```
/****************/
/* SQL SERVER   */
/*    INDEX     */
/****************/

--24-- BUILT IN STRING FUNCTIONS (SYSTEM FUNCTIONS) CONT. II
--25-- DATETIME DATA TYPES AND FUNCTIONS
--26-- BUILT IN DATETIME FUNCTIONS (SYSTEM FUNCTIONS) CONT.
--27-- BUILT IN DATETIME FUNCTIONS (SYSTEM FUNCTIONS) CONT. II
--28-- BUILT IN FUNCTIONS: CAST AND CONVERT
--29-- BUILT IN FUNCTIONS: MATHEMATICAL FUNCTIONS
--30-- USER DEFINED FUNCTIONS: SCALAR VALUED FUNCTIONS
--31-- USER DEFINED FUNCTIONS: IN LINE TABLE VALUED FUNCTIONS
--32-- USER DEFINED FUNCTIONS: MULTI STATEMENT TABLE VALUED FUNCTIONS
--33-- FUNCTIONS: IMPORTANT CONCEPTS
--34-- TEMPORARY TABLES
--35-- INDEXES
--36-- INDEXES: CLUSTERED AND NON CLUSTRED
--37-- INDEXES: UNIQUE AND NON-UNIQUE
--38-- INDEXES: ADVANTAGES AND DISADVANTAGES
--39-- VIEWS
--40-- VIEWS: UPDATABLE VIEWS
--41-- VIEWS: INDEXED VIEWS
--42-- VIEW LIMITATIONS
--43-- AFTER DML TRIGGERS: INSERT, DELETE
--44-- AFTER DML TRIGGERS: UPDATE
--45-- INSTEAD OF DML TRIGGERS: INSERT
--46-- INSTEAD OF DML TRIGGERS: UPDATE
--47-- INSTEAD OF DML TRIGGERS: DELETE

/***************/
/* SQL SERVER */
/***************/

-------------------------------------------------------
-- 24 -- BUILT IN STRING FUNCTIONS (CONT.) II -- 24 --
-------------------------------------------------------

SELECT REPLICATE('Pragim', 3)
-- will repeat Pragim three times

-- mask the email address with five star symbols
SELECT FirstName, LastName,
                SUBSTRING(Email, 1, 2) + REPLICATE('*', 5) +
                SUBSTRING(Email, CHARINDEX('@', Email), LEN(Email) -
CHARINDEX('@',Email) + 1) AS Email
FROM tblEmployee
```

```
SELECT '    '

SELECT SPACE(5)
-- you can be sure to add exactly 5 spaces


SELECT FirstName + SPACE(5) + LastName AS   FullName
FROM tblEmployee



-- returns the starting position of a pattern in a specified expression
-- similar to char index. Returns the index of a string.
-- with PATINDEX you can use wildcards

SELECT Email, PATINDEX('%@aaa.com', Email) AS FirstOccurrence
FROM tblEmployee
WHERE PATINDEX('%@aaa.com', Email) > 0
-- will return the full email addresses and the first ocurrence of
-- the expression (the index) only for email addresses ending in @aaa.com


SELECT Email, REPLACE(Email, '.com', '.net') AS ConvertedEmail
FROM tblEmployee
-- all emails ending in .com are replaced with ending in .net

SELECT FirstName, LastName, Email, STUFF(Email, 2, 3, '*****') AS StuffedEmail
FROM tblEmployee
-- return first name, last name, email, and returns the email address, with the
-- following replacements, from the second character on, for three characters,
-- iti will enter five stars, inserting those stars instead of whatever characters
-- would be in their way


-------------------------------------
-- 25 -- DATETIME FUNCTIONS -- 25 --
-------------------------------------

-- DATETIME DATA TYPES

-- Choose depending on performance, and intention

-- time, date (e.g. only the date of birth of a person)
-- smalldatetime, datetime (e.g. the date and time of the birth of a person)
-- datetime2(more precise to nanoseconds),
-- datetimeoffset (includes the timezone offset)

-- UTC -- Coordinated Universal Time
```

```
-- Time based on which the world regulates time
-- GMT most of the time is synonymous with UTC


-- DATETIME DATA FUNCTIONS

SELECT * FROM tblDateTime
INSERT INTO tblDateTime VALUES (GETDATE(), GETDATE(),GETDATE(),
GETDATE(),GETDATE(), GETDATE())

UPDATE tblDateTime SET c_datetimeoffset = '2012-08-31 21:34:36.9200000 +10:00'
WHERE c_datetimeoffset = '2012-08-31 21:24:36.9200000 +00:00'

-------

SELECT GETDATE(), 'GETDATE()' -- will get date time from machine right now
-- can get date time of where you have sqlserver installed rather than your computer
SELECT CURRENT_TIMESTAP, 'CURRENT_TIMESTAP' -- ansi sql equivalent to get date
-- other wise it is the same
SELECT SYSDATETIME(), 'SYSDATETIME()' -- gives fractional seconds precision
SELECT SYSDATETIMEOFFSET(), 'SYSDATETIMEOFFSET()' -- will give you the time in
-- utc plus the offset from utc (one hour ahead will be +1:00)
SELECT GETUTCDATE(), 'GETUTCDATE()' -- gets the utc date
SELECT SYSUTCDATETIME(), 'SYSUTCDATETIME' -- gets the utc date and time


--------------------------------------------
-- 26 -- DATETIME FUNCTIONS (CONT.) -- 26 --
--------------------------------------------

-- ISDATE, checks  if the value is a valid date, time or datetime

SELECT ISDATE('Pragim') -- returns 0
SELECT ISDATE(GETDATE()) -- returns 1, so it is a valid date, time or datetime
SELECT ISDATE('2012-08-31 21:02:04.167') -- returns 1
SELECT ISDATE('2012-08-31 21:02:04.1918447') -- returns 0 (for datetime2 function
-- and for datetimeoffset)

SELECT DAY(GETDATE()) -- returns the current day corresponding to the date
SELECT DAY('01/31/2012') -- returns 31

SELECT MONTH(GETDATE()) -- returns the current month corresponding to the date
SELECT MONTH('01/31/2012') -- returns 1

SELECT YEAR(GETDATE()) -- returns the current year corresponding to the date
SELECT YEAR('01/31/2012') -- returns 2012

SELECT DAY(GETDATE()) -- returns the current day corresponding to the date
SELECT DAY(01/31/2012) -- returns 31
```

```
SELECT DATENAME(DAY,'2012-08-31 21:02:04.167' ) -- returns the number of the day, 31
SELECT DATENAME(WEEKDAY,'2012-08-31 21:02:04.167' ) -- returns the name of the weekday
SELECT DATENAME(MONTH,'2012-08-31 21:02:04.167') -- returns august

-- Date parts you can use as first argument of DATENAME()
-- year, quarter, month, dayofyear, day, week, weekday, hour
-- minute, second, millisecond, microsecond, nanosecond, TZoffset

SELECT Name, DateofBirth, DATENAME(WEEKDAY, DateofBirth) AS [Day], -- we use brackets
-- because DAY is a reserved keyword, but here we want to print out the string
-- Day
      MONTH(DateofBirth) AS MonthNumber,
      DATENAME(MONTH, DateofBirth) AS MonthName,
      YEAR(DateofBirth) AS [Year]
FROM tblEmployee



-----------------------------------------------
-- 27 -- DATETIME FUNCTIONS (CONT.) II -- 27 --
-----------------------------------------------

-- DATEPART returns an integer, whereas DATENAME returns a VARCHAR
SELECT DATEPART(WEEKDAY, '2012-08-30 19:45:31.793') -- returns 5
SELECT DATENAME(WEEKDAY, '2012-08-30 19:45:31.793') -- returns Thursday

SELECT DATEADD(DAY, 20, '2012-08-30 19:45:31.793')
-- returns 2012-09-19 19:45:31.793
SELECT DATEADD(DAY, -20, '2012-08-30 19:45:31.793')
-- returns 2012-08-10 19:45:31.793

SELECT DATEDIFF(MONTH, '31/30/2005', '01/31/2006') -- retuns 2
SELECT DATEDIFF(DAY, '31/30/2005', '01/31/2006') -- retuns 62


DECLARE @DOB DATETIME, @tmpdate DATETIME, @years INT, @months INT, @days INT
SET @DOB = '10/08/1982'

SELECT @tempdate = @DOB

SELECT @years = DATEDIFF(YEAR, @tempdate, GETDATE()) -
                CASE
                        WHEN (MONTH(@DOB) > MONTH(GETDATE())) OR
                                (MONTH(@DOB) = MONTH(GETDATE()) AND DAY(@DOB) >
DAY(GETDATE()))
                        THEN 1
                        ELSE 0
                END
SELECT @tempdate = DATEADD(YEAR, @years, @tempdate)
```

```
-- we add years to tempdate

SELECT @months = DATEDIFF(MONTH, @tempdate, GETDATE()) -
		CASE
			WHEN DAY(@DOB) > DAY(GETDATE())
			THEN 1
			ELSE 0
		END
SELECT @tempdate = DATEADD(MONTH, @months, @tempdate)

SELECT @days = DATEDIFF(DAY, @tempdate, GETDATE())

SELECT @years AS Years, @months AS Months, @days AS days
-- it will calculate the years, months and days of the person born a given
-- date of birth


-- let's put it in a function

CREATE FUNCTION fnComputeAge(@DOB DATETIME)
RETURNS NVARCHAR(50)
AS
BEGIN
	DECLARE @DOB DATETIME, @tmpdate DATETIME, @years INT, @months INT, @days
INT

	SELECT @tempdate = @DOB

	SELECT @years = DATEDIFF(YEAR, @tempdate, GETDATE()) -
		CASE
			WHEN (MONTH(@DOB) > MONTH(GETDATE())) OR
				(MONTH(@DOB) = MONTH(GETDATE()) AND DAY(@DOB) >
DAY(GETDATE()))
			THEN 1
			ELSE 0
		END
	SELECT @tempdate = DATEADD(YEAR, @years, @tempdate)
	-- we add years to tempdate

	SELECT @months = DATEDIFF(MONTH, @tempdate, GETDATE()) -
		CASE
			WHEN DAY(@DOB) > DAY(GETDATE())
			THEN 1
			ELSE 0
		END
	SELECT @tempdate = DATEADD(MONTH, @months, @tempdate)

	SELECT @days = DATEDIFF(DAY, @tempdate, GETDATE())
```

```
        DECLARE @Age NVARCHAR(50)
        SET @Age = CAST(@years AS NVARCHAR(4)) + ' Years' +
                        CAST(@months AS NVARCHAR(2)) + ' Months' +
                        CAST(@days AS NVARCHAR(2)) + ' Days'
        RETURN @Age
END

-- now let's create a table

SELECT Id, Name, DateofBirth, dbo.fnComputeAge(DateofBirth) AS Age
FROM tblEmployee
-- here we passs DateOfBirth as parameter

-------------------------------------------
-- 28 -- CAST AND CONVERT FUNCTIONS -- 28 --
-------------------------------------------

-- to change data type you can use CAST or CONVERRT

SELECT Id, Name, DateofBirth, CAST(DateofBirth AS NVARCHAR) AS ConvertedDOB
FROM tblEmployee
-- specify what you want to convert AS the data type you want to convert it to

SELECT Id, Name, DateofBirth, CAST(DateofBirth AS NVARCHAR(5)) AS ConvertedDOB
FROM tblEmployee
-- same result but the output will be knocked of after 5 characters


SELECT Id, Name, DateofBirth, CONVERT(NVARCHAR, DateofBirth) AS ConvertedDOB
FROM tblEmployee
-- wrill return the same output

SELECT Id, Name, DateofBirth, CONVERT(NVARCHAR, DateofBirth, 103) AS ConvertedDOB
FROM tblEmployee

-- instead of 103, you wan enter the following
-- 101 mm/dd/yyyy
-- 102 yy.mm.dd
-- 103 dd/mm/yyyy
-- 104 dd.mm.yy
-- 105 dd-mm-yy
-- there are more in MSD documentation (some 30 or so styles)
-- these styles can only be used with convert and not with CAST


SELECT CAST(GETDATE(), AS DATE) -- will only return the date, not the time
SELECT CONVERT(DATE, GETDATE()) -- will do the same, but you can't use styles
-- styles (e.g. 103) can only be used when converting to var char
```

```
-- We want to concatenate values!
SELECT Id, Name, Name + ' - ' + CAST(Id AS NVARCHAR) AS [Name - Id]
FROM tblEmployee
-- we need too cast id to var char if we want to concatenate it


SELECT RegisteredDate, COUNT(id) AS TOTAL
FROM tblRegistrations
GROUP BY RegisteredDate
-- if RegisteredDate is set as DateTime, the grouping will be unlikely to
-- work since it is unlikely people registered by the same second
-- so we do cast to group by date registration date correctly

--
SELECT CAST(RegisteredDate AS DATE) AS RegistrationDate, COUNT(id) AS TOTAL
FROM tblRegistrations
GROUP BY RegisteredDate
-- now it will be grouped up to the day and not to the second


-- Differences Between CAST and CONVERT
        -- CAST is an ANSI standard, so it can be used on other database applications
        -- portability
        -- CONVERT is specific to SQL SERVER, and allows us to specify styles
        -- personalization

        -- use CAST whenever possible, unless you need to use the style functionality


---------------------------------------
-- 29 -- MATHEMATICAL FUNCTIONS -- 29 --
---------------------------------------

-- to find the functions in sql server go to databaes > sample >
-- programmability > functions

SELECT ABS(-101.5) -- returns 101.5, the absolute value

SELECT CEILING(15.2) -- returns 16
SELECT CEILING(15.8) -- returns 16
SELECT CEILING(-15.2) -- returns -15
-- returns the smaller integer values greater tha or equal to the parameter
SELECT FLOOR (15.2) -- returns 15
SELECT FLOOR (15.8) -- returns 15
SELECT FLOOR (-15.2) -- returns -16
-- returns the largest integer number less than or equal to the parameter

SELECT POW(2,3) -- raise 2 to the 3rd power (cube), 2x2x2
```

```
SELECT SQUARE(9) -- returns square of 9, 81

SELECT SQRT(81) -- reuurns the square root 9

SELECT RAND(1) -- Returns a random number, with seed value 1 (it could be anything)
-- you will always get the same random number
SELECT RAND() -- Will get a random number between 0 and 1
-- value changes every time the select is executed

-- if you want to get a random number between 1 and 100
SELECT FLOOR(RAND()*100)
-- FLOOR will knock off the decimal places, and *100 will push it between 0 and hundred

DECLARE @Counter INT
SET @Counter = 1
WHILE (@Counter <= 10)
BEGIN
        PRINT FLOOR(RAND()*1000)
        SET @Counter = @Counter +1
END
-- print 10 random numbers between 0 and 1000


SELECT ROUND(850.556, 2) -- round the number (first parameter) to 2 decimal
-- places 850.560

-- if you add 1 to third parameter, you will truncate the decimal, not round it
SELECT ROUND(850.556, 2, 1)
-- returns 850.550

SELECT ROUND(850.556, -2)
--  returns 900, because you are telling it to round the 50 of 850, so rounded to 900


----------------------------------------------------------------
-- 30 -- USER DEFINED FUNCTIONS: SCALAR VALUED FUNCTIONS -- 30 --
----------------------------------------------------------------

SELECT SQUARE(3) -- takes one parameter
SELECT GETDATE() -- doesn't take any parameter

-- USER DEFINED FUNCTIONS (UDF): SCALAR FUNCTIONS

-- A scalar function takes 0 or more parameters and returns a single scalar value

-- create a function
CREATE FUNCTION CalculateAge(@DOB DATE)
RETURNS INT
AS
```

```
BEGIN
DECLARE @Age INT
	-- function body
	SET @Age = DATEDIFF(YEAR, @DOB, GETDATE())
		CASE
			WHEN (MONTH(@DOB) > MONTH(GETDATE())) OR
				(MONTH(@DOB) = MONTH(GETDATE()) AND DAY(@DOB) >
DAY(GETDATE()))
			THEN 1
			ELSE 0
		END
RETURN @Age
END

-- invoke the function
SELECT dbo.CalculateAge('10/08/1982') AS Age --dbo stands for data base owner

-- write a query that will give you name, date of birth and age of the person
-- only for people older than 30
SELECT Name, DateOfBirth, dbo.CalculateAge(DateofBirth) AS Age
FROM tblEmployee
WHERE Age > 30


-- you can convert the function to a stored procedure

sp_helptext CalculateAge
-- this invokes the code of the function Calculate Age

-- next we modify the function to make it into a procedure
CREATE PROC spCalculateAge -- MODIFIED
@DOB DATE -- MODIFIED
AS
BEGIN
DECLARE @Age INT
	-- function body
	SET @Age = DATEDIFF(YEAR, @DOB, GETDATE())
		CASE
			WHEN (MONTH(@DOB) > MONTH(GETDATE())) OR
				(MONTH(@DOB) = MONTH(GETDATE()) AND DAY(@DOB) >
DAY(GETDATE()))
			THEN 1
			ELSE 0
		END
SELECT @Age -- MODIFIED
END

-- a function can be used in a SELECT and WHERE clause... but you cannot do that
-- to select a stored procedure
```

```
-- we use ALTER FUNCTION, AND DROP FUNCTION, to modify and delete a function

-----------------------------------------------------------------------
-- 31 -- USER DEFINED FUNCTIONS: IN LINE TABLE VALUED FUNCTIONS -- 31 --
-----------------------------------------------------------------------

-- USER DEFINED FUNCTIONS II: IN LINE TABLE VALUED FUNCTIONS
-- returns a table

CREATE FUNCTIONS fn_EmployeeByGender(@Gender NVARCHAR(10))
RETURNS TABLE
AS
RETURN (SELECT Id, Name, DateOfBirth, Gender, DepartmentId
                FROM tblEmployee
                WHERE Gender = @Gender)

-- RETURNS TABLE, and not any of the scalar data types
-- we do not use the begin and end
-- the structure of the table will be determined by the select statement

-- Call the function
-- treat is as a table
SELECT * FROM fn_EmployeeByGender('Male')

-- you can use the WHERE clause
SELECT * FROM fn_EmployeeByGender('Female')
WHERE Name = 'Pam'

-- In line table valued functions can be used to achieve the functionality
-- of parmetrized views.
-- You can use it in JOINS with other tables
SELECT Name, Gender, DepartmentName
FROM fn_EmployeeByGender('Male') e
JOIN tblDepartment d ON d.Id = e.DepartmentId

-----------------------------------------------------------------------------
-- 32 -- USER DEFINED FUNCTIONS: MULTI STATEMENT TABLE VALUED FUNCTIONS -- 32
--
-----------------------------------------------------------------------------

-- In Line Table Valued Function
CREATE FUNCTION fn_ILTVF_GetEmployees()
RETURNS TABLE
AS
RETURN (SELECT Id, Name, CAST(DateofBirth AS DATE)
FROM tblEmployee)
-- you do not specify the structure of the table to be returned with ILTVF
-- you no not add BEGIN and END block
-- better for performance than MSTVF, because internally an ILTVF is treated
```

```
-- similar to a VIEW, whereas MSTVF is treated as a Stored Procedure

CREATE FUNCTION fn_MSTVF_GetEmployees()
RETURNS @Table TABLE (Id INT, Name VARCHAR(20), DOB DATE)
AS
BEGIN
	INSERT INTO @Table
	SELECT Id, Name, CAST(DateofBirth AS DATE)
	FROM tblEmployee

	RETURN
END
-- you create a table variable with explicitly saying the structure of the table
-- you do use a BEGIN and END block
-- it can contain multiple statements, for example INSERT and SELECT


-- calling both ILTVF, MSTVF
SELECT * FROM dbo.fn_ILTVF_GetEmployees
SELECT * FROM dbo.fn_MSTVF_GetEmployees


-- updating an ILTVF
UPDATE fn_ILTVF_GetEmployees()
SET Name = 'Sam'
WHERE Id = 1
-- name which was not Sam, is now updated to Sam

-- it's not possible too update a MSTVF like we do the ILTVF
-- why? because there are intermediate processing steps it's
-- impossible for SQL Server, which table the information
-- we want to modify is coming from

---------------------------------------------
-- 33 -- FUNCTIONS: IMPORTANT CONCEPTS -- 33 --
---------------------------------------------

-- DETERMINISTIC FUNCTION
-- no matter how many times i execute a query, if the db has not changed
-- and the input parameters are the same, i will get the same result
SELECT SQUARE(9)
-- will yield the same result no matter how many times i execute it
-- so long as the db is the same and the input parameter is the same
-- all aggregate functions are deterministic functions

-- NON DETERMINISTIC FUNCTION
-- may return different values, for the same set of input values
-- even if the database has no changed
SELECT GETDATE()
```

```
SELECT RAND(3) -- is deterministic, because the input parameter
-- for the given seed, you will get the same output
SELECT RAND() -- is non deterministic


-- ENCRYPTION THE FUNCTION DEFINITION
-- is the same as encrypting the Stored Procedure
CREATE FUNCTION fn_GetNameById(@id INT)
RETURNS NVARCHAR(30)
AS
BEGIN
        RETURN (SELECT Name FROM tblEmployee WHERE Id = @Id)
END

-- call the function
SELECT dbo.fn_GetNameById(1)
-- returns "Sam", 1 record

sp_helptext fn_GetNameById
-- to get the definition of the function

-- can we encrypt so that the step above won't show the definition of
-- the function

ALTER FUNCTION fn_GetNameById(@Id INT)
RETURNS NVARCHAR(30)
WITH ENCRYPTION
AS
BEGIN
        RETURN (SELECT Name FROM tblEmployee WHERE Id = @Id)
END

-- now we can't view the definition anymore
-- we can alter again without with encryption to decrypt it

-- SCHEMABINDING A FUNCTION
-- suppose some does
-- DROP TABLE tblEmployee
-- now the table is delted

-- when I try to execute the fn_GetNameById() function
-- will get an error message
-- why? because the function refers to the table
-- but i don't want anybody to touch the table so that the
-- function is dangling

ALTER FUNCTION fn_GetNameById(@Id INT)
RETURNS NVARCHAR(30)
```

```
WITH SCHEMABINDING
AS
BEGIN
	RETURN (SELECT Name FROM dbo.tblEmployee WHERE Id = @Id) -- include dbo.
END

-- now we can't DROP TABLE tblEmployee
-- we can't drop Name column either
-- or do anything that will affect the integrity of the function
-- the function definition must be modified first before altering
-- any column or table that the function is referencing

---------------------------------
-- 34 -- TEMPORARY TABLES -- 34 --
---------------------------------

-- get created in the temp db and get deleted after they are used

-- Permanent Tables are created with CREATE TABLE
-- Temporary Tables are Created as follows -- note the hash

CREATE TABLE #PersonDetails(Id INT, Name NVARCHAR(20))

-- HOW TO CHECK THAT THE TEMPORARY TABLE IS CREATED:

-- System Databases > Tempdb > Temporary Tables
-- OR
SELECT Name
FROM tempdb..systobjects
WHERE Name LIKE "#PersonDetails%" -- do not use =, use LIKE

-- the temporary table is available only for the duration of the connection
-- that created the table
-- when the connection is closed, the temporary table is dropped

-- you can drop the table yourself
DROP TABLE #PersonDetails

-- STORED PROCEDURE CREATES TEMPORARY TABLE
-- if the temporary table is created within the stored procedure
-- the table is dropped automatically

CREATE PROCEDURE spCreateLocalTempTable
AS
BEGIN
	CREATE TABLE #PersonDetails (Id INT, Name NVARCHAR(20))

	INSERT INTO #PersonDetails VALUES(1, 'Mike')
	INSERT INTO #PersonDetails VALUES(2, 'Dave')
```

```
        INSERT INTO #PersonDetails VALUES(3, 'Ben')

        SELECT * FROM #PersonDetails
END

EXECUTE spCreateLocalTempTable
-- i get the data, but the table is dropped afterwards

-- if temporary tables with the same name are created by different
-- users/connections, there is no problem because sql server
-- distinguishes the two by appending a number to the end of the tables


-- GLOBAL TEMPORARY TABLE
-- uses two hash prefixes, and are visible for all connections
-- they are destroyed when the last connection that is referencing it is closed
-- global temporary tables require that their name be unique
-- because they are shared by multiple users/connections/sessions
CREATE TABLE ##EmployeeDetails(Id INT, Name NVARCHAR(20))

SELECT * FROM ##EmployeeDetails
-- you can do this from a different user/connection

-------------------------
-- 35 -- INDEXES -- 35 --
-------------------------

-- WHAT ARE INDEXES
-- Are used by queries to find data quickly
-- Helps reduce the time it takes to find data quickly
-- The existence of the right indexes drastically improves performance
-- When there is no index to help a query, the database engine has to
-- scan every row from begining to end. If the table is large, this can
-- become a serious performance issue
-- this is called a "TABLE SCAN"

CREATE INDEX IX_tblEmployee_Salary
ON tblEmployee (Salary ASC)
-- we are creating an index on the salary column of the tblEmployee table
-- in the index, the Salary is the key... ASC is telling sql server
-- to arrange salaries in ascending order

-- to find the indexes
sp_Helpindex tblEmployee

-- DROP INDEX
DROP INDEX tblEmployee.IX_tblEmployee_Salary
```

```
---------------------------------------------
-- 36 -- INDEXES: CLUSTERED AND NON CLUSTRED  --
---------------------------------------------

-- CLUSTERED, NON CLUSTERED, UNIQUE, FILTERED
-- XML, FULL TEXT, SPATIAL, COLUMNSTORE, INDEX WITH INCLUDED COLUMNS,
-- INDEX ON COMPUTED COLUMNS

-- A table can have only ONE clustered index.
-- Deterimines the physical order of data on a table

-- a primary key constraint will automatically create a clustered index on a table

CREATE TABLE [tblEmployee]
(
	[id] INT PRIMARY key,
	[Name] NVARCHAR(50),
	[Salary] INT,
	[Gender] NVARCHAR(10),
	[City] NVARCHAR(50)
)

-- check if there is a clustered index
EXECUTE sp_helpindex tblEmployee
-- THE clustered index was created automatically during table creation because
-- we used primary key

INSERT INTO tblEmployee VALUES (3,'John',4500,'Male','New York') -- we
INSERT INTO tblEmployee VALUES (1,'Sam',2500,'Male','London') -- we
INSERT INTO tblEmployee VALUES (4,'Sarah',5500,'Female','Tokyo') -- we
INSERT INTO tblEmployee VALUES (5,'Tom',3100,'Male','Toronto') -- we
INSERT INTO tblEmployee VALUES (2,'Pam',6500,'Female','Sydney') -- we


-- insert data in disorder (the ids). But even if we insert in the wrong order
-- when i do SELECT * the data should be ordered

SELECT * FROM tblEmployee

-- in a table the data is arranged by the clustered index key.
-- the clustered index need be one, but it can contain multiple COLUMNs
-- this is known as a composite clustered index.

-- a non clustered index made up of multiple columns is a composite non clustered index

-- let´s create a composite clusterd index by gender and Salary
-- sort the data first by gender, then by Salary

-- FIRST!!!!
```

```
-- Go to the object explorer and delete the primary key clustered index

CREATE CLUSTERED INDEX IX_tblEmployee_Gender_Salary
ON tblEmployee(Gender DESC, Salary ASC)
-- now the data is first arranged by gender Male, Female, and then within
-- the gender in ascending order of the salaries


-- a non clustered index is stored separately from the index... it is not
-- made up of the record data.
-- if you create a nonclusterd index on the Name column, orders the table
-- by name alphabetically.
-- you can have more than one non clustered index, as many as you want.

CREATE NONCLUSTERED INDEX IX_tblEmployee_Name
ON tblEmployee(Name)

-- how the data is stored is determined by the clustered index.

-- DIFFERENCES BETWEEN CLUSTERED AND NON CLUSTERED INDEXES

--1) one clustered index only allowed per table, many non clustered possible
--2) clustered index is faster because the clustered index has to refer back to the table
-- in nonclustered index, because it is separate from the table, when searching, sql server
-- has to look at the table with non clustered index table first and then look at the actual
-- table, this is because clustered index is embedded in the table
--3) Non clustered indexes are stored separately, with the index being dublicated and
-- stored separately, requiring additional disk space



---------------------------------------------
-- 37 -- INDEXES: UNIQUE AND NON-UNIQUE -- 37 --
---------------------------------------------

-- A unique index is used to enforce uniquenes of key values in the index

CREATE TABLE [tblEmployee]
(
        [id] INT PRIMARY key,
        [Name] NVARCHAR(50),
        [Salary] INT,
        [Gender] NVARCHAR(10),
        [City] NVARCHAR(50)
)

-- a unique clustered index is created when we create the primary key
-- check:
EXECUTE sp_helpindex tblEmployee
```

```
-- a UNIQUE CLUSTERED index was created
-- the primary key constraint uses unique to enforce the primary key constraint

-- We delete the index from the object explorer
-- so we can insert duplicate records now
-- so a primary key constraint uses uniquness!

-- the UNIQUE is a property of an index, so a CLUSTERED AND NON CLUSTERED, can or
-- cannot be UNIQUE

CREATE UNIQUE NONCLUSTERED INDEX UIX_tblEmployee_FirstName_LastName
ON tblEmployee(FirstName, LastName)
-- so we make sure no one can have the same last and first name


-- No major differences between UNIQUE index and UNIQUE constraint
-- Example: I add a UNIQUE constraint
ALTER TABLE tblEmployee
ADD CONSTRAINT UQ_tblEmployee_City
UNIQUE (City)
-- we are adding a unique constraint
-- an index is automatically created, a non clustered unique index is created
-- this is by default

-- if we had wanted to create a unique clustered index created we would do:
ALTER TABLE tblEmployee
ADD CONSTRAINT UQ_tblEmployee_City
UNIQUE CLUSTERED (City)

-- WE CAN CREATE UNIQUE INDEX, BY ADDING A UNIQUE CONSTRAINT (INDIRECTLY),
-- OR CREATING A UNIQUE INDEX DIRECTLY

-- Create a unique constraint when data integrity is the objective.
-- In either case, data is validadted in the same manner, and the
-- query optimizer does not differentiate between a unique index created
-- by a unique constraint or manually created

-- A UNIQUE constraint or a UNIQUE index CANNOT BE CREATED ON AN EXISTING
-- TABLE, IF THE TABLE CONTAINS DUPLICATE VALUES IN THE KEY COLUMNS.
-- OBVIOUSLY, TO SOLVE THIS, REMOVE THE KEY COLUMNS FROMT HE INDEX
DEFINITION
--- OR DELETE OR UPDATE THE DUPLICATE VALUES.



-----------------------------------------------------
-- 38 -- INDEXES: ADVANTAGES AND DISADVANTAGES -- 38 --
-----------------------------------------------------
```

```
-- ADVANTAGES:
-- INDEXES ARE USED BY QUERIES TO FIND DATA QUICKLY

-- If you have a nonclustered index ordered by salary, and you do
-- a query to search WHERE salary falls within a range, then this is
-- a scenario in which the non clustered index is best
SELECT * FROM tblEmployee
WHERE Salary > 4000 AND Salary < 8000

DELETE FROM tblEmployee
WHERE Salary = 2500

UPDATE tblEmployee SET Salary = 9000
WHERE Salary =7500

SELECT * FROM tblEmployee
ORDER BY Salary

SELECT * FROM tblEmployee
ORDER BY Salary DESC

SELECT Salary, COUNT(Salary) AS Total
FROM tblEmployee
GROUP BY Salary


-- DISADVANTAGES:
-- NON CLUSTERED INDEXES REQUIRE ADDITIONAL DISK space
-- but these days disk storage is not costly
-- the amount of space required will depend on the size of the table

-- INSERT, UPDATE AND DELETE queries can become slow
-- if the tables are huge, when you update or delete,
-- IF THERE ARE MANY NON CLUSTERED INDEXES, all of those require
-- additional work (because the non clustered indexes are stored separately)


-- COVERING QUERY:
-- If all the columns that you have requested in the SELECT clause of the query
-- are present in the index, then there is no need to lookup in the table
-- again. The requested columns data can simply be retuned from the index
-- FOR THIS REASON, A CLUSTERED INDEX ALWAYS COVERS A QUERY, because it is
-- embedded in the table.


----------------------
-- 39 -- VIEWS -- 39 --
----------------------
```

```
-- A view is a saved sql query. Also considered a virtual table.
CREATE VIEW vWEmployeesByDepartment
AS
SELECT Id, Name, Salary, Gender, DeptName
FROM tblEmployee
JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.DeptId
-- this view is returning employees by departmnt

-- you can treat the view just like a table
SELECT * FROM vWEmployeesByDepartment
-- the database engine knows that the view is getting the data
-- from tblEmployee and tblDepartement
-- so SQL SERVER executes the SELECT statement.
-- it is called a virtual table, because the view iteself doesnt
-- actually stored any data, it is just a saved SELECT query

-- to see the view definition
sp_helptext vWEmployeesByDepartment


-- ADVANTAGES OF VIEWS:

--1) TO REDUCE THE COMPLEXITY OF THE DATABASE SCHEMA FOR NON IT USERS
-- Non IT users dont know how to do JOINS for example
-- writing a join query is not simple.
-- What you can do is write a view, and give access to that user to the view
-- and that user treats the join as a single table

--2) TO IMPLEMENT ROW AND COLUMN LEVEL SECURITY
-- You can grant access only to a a view, so that a user can´t actually
-- access the underlying base tables. -- this is row level security

-- You don´t want to allow access to Salary column for example
CREATE VIEW vWNonConfidentialData
AS
SELECT Id, Name, Gender, DeptName -- except the salary, all the columns
FROM tblEmployee
JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.DeptId

-- 3) TO PROVIDE USERS ONLY AGGREGATE DATA
CREATE VIEW vWSummarizedData
AS
SELECT DeptName, COUNT(Id) as TotalEmployees
FROM tblEmployee
JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.DeptId
```

```
GROUP BY DeptName

-- To modify a view we use ALTER VIEW [View Name]
-- To drop a view we use DROP VIEW [View Name]



----------------------------------------
-- 40 -- VIEWS: UPDATABLE VIEWS -- 40 --
----------------------------------------

CREATE VIEW vWEmployeesDataExceptSalary
AS
SELECT Id, Name, Gender, DepartmentId
FROM tblEmployee
-- selects all columns except salary

SELECT * FROM vWEmployeesDataExceptSalary
-- the view gets its data from tblEmployee, which is the underlying base table
-- the view doesn´t store any data

-- It is possible to INSERT, UPDATE AND DELETE from the base table thru a view
-- Examples:

--1)
UPDATE vWEmployeesDataExceptSalary
SET Name = 'Mikey'
WHERE Id = 1
-- now we changed the underlying base table:

SELECT * FROM tblEmployee
-- the name Mike was changed to Mikey

--2)
DELETE FROM vWEmployeesDataExceptSalary
WHERE Id = 2

SELECT * FROM tblEmployee
-- the record was deleted

--3)
INSERT INTO vWEmployeesDataExceptSalary
VALUES (2, 'Mikey', Male, 2)

SELECT * FROM tblEmployee
-- the record was added

-- Views could be based on multiple tables
CREATE VIEW vWEmployeeDetailsByDepartment
```

```
AS
SELECT Id, Name, Salary, Gender, DeptName
FROM tblEmployee
JOIN tblDepartement
ON tblEmployee.DepartmentId = tblDepartement.DeptId

-- suppose we update records on multiple tables
-- it is updating the underlying tables
-- when we do:
UPDATE vWEmployeeDetailsByDepartment
SET DeptName = 'IT'
WHERE Name = 'John'

-- the view will update all records who have HR to IT
-- not only Johns. It updates the deparment name for all records

-- IF A VIEW IS BASED ON MULTIPLE TABLES, IT MAY NOT UPDATE
-- THE UNDERLYING TABLES LIKE WE INTENDED THEM TO

-- TO DO SO WE NEED TO USE
-- INSTEAD OF
-- TRIGGERS


--------------------------------------
-- 41 -- VIEWS: INDEXED VIEWS -- 41 --
--------------------------------------

-- A standard non indexed view, is just a stored SQL query, a virtual table.
-- Once we create an index in a view, the view gets materialized and is now
-- capable of storing data.
CREATE VIEW vWTotalSalesByProduct
WITH SCHEMABINDING -- this means that you cannot change the underlying
-- objects in any way that can affect the VIEW (or STORED PROCEDURE)
AS
SELECT Name, SUM(ISNULL((QuantitySold * UnitPrice), 0)) AS TotalSales,
COUNT_BIG(*) AS TotalTransactions
-- RULE IN THE USE OF AGGREGATE FUNCTIONS
-- we use an aggregate function SUM, and if there is a possibilty for
-- that expression to result is null, we need to make sure to convert
-- those rows to 0, this is why we use ISNULL, when we do SUM
FROM dbo.tblProductSales
JOIN dbo.tblProduct
-- use dbo. schemaname.table name... this is also a RULE
ON dbo.tblProduct.ProductId = dbo.tblProductSales.ProductId
GROUP BY Name
-- RULE:
-- If GROUP BY is specified, the view select list must containt COUNT_BIG(*)
-- expression
```

```
SELECT * FROM vWTotalSalesByProduct
-- right now, every time we do this command, the select statemnt is invoked in its entirety
-- we can create an index so that sql server doesn't have to go to the base tables every time


CREATE UNIQUE CLUSTERED INDEX UIX_vWTotalSalesByProduct_Name
ON vWTotalSalesByProduct (Name)
-- we create a clustred index, because we cannot create a non clustered
-- index on the view because there is no table on the view, the view
-- is just a select statement

-- now when we do a select statement
-- the SELECT becomes more performant
SELECT * FROM vWTotalSalesByProduct

-- this is ideal for data that are not changed frequently
-- if the data is changed frequently it will not be performant
-- because the indexed views are continuously readjusted with every change
-- the cost of maintining indexed views is more than tables




---------------------------------
-- 42 -- VIEW LIMITATIONS -- 42 --
---------------------------------

-- YOU CANNOT PASS PARAMETERS TO A VIEW
-- But you CAN use the WHERE clause to make for instance Gender = Male
-- Or, you can use TABLE VALUED FUNCTIONS that can act as a replacement
-- for parametrized views:

CREATE FUNCTION fnEmployeeDetails(@Gender NVARCHAR(20))
RETURNS TABLE
AS
RETURN (SELECT Id, Name, Gender, DepartmentId)
FROM tblEmployee
WHERE Gender = @Gender

SELECT * FROM dbo.fnEmployeeDetails

-- RULES AND DEFAULTS CANNOT BE ASSOCIATED WITH VIEWS
-- because except for indexed views they don't stored any data

-- YOU CANNOT USE AN ORDER BY CLAUSE IN A VIEW
-- Unless you use the TOP or FOR XML commands

-- YOU CANNOT CREATE VIEWS (or FUNCTIONS) ON TEMPORARY TABLES (##tblName)
```

```
-----------------------------
-- 43 -- DML TRIGGERS -- 43 --
-----------------------------

-- DML, DDL, LOGON TRIGGERS

-- DML: DATA MANIPULATION LANGUAGE
-- INSERT, UPDATE, DELETE ARE DML STATEMENTS
-- fire automatically in response to DML STATEMENTS

-- AFTER TRIGGERS OR FOR TRIGGERS
-- they fire after INSERT, UPDATE, or DELETE
-- INSTEAD OF TRIGGERS
-- they fire instead of INSERT, UPDATE, or DELETE

-- AFTER DML TRIGGERS
-- INSERT
CREATE TRIGGER tr_tblEmployee_ForInsert
ON tblEmployee
FOR INSERT
AS
BEGIN
        DECLARE @Id INT
        SELECT @Id = Id
        FROM inserted -- inserted is a table availabe only in the
        -- context of the trigger... it is a magic table... which
        -- retains a copy of the row inserted will be maintained
        -- in this inserted table. can only be accessed here in this way
        -- you cannot access inserted from outside of this context (of a trigger)

        INSERT INTO tblEmployeeAudit
        VALUES ("New employee with Id = " + CAST(@Id AS VARCHAR(5)) +
                " is added at " + CAST(GETDATE() AS NVARCHAR(20))
                )
END

-- now we insert a row into tblEmployee table
INSERT INTO tblEmployee VALUES (8, 'Jimmy', 1800, 'Male', 3)
-- immediately upon execution, a row gets inserted into tblEmployee
-- and the trigger makes the audit row be inserted into tblEmployeeAudit table

-- AFTER DML TRIGGER
-- DELETE
CREATE TRIGGER tr_tblEmployee_ForDelete
ON tblEmployee
FOR DELETE
```

```
AS
BEGIN
	DECLARE @Id INT
	SELECT @Id = Id
	FROM deleted -- deleted is a table availabe only in the
	-- context of the trigger... it is a magic table... which
	-- retains a copy of the row deleted will be maintained
	-- in this deleted table. can only be accessed here in this way
	-- you cannot access deleted from outside of this context (of a trigger)

	INSERT INTO tblEmployeeAudit
	VALUES ("An existing employee with Id = " + CAST(@Id AS VARCHAR(5)) +
		" is deleted at " + CAST(GETDATE() AS NVARCHAR(20))
		)
END

DELETE FROM tblEmployee
WHERE Id = 1
-- a row is deleted from tblEmployee
-- the delete trigger executes and a record value is inserted into tblEmployeeAudit


-------------------------------------------
-- 44 -- AFTER DML TRIGGERS: UPDATE -- 44 --
-------------------------------------------

-- fires after the triggering action
CREATE TRIGGER  tr_tblEmployeee_ForUpdate
ON tblEmployee
FOR UPDATE
AS
BEGIN
	SELECT * FROM deleted
	SELECT * FROM inserted
END

UPDATE tblEmployee SET Name = 'James', Salary = 2000, Gender = 'Male'
WHERE Id = 8
-- deleted table will show the data had before the update action
-- insert table will show the ddata after the update action

ALTER TRIGGER  tr_tblEmployeee_ForUpdate
ON tblEmployee
FOR UPDATE
AS
BEGIN
	DECLARE @Id INT
	DECLARE @OldName VARCHAR(20), @NewName NVARCHAR(20)
	DECLARE @OldSalary INT, @NewSalary INT
```

```
	DECLARE @OldGender VARCHAR(20), @NewGender NVARCHAR(20)
	DECLARE @OldDeptId INT, @NewDeptId INT
	DECLARE @AuditString NVARCHAR(1000)

	SELECT *
	INTO #TempTable -- this table contains the new data
	FROM inserted
	-- select all the rows from the inserted table, and put it into the temp table

	WHILE(EXISTS(SELECT Id FROM @TempTable)) -- there may be many rows in the temp
table
	-- so we loop
	BEGIN
		SET @AuditString = ''

		-- select the first row from the temp table (inserted) with the new data
		-- and compare it to the data in the old table (deleted)
		SELECT TOP 1 @Id = Id, @NewName = Name, @OldSalary = Salary,
		 @OldDeptId = DepartmentId, @NewGender = Gender
		FROM #TempTable

		SELECT @OldName = Name, @OldSalary = Salary,
		 @OldDeptId = DepartmentId, @OldGender = Gender
		FROM deleted -- this table contains the old data
		WHERE Id = @Id

		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 ' + @OldSalary + ' to ' +
@NewSalary
		IF (@OldDeptId <> @NewDeptId)
			SET @AuditString = @AuditString + ' Department from ' + @OldDeptId + ' to '
+ @NewDeptId

		INSERT INTO tblEmployeeAudit VALUES (@AuditString)

		DELETE FROM #TempTable
		WHERE Id = @Id -- this will prevent the loop from being infinite, it is the change
condition
	END
END
```

```
END

UPDATE tblEmployee SET Name = 'Todd', Salary = 2000, Gender = 'Female'
WHERE Id IN (1,2,4,8) -- many records require the while loop in the trigger above

SELECT * FROM tblEmployee
SELECT * FROM tblEmployeeAudit



-------------------------------------------------
-- 45 -- INSTEAD OF DML TRIGGERS: INSERT -- 45 --
-------------------------------------------------

-- i want to create a view based on two tables (will use join)
-- when i try to insert a row into this view (a view doesnt contain data)
-- i want data to be inserted into the base tables
-- sql server doesnt now into which base table should this row be inserted
-- so sql server will throw an error

-- instead of triggers are usually used to update views correctly that are based
-- on multiple underlying tables

-- first we create this view
CREATE VIEW vwEmployeeDetails
AS
SELECT Id, Name, Gender, DeptName
FROM tblEmployee
JOIN tblDepartment
ON tblDepartment.DeptId = tblEmployee.DepartmentId
-- we created the view

-- select data from that view
SELECT * FROM vwEmployeeDetails

--let´s try to insert a row
INSERT INTO vwEmployeeDetails VALUES (7, 'Valerie', 'Female', 'IT')
-- we get an error because there are multiple underlying tables for this view

--we create an INSTEAD OF trigger
CREATE TRIGGER tr_vWEmployeeDetails_InseadOfInsert
ON vwEmployeeDetails
INSTEAD OF insert
AS
BEGIN
        SELECT * FROM inserted
        SELECT * FROM deleted
END
```

```
--
--let´s try to insert a row
INSERT INTO vwEmployeeDetails VALUES (8, 'Valerie', 'Female', 'IT')
-- we'll execute the select statements from the trigger instead, no error message

--we create an INSTEAD OF trigger
ALTER TRIGGER tr_vWEmployeeDetails_InseadOfInsert
ON vwEmployeeDetails
INSTEAD OF insert
AS
BEGIN
	DECLARE @DeptId INT

	-- check if there is a valid DepartemntId
	-- for the given DepartmentName
	-- saving the department id from tblDepartment and save it into the variable
	SELECT @DeptId = DeptId
	FROM tblDepartment
	JOIN inserted
	ON inserted.DeptName = tblDepartment.DeptName

	-- if DepartmentId is null throw an error
	-- and stop processing
	-- this will happen when the insert statement has an invalid department name
	IF (@DeptId IS NULL)
	BEGIN
		RAISEERROR('Invalid Department Name, Statement Terminated', 16, 1)
		-- 16 is the severity level, 16 means the user can correct the error,
		-- 1 is the state
		RETURN
	END

	-- finally insert into tblEmployee table
	INSERT INTO tblEmployee(Id, Name, Gender, DepartmentId)
	SELECT Id, Name, Gender, @DeptId
	FROM inserted
END

--let´s try to insert a row
INSERT INTO vwEmployeeDetails VALUES (9, 'Valerie', 'Female', 'invaliddepartment')
-- the trigger will not find a valid department, and DeptId will be null
-- and the error will be raised

--let´s try to insert a row
INSERT INTO vwEmployeeDetails VALUES (9, 'Valerie', 'Female', 'IT')
-- the trigger will execute and the VALUES will be inserted into tblEmployee

-- we verify
```

```
SELECT * FROM tblEmployee
SELECT * FROM vWEmployeeDetails

-- to get the department id we are joining inserted table


-----------------------------------------------
-- 46 -- INSTEAD OF DML TRIGGERS: UPDATE -- 46 --
-----------------------------------------------

-- a view based on two tables, tblEmployee and tblDepartment

SELECT * tblEmployee
SELECT * tblDepartment

CREATE VIEW vWEmployeeDetails
AS
SELECT Id, Name, Gender, DeptName
FROM tblEmployee
JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.DeptId

SELECT * FROM vWEmployeeDetails

-- let´s try to update the view, so that underlying tables are affected
UPDATE vWEmployeeDetails SET Name = 'Johnny', DeptName = 'IT'
WHERE Id = 1
-- will result in an error message, you cannot update a view
-- when it depends on multiple underlying tables

-- if we were just updating the department name DeptName only
-- it will modify it without problems, but it will modify
-- references to both tables for all records, not just where id = 1
-- we want to avoid this
-- SO WHENEVER YOU ARE UPDATING A VIEW WITH UNDERLYING TABLES
-- YOU SHOULD ALWAYS USE INSTEAD OF UPDATE TRIGGERS
-- REGARDLESS OF WHETHER THE UPDATE STATEMENT AFFECTS ONE OR MULTIPLE
-- UNDERLYING TABLES

-- we can create an INSTEAD OF UPDATE TRIGGER
CREATE TRIGGER tr_vWEmployeeDetails_InsteadOfUpdate
ON vwEmployeeDetails
INSTEAD OF UPDATE
AS
BEGIN
        -- If EmployeeId is updated
        IF (UPDATE(Id)) -- Returns false if an update statement is executed
        -- and the updated field is not Id column, if it is then returns true
        BEGIN
```

```
            RAISEERROR('Id cannot be changed', 16, 1)
            RETURN
        END

        -- If DeptName is updated
        IF (UPDATE(DeptName))
        BEGIN
            DECLARE @DeptId INT

            -- get the department id column from tblDepartment
            SELECT @DeptId = DeptId
            FROM tblDepartment
            JOIN inserted -- this contains the new data, the updated data
            ON inserted.DeptName = tblDepartment.DeptName

            IF(@DeptId IS NULL) -- it will be null only when there is an invalid
            -- department names insertion attempts
            BEGIN
                RAISEERROR('Invalid Department Name', 16, 1)
                RETURN
            END

            UPDATE tblEmployee SET DepartmentId = @DeptId
            FROM inserted
            JOIN tblEmployee
            ON tblEmployee.Id = inserted.id

        -- if Gender is updated
        IF(UPDATE(Gender))
        BEGIN
            UPDATE tblEmployee SET Gender = inserted.Gender
            FROM inserted
            JOIN tblEmployee
            ON tblEmployee.Id = inserted.id
        END

        -- if Name is updated
        IF(UPDATE(Name))
        BEGIN
            UPDATE tblEmployee SET Name = inserted.Name
            FROM inserted
            JOIN tblEmployee
            on tblEmployee.Id = inserted.id
        END
END

-- now let´s try to update
UPDATE vWEmployeeDetails SET DeptName = 'IT'
WHERE Id = 1
```

```
-- now we are only updating the record for John (id = 1)

-- now let´s try to update with multiple columns
UPDATE vWEmployeeDetails SET DeptName = 'Payroll', Name = 'Mark', Gender = 'Female'
WHERE Id = 1
-- now it succeeds

-- let's verify
SELECT * FROM vWEmployeeDetails
SELECT * FROM tblEmployee
SELECT * FROM tblDeptment


-------------------------------------------------
-- 47 -- INSTEAD OF DML TRIGGERS: DELETE -- 47 --
-------------------------------------------------

-- used to delete a row from a view that is based on multiple tables
-- we have a view based on tblEmployee, and tblDepartment
-- we are trying to delete Johns record from a view
-- sql server doesn't know which underlying tables should be affected
-- so it will throw an error message


SELECT * tblEmployee
SELECT * tblDepartment

CREATE VIEW vWEmployeeDetails
AS
SELECT Id, Name, Gender, DeptName
FROM tblEmployee
JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.DeptId

SELECT * FROM vWEmployeeDetails

DELETE FROM vwEmployeeDetails
WHERE Id = 1
-- will raise an error
-- because modification delete affects multiple tables

CREATE TRIGGER tr_vWEmployeeDetails_InsteaOfDelete
ON vWEmployeeDetails
INSTEAD OF DELETE
AS
BEGIN
        DELETE tblEmployee
        FROM tblEmployee
        JOIN deleted -- deleted will contain the deleted records
```

```
        ON tblEmployee.Id = deleted.id

        -- Alternatively you could write a subquery instead of the previous 4 lines:
        -- DELETE FROM tblEmployee
        -- WHERE Id IN (SELECT Id FROM deleted)
END

SELECT * FROM vWEmployeeDetails

-- now let's try to delete from the view
DELETE FROM vwEmployeeDetails
WHERE Id IN (1,2)
-- two rows will have been deleted

-- let´s verify the records are gone from the view and the underlying tables
SELECT * FROM vWEmployeeDetails
SELECT * tblEmployee
SELECT * tblDepartment
```