In general, a trigger is a special kind of stored procedure that automatically executes when an event occurs in the database server.

DML stands for Data Manipulation Language. INSERT, UPDATE, and DELETE statements are DML statements. DML triggers are fired, when ever data is modified using INSERT, UPDATE, and DELETE events.

DML triggers can be again classified into 2 types.
1. After triggers (Sometimes called as FOR triggers)
2. Instead of triggers

After triggers, as the name says, fires after the triggering action. The INSERT, UPDATE, and DELETE statements, causes an after trigger to fire after the respective statements complete execution.

On ther hand, as the name says, INSTEAD of triggers, fires instead of the triggering action. The INSERT, UPDATE, and DELETE statements, can cause an INSTEAD OF trigger to fire INSTEAD OF the respective statement execution.

We will use tblEmployee and tblEmployeeAudit tables for our examples

SQL Script to create tblEmployee table:
```
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Salary int,
  Gender nvarchar(10),
  DepartmentId int
)
```

Insert data into tblEmployee table
```
Insert into tblEmployee values (1,'John', 5000, 'Male', 3)
Insert into tblEmployee values (2,'Mike', 3400, 'Male', 2)


Insert into tblEmployee values (3,'Pam', 6000, 'Female', 1)
```

tblEmployee

Id	Name	Salary	Gender	DepartmentId
1	John	5000	Male	3
2	Mike	3400	Male	2
3	Pam	6000	Female	1
4	Todd	4800	Male	4
5	Sara	3200	Female	1
6	Ben	4800	Male	3

SQL Script to create tblEmployeeAudit table:

```
CREATE TABLE tblEmployeeAudit
(
  Id int identity(1,1) primary key,
  AuditData nvarchar(1000)
)
```

When ever, a new Employee is added, we want to capture the ID and the date and time, the new employee is added in tblEmployeeAudit table. The easiest way to achieve this, is by having an AFTER TRIGGER for INSERT event.

Example for AFTER TRIGGER for INSERT event on tblEmployee table:

```
CREATE TRIGGER tr_tblEMployee_ForInsert
ON tblEmployee
FOR INSERT
AS
BEGIN
 Declare @Id int
 Select @Id = Id from inserted

 insert into tblEmployeeAudit
 values('New employee with Id  = ' + Cast(@Id as nvarchar(5)) + ' is
added at ' + cast(Getdate() as nvarchar(20)))
END
```

In the trigger, we are getting the id from inserted table. So, what is this inserted table? INSERTED table, is a special table used by DML triggers. When you add a new row into tblEmployee table, a copy of the row will also be made into inserted table, which only a trigger can access. You cannot access this table outside the context of the trigger.

The structure of the inserted table will be identical to the structure of tblEmployee table.

So, now if we execute the following INSERT statement on tblEmployee. Immediately, after inserting the row into tblEmployee table, the trigger gets fired (executed automatically), and a row into tblEmployeeAudit, is also inserted.

```
Insert into tblEmployee values (7,'Tan', 2300, 'Female', 3)
```

Along, the same lines, let us now capture audit information, when a row is deleted from the table, tblEmployee.
Example for AFTER TRIGGER for DELETE event on tblEmployee table:

```
CREATE TRIGGER tr_tblEMployee_ForDelete
ON tblEmployee
FOR DELETE
AS
BEGIN
 Declare @Id int
 Select @Id = Id from deleted

 insert into tblEmployeeAudit
 values('An existing employee with Id  = ' + Cast(@Id as nvarchar(5)) + '
is deleted at ' + Cast(Getdate() as nvarchar(20)))
END
```

The only difference here is that, we are specifying, the triggering event as **DELETE** and retrieving the deleted row ID from **DELETED** table. DELETED table, is a special table used by DML triggers. When you delete a row from tblEmployee table, a copy of the deleted row will be made available in DELETED table, which only a trigger can access. Just like INSERTED table, DELETED table cannot be accessed, outside the context of the trigger and, the structure of the DELETED table will be identical to the structure of tblEmployee table.

In the next session, we will talk about AFTER trigger for UPDATE event. **Triggers make use of 2 special tables**, INSERTED and DELETED. The inserted table contains the updated data and the deleted table contains the old data. The After trigger for UPDATE event, makes use of both inserted and deleted tables.

Create AFTER UPDATE trigger script:

```
Create trigger tr_tblEmployee_ForUpdate
```

```
on tblEmployee
for Update
as
Begin
 Select * from deleted
 Select * from inserted
End
```

Now, execute this query:

```
Update tblEmployee set Name = 'Tods', Salary = 2000,
Gender = 'Female' where Id = 4
```

Immediately after the UPDATE statement execution, the AFTER UPDATE trigger gets fired, and you should see the contenets of INSERTED and DELETED tables.

The following AFTER UPDATE trigger, audits employee information upon UPDATE, and stores the audit data in tblEmployeeAudit table.

```
Alter trigger tr_tblEmployee_ForUpdate
on tblEmployee
for Update
as
Begin
    -- Declare variables to hold old and updated data
    Declare @Id int
    Declare @OldName nvarchar(20), @NewName nvarchar(20)
    Declare @OldSalary int, @NewSalary int
    Declare @OldGender nvarchar(20), @NewGender nvarchar(20)
    Declare @OldDeptId int, @NewDeptId int

    -- Variable to build the audit string
    Declare @AuditString nvarchar(1000)

    -- Load the updated records into temporary table
    Select *
    into #TempTable
    from inserted

    -- Loop thru the records in temp table
    While(Exists(Select Id from #TempTable))
    Begin
        --Initialize the audit string to empty string
        Set @AuditString = ''

        -- Select first row data from temp table
```

```
            Select Top 1 @Id = Id, @NewName = Name,
            @NewGender = Gender, @NewSalary = Salary,
            @NewDeptId = DepartmentId
            from #TempTable

            -- Select the corresponding row from deleted table
            Select @OldName = Name, @OldGender = Gender,
            @OldSalary = Salary, @OldDeptId = DepartmentId
            from deleted where Id = @Id

    -- Build the audit string dynamically
            Set @AuditString = 'Employee with Id = ' + Cast(@Id as
nvarchar(4)) + ' changed'
            if(@OldName <> @NewName)
                Set @AuditString = @AuditString + ' NAME from ' +
@OldName + ' to ' + @NewName

            if(@OldGender <> @NewGender)
                Set @AuditString = @AuditString + ' GENDER from ' +
@OldGender + ' to ' + @NewGender

            if(@OldSalary <> @NewSalary)
                Set @AuditString = @AuditString + ' SALARY from
' + Cast(@OldSalary as nvarchar(10))+ ' to ' + Cast(@NewSalary as
nvarchar(10))

    if(@OldDeptId <> @NewDeptId)
                Set @AuditString = @AuditString + ' DepartmentId from
' + Cast(@OldDeptId as nvarchar(10))+ ' to ' + Cast(@NewDeptId as
nvarchar(10))

            insert into tblEmployeeAudit values(@AuditString)

            -- Delete the row from temp table, so we can move to the next
row
            Delete from #TempTable where Id = @Id
     End
End
```

Id	Name	Gender	DepartmentId
1	Sam	Male	1
2	Ram	Male	1
3	Sara	Female	3
4	Todd	Male	2
5	John	Male	3
6	Sana	Female	2
7	James	Male	1
8	Rob	Male	2
9	Steve	Male	1
10	Pam	Female	2

Creating a simple stored procedure without any parameters: This stored procedure, retrieves Name and Gender of all the employees. To create a stored procedure we use, **CREATE PROCEDURE** or **CREATE PROC** statement.

```
Create Procedure spGetEmployees
as
Begin
  Select Name, Gender from tblEmployee
End
```

Note: When naming user defined stored procedures, Microsoft recommends not to use **"sp_"** as a prefix. All system stored procedures, are prefixed with **"sp_"**. This avoids any ambiguity between user defined and system stored procedures and any conflicts, with some future system procedure.

To execute the stored procedure, you can just type the procedure name and press F5, or use EXEC or EXECUTE keywords followed by the procedure name as shown below.

1. spGetEmployees
2. EXEC spGetEmployees
3. Execute spGetEmployees

Note: You can also right click on the procedure name, in object explorer in SQL Server Management Studio and select EXECUTE STORED PROCEDURE.

Creating a stored procedure with input parameters: This SP, accepts GENDER and DEPARTMENTID parameters. Parameters and variables have an @ prefix in their name.

```
Create Procedure spGetEmployeesByGenderAndDepartment
@Gender nvarchar(50),
@DepartmentId int
as
Begin
  Select Name, Gender from tblEmployee Where Gender =
@Gender and DepartmentId = @DepartmentId
End
```

To invoke this procedure, we need to pass the value for @Gender and @DepartmentId parameters. If you don't specify the name of the parameters, you have to first pass value for @Gender parameter and then for @DepartmentId.

```
EXECUTE spGetEmployeesByGenderAndDepartment 'Male', 1
```

On the other hand, if you change the order, you will get an error stating "Error converting data type varchar to int." This is because, the value of **"Male"** is passed into @DepartmentId parameter. Since @DepartmentId is an integer, we get the type conversion error.

```
spGetEmployeesByGenderAndDepartment 1, 'Male'
```

When you specify the names of the parameters when executing the stored procedure the order doesn't matter.

```
EXECUTE spGetEmployeesByGenderAndDepartment
@DepartmentId=1, @Gender = 'Male'
```

To view the text, of the stored procedure

1. Use system stored procedure sp_helptext 'SPName'

OR

2. Right Click the SP in Object explorer -> Scrip Procedure as -> Create To -> New Query Editor Window

To change the stored procedure, use ALTER PROCEDURE statement:

```
Alter Procedure spGetEmployeesByGenderAndDepartment
@Gender nvarchar(50),
@DepartmentId int
```

```
as
Begin
  Select Name, Gender from tblEmployee Where Gender = @Gender
and DepartmentId = @DepartmentId order by Name
End
```

To encrypt the text of the SP, use WITH ENCRYPTION option. Once, encrypted, you cannot view the text of the procedure, using sp_helptext system stored procedure. There are ways to obtain the original text, which we will talk about in a later session.

```
Alter Procedure spGetEmployeesByGenderAndDepartment
@Gender nvarchar(50),
@DepartmentId int
WITH ENCRYPTION
as
Begin
  Select Name, Gender from tblEmployee Where Gender = @Gender
and DepartmentId = @DepartmentId
End
```

To delete the SP, use DROP PROC 'SPName' or DROP PROCEDURE 'SPName'

In the next seesion, we will learn creating stored procedures with OUTPUT parameters.

Stored procedures with output parameters - Part 19

In this video, we will learn about, creating stored procedures with output parameters. Please watch Part 18 of this video series, before watching this video.

Id	Name	Gender	DepartmentId
1	Sam	Male	1
2	Ram	Male	1
3	Sara	Female	3
4	Todd	Male	2
5	John	Male	3
6	Sana	Female	2
7	James	Male	1
8	Rob	Male	2
9	Steve	Male	1
10	Pam	Female	2

To create an SP with output parameter, we use the keywords OUT or OUTPUT. @EmployeeCount is an OUTPUT parameter. Notice, it is specified with OUTPUT keyword.

```
Create Procedure spGetEmployeeCountByGender
@Gender nvarchar(20),
@EmployeeCount int Output
as
Begin
 Select @EmployeeCount = COUNT(Id)
 from tblEmployee
 where Gender = @Gender
End
```

To execute this stored procedure with OUTPUT parameter

1. First initialise a variable of the **same datatype** as that of the **output parameter**. We have declared @EmployeeTotal integer variable.
2. Then pass the @EmployeeTotal variable to the SP. You have to specify the **OUTPUT** keyword. If you don't specify the OUTPUT keyword, the variable will be **NULL**.

3. Execute

```
Declare @EmployeeTotal int
Execute spGetEmployeeCountByGender 'Female',
@EmployeeTotal output
Print @EmployeeTotal
```

If you don't specify the OUTPUT keyword, when executing the stored procedure, the @EmployeeTotal variable will be NULL. Here, we have not specified OUTPUT keyword. When you execute, you will see **'@EmployeeTotal is null'** printed.

```
Declare @EmployeeTotal int
Execute spGetEmployeeCountByGender 'Female', @EmployeeTotal
if(@EmployeeTotal is null)
 Print '@EmployeeTotal is null'
else
 Print '@EmployeeTotal is not null'
```

You can pass parameters in any order, when you use the parameter names. Here, we are first passing the OUTPUT parameter and then the input @Gender parameter.

```
Declare @EmployeeTotal int
Execute spGetEmployeeCountByGender @EmployeeCount =
@EmployeeTotal OUT, @Gender = 'Male'
Print @EmployeeTotal
```

The following system stored procedures, are extremely useful when working procedures.
sp_help SP_Name : View the information about the stored procedure, like parameter names, their datatypes etc. sp_help can be used with any database object, like tables, views, SP's, triggers etc. Alternatively, you can also press ALT+F1, when the name of the object is highlighted.

sp_helptext SP_Name : View the Text of the stored procedure

sp_depends SP_Name : View the dependencies of the stored procedure. This system SP is very useful, especially if you want to check, if there are any stored procedures that are referencing a table that you are abput to drop. sp_depends can also be used with other database objects like table etc.

Note: All parameter and variable names in SQL server, need to have the @symbol.