T.C. İSTANBUL KÜLTÜR ÜNİVERSİTESİ

CSE5041 - DATABASE DESIGN AND DEVELOPMENT

LAB 07 – Introduction to Views and, the code scripts in SQL.

After completing this Lab, you will be able to

- Understand views
- Create, update, alter, and drop views
- Insert or delete rows through a view in SQL
- Use the View Designer in SQL
- Work with code scripts in SQL
- Work with scaler variables and table variables
- Work with temporary tables and derived tables
- Control the execution of a script
- Learn to IF… ELSE statement and WHILE loop
- Learn to TRY… CATCH statement to handle errors in SQL

PROCEDURE 1 – How to create a view in SQL.

A VIEW in SQL Server is like a virtual table that contains data from one or multiple tables. It does not hold any data and does not exist physically in the database. Similar to a SQL table, the view name should be unique in a database. It contains a set of predefined SQL queries to fetch data from the database. It can contain database tables from single or multiple databases as well. You can add SQL statements and functions to a view and present the data as if the data were coming from one single table. A view is created with the **CREATE VIEW** statement. The following statement defines the syntax of a view.

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Note: A view always shows up-to-date data! The database engine recreates the view, every time a user query it.

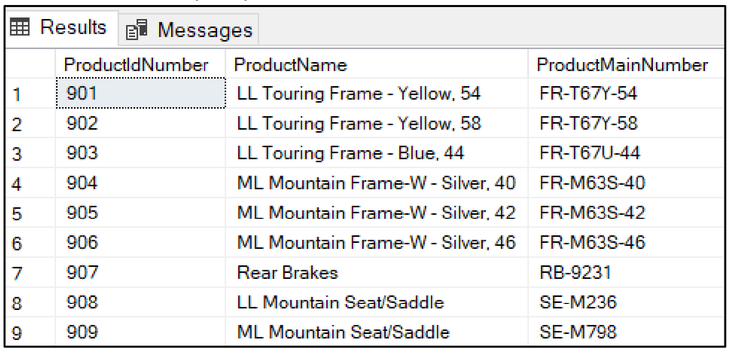
Step 1 - At first, we need to specify the **CREATE VIEW** statement and then we must give a name to the view. In the second step, we define the **SELECT** statement after the **AS** keyword. The following example will create a view that will be named as **VProductSpecialList. VProductSpecialList** view fetches data from the **Product** table and it only returns the **ProductID**, **Name** and **ProductNumber** columns for products with **ProductID** greater than 900 from the **Product** table.

```
USE AdventureWorks2019;
GO
CREATE VIEW VProductSpecialList
AS
SELECT p.ProductID AS [ProductIdNumber],
p.Name AS [ProductName],
p.ProductNumber AS [ProductMainNumber]
FROM [Production].[Product] as p
where ProductID>900
```

Step 2- After the creation of the view, we can retrieve data using a simple SELECT statement. The following example shows how to fetch data from the VProductSpecialList view:

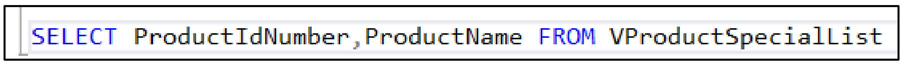
```
SELECT * FROM VProductSpecialList
```

** Execute the query and review the results.



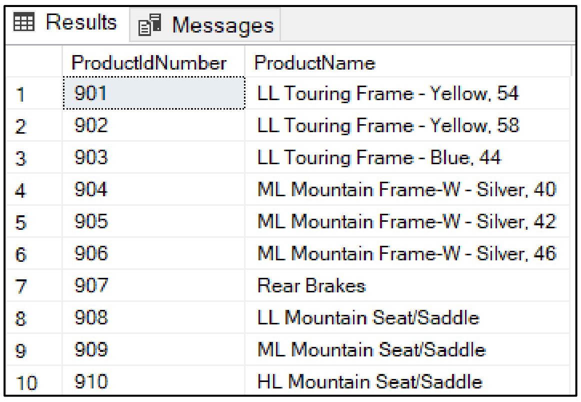
	ProductIdNumber	ProductName	ProductMainNumber
1	901	LL Touring Frame - Yellow, 54	FR-T67Y-54
2	902	LL Touring Frame - Yellow, 58	FR-T67Y-58
3	903	LL Touring Frame - Blue, 44	FR-T67U-44
4	904	ML Mountain Frame-W - Silver, 40	FR-M63S-40
5	905	ML Mountain Frame-W - Silver, 42	FR-M63S-42
6	906	ML Mountain Frame-W - Silver, 46	FR-M63S-46
7	907	Rear Brakes	RB-9231
8	908	LL Mountain Seat/Saddle	SE-M236
9	909	ML Mountain Seat/Saddle	SE-M798

Step 3 - For the different circumstances, we may need some particular columns of the view for this we can only use these column names in the SELECT statement:



```
SELECT ProductIdNumber,ProductName FROM VProductSpecialList
```

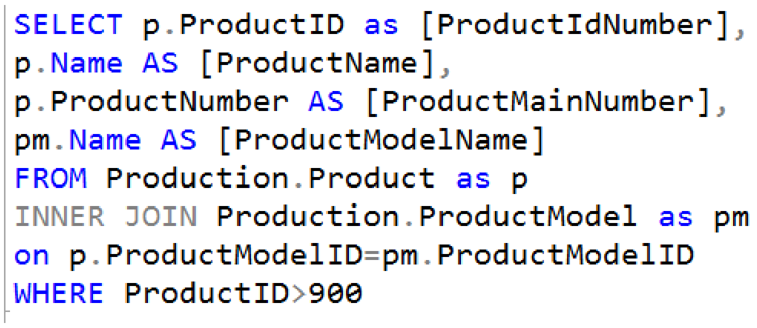
** Execute the query and review the results.



Results | Messages

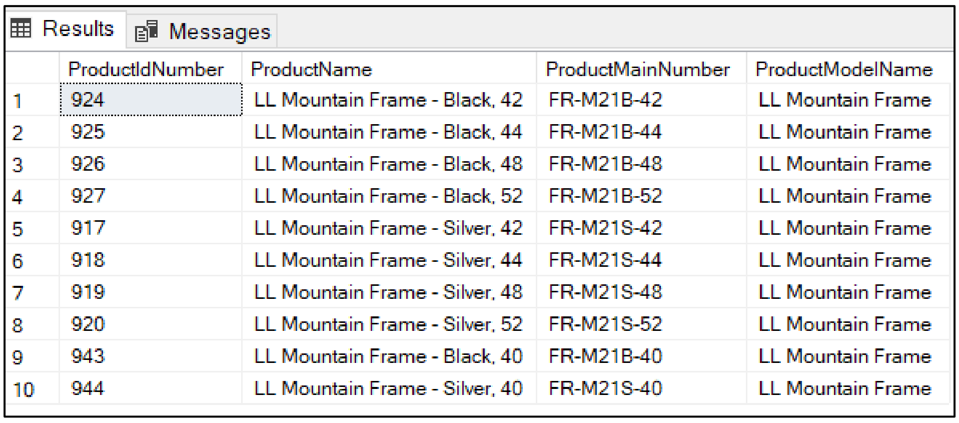
	ProductIdNumber	ProductName
1	901	LL Touring Frame - Yellow, 54
2	902	LL Touring Frame - Yellow, 58
3	903	LL Touring Frame - Blue, 44
4	904	ML Mountain Frame-W - Silver, 40
5	905	ML Mountain Frame-W - Silver, 42
6	906	ML Mountain Frame-W - Silver, 46
7	907	Rear Brakes
8	908	LL Mountain Seat/Saddle
9	909	ML Mountain Seat/Saddle
10	910	HL Mountain Seat/Saddle

Step 4 - In the previous example, we created a view for a single table, but we can also create a view for joined multiple tables. The following example will create a view that will be named as **VProductDetailList**. **VProductDetailList** view fetches data from the **Product** and **ProductModel** tables, and it only returns the **ProductID**, **Name** and **ProductNumber** columns for products with **ProductID** greater than 900 from the **Product** table, and **name** column of **ProductModel** table.



```
SELECT p.ProductID as [ProductIdNumber],
p.Name AS [ProductName],
p.ProductNumber AS [ProductMainNumber],
pm.Name AS [ProductModelName]
FROM Production.Product as p
INNER JOIN Production.ProductModel as pm
on p.ProductModelID=pm.ProductModelID
WHERE ProductID>900
```

** Execute the query and review the results.



Results | Messages

	ProductIdNumber	ProductName	ProductMainNumber	ProductModelName
1	924	LL Mountain Frame - Black, 42	FR-M21B-42	LL Mountain Frame
2	925	LL Mountain Frame - Black, 44	FR-M21B-44	LL Mountain Frame
3	926	LL Mountain Frame - Black, 48	FR-M21B-48	LL Mountain Frame
4	927	LL Mountain Frame - Black, 52	FR-M21B-52	LL Mountain Frame
5	917	LL Mountain Frame - Silver, 42	FR-M21S-42	LL Mountain Frame
6	918	LL Mountain Frame - Silver, 44	FR-M21S-44	LL Mountain Frame
7	919	LL Mountain Frame - Silver, 48	FR-M21S-48	LL Mountain Frame
8	920	LL Mountain Frame - Silver, 52	FR-M21S-52	LL Mountain Frame
9	943	LL Mountain Frame - Black, 40	FR-M21B-40	LL Mountain Frame
10	944	LL Mountain Frame - Silver, 40	FR-M21S-40	LL Mountain Frame

PROCEDURE 2 – How to update a view in SQL.

The SQL **UPDATE VIEW** command can be used to modify the data of a view. All views are not updatable. So, UPDATE command is not applicable to all views. An updatable view is one which allows performing a UPDATE command on itself without affecting any other table.

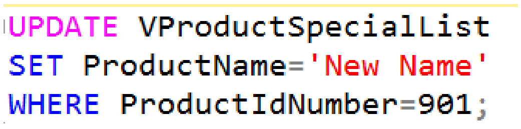
The SELECT statement that defines the view must not contain any of the following elements:

- Aggregate functions such as MIN, MAX, SUM, AVG, and COUNT.
- DISTINCT or TOP clause in its definition.
- GROUP BY or HAVING clause in its definition.
- UNION or UNION ALL clause in its definition.
- Left join or outer join.
- Subquery in the SELECT clause or in the WHERE clause that refers to the table appeared in the FROM clause.
- Reference to non-updatable view in the FROM clause.
- Reference only to literal values.
- Multiple references to any column of the base table.
- Calculated column.

The following statement defines the syntax to update a view.

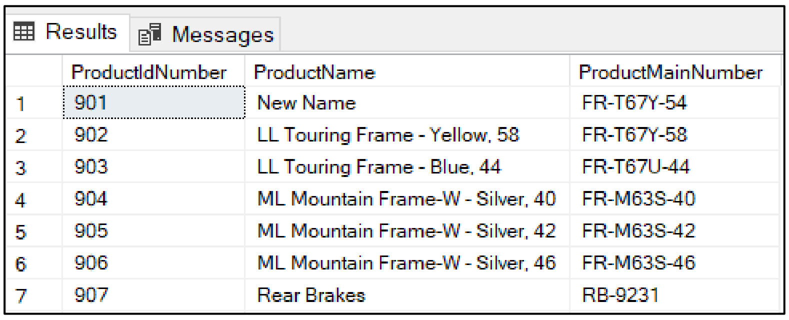
```
UPDATE < view_name > SET<column1>=<value1>,<column2>=<value2>,.....
WHERE <condition>;
```

Step 1 - If we want to update the Product Name information to “New Name” from “LL Touring Frame-Yellow’, 54” for ProductIDNumber = 901, we will need to update the VProductSpecialList view.



```sql
UPDATE VProductSpecialList
SET ProductName='New Name'
WHERE ProductIdNumber=901;
```

** Execute the query and review the results.

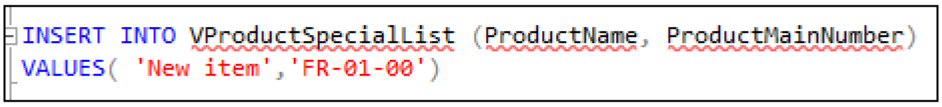


	ProductIdNumber	ProductName	ProductMainNumber
1	901	New Name	FR-T67Y-54
2	902	LL Touring Frame - Yellow, 58	FR-T67Y-58
3	903	LL Touring Frame - Blue, 44	FR-T67U-44
4	904	ML Mountain Frame-W - Silver, 40	FR-M63S-40
5	905	ML Mountain Frame-W - Silver, 42	FR-M63S-42
6	906	ML Mountain Frame-W - Silver, 46	FR-M63S-46
7	907	Rear Brakes	RB-9231

PROCEDURE 3 - How to insert or delete rows through a view in SQL.

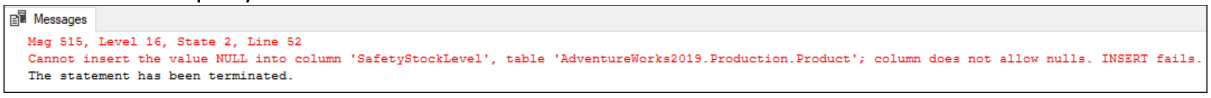
Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the **INSERT** command. You can insert rows into a view only if the view is modifiable and contains no derived columns. The reason for the second restriction is that an inserted row must provide values for all columns, but the database server cannot tell how to distribute an inserted value through an expression. When a modifiable view contains no derived columns, you can insert into it as if it were a table. The database server, however, uses NULL as the value for any column that is not exposed by the view. If such a column does not allow NULL values, an error occurs, and the insert fails.

Step 1 - We're going to use the **VProductSpecialList** to insert a record into our **Product** table.



```
INSERT INTO VProductSpecialList (ProductName, ProductMainNumber)
VALUES( 'New item','FR-01-00')
```

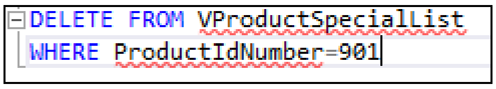
** Execute the query and review the results.



```
Msg 515, Level 16, State 2, Line 52
Cannot insert the value NULL into column 'SafetyStockLevel', table 'AdventureWorks2019.Production.Product'; column does not allow nulls. INSERT fails.
The statement has been terminated.
```

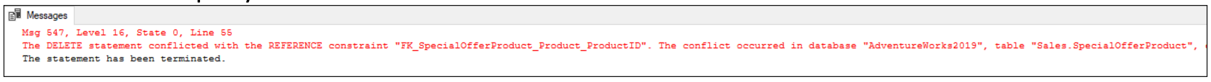
** An INSERT statement fails due to columns with null values. Here, we cannot insert rows in the VProductSpecialList because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Step 2 - Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the **DELETE** command. Following is an example to delete a record having ProductIdNumber = 901.



```
DELETE FROM VProductSpecialList
WHERE ProductIdNumber=901
```

** Execute the query and review the results.



```
Msg 547, Level 16, State 0, Line 55
The DELETE statement conflicted with the REFERENCE constraint "FK_SpecialOfferProduct_Product_ProductID". The conflict occurred in database "AdventureWorks2019", table "Sales.SpecialOfferProduct",
The statement has been terminated.
```

** DELETE statement fails because the Product table contains rows related to the SpecialOfferProduct table.

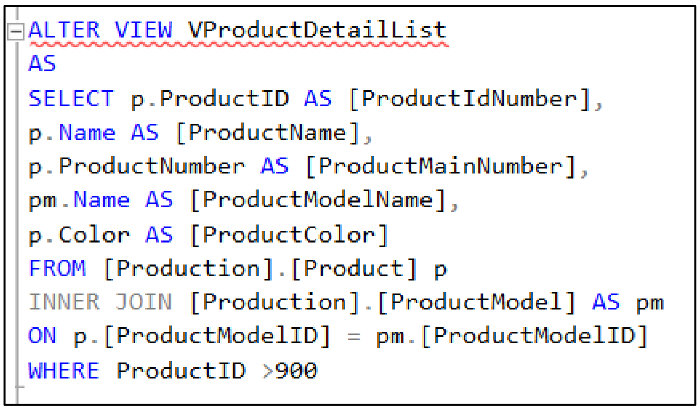
PROCEDURE 4 - How to modify or delete a view in SQL.

The SQL **ALTER VIEW** command modifies a previously created view. The following statement defines the syntax to modify a view.

```
ALTER VIEW view_name [(column_name_1 [, column_name_2]...)]
[WITH {ENCRYPTION|SCHEMABINDING|ENCRYPTION,SCHEMABINDING}]
AS
select_statement
[WITH CHECK OPTION]
```

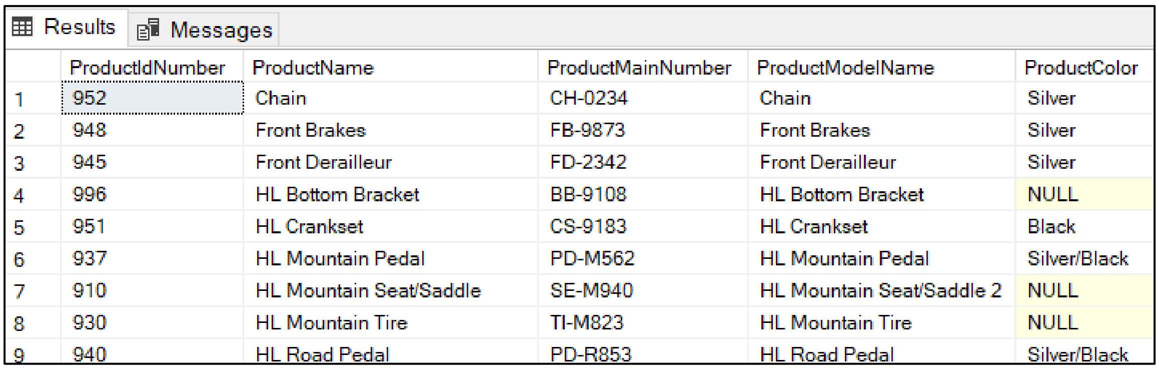
Note that; you cannot use this command to change the definition for a view. To change the view definition, you must drop the view and then recreate it.

Step 1 - If we need the color information of the product other than the Product ID, Name, Product Number and the name of the Product model, we will need to modify the **VProductDetailList** view.



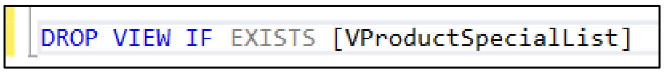
```
ALTER VIEW VProductDetailList
AS
SELECT p.ProductID AS [ProductIdNumber],
p.Name AS [ProductName],
p.ProductNumber AS [ProductMainNumber],
pm.Name AS [ProductModelName],
p.Color AS [ProductColor]
FROM [Production].[Product] p
INNER JOIN [Production].[ProductModel] AS pm
ON p.[ProductModelID] = pm.[ProductModelID]
WHERE ProductID >900
```

** Execute the query and review the results.



	ProductIdNumber	ProductName	ProductMainNumber	ProductModelName	ProductColor
1	952	Chain	CH-0234	Chain	Silver
2	948	Front Brakes	FB-9873	Front Brakes	Silver
3	945	Front Derailleur	FD-2342	Front Derailleur	Silver
4	996	HL Bottom Bracket	BB-9108	HL Bottom Bracket	NULL
5	951	HL Crankset	CS-9183	HL Crankset	Black
6	937	HL Mountain Pedal	PD-M562	HL Mountain Pedal	Silver/Black
7	910	HL Mountain Seat/Saddle	SE-M940	HL Mountain Seat/Saddle 2	NULL
8	930	HL Mountain Tire	TI-M823	HL Mountain Tire	NULL
9	940	HL Road Pedal	PD-R853	HL Road Pedal	Silver/Black

Step 2- In order to delete a view in a database, we can use the **DROP VIEW** statement. However, the DROP VIEW statement may return an error if the view we want to delete do not exists in the database. To overcome this issue, we can use the **IF EXISTS** keyword with the DROP VIEW statement. The following script deletes the **vProductSpecialList** from the database:

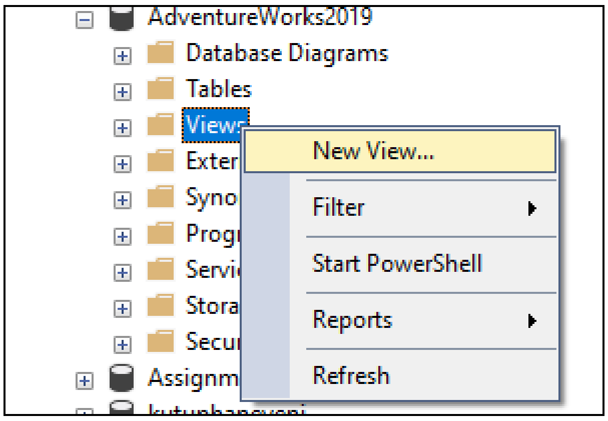


```
DROP VIEW IF EXISTS [VProductSpecialList]
```

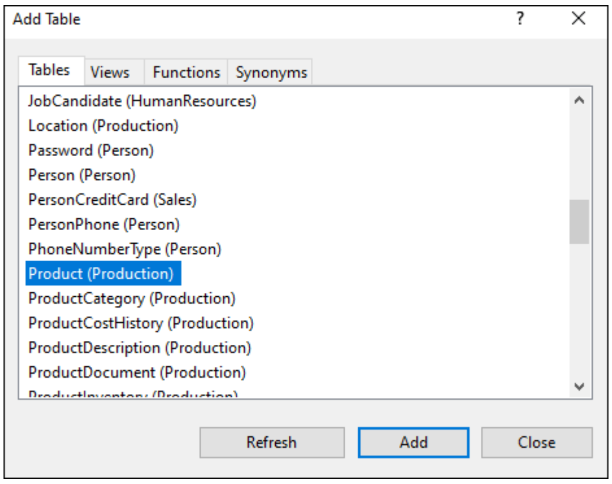
PROCEDURE 5 - How to use the View Designer in SQL.

Step 1 – We can create views with SSMS. The following example will create a view that will be named as **VProductSpecialList. VProductSpecialList** view fetches data from the **Product** table and it only returns the **ProductID**, **Name** and **ProductNumber** columns for products with **ProductID** greater than 900 from the **Product** table.

- To create a view| Expand the server node, and then expand the Databases folder |Expand the AdventureWorks2019 database |Right-click the folder labeled Views and select New View from the context menu.

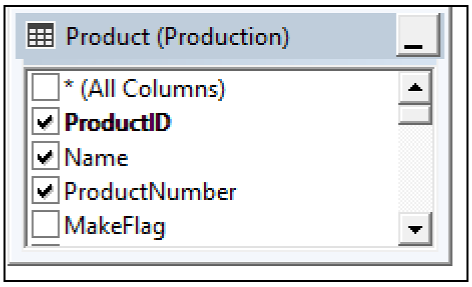


- The Add Table dialog box will appear. Scroll down the list, and locate and select the Product (Production) table. Click Add. The Production table will appear in the Diagram pane.

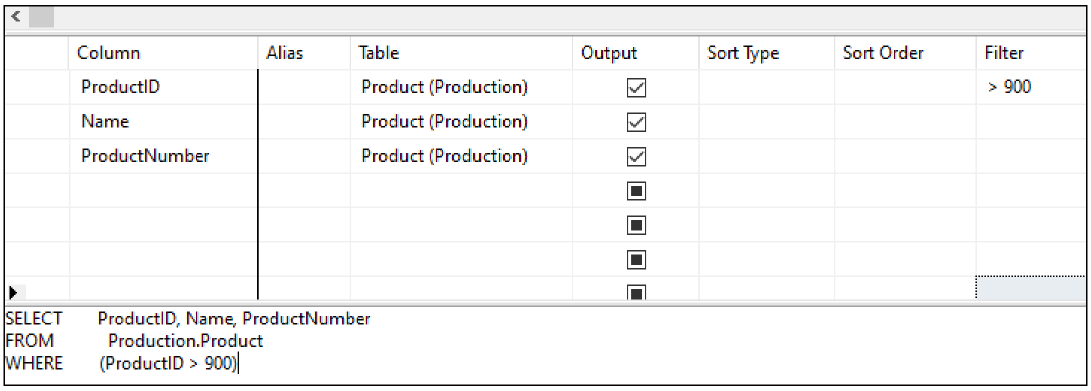


JobCandidate (HumanResources)
Location (Production)
Password (Person)
Person (Person)
PersonCreditCard (Sales)
PersonPhone (Person)
PhoneNumberType (Person)
Product (Production)
ProductCategory (Production)
ProductCostHistory (Production)
ProductDescription (Production)
ProductDocument (Production)

- In the diagram pane, click the box next to the ProductID, Name and ProductNumber columns in the Product table.



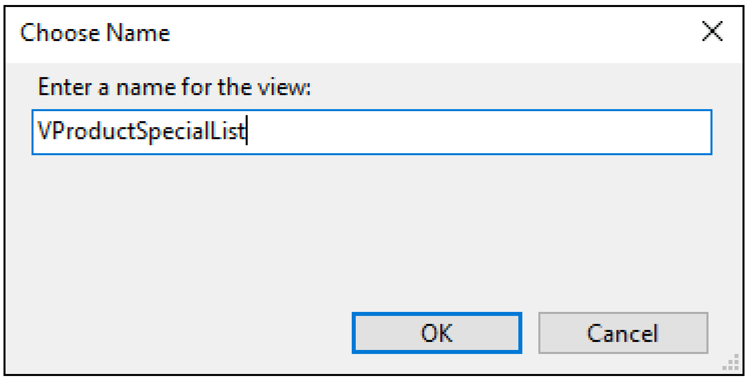
- Now your screen should resemble the following image.



	Column	Alias	Table	Output	Sort Type	Sort Order	Filter
	ProductID		Product (Production)	✓			> 900
	Name		Product (Production)	✓			
	ProductNumber		Product (Production)	✓			
				▣			
				▣			
				▣			
▶				▣			

```
SELECT     ProductID, Name, ProductNumber
FROM        Production.Product
WHERE      (ProductID > 900)
```

- Click the save icon in the SSMS menu bar and the Choose Name dialog box will appear. Type **VProductSpecialList** in the Enter a Name for the View text box and click OK.



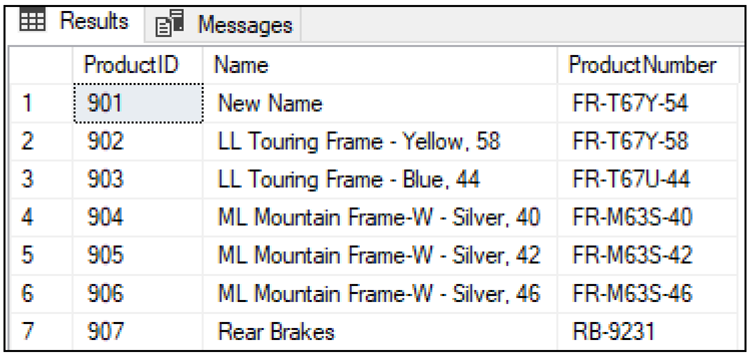
Choose Name

Enter a name for the view: VProductSpecialList

- Open a new query window and type the following code:

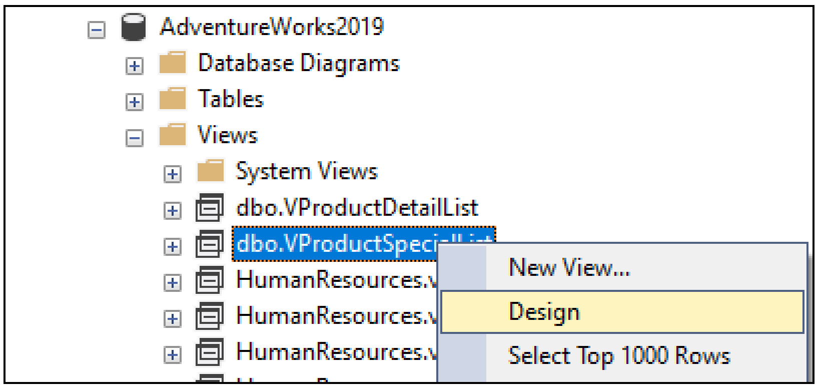
```
SELECT * FROM VProductSpecialList
```

** Execute the query and review the results.

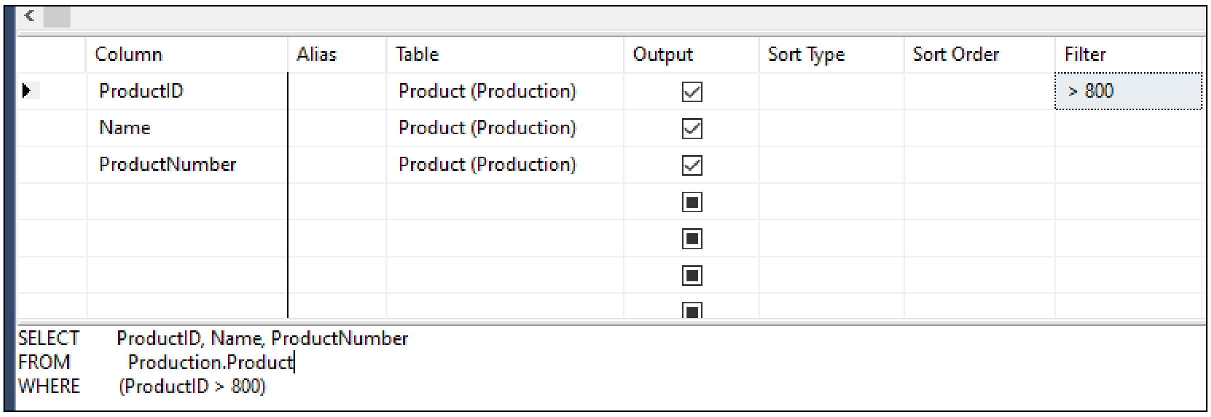


	ProductID	Name	ProductNumber
1	901	New Name	FR-T67Y-54
2	902	LL Touring Frame - Yellow, 58	FR-T67Y-58
3	903	LL Touring Frame - Blue, 44	FR-T67U-44
4	904	ML Mountain Frame-W - Silver, 40	FR-M63S-40
5	905	ML Mountain Frame-W - Silver, 42	FR-M63S-42
6	906	ML Mountain Frame-W - Silver, 46	FR-M63S-46
7	907	Rear Brakes	RB-9231

Step 2 – To alter views, Expand the server node, and then expand the Databases folder |Expand the AdventureWorks2019 database |Expand the Views folder | Right-click the dbo.VProductSpecialList view and select Design from the context menu.



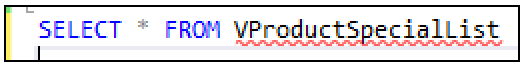
- We change the filter condition to 800 from 900.



Column	Alias	Table	Output	Sort Type	Sort Order	Filter
ProductID		Product (Production)	✓			> 800
Name		Product (Production)	✓			
ProductNumber		Product (Production)	✓			

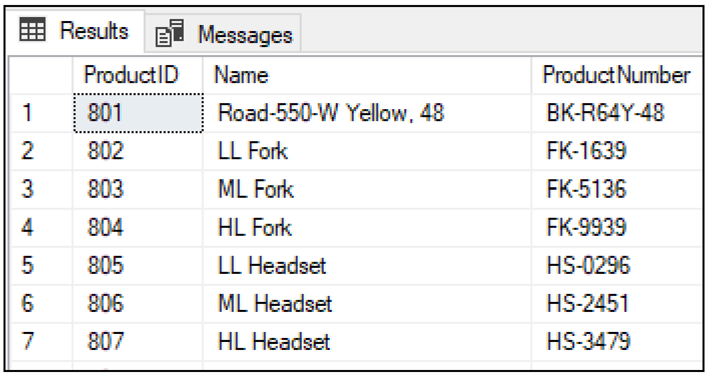
```
SELECT      ProductID, Name, ProductNumber
FROM          Production.Product
WHERE       (ProductID > 800)
```

- Open a new query window and execute the following query:



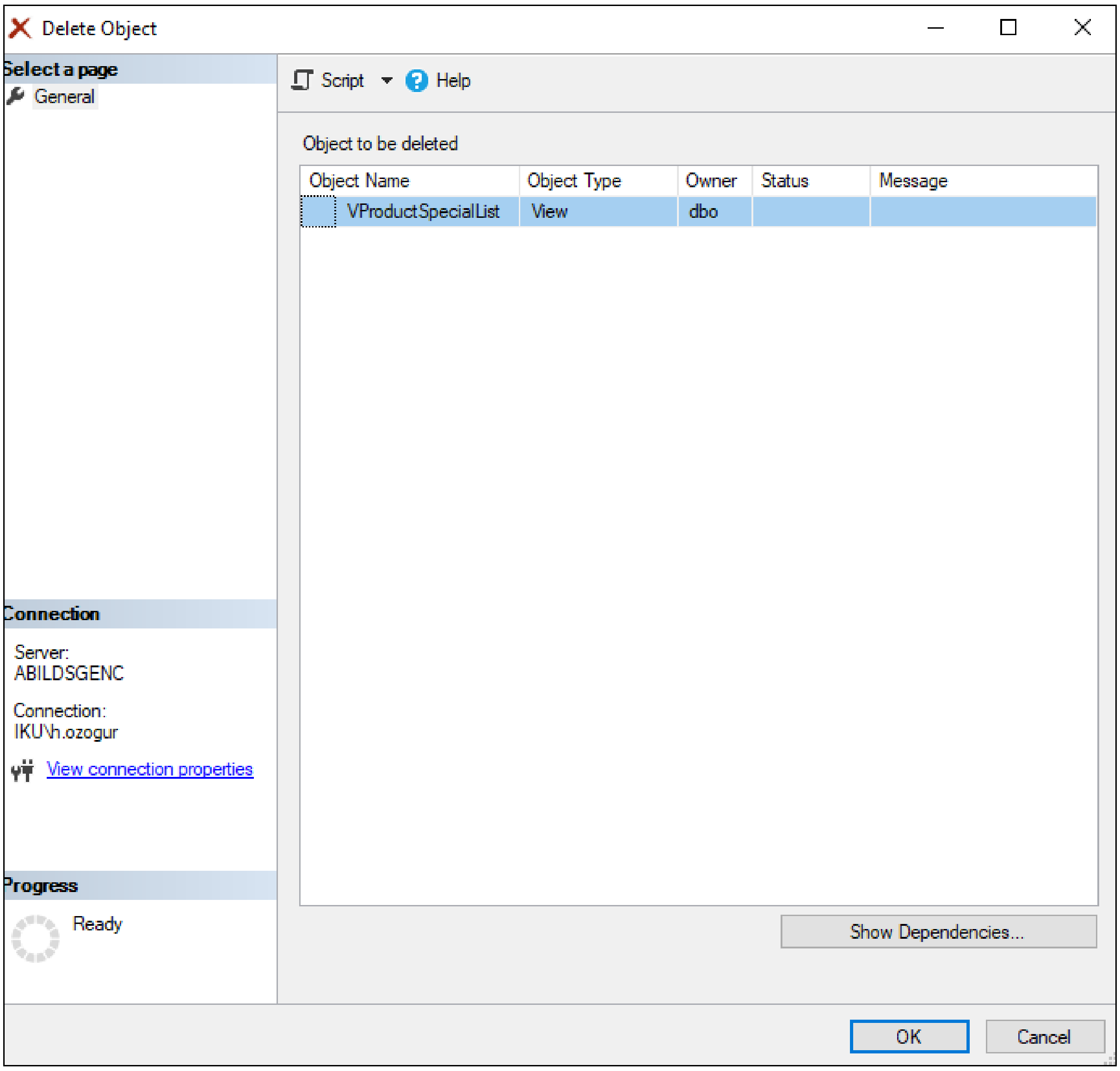
```
SELECT * FROM VProductSpecialList
```

** Execute the query and review the results.



	ProductID	Name	ProductNumber
1	801	Road-550-W Yellow, 48	BK-R64Y-48
2	802	LL Fork	FK-1639
3	803	ML Fork	FK-5136
4	804	HL Fork	FK-9939
5	805	LL Headset	HS-0296
6	806	ML Headset	HS-2451
7	807	HL Headset	HS-3479

Step 3 – To drop the view, right-click the dbo.vwEmployeeInformation view in Object Explorer and select Delete from the context menu The Delete Object dialog box will appear.



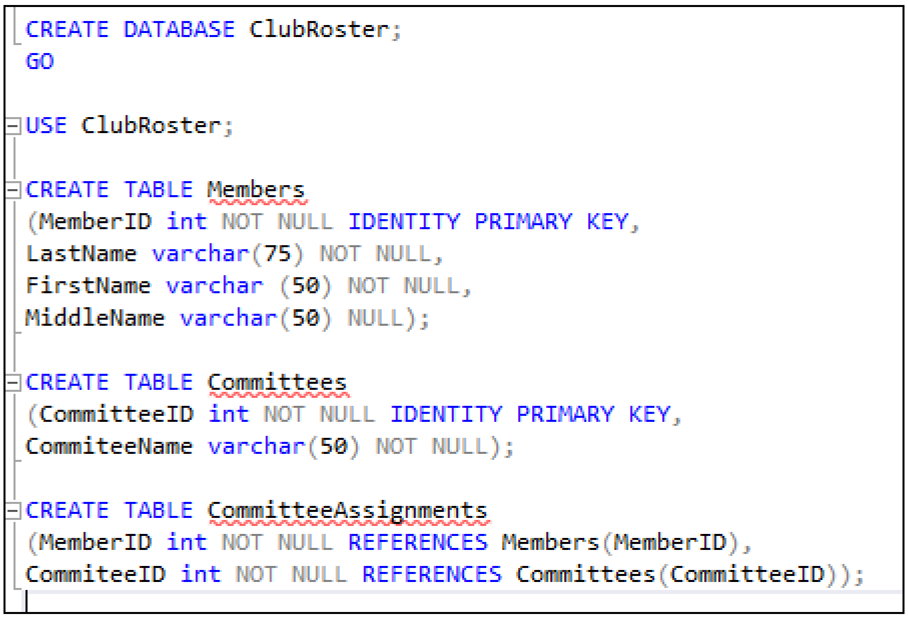
Object to be deleted

Object Name	Object Type	Owner	Status	Message
VProductSpecialList	View	dbo		

PROCEDURE 6 – How to code scripts in SQL.

A **script** can include any number of statements, and those statements can be divided into one or more batches. You can use SQL Scripts to create, edit, view, run, and delete script files. To indicate the end of a batch, you code **GO** command.

Step 1- We write a script that consists of two batches. The first one creates a database, and the second one creates three tables in that database.

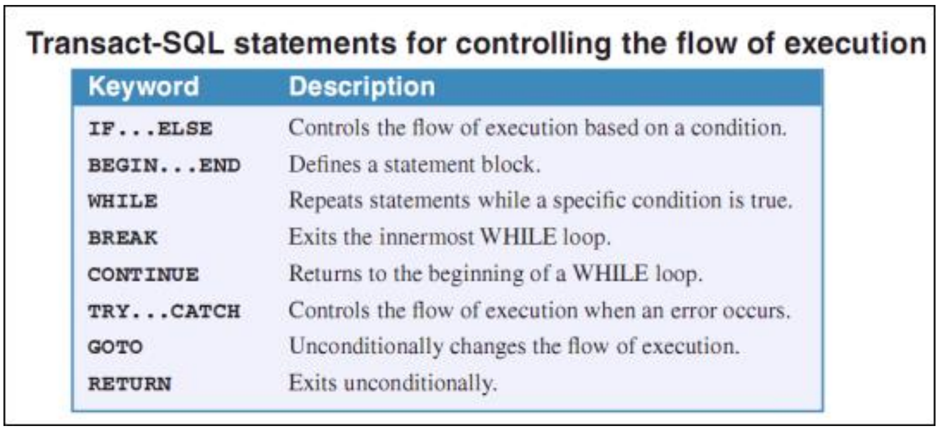


```
CREATE DATABASE ClubRoster;
GO

USE ClubRoster;

CREATE TABLE Members
(MemberID int NOT NULL IDENTITY PRIMARY KEY,
LastName varchar(75) NOT NULL,
FirstName varchar (50) NOT NULL,
MiddleName varchar(50) NULL);

CREATE TABLE Committees
(CommitteeID int NOT NULL IDENTITY PRIMARY KEY,
CommiteeName varchar(50) NOT NULL);

CREATE TABLE CommitteeAssignments
(MemberID int NOT NULL REFERENCES Members(MemberID),
CommiteeID int NOT NULL REFERENCES Committees(CommitteeID));
```

Statements that must be in their own batch.

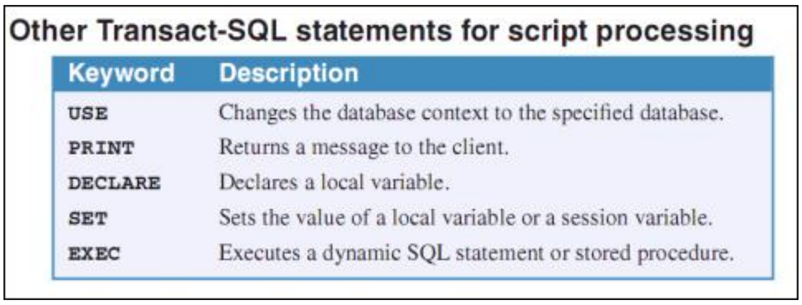
- CREATE VIEW
- CREATE PROCEDURE
- CREATE FUNCTION
- CREATE TRIGGER
- CREATE SCHEMA

Step 2 – Transact-SQL statements are used within SQL scripts to add functionality similar to the procedural programming languages.



Transact-SQL statements for controlling the flow of execution

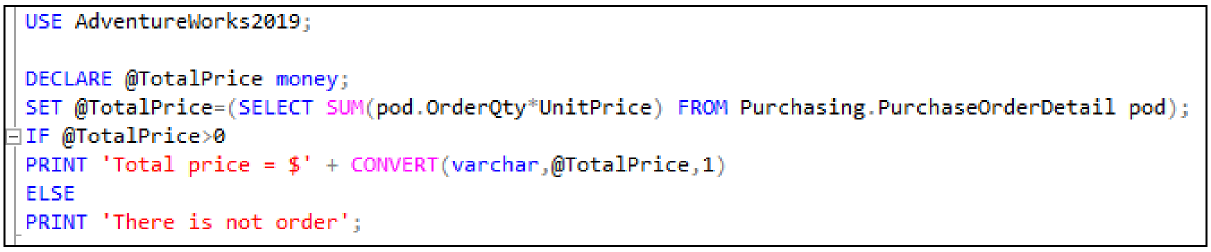
Keyword	Description
IF...ELSE	Controls the flow of execution based on a condition.
BEGIN...END	Defines a statement block.
WHILE	Repeats statements while a specific condition is true.
BREAK	Exits the innermost WHILE loop.
CONTINUE	Returns to the beginning of a WHILE loop.
TRY...CATCH	Controls the flow of execution when an error occurs.
GOTO	Unconditionally changes the flow of execution.
RETURN	Exits unconditionally.



Other Transact-SQL statements for script processing

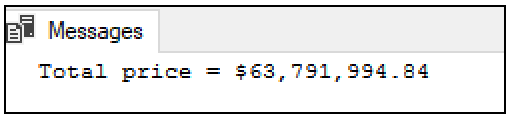
Keyword	Description
`USE`	Changes the database context to the specified database.
`PRINT`	Returns a message to the client.
`DECLARE`	Declares a local variable.
`SET`	Sets the value of a local variable or a session variable.
`EXEC`	Executes a dynamic SQL statement or stored procedure.

** Let's write a script to find the total price of the products sold in the **PurchaseOrderDetail** table in the **AdventureWorks2019** database.



```sql
USE AdventureWorks2019;

DECLARE @TotalPrice money;
SET @TotalPrice=(SELECT SUM(pod.OrderQty*UnitPrice) FROM Purchasing.PurchaseOrderDetail pod);
IF @TotalPrice>0
PRINT 'Total price = $' + CONVERT(varchar,@TotalPrice,1)
ELSE
PRINT 'There is not order';
```

** Execute the query and review the results.



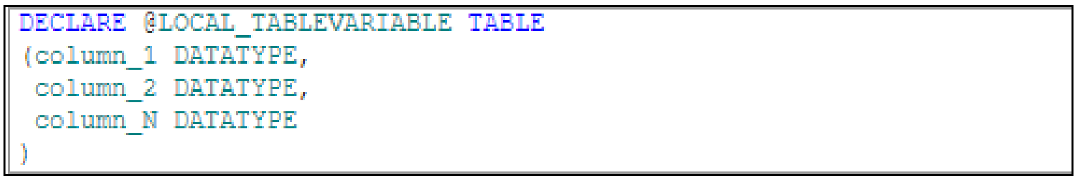
Messages

```
Total price = $63,791,994.84
```

Note: To create a variable, you use the **DECLARE** statement. The initial value of a variable is always null. The name of a variable must always start with an at **sign (@).** Whenever possible, you should use long, descriptive names for variables. To assign a value to a variable, you use **SET** statement. Alternatively, you can use the **SELECT** statement to assign a value to one or more variables.

Step 3 – You can use to create the table variables the syntax of the **DECLARE** statement. To create this type of variable, you specify the table data type in the **DECLARE** statement rather than one of the standard SQL data types. Then, you define the columns and constraints for the table using the same syntax that you for the **CREATE TABLE** statement.

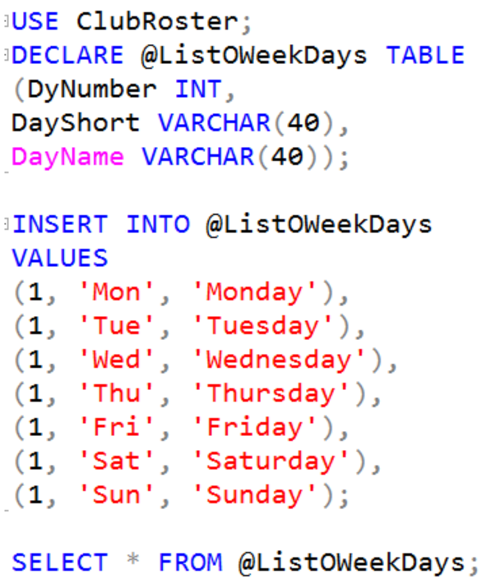
The following syntax describes how to declare a table variable:



```sql
DECLARE @LOCAL_TABLEVARIABLE TABLE
(column_1 DATATYPE,
 column_2 DATATYPE,
 column_N DATATYPE
)
```

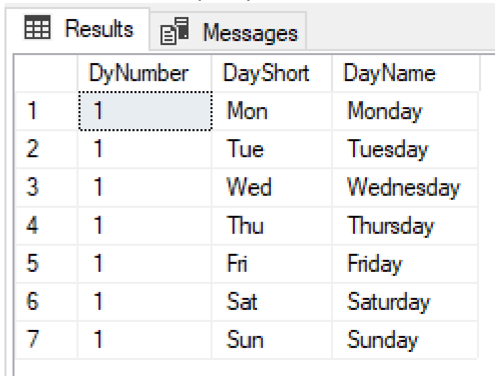
If we want to declare a table variable, we have to start the DECLARE statement which is similar to local variables. The name of the local variable must start with at(@) sign. The TABLE keyword specifies that this variable is a table variable. After the TABLE keyword, we have to define column names and datatypes of the table variable in SQL Server.

** In the following example, we will declare a table variable and insert the days of the week and their abbreviations to the table variable:



```
USE ClubRoster;
DECLARE @ListOWeekDays TABLE
(DyNumber INT,
DayShort VARCHAR(40),
DayName VARCHAR(40));

INSERT INTO @ListOWeekDays
VALUES
(1, 'Mon', 'Monday'),
(1, 'Tue', 'Tuesday'),
(1, 'Wed', 'Wednesday'),
(1, 'Thu', 'Thursday'),
(1, 'Fri', 'Friday'),
(1, 'Sat', 'Saturday'),
(1, 'Sun', 'Sunday');

SELECT * FROM @ListOWeekDays;
```

** Execute the query and review the results.

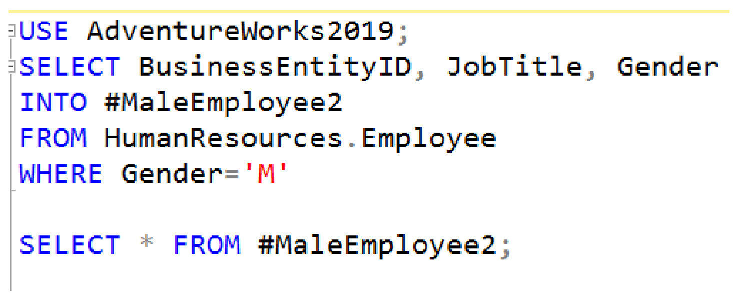


Results | Messages

	DyNumber	DayShort	DayName
1	1	Mon	Monday
2	1	Tue	Tuesday
3	1	Wed	Wednesday
4	1	Thu	Thursday
5	1	Fri	Friday
6	1	Sat	Saturday
7	1	Sun	Sunday

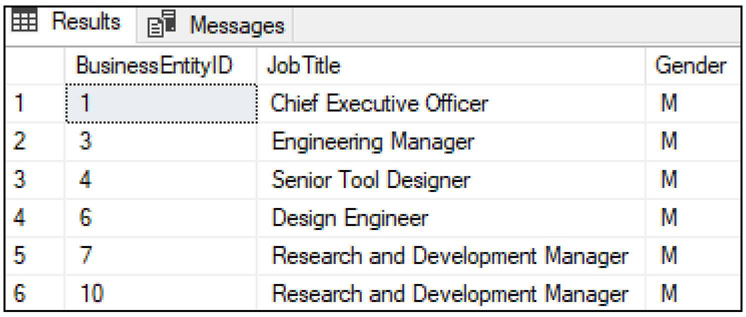
Step 4 - A **temporary table** in SQL Server is a database table that exists temporarily on the database server. A temporary table stores a subset of data from a normal table for a certain period of time. Temporary tables are particularly useful when you have a large number of records in a table and you repeatedly need to interact with a small subset of those records. In such cases instead of filtering the data again and again to fetch the subset, you can filter the data once and store it in a temporary table. You can then execute your queries on that temporary table. Temporary tables are stored inside "**tempdb**" which is a system database.

- The simplest way of creating a **temporary table** is by using an **INTO** statement within a SELECT query. The name of a temporary table must start with a **hash (#).** Let's create a **local temporary table** that contains the **BusinessEntityID**, **JobTitle**, and **Gender** of all the male employee records from the **Employee** table.



```sql
USE AdventureWorks2019;
SELECT BusinessEntityID, JobTitle, Gender
INTO #MaleEmployee2
FROM HumanResources.Employee
WHERE Gender='M'

SELECT * FROM #MaleEmployee2;
```

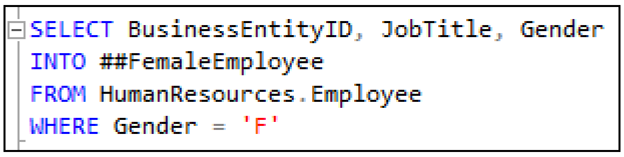
** Execute the query and review the results.



	BusinessEntityID	JobTitle	Gender
1	1	Chief Executive Officer	M
2	3	Engineering Manager	M
3	4	Senior Tool Designer	M
4	6	Design Engineer	M
5	7	Research and Development Manager	M
6	10	Research and Development Manager	M

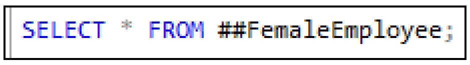
Step 5- It is pertinent to mention here that; a temporary table is only accessible to the connection that created that temporary table. It is not accessible to other connections. However, we can create temporary tables that are accessible to all the open connections. Such temporary tables are called **global temporary tables**. The name of the global temporary table starts with a **double hash symbol (##)**.

- Let's create a global temporary table that contains records of all female employees from the Employee table.

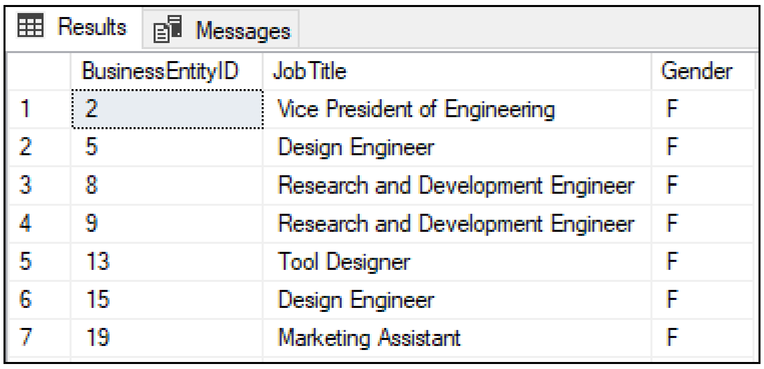


```sql
SELECT BusinessEntityID, JobTitle, Gender
INTO ##FemaleEmployee
FROM HumanResources.Employee
WHERE Gender = 'F'
```

** Execute the query and review the results.

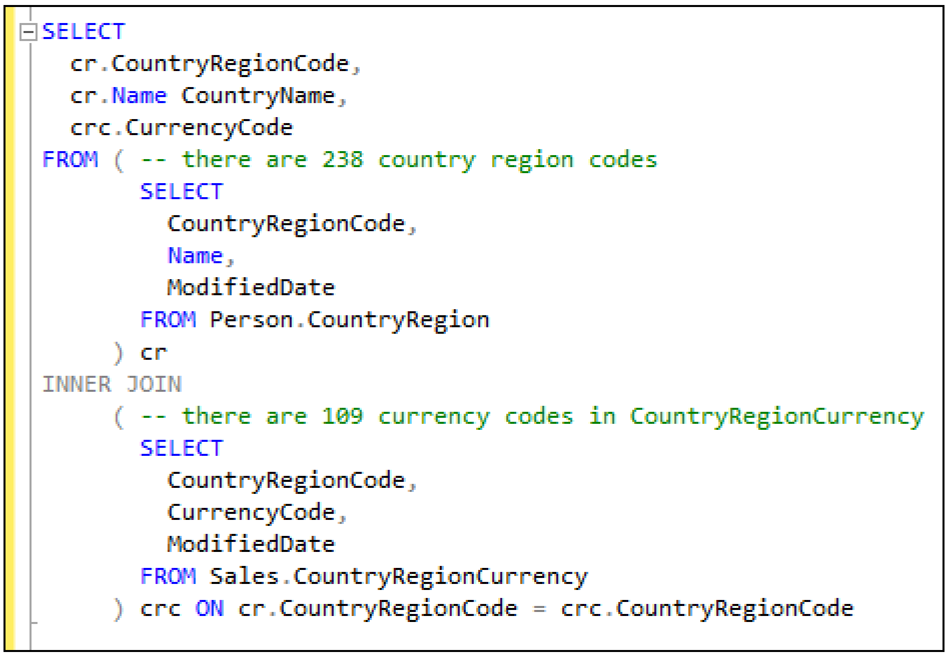


```sql
SELECT * FROM ##FemaleEmployee;
```



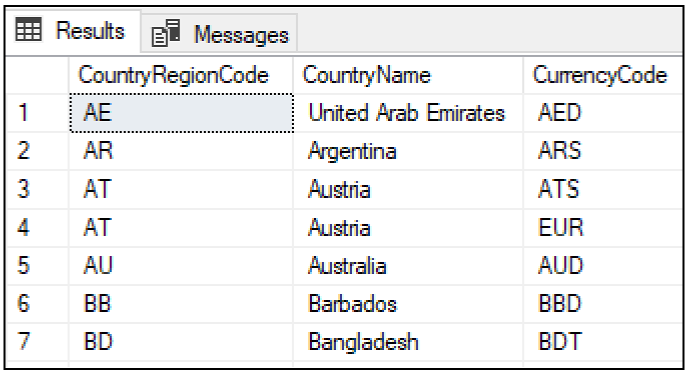
	BusinessEntityID	JobTitle	Gender
1	2	Vice President of Engineering	F
2	5	Design Engineer	F
3	8	Research and Development Engineer	F
4	9	Research and Development Engineer	F
5	13	Tool Designer	F
6	15	Design Engineer	F
7	19	Marketing Assistant	F

Step 6- A **derived table** is a subquery nested within a **FROM** clause. Because of being in a FROM clause, the subquery's result set can be used similarly to a SQL Server table. The subquery in the FROM clause must have a name. One reason for including a derived table in an outer query is to simplify the outer query.



```
SELECT
   cr.CountryRegionCode,
   cr.Name CountryName,
   crc.CurrencyCode
FROM ( -- there are 238 country region codes
        SELECT
           CountryRegionCode,
           Name,
           ModifiedDate
        FROM Person.CountryRegion
      ) cr
INNER JOIN
      ( -- there are 109 currency codes in CountryRegionCurrency
        SELECT
           CountryRegionCode,
           CurrencyCode,
           ModifiedDate
        FROM Sales.CountryRegionCurrency
      ) crc ON cr.CountryRegionCode = crc.CountryRegionCode
```

** Execute the query and review the results.



	CountryRegionCode	CountryName	CurrencyCode
1	AE	United Arab Emirates	AED
2	AR	Argentina	ARS
3	AT	Austria	ATS
4	AT	Austria	EUR
5	AU	Australia	AUD
6	BB	Barbados	BBD
7	BD	Bangladesh	BDT

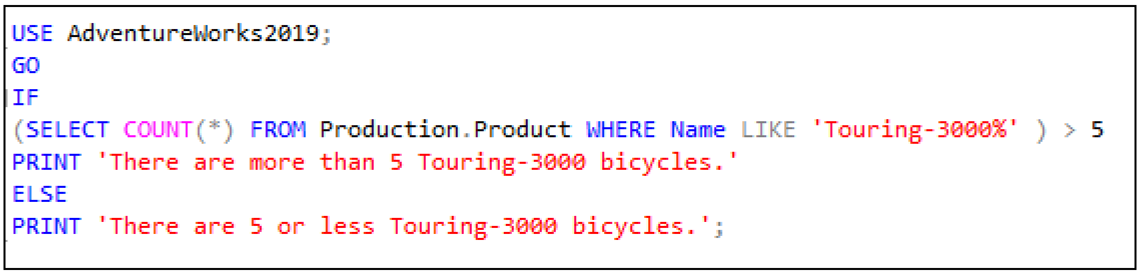
The script shows how to specify an inner join between two derived tables – one based on the CountryRegion table and the other based on the CountryRegionCurrency table.

- The derived table labelled cr returns a result set of 238 rows from the **CountryRegion** table.
- The derived table labelled crc returns a result set of 109 rows from the **CountryRegionCurrency** table.
- The inner join operator and its matching on keywords (CountryRegionCode) indicate that each row in the cr derived table result set should be matched to as many rows in crc derived table result set whenever the CountryRegionCode value is the same in both tables.
- Because there are just 109 rows in the crc derived table and each of its rows match one row in the cr derived table, the result set for the outer query contains 109 rows.

PROCEDURE 7 – How to control the execution of a script.

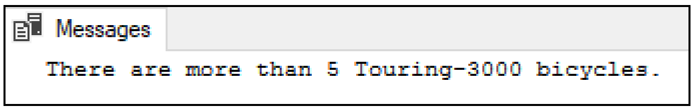
Step 1 - The **IF...ELSE** statement is a control-flow statement that allows you to execute or skip a statement block based on a specified condition. The T-SQL statement that follows an **IF** keyword and its condition is executed if the condition is satisfied: the Boolean expression returns **TRUE**. The optional **ELSE** keyword introduces another T-SQL statement that is executed when the IF condition is not satisfied: the Boolean expression returns **FALSE**.

- The following example executes a query as part of the Boolean expression. Because there are 10 bikes in the Product table that meet the **WHERE** clause, the first print statement will execute.



```
USE AdventureWorks2019;
GO
IF
(SELECT COUNT(*) FROM Production.Product WHERE Name LIKE 'Touring-3000%' ) > 5
PRINT 'There are more than 5 Touring-3000 bicycles.'
ELSE
PRINT 'There are 5 or less Touring-3000 bicycles.';
```

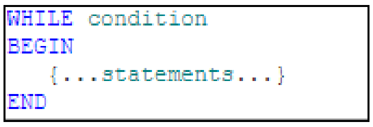
** Execute the query and review the results.



Messages

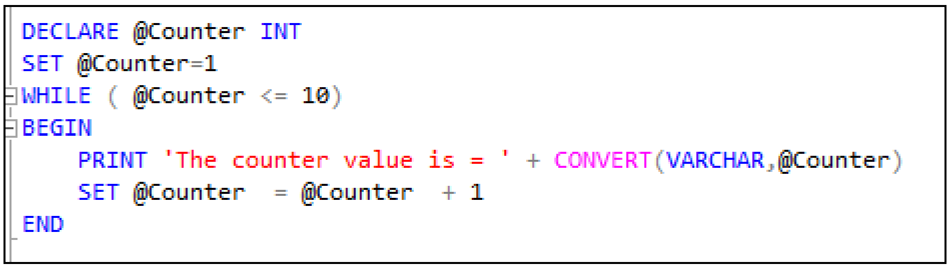
```
There are more than 5 Touring-3000 bicycles.
```

Step 2 - SQL **WHILE** loop provides us with the advantage to execute the SQL statement(s) repeatedly until the specified condition result turn out to be false. The syntax of the WHILE loop in SQL looks like as follows:



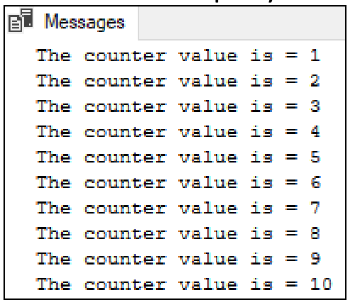
```
WHILE condition
BEGIN
   {...statements...}
END
```

- In the example given below, the **WHILE** loop example will write a value of the variable ten times, and then the loop will be completed:



```
DECLARE @Counter INT
SET @Counter=1
WHILE ( @Counter <= 10)
BEGIN
    PRINT 'The counter value is = ' + CONVERT(VARCHAR,@Counter)
    SET @Counter  = @Counter  + 1
END
```

** Execute the query and review the results.



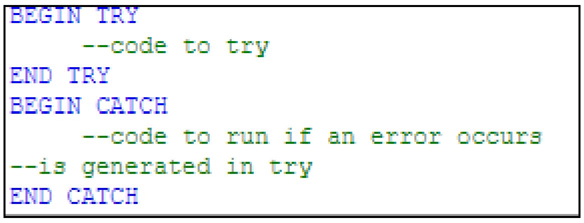
Messages

```
The counter value is = 1
The counter value is = 2
The counter value is = 3
The counter value is = 4
The counter value is = 5
The counter value is = 6
The counter value is = 7
The counter value is = 8
The counter value is = 9
The counter value is = 10
```

PROCEDURE 8 – How to handle errors in SQL.

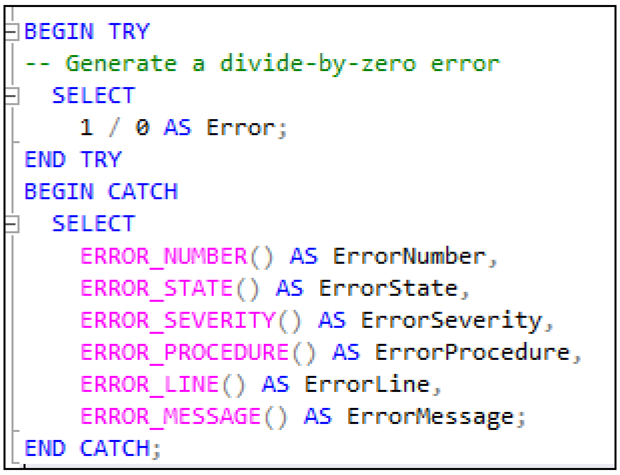
Error handling in SQL Server gives us control over the Transact-SQL code. For example, when things go wrong, we get a chance to do something about it and possibly make it right again.

Step 1 - A **TRY...CATCH** construct catches all execution errors that have a severity higher than 10 that do not close the database connection. A **TRY** block must be immediately followed by an associated **CATCH** block. Including any other statements between the END TRY and BEGIN CATCH statements generates a syntax error.



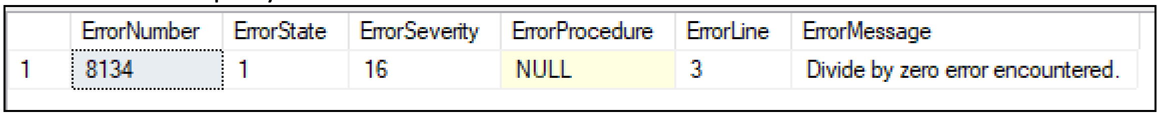
```
BEGIN TRY
     --code to try
END TRY
BEGIN CATCH
     --code to run if an error occurs
--is generated in try
END CATCH
```

- This is an example of how it looks and how it works. The only thing we're doing in the BEGIN TRY is dividing 1 by 0, which, of course, will cause an error. So, as soon as that block of code is hit, it's going to transfer control into the CATCH block and then it's going to select all of the properties using the built-in functions.



```
BEGIN TRY
-- Generate a divide-by-zero error
  SELECT
    1 / 0 AS Error;
END TRY
BEGIN CATCH
  SELECT
    ERROR_NUMBER() AS ErrorNumber,
    ERROR_STATE() AS ErrorState,
    ERROR_SEVERITY() AS ErrorSeverity,
    ERROR_PROCEDURE() AS ErrorProcedure,
    ERROR_LINE() AS ErrorLine,
    ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

** Execute the query and review the results.



	ErrorNumber	ErrorState	ErrorSeverity	ErrorProcedure	ErrorLine	ErrorMessage
1	8134	1	16	NULL	3	Divide by zero error encountered.

** We got two result grids because of two SELECT statements: the first one is 1 divided by 0, which causes the error and the second one is the transferred control that actually gave us some results. From left to right, we got ErrorNumber, ErrorState, ErrorSeverity; there is no procedure in this case (NULL), ErrorLine, and ErrorMessage.

SUMMARY

In this LAB, you learned how to create and use a view in SQL. You learned to create, update, alter, and drop views and to insert or delete rows through a view in SQL. In addition, you learned to create a view by using the View Designer in SQL. You have learned how to code procedural scripts in T-SQL. By using the techniques, you have learned here, you will be able to code script that are more general, more useful, and less susceptible to failure.

PROCEDURE 9 - ASSIGNMENT (Upload the solution to the CATs by your ID)

You can access your assignment at the time your section starts. Then, you can download the assignment from the Assignments section in CATS.