a predetermined number of times

```
WHILE <Boolean expression>
      <sql statement> |
[BEGIN
      <statement block>
      [BREAK]
      <sql statement> | <statement block>
      [CONTINUE]
END]
```

```
DECLARE @i AS INT = 1;
WHILE @i <= 10
BEGIN
    PRINT @i;
    SET @i = @i + 1;
END;
```

# The WHILE Flow Element

- The BREAK command allow we to break out of the current loop and proceed to execute the statement that appears after the loop's body
- The CONTINUE command skips the rest of the activity in the current iteration and evaluate the loop's predicate again

```
DECLARE @i AS INT = 1;
WHILE @i <= 10
BEGIN
    IF @i = 6 BREAK;
    PRINT @i;
    SET @i = @i + 1;
END;
```

```
1
2
3
4
5
```

```
DECLARE @i AS INT = 0;
WHILE @i < 10
BEGIN
    SET @i = @i + 1;
    IF @i = 6 CONTINUE;
    PRINT @i;
END;
```

```
1
2
3
4
5
7
8
9
10
```

# Cursors

- A query with an ORDER BY clause returns what standard SQL calls a cursor - a nonrelational result with order guaranteed among rows.
- T-SQL also supports an object called cursor that allows we to process rows from a result set of a query one at a time and in a requested order.
- This is in contrast to using set-based queries - normal queries without a cursor for which we manipulate the set or multiset as a whole and cannot rely on order.

# Working with a Cursor

- Working with a cursor generally involves the following steps:
  1. Declare the cursor based on a query.
  2. Open the cursor.
  3. Fetch attribute values from the first cursor record into variables.
  4. Until the end of the cursor is reached (while the value of a function called @@FETCH_STATUS is 0), loop through the cursor records; in each iteration of the loop, fetch attribute values from the current cursor record into variables and perform the processing needed for the current row.
  5. Close the cursor.
  6. Deallocate the cursor.

```
DECLARE @Result TABLE
(
    CustomerID      INT,
    OrderDate       DATETIME,
    TotalDue        MONEY,
    RunningTotal    MONEY
);

DECLARE
    @CustomerID    AS INT,
    @PrvCustomerID AS INT,
    @OrderDate     DATETIME,
    @TotalDue      MONEY,
    @RunningTotal  MONEY;

DECLARE C CURSOR FAST_FORWARD FOR --/* read only, forward only */
    SELECT    CustomerID, OrderDate, TotalDue
    FROM      Sales.SalesOrderHeader
    ORDER BY CustomerID, OrderDate;
```

```
OPEN C;

FETCH NEXT FROM C INTO @CustomerID, @OrderDate, @TotalDue;

SELECT @PrvCustomerID = @CustomerID, @RunningTotal = 0;

WHILE @@FETCH_STATUS = 0
BEGIN
    IF @CustomerID <> @PrvCustomerID
        SELECT @PrvCustomerID = @CustomerID, @RunningTotal = 0;

    SET @RunningTotal = @RunningTotal + @TotalDue;
    INSERT INTO @Result VALUES(@CustomerID, @OrderDate, @TotalDue,
    @RunningTotal);

    FETCH NEXT FROM C INTO @CustomerID, @OrderDate, @TotalDue;
END

CLOSE C;

DEALLOCATE C;
```

```
SELECT  CustomerID,
        CONVERT(VARCHAR(7), OrderDate, 121) AS OrderMonth,
        TotalDue,
        RunningTotal,
        SUM(TotalDue) OVER ( PARTITION BY CustomerID
                             ORDER BY OrderDate
                             ROWS BETWEEN UNBOUNDED PRECEDING
                              AND CURRENT ROW ) AS RunningTotal2
FROM    @Result
ORDER BY CustomerID, OrderDate;
```

1. When you use cursors you pretty much go against the relational

| CustomerID | OrderMonth | TotalDue | RunningTotal | RunningTotal2 |
|---|---|---|---|---|
| 11000 | 2011-06 | 3756.989 | 3756.989 | 3756.989 |
| 11000 | 2013-06 | 2587.8769 | 6344.8659 | 6344.8659 |
| 11000 | 2013-10 | 2770.2682 | 9115.1341 | 9115.1341 |
| 11001 | 2011-06 | 3729.364 | 3729.364 | 3729.364 |
| 11001 | 2013-06 | 2674.0227 | 6403.3867 | 6403.3867 |
| 11001 | 2014-05 | 650.8008 | 7054.1875 | 7054.1875 |
| 11002 | 2011-06 | 3756.989 | 3756.989 | 3756.989 |
| 11002 | 2013-06 | 2535.964 | 6292.953 | 6292.953 |
| 11002 | 2013-07 | 2673.0613 | 8966.0143 | 8966.0143 |
| 11003 | 2011-05 | 3756.989 | 3756.989 | 3756.989 |
| 11003 | 2013-06 | 2562.4508 | 6319.4398 | 6319.4398 |
| ... | | | | |

# Overview

- Variables
- Flow Elements
- **Temporary Tables**
  - Local/Global Temporary Tables
  - Table Variables
  - Dynamic SQL
- Routines

# Temporary Tables

- When we need to temporarily store data in tables, in certain cases we might prefer not to work with permanent tables.
  - Suppose we need the data to be visible only to the current session, or even only to the current batch.
- SQL Server supports three kinds of temporary tables that we might find more convenient to work with than permanent tables in such cases:
  - local temporary tables,
  - global temporary tables, and
  - table variables.

# Local Temporary Tables

- We create a local temporary table by naming it with a *single number sign* as a prefix, such as #T1.
- All three kinds of temporary tables are created in the tempdb database.
- A local temporary table is visible only to the session that created it, in the creating level and all inner levels in the call stack (inner procedures, functions, triggers, etc).
- A local temporary table is destroyed automatically by SQL Server when the creating level in the call stack goes out of scope.

# Local Temporary Tables

- One obvious scenario for which local temporary tables are useful is when we have a process that needs to store intermediate results temporarily - such as during a loop - and later query the data.
- Another scenario is when we need to access the result of some expensive processing multiple times.
  - Example, suppose that we need to aggregate order quantities by order year, and join two instances of the aggregated data to compare each year’s total quantity with the previous year.

```
IF OBJECT_ID('tempdb.#MyOrderTotalsByYear') IS NOT NULL
    DROP TABLE #MyOrderTotalsByYear;
GO

CREATE TABLE #MyOrderTotalsByYear
(
    OrderYear INT NOT NULL PRIMARY KEY,
    Total     INT NOT NULL
);

INSERT INTO #MyOrderTotalsByYear
    SELECT YEAR(OrderDate) AS OrderYear, SUM(SubTotal) AS Total
    FROM   Sales.SalesOrderHeader
    GROUP BY YEAR(OrderDate);

SELECT    Cur.OrderYear, Cur.Total AS CurYearTotal,
                         Prv.Total AS PrvYearTotal
FROM      #MyOrderTotalsByYear AS Cur
LEFT JOIN #MyOrderTotalsByYear AS Prv
       ON Cur.OrderYear = Prv.OrderYear + 1;
```

```
OrderYear   CurYearTotal PrvYearTotal
----------- ------------ ------------
2011        12641672     NULL
2012        33524301     12641672
2013        43622479     33524301
2014        20057929     43622479
```

```
INSERT INTO #MyOrderTotalsByYear
    SELECT YEAR(OrderDate) AS OrderYear, SUM(SubTotal) AS Total
    FROM   Sales.SalesOrderHeader
    GROUP BY YEAR(OrderDate);

SELECT    Cur.OrderYear, Cur.Total AS CurYearTotal,
                         Prv.Total AS PrvYearTotal
FROM      #MyOrderTotalsByYear AS Cur
LEFT JOIN #MyOrderTotalsByYear AS Prv
       ON Cur.OrderYear = Prv.OrderYear + 1;
```

```
SELECT YEAR(OrderDate) AS OrderYear, SUM(SubTotal) AS CurYearTotal,
       LAG(SUM(SubTotal)) OVER(ORDER BY YEAR(OrderDate)) AS
                                                    PrvYearTotal
FROM   Sales.SalesOrderHeader
GROUP BY YEAR(OrderDate);
```

```
OrderYear   CurYearTotal PrvYearTotal
----------- ------------ ------------
2011        12641672     NULL
2012        33524301     12641672
2013        43622479     33524301
2014        20057929     43622479
```

# Global Temporary Tables

- When we create a global temporary table, it is visible to all other sessions.
- Global temporary tables are destroyed automatically by SQL Server when the creating session disconnects and there are no active references to the table.
- We create a global temporary table by naming it with two number signs as a prefix, such as ##T1.
- Global temporary tables are useful when we want to share temporary data with everyone.

# Global Temporary Tables

- For example, the following code creates a global temporary table called ##Globals with columns called id and val.

```
CREATE TABLE ##Globals
(
    id sysname NOT NULL PRIMARY KEY,
    val SQL_VARIANT NOT NULL
);
```

- Anyone can insert rows into the table.

```
INSERT INTO ##Globals(id, val) VALUES(N'i', CAST(10 AS INT));
```

- Anyone can modify and retrieve data from the table.

```
SELECT val FROM ##Globals WHERE id = N'i';
```

# Table Variables

- Table variables are similar to local temporary tables in some ways and different in others.
- We declare table variables much like we declare other variables, by using the DECLARE statement.
- As with local temporary tables, table variables have a physical presence as a table in the tempdb database, contrary to the common misconception that they exist only in memory.
- Like local temporary tables, table variables are visible only to the creating session, but they have a more limited scope: only the current batch.

# Table Variables

- Table variables are visible neither to inner batches in the call stack nor to subsequent batches in the session.
- If an explicit transaction is rolled back, changes made to temporary tables in that transaction are rolled back as well; however, changes made to table variables by statements that completed in the transaction aren't rolled back.
- For example, creating a table variable

```
DECLARE @MyOrderTotalsByYear TABLE
(
    OrderYear INT NOT NULL PRIMARY KEY,
    Total     INT NOT NULL
);
```

```
DECLARE @MyOrderTotalsByYear TABLE
(
    OrderYear INT NOT NULL PRIMARY KEY,
    Total     INT NOT NULL
);

INSERT INTO @MyOrderTotalsByYear
    SELECT YEAR(OrderDate) AS OrderYear, SUM(SubTotal) AS Total
    FROM   Sales.SalesOrderHeader
    GROUP BY YEAR(OrderDate);

SELECT    Cur.OrderYear, Cur.Total AS CurYearTotal,
                         Prv.Total AS PrvYearTotal
FROM      @MyOrderTotalsByYear AS Cur
LEFT JOIN @MyOrderTotalsByYear AS Prv
       ON Cur.OrderYear = Prv.OrderYear + 1;
```

```
OrderYear   CurYearTotal PrvYearTotal
----------- ------------ ------------
2011        12641672     NULL
2012        33524301     12641672
2013        43622479     33524301
2014        20057929     43622479
```

# Dynamic SQL

- SQL Server allows we to construct a batch of T-SQL code as a character string and then execute that batch. This capability is called dynamic SQL.
- SQL Server provides two ways of executing dynamic SQL: using the EXEC (short for EXECUTE) command, and using the sp_executesql stored procedure.
- Dynamic SQL is useful for several purposes, for example, constructing elements of the code based on querying the actual data
  - Constructing a PIVOT query dynamically when we don't know ahead of time which elements should appear in the IN clause of the PIVOT operator

# The EXEC Command

- The EXEC command is the original technique provided in T-SQL for executing dynamic SQL.
- EXEC accepts a character string in parentheses as input and executes the batch of code within the character string.
- For example,

```
DECLARE @sql AS VARCHAR(100);
SET @sql = 'PRINT ''This message was printed by a dynamic SQL
batch.'';';
EXEC(@sql);
```

This example stores a character string with a PRINT statement in the variable @sql and then uses the EXEC command to invoke the batch of code stored within the variable.

# Using PIVOT with Dynamic SQL

- PIVOT relational operator can be used to transform columns distinct values as Columns in the result set.
  - Mentioning all the distinct column values in the PIVOT operators PIVOT columns IN clause.
- This type of PIVOT query is called Static PIVOT query.
  - The PIVOT query result unless it is mentioned in the PIVOT Columns IN clause.
- For example,

```
SELECT *
FROM (SELECT ShipMethodID, YEAR(OrderDate) AS OrderYear, Freight
  FROM Sales.SalesOrderHeader) AS D
  PIVOT(SUM(Freight) FOR OrderYear IN([2011],[2012],[2013],[2014]))
                                                          AS P;
```

- For example,

```
SELECT *
FROM (SELECT ShipMethodID, YEAR(OrderDate) AS OrderYear, Freight
  FROM Sales.SalesOrderHeader) AS D
  PIVOT(SUM(Freight) FOR OrderYear IN([2011],[2012],[2013],[2014]))
                                                          AS P;
```

| ShipMethodID | 2011 | 2012 | 2013 | 2014 |
|---|---|---|---|---|
| 1 | 96578.0647 | 159765.1035 | 268303.4914 | 209321.0124 |
| 5 | 263904.6728 | 828663.3013 | 1003936.9575 | 352957.6482 |

- With the static query, we have to know ahead of time which values (order years in this case) to specify in the IN clause of the PIVOT operator. This means that we need to revise the code every year.
- Instead, we can query the distinct order years from the data, construct a batch of dynamic SQL code based on the years that we queried, and execute the dynamic SQL batch.

```
DECLARE @Sql AS NVARCHAR(1000), @OrderYear AS INT, @First AS INT;

DECLARE C CURSOR FAST_FORWARD FOR
    SELECT DISTINCT(YEAR(OrderDate)) AS OrderYear
    FROM   Sales.SalesOrderHeader
    ORDER BY OrderYear;

SET @First = 1;
SET @Sql = N'SELECT *
FROM (SELECT ShipMethodID, YEAR(OrderDate) AS OrderYear, Freight
      FROM Sales.SalesOrderHeader) AS D
      PIVOT(SUM(Freight) FOR OrderYear IN(';

OPEN C;
FETCH NEXT FROM C INTO @OrderYear;
WHILE @@fetch_status = 0
BEGIN
    IF @First = 0
        SET @sql = @sql + N','
    ELSE
        SET @First = 0;

    SET @Sql = @Sql + '[' + CAST(@OrderYear AS NVARCHAR(4)) + ']';
    FETCH NEXT FROM C INTO @OrderYear;
END
CLOSE C;
DEALLOCATE C;

SET @Sql = @Sql + N')) AS P;';

EXEC(@Sql);
```

# Overview

- Variables
- Flow Elements
- Temporary Tables
- **Routines**
  - User-Defined Functions
  - Stored Procedures
  - Triggers

# Routines

- Routines are programmable objects that encapsulate code to calculate a result or to execute activity.
- SQL Server supports three types of routines: user-defined functions, stored procedures, and triggers.
- SQL Server allows us to choose whether to develop a routine with T-SQL or with Microsoft .NET code based on the CLR integration in the product.
  - When the task at hand mainly involves data manipulation, T-SQL is usually a better choice.
  - When the task is more about iterative logic, string manipulation, or computationally intensive operations, .NET code is usually a better choice.

# User-Defined Functions

- The purpose of a user-defined function (UDF) is to encapsulate logic that calculates something, possibly based on input parameters, and return a result.
- SQL Server supports scalar and table-valued UDFs. Scalar UDFs return a single value; table-valued UDFs return a table.
- One benefit of using UDFs is that we can incorporate them in queries.
  - Scalar UDFs can appear anywhere in the query where an expression that returns a single value can appear (for example, in the SELECT list).
  - Table UDFs can appear in the FROM clause of a query.

# User-Defined Functions

- UDFs are not allowed to have any side effects.
- This obviously means that UDFs are not allowed to apply any schema or data changes in the database.
  - For example, in SQL Server, invoking the RAND function to return a random value or the NEWID function to return a globally unique identifier (GUID) has side effects.
  - RAND and NEWID have side effects, we ’re not allowed to use them in our UDFs.

# User-Defined Functions

- A function can have more than just a RETURN clause in its body. It can have code with flow elements, calculations, and more.
- But the function must have a RETURN clause that returns a value.

```
CREATE FUNCTION <function name>( [ <@parameter name> [AS] <data type>
                                  [ = <default value> [READONLY]]
                                  [ ,...n ] ] )
RETURNS {<scalar type> | TABLE [(<table definition>)]}
    [ WITH [ENCRYPTION] | [SCHEMABINDING] |
      [RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT] |
      [EXECUTE AS { CALLER|SELF|OWNER|<'user name'> } ]
    ] [AS]
    { EXTERNAL NAME <external method> |
BEGIN
    [<function statements>]
    {RETURN <type as defined in RETURNS clause>|RETURN (<SELECT statement>)}
END }[;]
```

- For example, the following code creates a UDF called GetAge that returns the age of a person with a specified birth date at a specified event date.

```
IF OBJECT_ID('GetAge') IS NOT NULL DROP FUNCTION GetAge;
GO

CREATE FUNCTION GetAge
(
    @birthdate AS DATE,
    @eventdate AS DATE
)
RETURNS INT
AS
BEGIN
RETURN
    DATEDIFF(year, @birthdate, @eventdate)
    - CASE WHEN 100 * MONTH(@eventdate) + DAY(@eventdate)
                  < 100 * MONTH(@birthdate) + DAY(@birthdate)
           THEN 1 ELSE 0
      END;
END;
GO
```

Note: minus 1 year in cases for which the year, the event month, and the day are smaller than the birth month and day.

- To demonstrate using a UDF in a query, the following code queries the Employees table and invokes the GetAge function in the SELECT list to calculate the age of each employee today.

```
SELECT P.FirstName, P.LastName, E.BirthDate,
       GetAge(E.BirthDate, SYSDATETIME()) AS Age
FROM   HumanResources.Employee AS E
JOIN   Person.Person AS P
  ON   P.BusinessEntityID = E.BusinessEntityID
```

```
FirstName            LastName             BirthDate           Age
-------------------- -------------------- ---------- ----------
Ken                  Sánchez              1969-01-29          46
Terri                Duffy                1971-08-01          44
Roberto              Tamburello           1974-11-12          41
Rob                  Walters              1974-12-23          40
Gail                 Erickson             1952-09-27          63
...
```

# Stored Procedures

- Stored procedures are server-side routines that encapsulate T-SQL code.
- Stored procedures can have input and output parameters, they can return result sets of queries, and they are allowed to invoke code that has side effects.
- Not only can we modify data through stored procedures, we can also apply schema changes through them.

# Stored Procedures

- Compared to using ad-hoc code, the use of stored procedures gives we many benefits:
  - Stored procedures encapsulate logic.
  - Stored procedures give us better control of security.
  - Stored procedures give us performance benefits.

```
CREATE PROCEDURE | PROC <sproc name>
    [<parameter name> <data type> [VARYING] [= <default value>]
            [OUT[PUT]] [READONLY]
    [,<parameter name> <data type>[VARYING] [= <default value>]
           [OUT[PUT]] [READONLY] [, ... ] ...]
    [WITH RECOMPILE | ENCRYPTION |
         [EXECUTE AS { CALLER|SELF|OWNER|<'user name'>}]
    [FOR REPLICATION]
AS
    <code> | EXTERNAL NAME <assembly name>.<assembly class>.<method>
```

- For example, the following code creates a stored procedure called Sales.GetCustomerOrders.

```
IF OBJECT_ID('Sales.GetCustomerOrders', 'P') IS NOT NULL
    DROP PROC Sales.GetCustomerOrders;
GO

CREATE PROC Sales.GetCustomerOrders
    @CustomerID AS INT,
    @FromDate   AS DATETIME = '19000101',
    @ToDate     AS DATETIME = '99991231',
    @NumRows    AS INT OUTPUT
AS

    SET NOCOUNT ON;
    SELECT SalesOrderID, CustomerID, SalesPersonID, OrderDate
    FROM   Sales.SalesOrderHeader
    WHERE  CustomerID = @CustomerID
      AND  OrderDate >= @FromDate
      AND  OrderDate  < @ToDate;

    SET @NumRows = @@rowcount;
GO
```

- Here's an example of executing the procedure, requesting information about orders placed by the customer with the ID of 11101 in the year 2011.

```
DECLARE @RowCount AS INT;

EXEC Sales.GetCustomerOrders
     @CustomerID = 11101,
     @FromDate   = '20110101',
     @ToDate     = '20120101',
     @NumRows    = @RowCount OUTPUT;

SELECT @RowCount AS NumRows;
```

```
SalesOrderID CustomerID  SalesPersonID OrderDate
------------ ----------- ------------- -----------------------
44475        11101       NULL          2011-09-29 00:00:00.000

NumRows
-----------
1
```

# Recursion in Stored Procedures

- Recursion is one of those things that isn't used very often in programming.
- The brief version is that recursion is the situation where a piece of code calls itself.
- The recursion can go on and on up to a limit of 32 levels of recursion. Once SQL Server gets 32 levels deep, it raises an error and ends processing.

- Example, the factorial of 5 is 120 - that's 5*4*3*2*1.

```
CREATE PROC Factorial
    @ValueIn  INT,
    @ValueOut INT OUTPUT
AS
    DECLARE @InWorking  INT;
    DECLARE @OutWorking INT;

    IF @ValueIn > 1
    BEGIN
        SET  @InWorking = @ValueIn - 1;
        EXEC Factorial @InWorking, @OutWorking OUTPUT;
        SET  @ValueOut = @ValueIn * @OutWorking;
    END
    ELSE
    BEGIN
        SET @ValueOut = 1;
    END
GO

DECLARE @Out INT;
EXEC Factorial
     @ValueIn  = 5,
     @ValueOut = @Out OUTPUT;

SELECT @Out AS [5!]
```

| Feature | SP | Scalar UDF | Table UDF | View |
|---|---|---|---|---|
| Return tabular data | Yes | No | Yes | Yes |
| Return multiple sets of result | Yes | N/A | No | No |
| Update data | Yes | No | No | No |
| Create other objects | Yes | No | No | No |
| Call from a procedure | Yes | Yes | Yes | Yes |
| Can call a procedure | Yes | No | No | No |
| Can call within a SELECT list | No | Yes | No | No |
| Use to populate multiple columns in a table | Yes | No | Yes | Yes |
| Return value required | No | Yes | Yes (Table) | N/A |
| Return value optional | Yes | No | Yes | No |
| Takes parameters | Yes | Yes | Yes | No |
| Output parameters | Yes | No | No | No |

# Triggers

- A trigger is a special kind of stored procedure - one that CANNOT be executed explicitly. Instead, it is attached to an event.
- SQL Server supports the association of triggers with two kinds of events - data manipulation events (*DML triggers*) such as INSERT, and data definition events (*DDL triggers*) such as CREATE TABLE.
- Triggers in SQL Server fire per statement and not per modified row.

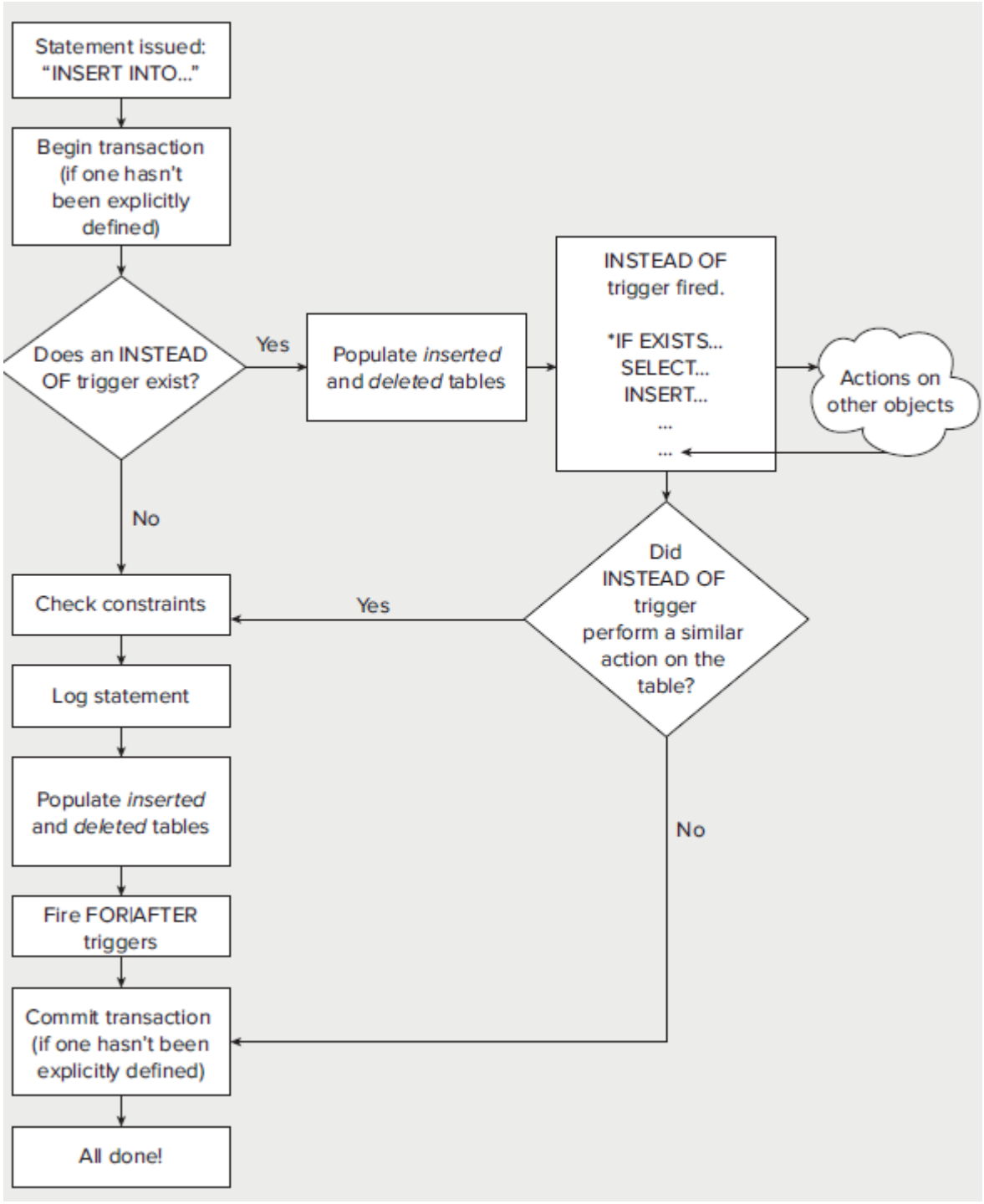
# DML Triggers

- SQL Server supports two kinds of *DML triggers* - *after* and *instead of*.
- An *after* trigger fires after the event it is associated with finishes and can only be defined on *permanent tables*.
- An *instead of* trigger fires instead of the event it is associated with and can be defined on *permanent tables* and *views*.
- In the trigger's code, we can access tables called *inserted* and *deleted* that contain the rows that were affected by the modification that caused the trigger to fire.

# DML Triggers

- The *inserted* table holds the new image of the affected rows in the case of INSERT and UPDATE actions.
- The *deleted* table holds the old image of the affected rows in the case of DELETE and UPDATE actions.
- In the case of instead of triggers, the *inserted* and *deleted* tables contain the rows that were supposed to be affected by the modification that caused the trigger to fire.

```
CREATE TRIGGER <trigger name>
    ON <table or view name>
    [WITH ENCRYPTION | EXECUTE AS <CALLER | SELF | <user> >]
    {{{FOR|AFTER} <[DELETE] [,] [INSERT] [,] [UPDATE]>} |INSTEAD OF}
    [NOT FOR REPLICATION]
AS
    < <sql statements> | EXTERNAL NAME <assembly method specifier> >
```


Statement issued: "INSERT INTO..."
Begin transaction (if one hasn't been explicitly defined)
Does an INSTEAD OF trigger exist?
Yes
Populate *inserted* and *deleted* tables
INSTEAD OF trigger fired.
*IF EXISTS... SELECT... INSERT... ... ...
Actions on other objects
No
Did INSTEAD OF trigger perform a similar action on the table?
Yes
Check constraints
Log statement
Populate *inserted* and *deleted* tables
Fire FOR|AFTER triggers
Commit transaction (if one hasn't been explicitly defined)
No
All done!

- For example, an after trigger audits inserts to a table.

```
IF OBJECT_ID('T1Audit', 'U') IS NOT NULL
    DROP TABLE T1Audit;

IF OBJECT_ID('T1', 'U') IS NOT NULL
    DROP TABLE T1;

CREATE TABLE T1
(
    Keycol  INT NOT NULL PRIMARY KEY,
    Datacol VARCHAR(10) NOT NULL
);

CREATE TABLE T1Audit
(
    AuditLsn    INT NOT NULL IDENTITY PRIMARY KEY,
    Dt DATETIME NOT NULL DEFAULT(SYSDATETIME()),
    LoginName   sysname NOT NULL DEFAULT(ORIGINAL_LOGIN()),
    Keycol      INT NOT NULL,
    Datacol     VARCHAR(10) NOT NULL
);
```

- To test the trigger, run the following code.

```
CREATE TRIGGER trg_T1_insert_audit
            ON T1
            AFTER INSERT
AS
    SET NOCOUNT ON;
    INSERT INTO T1Audit(Keycol, Datacol)
    SELECT Keycol, Datacol FROM inserted;
GO

INSERT INTO T1(Keycol, Datacol) VALUES(10, 'a');
INSERT INTO T1(Keycol, Datacol) VALUES(30, 'x');
INSERT INTO T1(Keycol, Datacol) VALUES(20, 'g');

SELECT * FROM T1Audit;
```

```
AuditLsn    Dt                      LoginName                    Keycol      Datacol
-------------------------------------------------------------------------------------
1           2015-11-16 18:21:36.117 A212-SLLUO\Administrator     10          a
2           2015-11-16 18:21:36.120 A212-SLLUO\Administrator     30          x
3           2015-11-16 18:21:36.127 A212-SLLUO\Administrator     20          g
```

# Example: Validating Data with a Trigger

```
CREATE TRIGGER Production.ProductIsRationed
            ON Production.ProductInventory
            FOR UPDATE
AS
    IF EXISTS
    (
        SELECT   1
          FROM   inserted AS I
          JOIN   deleted  AS D
            ON   I.ProductID  = D.ProductID
           AND   I.LocationID = D.LocationID
         WHERE  (D.Quantity - I.Quantity) > D.Quantity / 2
           AND   D.Quantity - I.Quantity > 0
    )
    BEGIN
        RAISERROR('Cannot reduce stock by more than 50%% at once.',16,1)
        ROLLBACK TRAN
    END
```

```
UPDATE Production.ProductInventory
SET    Quantity   = 1 -- Was 408 if we want to set it back
WHERE  ProductID  = 1
  AND  LocationID = 1
```

```
Msg 50000, Level 16, State 1, Procedure ProductIsRationed, Line
36 Cannot reduce stock by more than 50% at once.
Msg 3609, Level 16, State 1, Line 21 The transaction ended in
the trigger. The batch has been aborted.
```

# DDL Triggers

- SQL Server supports *DDL triggers*, which can be used for purposes such as auditing, policy enforcement, and change management.
- We create a database trigger for events with a database scope, such as CREATE TABLE.
- We create an all server trigger for events with a server scope, such as CREATE DATABASE. SQL Server supports only *after DDL triggers*; it DOESN'T support *instead of DDL triggers*.