

*the*knowledgeacademy

Advanced SQL



About The Knowledge Academy

The world's largest provider of classroom and online training courses

- ✓ World Class Training Solutions
- ✓ Subject Matter Experts
- ✓ Highest Quality Training Material
- ✓ Accelerated Learning Techniques
- ✓ Project, Programme, and Change Management, ITIL® Consultancy
- ✓ Bespoke Tailor Made Training Solutions
- ✓ PRINCE2®, MSP®, ITIL®, Soft Skills, and More

Course Syllabus

- Module 1: Creating Tables
- Module 2: Stored Procedure Basics
- Module 3: Variables
- Module 4: Parameters and Return Values
- Module 5: Avoiding Scalar Functions
- Module 6: Testing Conditions
- Module 7: Looping
- Module 8: Temporary Tables and Table Variables
- Module 9: Table values Functions
- Module 10: Derived Tables and CTEs
- Module 11: Subqueries
- Module 12: Cursors
- Module 13: Error-Handling



Module 1



Creating Tables

Description

1	Creating Tables in SQL
2	Inserting Data
3	Inserting Multiple Rows
4	View Table

Creating Tables

Creating Tables in SQL

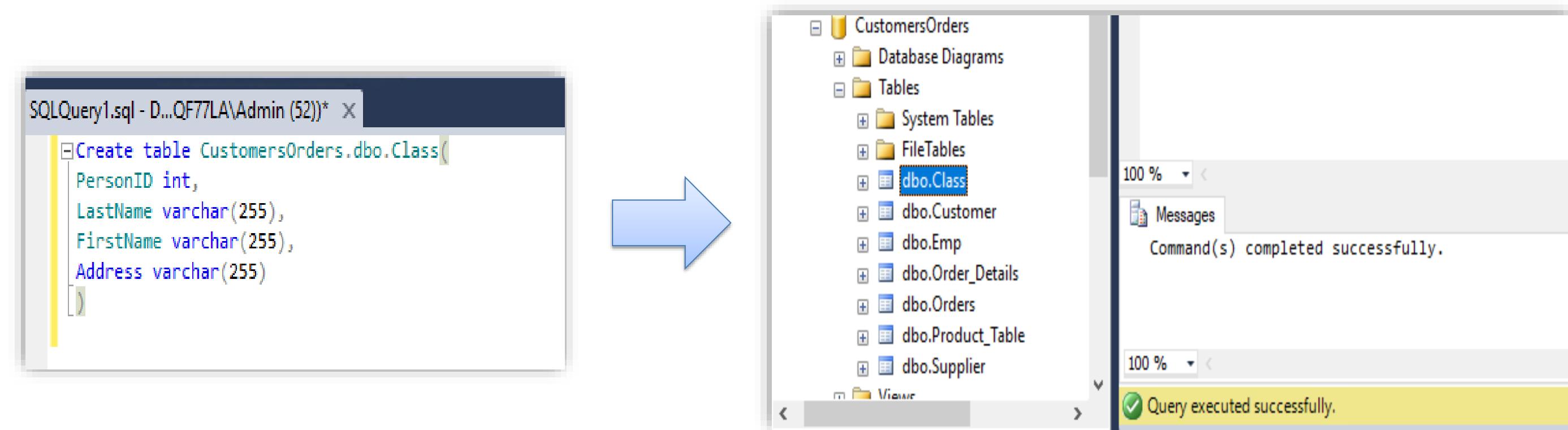
- ✓ The following commands are the syntax for creating table in SQL:

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ....
);
```

Creating Tables

Creating Tables in SQL

- ✓ Execute the following query to create the table:



Creating Tables

Inserting Data in SQL

- ✓ The following are the syntax of insert data query in SQL :

```
INSERT INTO table_name (column1, column2, column3, ...)  
VALUES (value1, value2, value3, ...);
```

Creating Tables

Inserting Data in SQL

- ✓ Execute the following query to insert data into table:

The image shows a screenshot of the SQL Server Management Studio (SSMS) interface. On the left, a query window titled 'SQLQuery1.sql - D...QF77LA\Admin (52)*' contains the following SQL code:

```
insert into CustomersOrders.dbo.Class (PersonID, LastName, FirstName, Address)
Values (123, 'xyz', 'abc', 'mno');
```

A large blue arrow points from the query window to the 'Messages' window on the right. The 'Messages' window has a title bar 'Messages' and a status bar at the bottom showing '100 %'. It displays the following text:

(1 row(s) affected)

Query executed successfully.

Creating Tables

Inserting Multiple Rows in SQL

- ✓ Execute the following query to insert multiple rows into table:

The screenshot shows the SQL Server Management Studio interface. On the left, a query window titled 'SQLQuery1.sql - D...QF77LA\Admin (52)* X' contains the following SQL code:

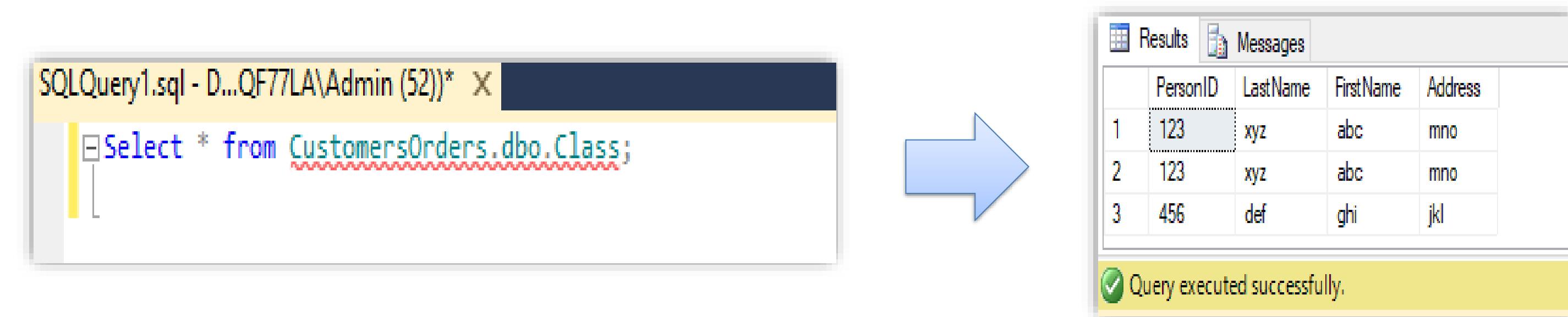
```
insert into CustomersOrders.dbo.Class (PersonID, LastName, FirstName, Address)
Values (123, 'xyz', 'abc', 'mno'),
(456, 'def', 'ghi', 'jkl');
```

A large blue arrow points from the query window to the 'Messages' window on the right. The 'Messages' window has a title bar 'Messages' and a status bar at the bottom showing '100 %'. It displays the message '(2 row(s) affected)' above a yellow success banner that says 'Query executed successfully.'

Creating Tables

View Table in SQL

- ✓ Execute the following query to view the table:



Module 2

Stored Procedure Basics

Description

1

Pros and Cons of Stored Procedures

2

Creating Stored Procedures

3

Two ways to Execute

4

System Stored Procedures

Stored Procedure Basics

Pros of Using Stored Procedure

- ✓ It provides an important security layer between the user interface and the database. It preserves the data integrity

- ✓ It helps to enhance productivity because statements in the stored procedure only need to be written once. It is modular, therefore easier to troubleshoot when any issue arises in an application

- ✓ It is also unable, as it diminishes the need to alter the GUI source code to improve its performance. It helps to reduce the network traffic between clients and servers

Stored Procedure Basics

Cons of Stored Procedures

- ✓ They are challenging to modify as they are written by experienced developers, and not all the members of the project have the right to change it
- ✓ Debugging is tough in stored procedures. Stored procedures are not faster than dynamic SQL's. No version control system is available for stored procedures



Stored Procedure Basics

Creating Stored Procedures

- ✓ A stored procedure is a prepared SQL code that you can save and reuse repeatedly. So, if you have a SQL query that you write frequently, save it as a stored procedure and then call it to execute it
- ✓ You can also pass parameters to a stored procedure to act based on the value(s) of the parameter(s) passed
- ✓ Stored Procedure Syntax:

```
CREATE PROCEDURE procedure_name  
AS  
sql_statement
```

Stored Procedure Basics

Creating Stored Procedures

- ✓ Execute a Stored Procedure Syntax:

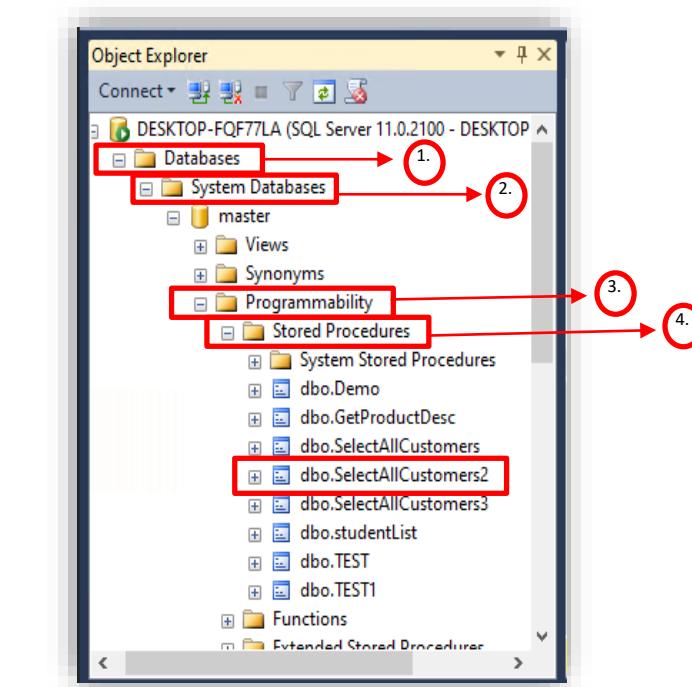
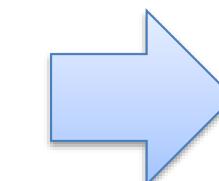
```
EXEC procedure_name;
```

- ✓ The below SQL statement creates a stored procedure named "TEST" that selects all records from the "Persons" table:

The screenshot shows a SQL query window titled "SQLQuery2.sql - D...QF77LA\Admin (53)*". Inside the window, the following SQL code is displayed:

```
CREATE PROCEDURE TEST
AS
select * from Persons
```

A red box highlights the first two lines of the code. At the bottom of the window, a message states: "Command(s) completed successfully."



Stored Procedure Basics

Creating Stored Procedures

- ✓ Execute the stored procedure:

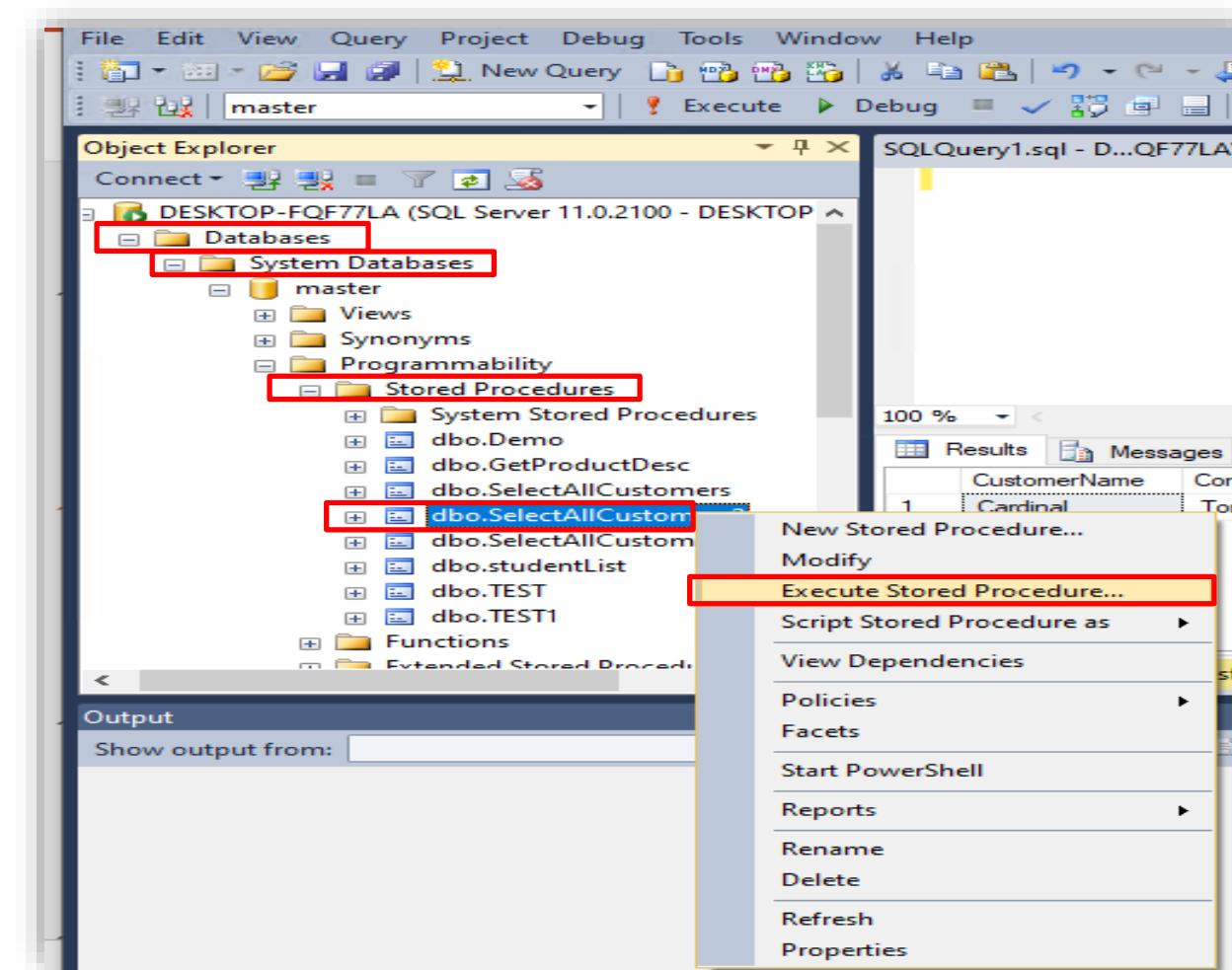
The screenshot shows a SQL Server Management Studio (SSMS) window titled "SQLQuery2.sql - D...QF77LA\Admin (53)*". In the query pane, the command "EXEC TEST;" is highlighted with a red box. The results pane displays a table with two rows of data:

	PersonID	LastName	FirstName	Address	City
1	1	Erichsen	Tom.B	Stavanger	Norway
2	2	Walden	Isaac G	3910 Cliffside Drive	Norway

Stored Procedure Basics

Two ways to Execute

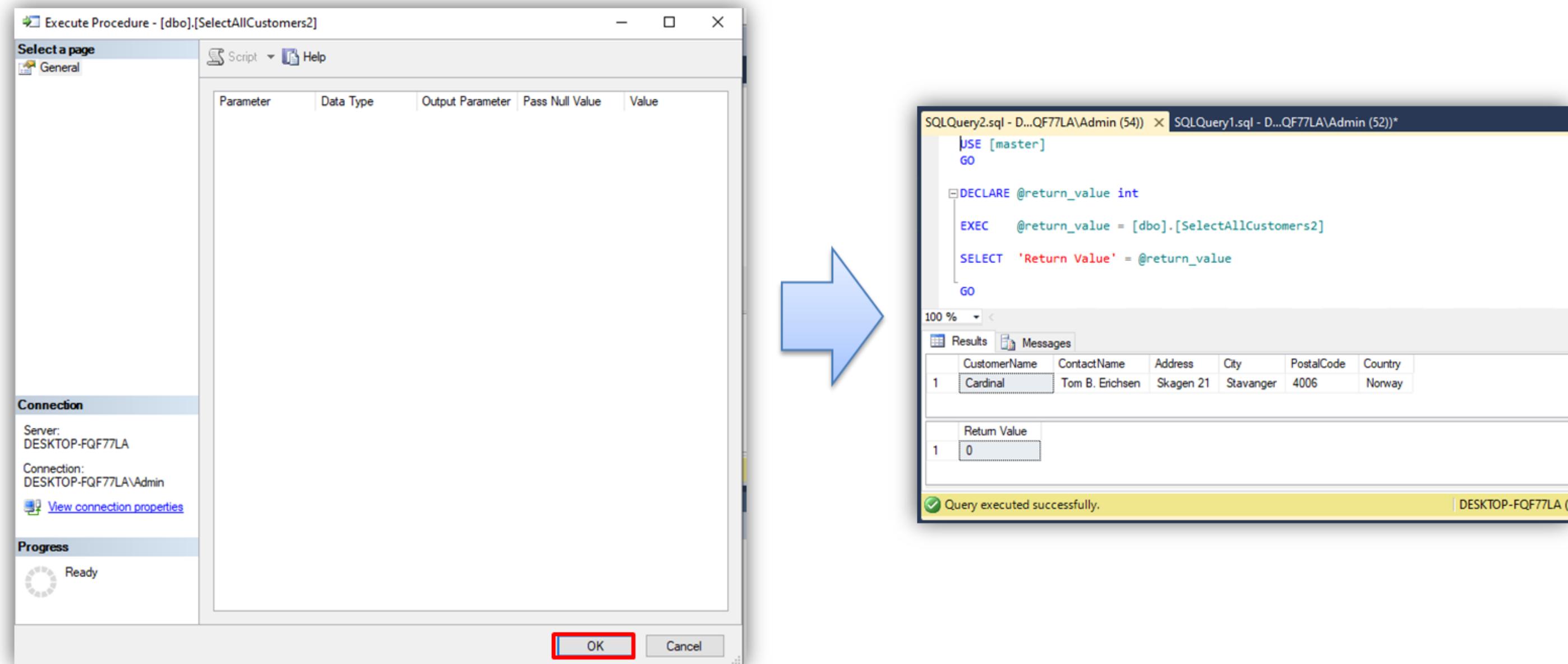
- ✓ There are three ways to execute the stored procedure:
- ✓ Using Object Explorer
- ✓ Step 1: Go to Database > System Database > Programmability > Stored Procedure. Right- click on the procedure, then click on the Execute Stored Procedure...



Stored Procedure Basics

Two ways to Execute

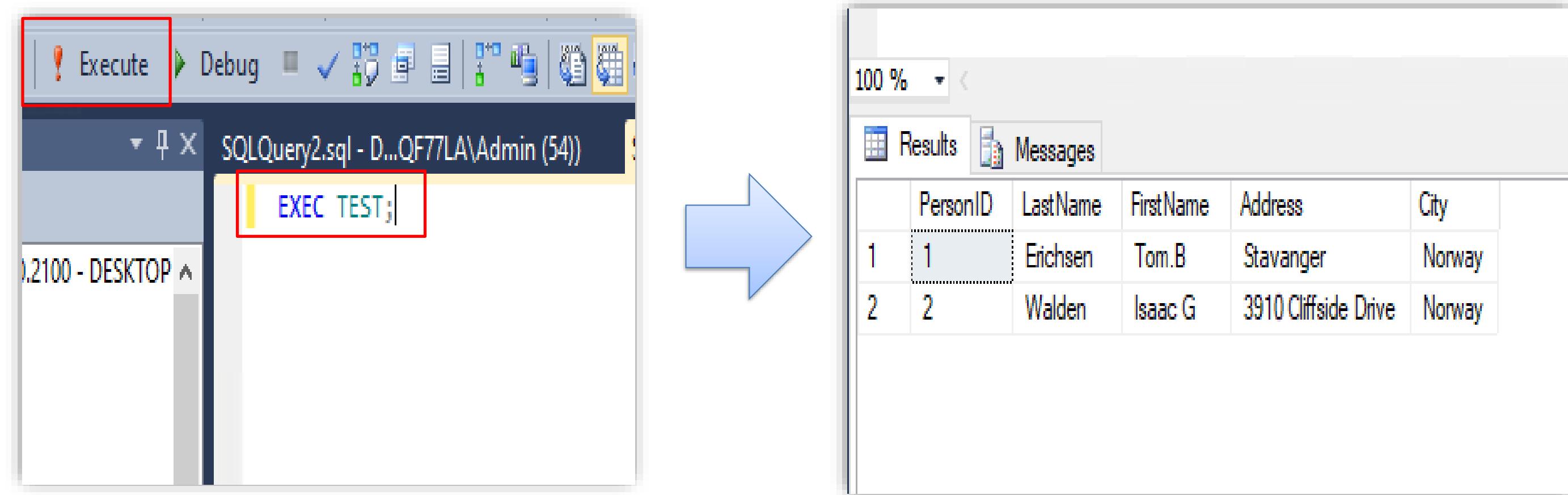
- ✓ Step 2: To see the results, click on OK in the Execute Procedure dialog box



Stored Procedure Basics

Two ways to Execute

- ✓ Using the Query Window
- ✓ To execute procedure from Query Window write the following statement, then click on Execute



Stored Procedure Basics

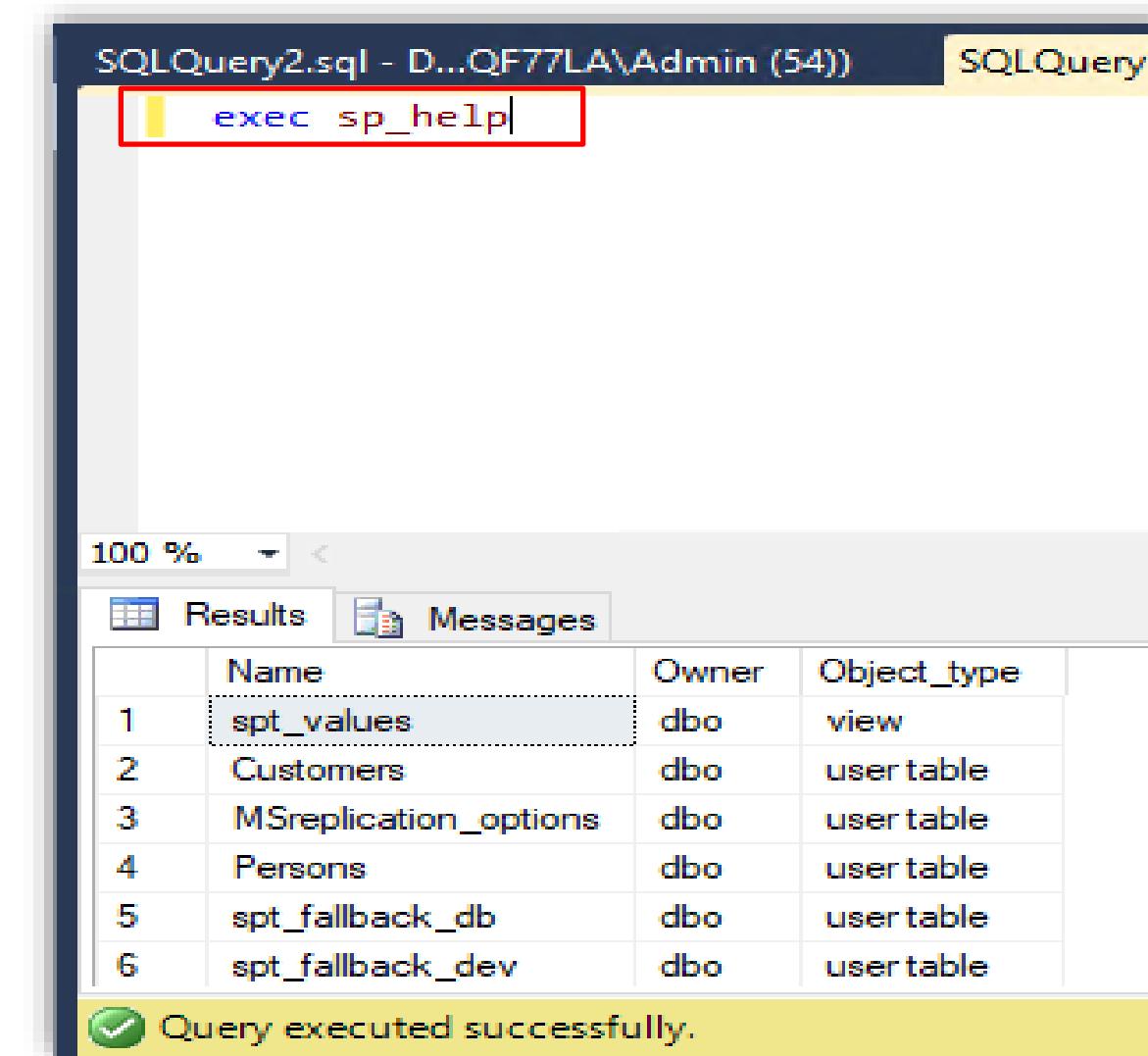
System Stored Procedures

- ✓ These procedures are beneficial in executing the administrative and informational activities in the SQL server
- ✓ Some of the most commonly used system stored procedures are given below:
 - ✓ **Sp_help <Object Name>**
 - ✓ The Sp_help reports information about a user-defined data type, the database object, or a data type
 - ✓ Sp_help will only return information for objects that can be referenced within the database you are currently in

Stored Procedure Basics

System Stored Procedures

- ✓ Write the below statement to find all database object information:



The screenshot shows a SQL Server Management Studio window with two panes: 'Results' and 'Messages'. The 'Results' pane displays a table of database objects. The 'Messages' pane at the bottom shows a successful query execution message.

	Name	Owner	Object_type
1	spt_values	dbo	view
2	Customers	dbo	user table
3	MSreplication_options	dbo	user table
4	Persons	dbo	user table
5	spt_fallback_db	dbo	user table
6	spt_fallback_dev	dbo	user table

Query executed successfully.

Stored Procedure Basics

System Stored Procedures

- ✓ To Find Table Information:

The screenshot shows the SQL Server Management Studio interface. In the top query window, the command `exec sp_help "Customers"` is highlighted with a red box. The results window displays two tables of information about the 'Customers' table.

Table 1: General Table Information

Name	Owner	Type	Created_datetime
Customers	dbo	user table	2022-03-09 06:27:50.123

Table 2: Column Details

Column_name	Type	Computed	Length	Prec	Scale	Nullable	Trim Trailing Blanks	FixedLenNullInSource	Collation
CustomerName	varchar	no	255			yes	no	yes	SQL_Latin1_General_CI_AS
ContactName	varchar	no	255			yes	no	yes	SQL_Latin1_General_CI_AS
Address	varchar	no	255			yes	no	yes	SQL_Latin1_General_CI_AS
City	varchar	no	255			yes	no	yes	SQL_Latin1_General_CI_AS
PostalCode	varchar	no	255			yes	no	yes	SQL_Latin1_General_CI_AS
Country	varchar	no	255			yes	no	yes	SQL_Latin1_General_CI_AS

At the bottom of the results window, a green checkmark icon and the message "Query executed successfully." are visible.

Stored Procedure Basics

System Stored Procedures

- ✓ **Sp_table<Object Name>**
- ✓ The Sp_help reports information about the list of the tables from the database. The system stored procedure shown below, returns all the tables first in the result set, followed by views

The screenshot shows a SQL Server Management Studio window with two tabs: 'SQLQuery2.sql - D...QF77LA\Admin (54) ' and 'SQLQuery1.sql - D...QF77LA\Admin (52)*'. The query in the active tab is:

```
EXEC sys.sp_tables |
```

The results grid displays the following data:

	TABLE_QUALIFIER	TABLE_OWNER	TABLE_NAME	TABLE_TYPE	REMARKS
1	master	dbo	sysmatrixageforget	SYSTEM TABLE	NULL
2	master	dbo	Customers	TABLE	NULL
3	master	dbo	MSreplication_options	TABLE	NULL
4	master	dbo	Persons	TABLE	NULL
5	master	dbo	spt_fallback_db	TABLE	NULL
6	master	dbo	spt_fallback_dev	TABLE	NULL
7	master	dbo	spt_fallback_usg	TABLE	NULL
8	master	dbo	spt_monitor	TABLE	NULL

At the bottom of the results pane, a green checkmark icon and the text 'Query executed successfully.' are visible.

Stored Procedure Basics

System Stored Procedures

- ✓ The given query will provide various information, from create date to file stream and other important information

The screenshot shows a SQL Server Management Studio window with two tabs: 'SQLQuery2.sql - D...QF77LA\Admin (54)' and 'SQLQuery1.sql - D...QF77LA\Admin (52)* X'. The 'Results' tab is selected, displaying the output of the following query:

```
SELECT * FROM sys.tables
```

The results grid shows the following data:

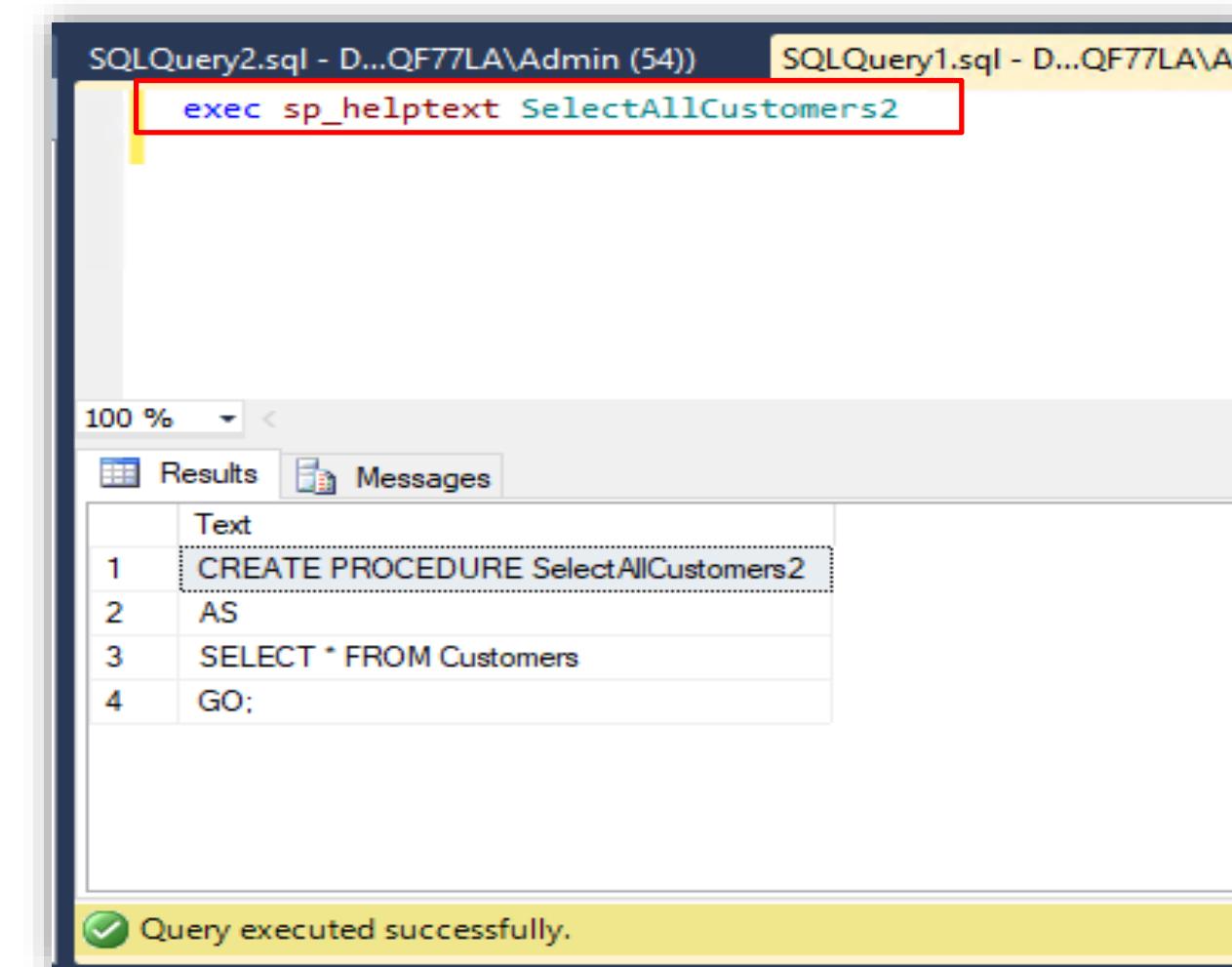
	name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date	is_ms_shipped	is_published
1	Persons	23671132	NULL	1	0	U	USER_TABLE	2022-03-09 05:08:17.843	2022-03-09 05:08:17.843	0	0
2	spt_fallback_db	117575457	NULL	1	0	U	USER_TABLE	2003-04-08 09:18:01.557	2012-02-10 21:14:52.657	1	0
3	spt_fallback_dev	133575514	NULL	1	0	U	USER_TABLE	2003-04-08 09:18:02.870	2012-02-10 21:14:52.667	1	0
4	Customers	135671531	NULL	1	0	U	USER_TABLE	2022-03-09 06:27:50.123	2022-03-09 06:27:50.123	0	0
5	spt_fallback_usg	149575571	NULL	1	0	U	USER_TABLE	2003-04-08 09:18:04.180	2012-02-10 21:14:52.673	1	0
6	spt_monitor	1483152329	NULL	1	0	U	USER_TABLE	2012-02-10 21:02:08.440	2012-02-10 21:14:52.683	1	0
7	MSreplication_options	1787153412	NULL	1	0	U	USER_TABLE	2012-02-10 21:14:19.113	2012-02-10 21:14:52.690	1	0

At the bottom of the results grid, a message states: 'Query executed successfully.' and shows the execution details: DESKTOP-FQF77LA (11.0 RTM) | DESKTOP-FQF77LA\Admin ... | master | 00:00:00 | 7 rows.

Stored Procedure Basics

System Stored Procedures

- ✓ **sp_helpText <Object Name>**
- ✓ The sp_helpText displays the definition that is used to generate an object in the multiple rows. The given example displays the definition of the procedure SelectAllCustomers2 in the master database



The screenshot shows a SQL Server Management Studio window with two tabs: 'SQLQuery2.sql - D...QF77LA\Admin (54)' and 'SQLQuery1.sql - D...QF77LA\Admin (54)'. The 'Results' tab is selected. In the results pane, there is a table with one column labeled 'Text'. The table contains four rows of T-SQL code:

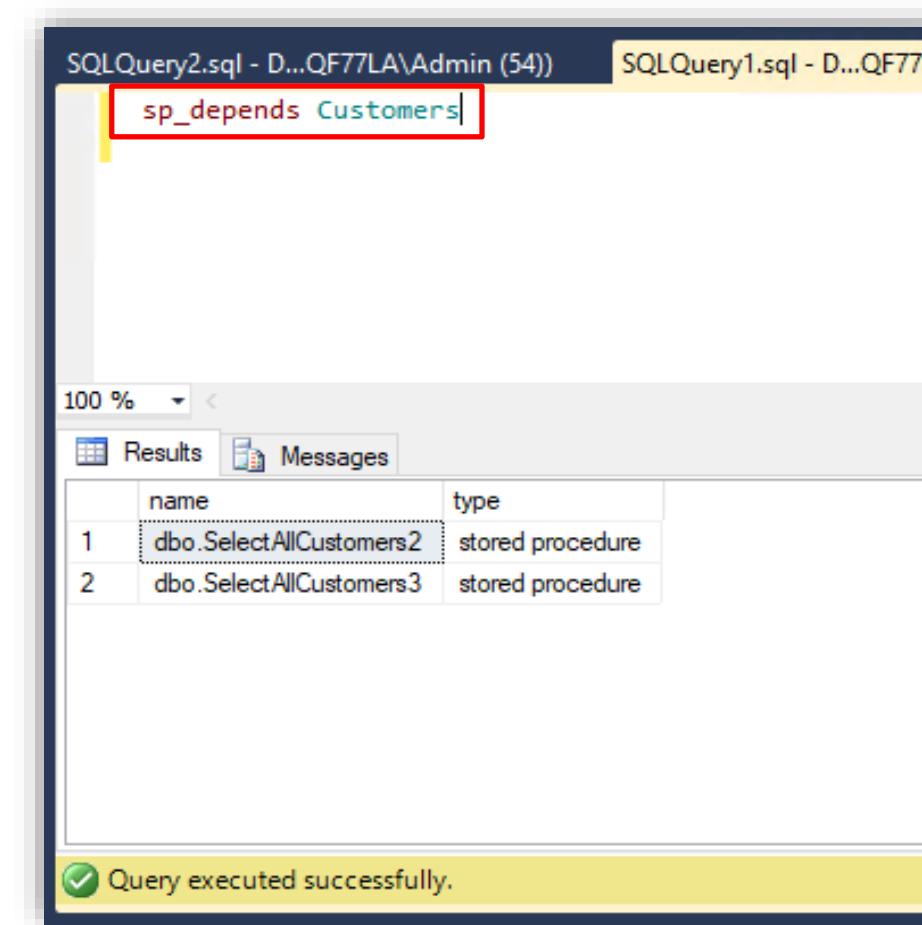
Text
1 CREATE PROCEDURE SelectAllCustomers2
2 AS
3 SELECT * FROM Customers
4 GO;

A red box highlights the command 'exec sp_helpText SelectAllCustomers2' in the top query editor. A green checkmark icon at the bottom of the results pane indicates that the query was executed successfully.

Stored Procedure Basics

System Stored Procedures

- ✓ **Sp_depends <Object Name>**
- ✓ The sp_helptext is used to get the dependent object details. If someone is working in a large database, then before changing a table or a Stored Procedure, one should know about the tables, Stored Procedures, and functions dependencies by using the sp_depends procedure



The screenshot shows a SQL Server Management Studio window with two tabs: 'SQLQuery2.sql - D...QF77LA\Admin (54)' and 'SQLQuery1.sql - D...QF77LA'. The 'Results' tab is selected, displaying the output of the 'sp_depends' command. The command 'sp_depends Customers' is highlighted with a red box. The results table has columns 'name' and 'type'. It shows two entries: 'dbo.SelectAllCustomers2' and 'dbo.SelectAllCustomers3', both categorized as 'stored procedure'. A message at the bottom of the results pane says 'Query executed successfully.'

	name	type
1	dbo.SelectAllCustomers2	stored procedure
2	dbo.SelectAllCustomers3	stored procedure

Module 3



Variables

Description

1	Declaring Variables
2	SET versus SELECT
3	Global Variables
4	Tricks with Variables

Variables

Declaring Variables

- ✓ During the code execution, a variable permits a programmer to store the data temporarily
- ✓ **Syntax for declaring a variable:**
 - ✓ A user can use DECLARE to create a variable for use. The syntax:

```
DECLARE @VariableName AS datatype
```

Variables

Declaring Variables

- ✓ Example:

The screenshot shows two windows from SQL Server Management Studio. On the left, the 'QLQuery1.sql' query editor window displays the following T-SQL code:

```
DECLARE @demo AS VARCHAR(100)='Training'  
SELECT @demo
```

A large blue arrow points from the query editor to the results window on the right. The results window, titled 'Results', shows the output of the query:

(No column name)
1 Training

At the bottom of the results window, a yellow status bar displays the message 'Query executed successfully.'

Variables

SET versus SELECT

- ✓ The following are the difference between Set and Select:

SET	SELECT
ANSI standard for variable assignment	Non-ANSI standard when assigning variables
A user can only assign one variable at a time SET @Index = 1 SET @LoopCount = 10 SET @InitialValue = 5	A user can assign values to more than one variable at a time SELECT @Index = 1, @LoopCount = 10, @InitialValue = 5

Variables

SET versus SELECT

SET	SELECT
<p>When assigning from a query and the query returns no result, SET will assign a NULL value to the variable</p> <pre>DECLARE @CustomerID NCHAR(5) SET @CustomerID = 'XYZ' SET @CustomerID = (SELECT [CustomerID] FROM [dbo].[Customers] WHERE [CustomerID] = 'ABC') SELECT @CustomerID -- Returns NULL</pre>	<p>When assigning from a query and the query returns no result, SELECT will not make the assignment and therefore not change the value of the variable</p> <pre>DECLARE @CustomerID NCHAR(5) SET @CustomerID = 'XYZ' SELECT @CustomerID = [CustomerID] FROM [dbo].[Customers] WHERE [CustomerID] = 'ABC' SELECT @CustomerID-- Returns XYZ</pre>

Variables

SET versus SELECT

SET	SELECT
<p>When assigning from a query that returns more than one value, SET will fail with an error</p> <pre>SET = (SELECT [CustomerID] FROM [dbo].[Customers])</pre> <p>Msg 512, Level 16, State 1, Line 3 Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <= , >, >= or when the subquery is used as an expression.</p>	<p>When assigning from a query that returns more than one value, SELECT will assign the last value returned by the query and hide the fact that the query returned more than one row</p> <pre>SELECT @CustomerID = [CustomerID] FROM [dbo].[Customers] -- No error generated</pre>

Variables

Global Variables

- ✓ Global variables are a particular type of variable
- ✓ The values in the global variables set by the database server
- ✓ Global variables refer to the scalar functions because they return one value
- ✓ Global variables always begin with an @@ prefix
- ✓ Global variables are system-defined functions; we cannot declare them

Variables

Global Variables

- ✓ The following is a list of global variables:

Variable name	Meaning
@@textsize	Current value of the SET TEXTSIZE option, which specifies the maximum length, in bytes, of text or image data to be returned with a select statement. The default setting is 32765, which is the largest byte string that can be returned using READTEXT
@@total_write	o (It is provided for compatibility with Transact-SQL.)
@@trancount	Nesting level of transactions
@@servername	Name of the current database server

Variables

Global Variables

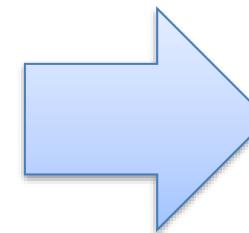
Variable name	Meaning
@@identity	Last value inserted into any IDENTITY or DEFAULT AUTOINCREMENT column by an INSERT or SELECT INTO statement
@@error	<ul style="list-style-type: none">• A Transact-SQL error code that checks the success or failure of the most recently executed statement. If the previous transaction succeeded, 0 is returned. If the previous transaction was unsuccessful, the last error number generated by the system is returned. To view descriptions of the values returned by @@error• A statement such as if @@error != 0 return causes an exit if an error occurs. Every statement resets @@error, comprising PRINT statements or IF tests, so the status check must instantly follow the statement whose success we want verified

Variables

Trick with Variables

- ✓ Reading Column Values into Variables
- ✓ Provided that a SELECT statement returns a single row of data, you can read the values into variables
- ✓ Example:

```
SQLQuery1.sql - D...QF77LA\Admin (52)* X
Declare @Name varchar(30)
Begin
Select @Name=CustomerName from CustomersOrders.dbo.Customer where Customerid=5604
Set @Name= 'Customer Name:' +@Name
Print @Name
End
```



Messages

Customer Name:Rajan Paul

100 % < >

Query executed successfully.

Module 4

Parameters and Return
Values

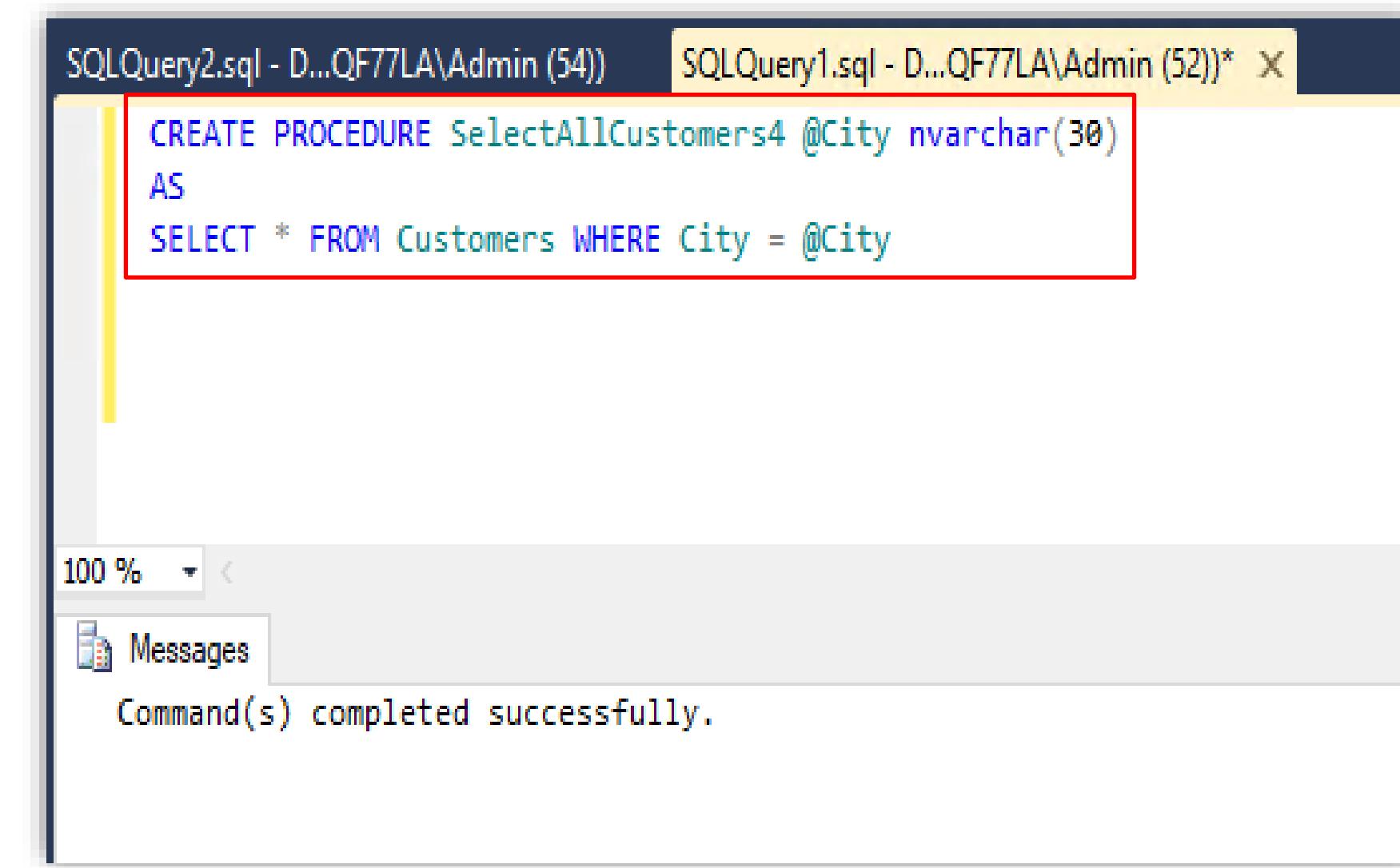
Description

1	Passing Parameters
2	Default Values and WHERE Clauses
3	Output Parameters
4	Using Return

Parameters and Return Values

Passing Parameters

- ✓ Stored Procedure With One Parameter
- ✓ The below SQL statement creates a stored procedure that selects Customers from a particular City from the "Customers" table:



The screenshot shows a SQL Server Management Studio window with two tabs: 'SQLQuery2.sql - D...QF77LA\Admin (54)' and 'SQLQuery1.sql - D...QF77LA\Admin (52)*'. The 'SQLQuery1.sql' tab is active and contains the following T-SQL code:

```
CREATE PROCEDURE SelectAllCustomers4 @City nvarchar(30)
AS
SELECT * FROM Customers WHERE City = @City
```

A red rectangular box highlights the code in the 'SQLQuery1.sql' tab. Below the tabs, there's a zoom control set to '100 %' and a 'Messages' pane. The 'Messages' pane displays the message: 'Command(s) completed successfully.'

Parameters and Return Values

Passing Parameters

- ✓ Execute the stored procedure:

The screenshot shows the SQL Server Management Studio interface. There are two query panes open: 'SQLQuery2.sql' and 'SQLQuery1.sql'. The 'SQLQuery1.sql' pane is active and contains the following SQL code:

```
EXEC SelectAllCustomers4 @City = 'Stavanger';
```

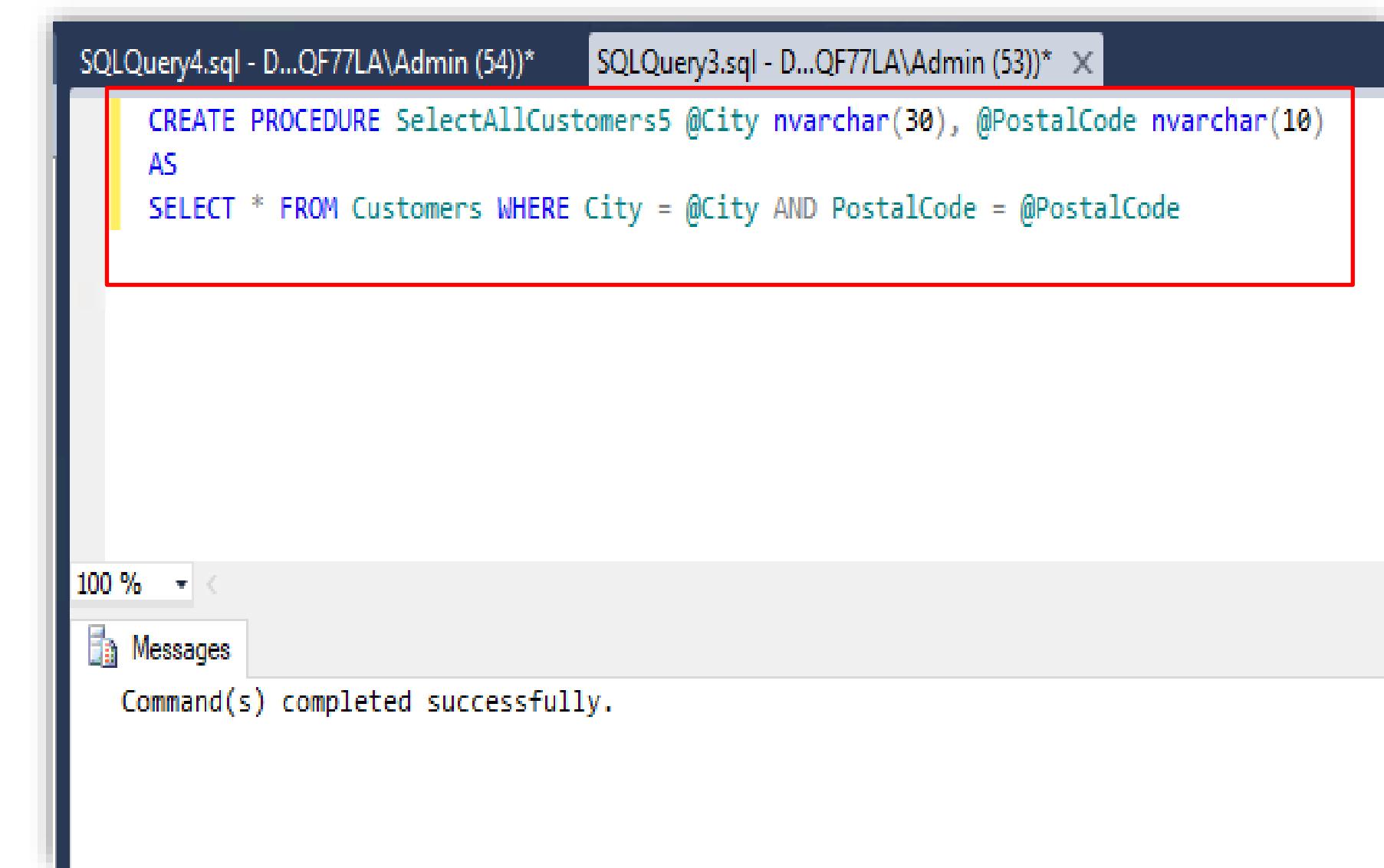
The results pane below shows a single row of data from the query:

	CustomerName	ContactName	Address	City	PostalCode	Country
1	Cardinal	Tom B. Erichsen	Skagen 21	Stavanger	4006	Norway

Parameters and Return Values

Passing Parameters

- ✓ It's simple to set up multiple parameters. Simply separate each parameter and data type with a comma, as seen below:



The screenshot shows a SQL Server Management Studio window with two tabs: 'SQLQuery4.sql - D...QF77LA\Admin (54)*' and 'SQLQuery3.sql - D...QF77LA\Admin (53))* X'. The code in the active tab is:

```
CREATE PROCEDURE SelectAllCustomers5 @City nvarchar(30), @PostalCode nvarchar(10)
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode
```

The entire code block is highlighted with a red rectangle. In the bottom pane, under the 'Messages' tab, it says 'Command(s) completed successfully.'

Parameters and Return Values

Passing Parameters

- ✓ Execute the stored procedure:

The screenshot shows a Windows application window titled "SQLQuery4.sql - D...QF77LA\Admin (54)*" and "SQLQuery3.sql - D...QF77LA\Admin (53)*". The main area contains the following SQL code:

```
EXEC SelectAllCustomers5 @City = 'Stavanger', @PostalCode = '4006';
```

Below the code, there is a "Results" tab showing a table with one row of data:

	CustomerName	ContactName	Address	City	PostalCode	Country
1	Cardinal	Tom B. Erichsen	Skagen 21	Stavanger	4006	Norway

Parameters and Return Values

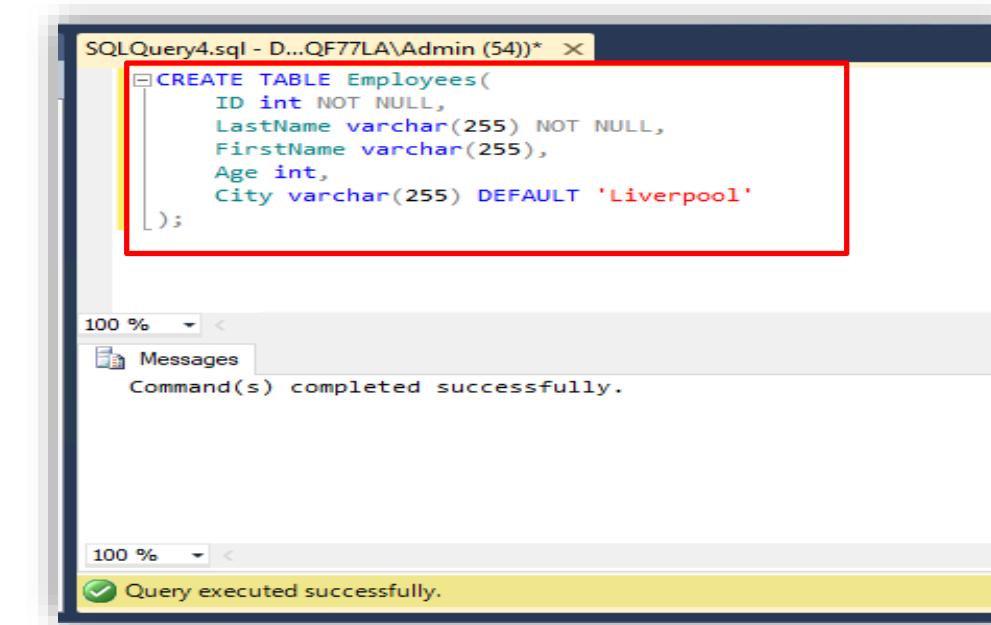
Default Values and WHERE Clauses

✓ SQL DEFAULT Constraint

- ✓ The DEFAULT constraint is used to set a column's default value. If no other value is specified, the default value will be added to all new records

✓ SQL DEFAULT on CREATE TABLE

- ✓ The below SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:



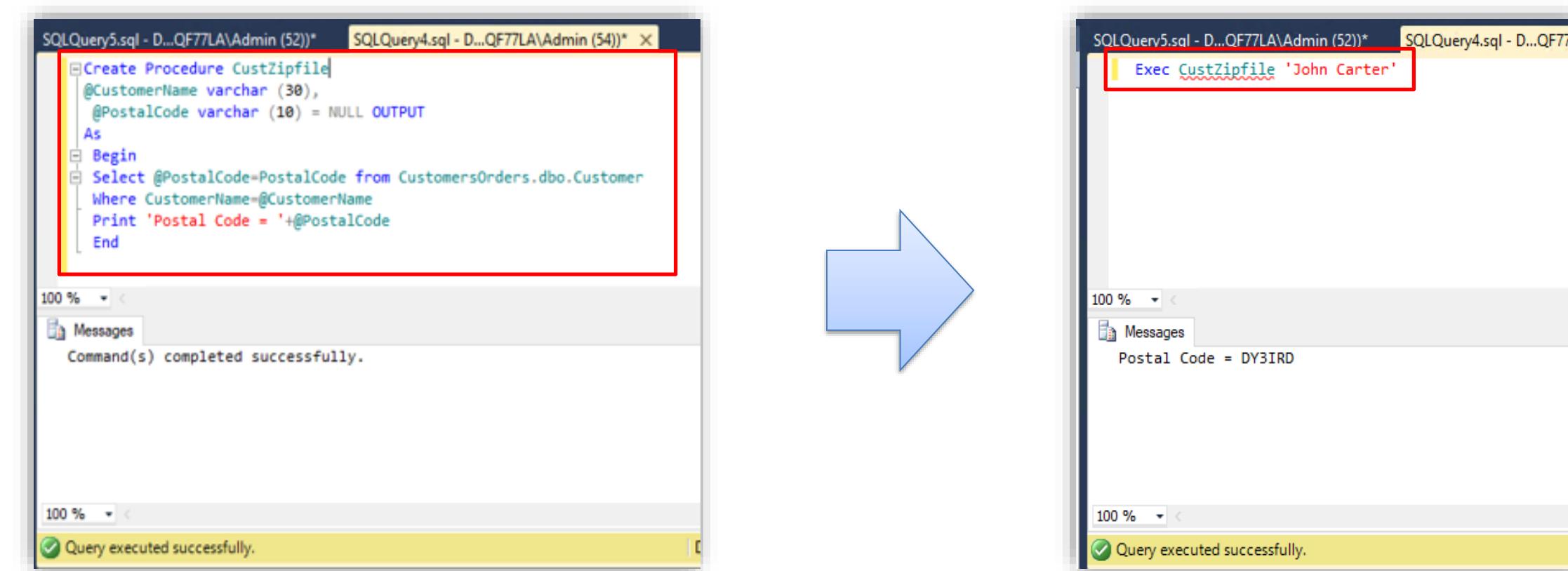
The screenshot shows a SQL query window titled "SQLQuery4.sql - D...QF77LA\Admin (54)*". The query is:`CREATE TABLE Employees(
 ID int NOT NULL,
 LastName varchar(255) NOT NULL,
 FirstName varchar(255),
 Age int,
 City varchar(255) DEFAULT 'Liverpool'
);`

The code is highlighted with a red rectangle. Below the query, the "Messages" pane shows "Command(s) completed successfully." At the bottom, a green bar indicates "Query executed successfully."

Parameters and Return Values

Output Parameters

- ✓ Output parameters enable us to return output values from the stored procedure. A modification to the output parameter within the stored procedure is reflected in the variable from the calling batch that was assigned to the output parameter. The output parameter is specified by the keyword `output`.
- ✓ To create a stored procedure with the `OUTPUT` parameter:



The screenshot shows two windows from SQL Server Management Studio. The left window displays the T-SQL code for creating a stored procedure named `CustZipfile`. The code includes an `OUTPUT` parameter `@PostalCode` and a `PRINT` statement that outputs the value of `@PostalCode`. The right window shows the result of executing the stored procedure with the parameter `'John Carter'`, which returns the postal code `DY3IRD`.

```
SQLQuery5.sql - D...QF77LA\Admin (52)* SQLQuery4.sql - D...QF77LA\Admin (54)*
Create Procedure CustZipfile
@CustomerName varchar (30),
@PostalCode varchar (10) = NULL OUTPUT
As
Begin
Select @PostalCode=PostalCode from CustomersOrders.dbo.Customer
Where CustomerName=@CustomerName
Print 'Postal Code = '+@PostalCode
End

100 % < Messages
Command(s) completed successfully.

100 % < Query executed successfully.

SQLQuery5.sql - D...QF77LA\Admin (52)* SQLQuery4.sql - D...QF77LA\Admin (54)*
Exec CustZipfile 'John Carter'

100 % < Messages
Postal Code = DY3IRD

100 % < Query executed successfully.
```

Parameters and Return Values

Using Return

- ✓ RETURN can be used at any point to exit from a procedure, batch, or statement block Statements that follow RETURN are not executed
- ✓ Syntax:

```
RETURN [ integer_expression ]
```

The screenshot shows the SQL Server Management Studio interface. In the center, there is a code editor window containing the following T-SQL script:

```
Create Procedure FindCustomerss  
@CustomerId numeric(4,0),  
@Country varchar(30)=NULL  
as  
Begin  
Select @Country = Country from CustomersOrders.dbo.Customer where CustomerId=@CustomerId  
If @Country='Australia'  
    Return 0  
Else  
    Return 1  
End
```

The code editor has a red rectangular box highlighting the entire script. Below the code editor, the 'Messages' pane displays the message: "Command(s) completed successfully.". At the bottom of the screen, a status bar shows the message: "Query executed successfully." and the computer name "DESKTOP-FQF77LA (11.0 RT)".

Parameters and Return Values

Using Return

- ✓ Using the Return Value:

The screenshot shows a SQL Server Management Studio window with two tabs: 'SQLQuery6.sql' and 'SQLQuery5.sql'. The code in 'SQLQuery6.sql' is highlighted with a red box and contains the following T-SQL:

```
Declare  
    @retValue INT  
Begin  
    EXEC @retValue=FindCustomerss '5605'  
If @retValue=0  
Print ' Not For Sale'  
else  
Print 'Hope You Like The Product'  
End
```

The 'Messages' pane below shows the output:

```
Hope You Like The Product
```

A yellow bar at the bottom indicates the query was executed successfully.

Parameters and Return Values

Using Return

- ✓ Using the Return Value:

The screenshot shows a SQL Server Management Studio window with two tabs: 'SQLQuery6.sql' and 'SQLQuery5.sql'. The code in 'SQLQuery6.sql' is highlighted with a red box and contains the following T-SQL:

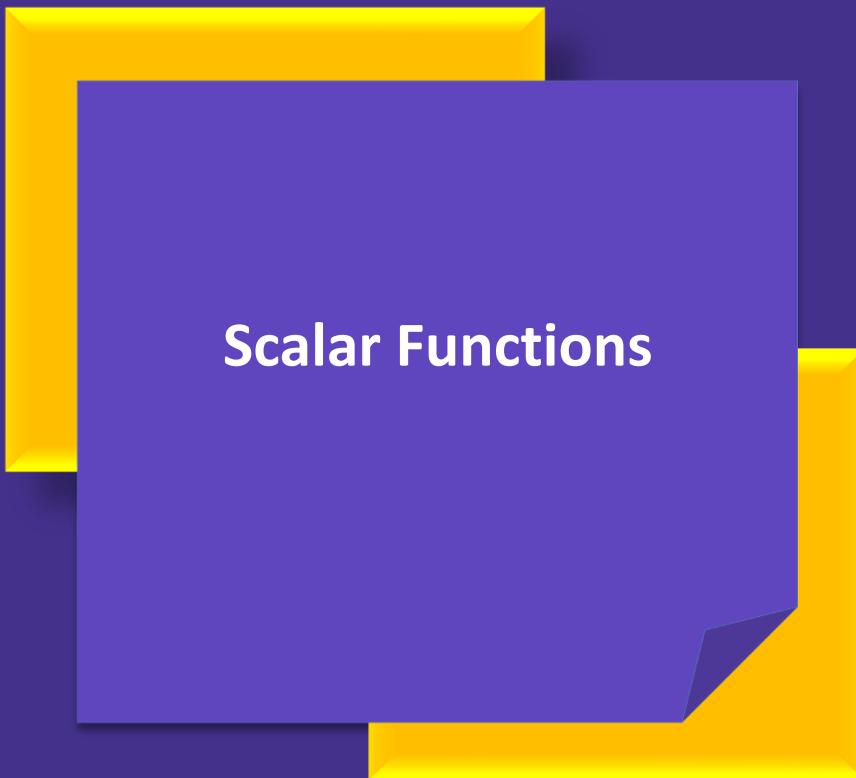
```
Declare  
    @retValue INT  
Begin  
    EXEC @retValue=FindCustomerss '5605'  
If @retValue=0  
Print ' Not For Sale'  
else  
Print 'Hope You Like The Product'  
End
```

The 'Messages' pane below shows the output:

```
Hope You Like The Product
```

A yellow bar at the bottom indicates the query was executed successfully.

Module 5



Scalar Functions

Description

1

Scalar Functions

2

Advantages

3

Disadvantages

Scalar Functions

Introduction

- ✓ SQL scalar functions return a single value based on the input value
- ✓ The following are some scalar functions:

UCASE()	Converts a field to upper case
LCASE()	Converts a field to lower case
MID()	Extract characters from a text field
LEN()	Returns the length of a text field
ROUND()	Rounds a numeric field to the number of decimals specified
NOW()	Returns the current system date and time
FORMAT()	Formats how a field is to be displayed

Scalar Functions

UCASE() function

- ✓ This function is used to convert the value of string columns to uppercase characters
- ✓ Syntax:

```
SELECT UPPER(column_name) from table-name;
```

Scalar Functions

UCASE()

- ✓ Using UCASE() function
- ✓ Syntax:

The image shows a screenshot of SQL Server Management Studio. On the left, a query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*" contains the following SQL code:

```
SELECT UPPER(CustomerName) As Uppercase FROM CustomersOrders.dbo.Customer;
```

A large blue arrow points from the query window to the results window on the right. The results window has tabs for "Results" and "Messages". The "Results" tab displays a table with one column labeled "Uppercase". The data rows are:

Uppercase
JOHN CARTER
TRIZA
MICHAEL
RAJAN PAUL

At the bottom of the results window, a yellow bar indicates the query was executed successfully.

Scalar Functions

LCASE() function

- ✓ This function is used to convert the value of string columns to lowercase characters
- ✓ Syntax:

```
SELECT LOWER(column_name) FROM table-name;
```

Scalar Functions

LCASE() function

- ✓ Using LCASE() function
- ✓ Syntax:

The screenshot shows the SQL Server Management Studio interface. On the left, a query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)* X" contains the following SQL code:

```
SELECT LOWER(CustomerName) As Lowercase FROM CustomersOrders.dbo.Customer;
```

A large blue arrow points from the query window to the results window on the right. The results window has tabs for "Results" and "Messages". The "Results" tab displays a table with four rows, each containing a number and a lowercase string:

	Lowercase
1	john carter
2	triza
3	michael
4	rajan paul

At the bottom of the results window, a yellow status bar displays the message "Query executed successfully."

Scalar Functions

MID() function

- ✓ MID function is used to extract substrings from column values of string type in a table
- ✓ Syntax:

```
SELECT MID(column_name, start, length) from table-name;
```

Scalar Functions

MID() function

- ✓ Using MID() function
- ✓ Syntax:

The screenshot shows a SQL query window and a results grid. The query is:

```
SQLQuery1.sql - D...QF77LA\Admin (52)* X
SELECT Substring(CustomerName,2,2) As Substring FROM CustomersOrders.dbo.Customer;
```

A large blue arrow points from the query window to the results grid. The results grid has a header row "Substring" and four data rows:

Substring
1 oh
2 ri
3 ic
4 aj

At the bottom of the results grid, there is a message: "Query executed successfully."

Scalar Functions

ROUND() function

- ✓ This function is used to round a numeric field to the nearest integer
- ✓ It is used on decimal point values
- ✓ Syntax:

```
SELECT ROUND(column_name, decimals) from table-name;
```

Scalar Functions

ROUND() function

- ✓ Using ROUND() function
- ✓ Syntax:

The screenshot shows a SQL query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)* X". The query is:

```
Select round(Product_Amount,2) As [Rounded Amount] from CustomersOrders.dbo.Order_Details;
```

An arrow points from the query window to the results window. The results window has tabs for "Results" and "Messages". The "Results" tab displays a table with four rows:

	Rounded Amount
1	10000
2	10702
3	4568
4	3567

The "Messages" tab at the bottom shows a green checkmark and the text "Query executed successfully."

Scalar Functions

LEN() function

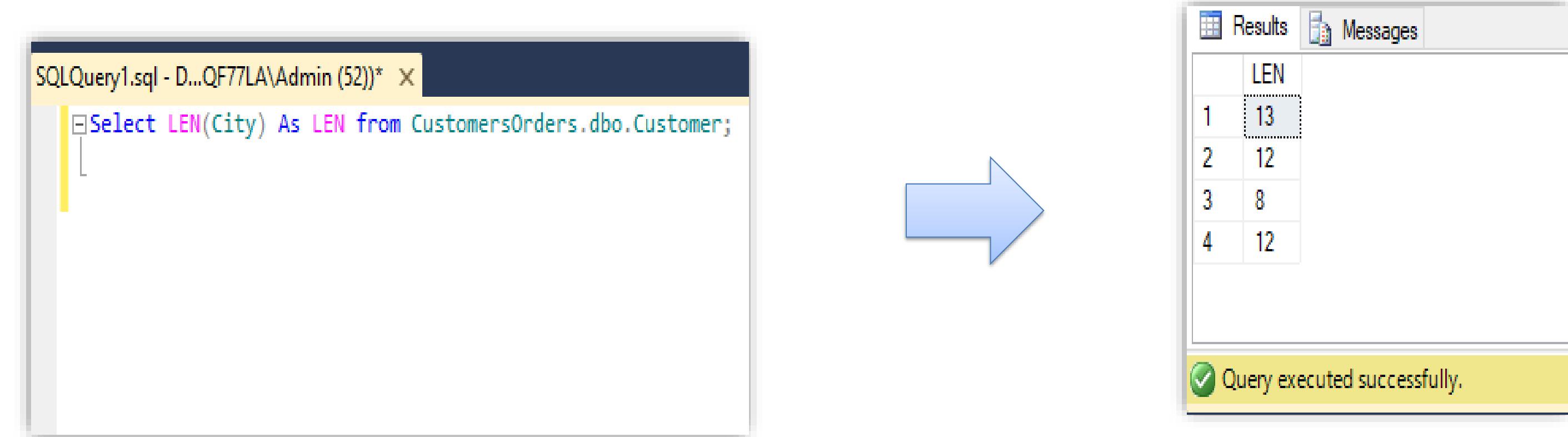
- ✓ This function returns the length of the value in the text field
- ✓ Syntax:

```
SELECT LEN(column_name) FROM table_name;
```

Scalar Functions

LEN() function

- ✓ Using LEN() function
- ✓ Syntax:



The image shows a screenshot of SQL Server Management Studio. On the left, a query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*" contains the following T-SQL code:

```
Select LEN(City) As LEN from CustomersOrders.dbo.Customer;
```

A large blue arrow points from the bottom right of the query window towards the results window. To the right of the arrow is a results window titled "Results". It displays a table with one column labeled "LEN" and four rows of data:

LEN
13
12
8
12

At the bottom of the results window, a yellow status bar indicates: "Query executed successfully."

Scalar Functions

GETDATE() function

- ✓ This function returns the current system date and time
- ✓ Syntax:

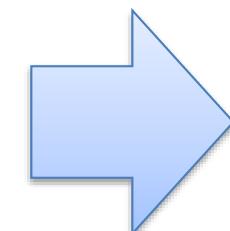
```
SELECT GETDATE() FROM table_name;
```

Scalar Functions

GETDATE() function

- ✓ Using GETDATE() function
- ✓ Syntax:

```
SQLQuery1.sql - D...QF77LA\Admin (52)* X
Select GETDATE() from CustomersOrders.dbo.Customer;
```



	(No column name)
1	2022-03-09 11:45:57.090
2	2022-03-09 11:45:57.090
3	2022-03-09 11:45:57.090
4	2022-03-09 11:45:57.090

Query executed successfully.

Scalar Functions

FORMAT() function

- ✓ This function is used to format how a field is to be displayed
- ✓ Syntax:

```
SELECT FORMAT(column_name,format) FROM table_name;
```

Scalar Functions

FORMAT() function

- ✓ Using FORMAT() function
- ✓ Syntax:

The screenshot shows the SQL Server Management Studio interface. On the left, a query window titled 'SQLQuery1.sql - D...QF77LA\Admin (52)* X' contains the following SQL code:

```
Select FORMAT(OrderDate, 'd', 'en-gb') As Date from CustomersOrders.dbo.Orders;
```

A large blue arrow points from the query window to the results pane on the right. The results pane has tabs for 'Results' and 'Messages'. The 'Results' tab displays a table with one column named 'Date' containing four rows of data:

Date
12/12/2017
10/01/2018
15/12/2017
18/01/2018

Below the table, a yellow status bar at the bottom of the results pane displays the message 'Query executed successfully.'

Scalar Functions

Advantages of Scalar Functions

- ✓ The advantages of SQL scalar functions can be classified into four sections:



Date/time Functions



Math Functions



String Functions



Miscellaneous Functions

Scalar Functions

Advantages of Scalar Functions

✓ Date/Time Functions

- ✓ The advantages of date/time SQL scalar functions can be used to determine or process date/time information

CURDATE	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURTIME	DAYNAME
DAYOFMONTH	DAYOFWEEK	DAYOFYEAR
EXTRACT	HOUR	MINUTE
MONTH	MONTHNAME	NOW
QUARTER	SECOND	TIMESTAMPADD
TIMESTAMPDIFF	WEEK	YEAR

Scalar Functions

Advantages of Scalar Functions

✓ Math Functions

- ✓ The advantages of math scalar functions provide a rich collection of arithmetic and trigonometric functions

ABS	ACOS	ASIN	ATAN	ATAN2
CEILING	COS	COT	DEGREES	EXP
FLOOR	LOG	LOG10	MOD	PI
POWER	RADIANS	RAND	ROUND	SIGN
SIN	SQRT	TAN	TRUNCATE	

Scalar Functions

Advantages of Scalar Functions

✓ String Functions

- ✓ The advantages of string functions are used to process and convert string expressions

ASCII	BIT_LENGTH	CHAR
CHAR_LENGTH	CHARACTER_LENGTH	COLLATE
CONCAT	INSERT	LCASE
LEFT	LENGTH	LOCATE
LOWER	LTRIM	OCTET_LENGTH
POSITION	REPEAT	REPLACE
RIGHT	RTRIM	SPACE
SUBSTRING	TRIM	UCASE

Scalar Functions

Advantages of Scalar Functions

- ✓ **Miscellaneous Functions**
 - ✓ The advantages of miscellaneous SQL scalar functions are as follows:

APPLICATIONID	CAST	COALESCE
CONTAINS	CONVERT	DATABASE
DIFFERENCE	IFNULL	IIF
ISNULL	LASTAUTOINC	NEWIDSTRING
SCORE	SCOREDISTINCT	SOUNDEX

Scalar Functions

Disadvantages of Scalar Functions

- ✓ Scalar functions are not allowed to return Binary Large Object (BLOB) data, e.g. text, timestamps, ntext, and image data-type variables

- ✓ Inside the scalar functions, the non-deterministic functions, e.g. newid () and rand () are not permitted

- ✓ Scalar functions are not allowed to use Try Catch block and Raiserror for error handling

- ✓ They are also not allowed to return table variables or cursor data types

Module 6



Testing Conditions

Description

1

IF/ELSE Conditions

2

Using Case where Possible

Testing Conditions

IF/ELSE Conditions

- ✓ In the IF/ELSE statement, the IF condition is used to execute when the condition is TRUE, and ELSE statement is used to execute when the condition is False

- ✓ Syntax:
 - ✓ The syntax of IF/ELSE statement in SQL Server is:

```
IF condition
  {...statements to execute when condition is TRUE...}

[ ELSE
  {...statements to execute when condition is FALSE...} ]
```

Testing Conditions

IF/ELSE Conditions

- ✓ Optional. ELSE condition is used to execute when the IF condition evaluated to FALSE
- ✓ Example:
 - ✓ IF...ELSE Statement

The screenshot shows two windows from SQL Server Management Studio. On the left, the 'SQLQuery1.sql' window displays a T-SQL script. The script declares variables @Units, @Amt, and @Amount, then performs a select operation. It calculates @Amount as @Units * @Amt. An IF statement checks if @Amount is greater than 10000. If true, it prints 'Avail 5%Discount'; otherwise, it prints 'No Discount'. The 'Messages' window on the right shows the output: 'Avail 5%Discount' and a green message at the bottom stating 'Query executed successfully.'

```
SQLQuery1.sql - D...QF77LA\Admin (52)* X
Declare @Units numeric (15,0), @Amt numeric(18,0), @Amount numeric (18,0)
Begin
    Select @Units=units_purchased, @Amt=Product_Amount from CustomersOrders.dbo.Order_Details where OrderId=1011
    Set @Amount = @Units * @Amt
    If @Amount > 10000
        Print 'Avail 5%Discount'
    Else
        Print 'No Discount'
    End
```

100 % < Messages
Avail 5%Discount

100 % < Query executed successfully.

Testing Conditions

IF/ELSE Conditions

- ✓ Example: No ELSE condition
 - ✓ Because the ELSE condition is optional, it is not required to include the ELSE condition in the IF/ELSE statement

The screenshot shows the SQL Server Management Studio interface. The top window is titled "SQLQuery1.sql - D...QF77LA\Admin (52)* X". It contains the following T-SQL code:

```
Declare @Units numeric (15, 0), @Amt numeric (18, 0), @Amount numeric (18, 0)
Begin
    Select @Units=units_purchased, @Amt=Product_Amount from CustomersOrders.dbo.Order_Details where OrderId=1011
    Set @Amount = @Units * @Amt
    If @Amount > 40000
        Print 'Avail 5%Discount'
    End
```

Below this is a "Messages" window with the status bar showing "100 %". It displays the message "Command(s) completed successfully." A large blue arrow points from the script window down to the messages window.

The bottom window is also a "Messages" window with the status bar showing "100 %". It displays the message "Query executed successfully." with a green checkmark icon.

Testing Conditions

IF/ELSE Conditions

- ✓ Example: Nested IF...ELSE Statements
 - ✓ In the IF/ELSE statement, we cannot write an ELSE/IF condition, so we need to nest multiple IF/ELSE statements

The image shows a screenshot of the SQL Server Management Studio interface. On the left, a query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*" displays the following T-SQL code:

```
Declare @Units numeric(15,0), @Amt numeric(18,0), @Amount numeric (18,0)
Begin
    Select @Units=units_purchased, @Amt=Product_Amount from CustomersOrders.dbo.Order_Details where OrderId=1011
    Set @Amount = @Units * @Amt
    If @Amount > 15000
        Print 'Avail 5% Discount'
    Else
        Begin
            If @Amount > 10000 and @Amount < 15000
                Print 'Avail 2% Discount'
            Else
                Print 'No Discount'
        End
End
```

A large blue arrow points from the query window to the right, leading to a "Messages" window. The "Messages" window has a title bar "Messages" and a status bar at the bottom showing "100 %". It contains two entries:

- "Avail 5% Discount"
- "Query executed successfully."

Testing Conditions

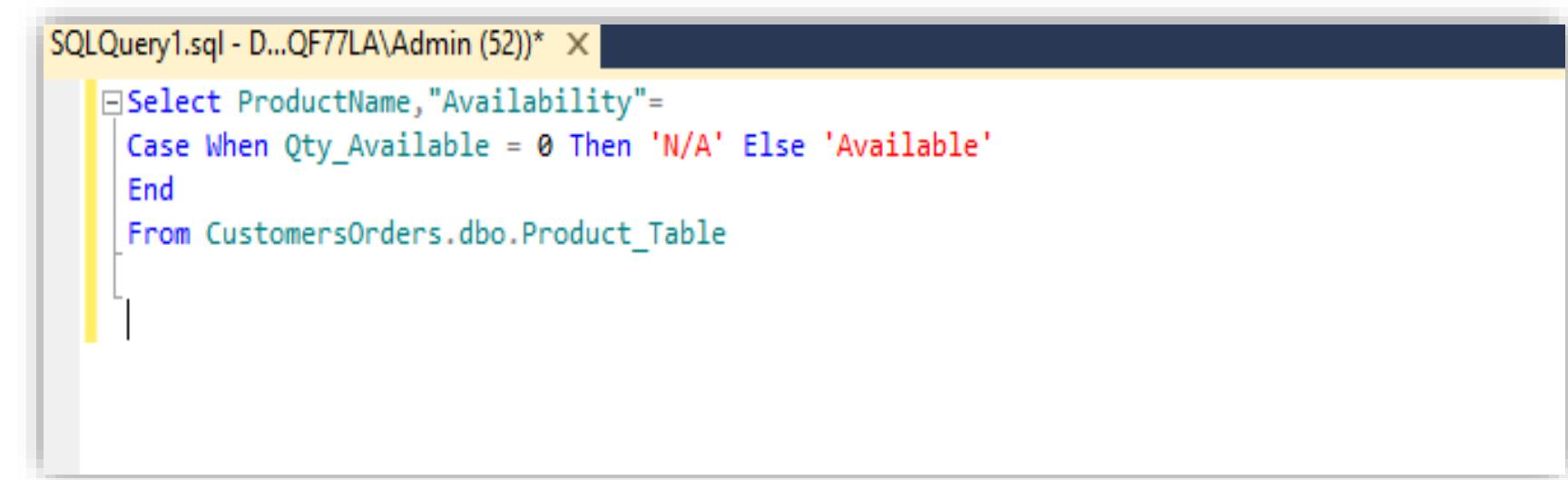
Using CASE Where Possible

- ✓ The CASE statement is a method of handling if/then logic in SQL
- ✓ It assesses a list of conditions and then returns one of multiple possible result expressions
- ✓ The case expression consists of two formats:
 - ✓ The simple CASE expression compares an expression to a set of simple expressions to determine the result
 - ✓ The searched CASE expression assesses a collection of Boolean expressions to determine the result

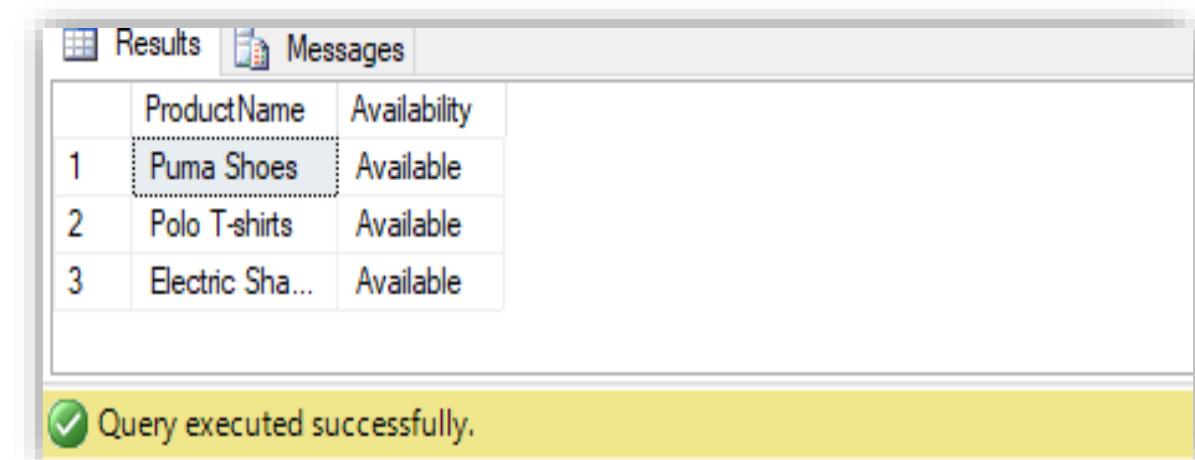
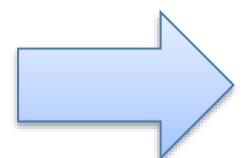
Testing Conditions

Using CASE Where Possible

- ✓ Here is a simple example of CASE:



```
SQLQuery1.sql - D...QF77LA\Admin (52)* X
Select ProductName, "Availability"=
Case When Qty_Available = 0 Then 'N/A' Else 'Available'
End
From CustomersOrders.dbo.Product_Table
```



The screenshot shows the execution results of the SQL query. The 'Results' tab is selected, displaying a table with three rows. The columns are 'ProductName' and 'Availability'. The data is as follows:

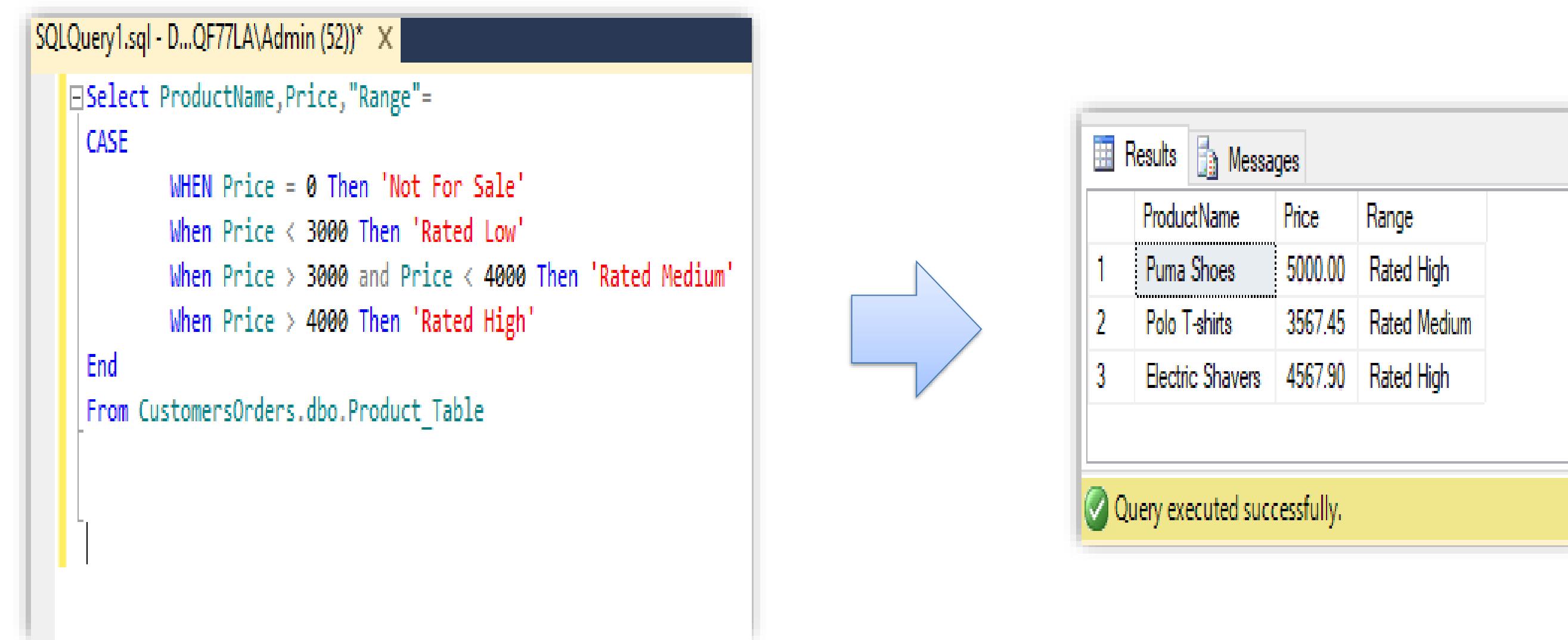
	ProductName	Availability
1	Puma Shoes	Available
2	Polo T-shirts	Available
3	Electric Sha...	Available

At the bottom of the results window, a green message bar states: 'Query executed successfully.'

Testing Conditions

Using CASE Where Possible

- ✓ Adding Multiple Conditions to a CASE statement
- ✓ A user can also specify multiple outcomes in a CASE statement by comprising as many WHEN/THEN statements as per the requirements:



```
SQLQuery1.sql - D...QF77LA\Admin (52)* X
Select ProductName, Price, "Range"=
CASE
    WHEN Price = 0 Then 'Not For Sale'
    When Price < 3000 Then 'Rated Low'
    When Price > 3000 and Price < 4000 Then 'Rated Medium'
    When Price > 4000 Then 'Rated High'
End
From CustomersOrders.dbo.Product_Table
```

Results Messages

	ProductName	Price	Range
1	Puma Shoes	5000.00	Rated High
2	Polo T-shirts	3567.45	Rated Medium
3	Electric Shavers	4567.90	Rated High

Query executed successfully.

Testing Conditions

Using CASE Where Possible

- ✓ Using CASE with aggregate functions
 - ✓ A user could use a CASE statement to assess the condition and produce null or non-null values according to the outcome:



The image shows a screenshot of a SQL Server Management Studio (SSMS) interface. On the left, the 'SQLQuery1.sql' window displays the following T-SQL code:

```
Select ProductId, Count(*) as TimesSold, AvailDiscount =  
Case  
When Count(*) > 3 Then 'Discount'  
Else  
'No Discount'  
End  
From CustomersOrders.dbo.Product_Table  
Group by ProductId
```

An orange arrow points from the 'SQLQuery1.sql' window to the 'Results' window on the right. The 'Results' window shows the output of the query:

	ProductId	TimesSold	AvailDiscount
1	9001	1	No Discount
2	9374	1	No Discount
3	9375	1	No Discount

At the bottom of the 'Results' window, a yellow bar indicates the query was executed successfully.

Module 7



Description

1	While Loop
2	Breaking out of a Loop
3	Basic Transactions
4	Using DELETE and UPDATE
5	Sys.Objects

Looping

While Loop

- ✓ The WHILE loop checks the condition first, then executes the SQL Statements block, repeating the process until the condition evaluates to false
- ✓ Syntax:

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

Looping

While Loop

- ✓ Using while loop
- ✓ Syntax:

The image shows a screenshot of SQL Server Management Studio. On the left, a code editor window displays a T-SQL script. The script uses a WHILE loop to iterate through a table named 'Product_Table' in the 'CustomersOrders.dbo' schema. It declares variables @PName and @I, initializes @I to 0, and then enters a loop where it selects the average price from the table, prints the product name if the average price is less than 15000, and updates the table by increasing the price by 300. The loop continues until the average price is no longer less than 15000. On the right, the 'Messages' window shows the output of the query. It lists three rows affected by the update statement, all labeled 'ProductNameElectric Shavers'. A large blue arrow points from the code editor to the messages window, indicating the flow of the execution.

```
SQLQuery6.sql - D...QF77LA\Admin (53)* X SQLQuery5.sql - D...QF77LA\Admin (52)* SQLQuery4.  
- Declare  
  - @PName varchar(30), @I numeric(3,0)=0  
- Begin  
  - While (Select Avg(Price) from CustomersOrders.dbo.Product_Table )< 15000  
    - Begin  
      Select @PName=ProductName from CustomersOrders.dbo.Product_Table  
      Print 'ProductName'+@PName  
      Update CustomersOrders.dbo.Product_Table  
      Set Price=Price+300  
    End  
  End
```

Messages

ProductNameElectric Shavers
(3 row(s) affected)
ProductNameElectric Shavers
(3 row(s) affected)
ProductNameElectric Shavers

100 %

Query executed successfully.

Looping

Breaking out of a Loop

- ✓ The BREAK statement is used to exit from a WHILE LOOP and execute the next statements after the loop's END statement
- ✓ Syntax:

```
WHILE Boolean_expression
BEGIN
    -- statements
    IF condition
        BREAK;
    -- other statements
END
```

Looping

Breaking out of a Loop

- ✓ Using Break Statement
- ✓ Syntax:

The screenshot shows two tabs in the SSMS interface: 'SQLQuery6.sql - D...QF77LA\Admin (53)*' and 'SQLQuery5.sql - D...QF77LA\Admin (52)*'. The code in the left tab is:

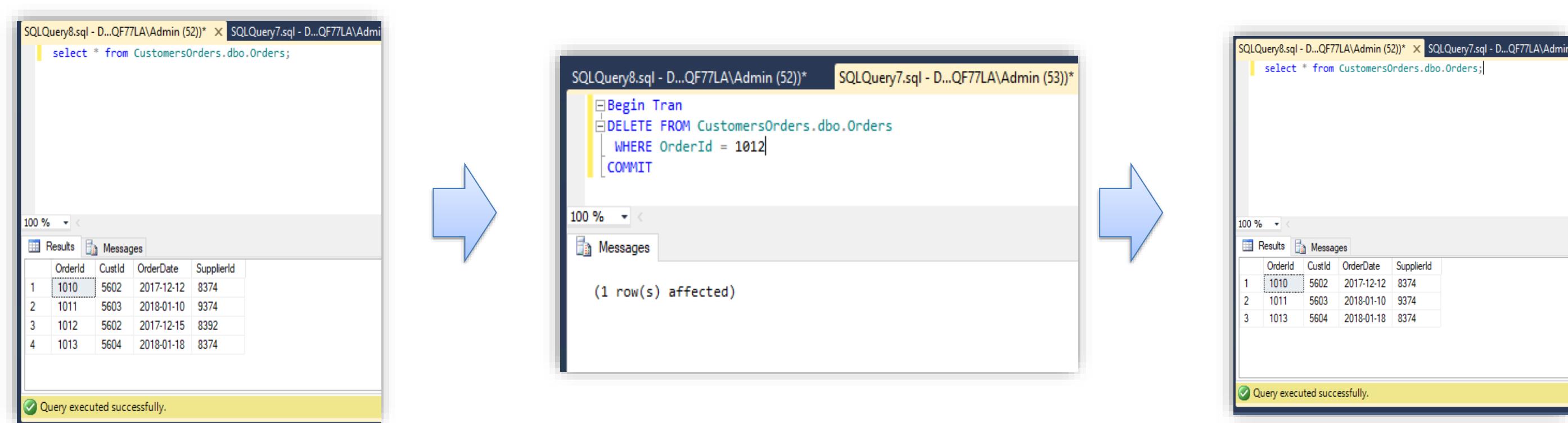
```
DECLARE @counter INT = 0;
WHILE @counter <= 6
BEGIN
    SET @counter = @counter + 1;
    IF @counter = 5
        BREAK;
    PRINT @counter;
END
```

A large blue arrow points from the code window to the 'Messages' window on the right. The 'Messages' window displays the output of the query, which consists of four lines of text: '1', '2', '3', and '4'. At the bottom of the 'Messages' window, there is a yellow status bar with the message 'Query executed successfully.'

Looping

Basic Transactions

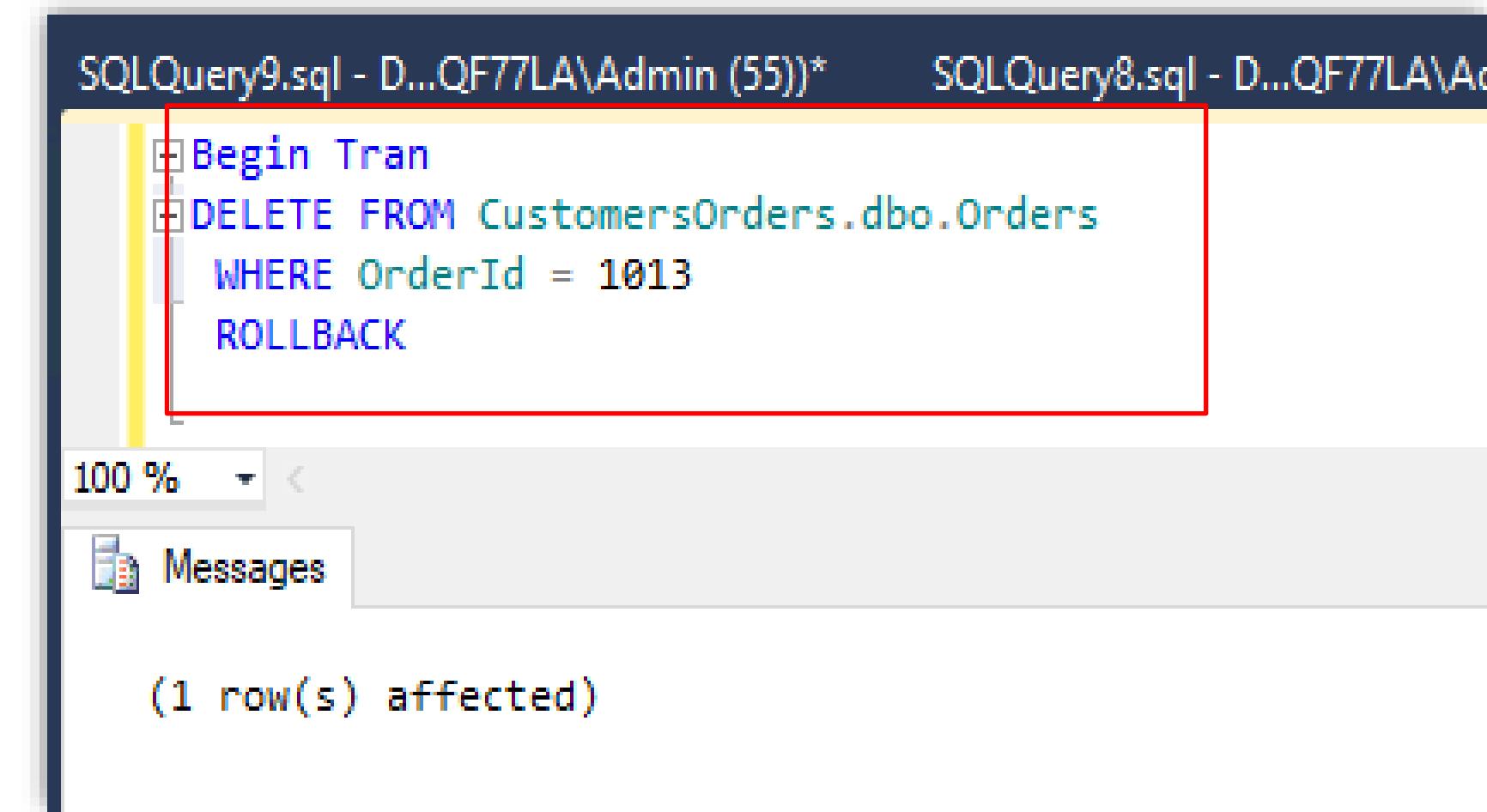
- ✓ The Transaction is the work unit performed opposite the database. Any transaction that reads or writes to a database. The transaction is the spread of one or various modifications to a database
- ✓ The transactions are controlled by the commands. These are the commands listed below:
 - ✓ **COMMIT:** It is used to save the changes



Looping

Basic Transactions

- ✓ **ROLLBACK:** ROLLBACK is used to undo the changes



The screenshot shows a SQL Server Management Studio window with two tabs: 'SQLQuery9.sql - D...QF77LA\Admin (55)*' and 'SQLQuery8.sql - D...QF77LA\Admin (55)*'. The code in the active tab is:

```
Begin Tran  
DELETE FROM CustomersOrders.dbo.Orders  
WHERE OrderId = 1013  
ROLLBACK
```

A red box highlights the entire transaction block. Below the code, the status bar shows '100 %' and the 'Messages' tab is selected. The message '(1 row(s) affected)' is displayed in the messages pane.

Looping

Basic Transactions

- ✓ **SAVEPOINTS:** Savepoints can be beneficial when it is required to roll back part of a SQL Server transaction. Usually, this is the case when there is a low possibility of error in part of the transaction, as well as the prior validation of the operation's accuracy is too expensive. Moreover, Savepoints can be used in stored procedures to be capable of successfully managing transactions in the nesting process.

The screenshot shows a SQL query window titled "SQLQuery1.sql - D...QF77LA\Admin (52) x". A red box highlights the command `begin transaction`. A red arrow points from this box to a red circle labeled "1." at the bottom right. Below the query window, the status bar says "Query executed successfully.".

The screenshot shows the same SQL query window. A red box highlights the command `insert into CustomersOrders.dbo.Class (PersonID, LastName, FirstName, Address) Values (123, 'xyz', 'abc', 'mno');`. A red arrow points from this box to a red circle labeled "2." at the bottom right. Below the query window, the status bar says "Query executed successfully.".

Looping

Using DELETE and UPDATE

- ✓ **Update Statement**
- ✓ The UPDATE statement is used for modifying the existing records. You can use the WHERE clause with the UPDATE query for updating the specific rows. Otherwise, all rows would be affected
- ✓ **Syntax:**

```
UPDATE table_name  
SET column1 = value1, column2 = value2...., columnN = valueN  
WHERE [condition];
```

Looping

Using DELETE and UPDATE

- ✓ Using Update Statement
- ✓ Example:

The screenshot shows a SQL Server Management Studio window with three tabs: 'SQLQuery8.sql - D...QF77LA\Admin (52)*' (selected), 'SQLQuery7.sql - D...QF77LA\Admin (53)*', and 'SQLQuery4'. The 'SQLQuery8.sql' tab contains the following SQL code:

```
UPDATE CustomersOrders.dbo.Order_Details
SET Product_Amount = 5000
WHERE ProductId = 9475;
```

The code is highlighted with a red rectangle. Below the code, the 'Messages' pane displays the output: '(1 row(s) affected)'. At the bottom of the window, a yellow status bar shows 'Query executed successfully.'

Looping

Using DELETE and UPDATE

- ✓ **Delete Statement**
- ✓ The DELETE Query is used for deleting the existing records, which you want to remove from the table. You can use the WHERE clause with the DELETE command to delete the rows you have selected. Otherwise, the record is deleted
- ✓ **Syntax:**

```
DELETE FROM table_name  
WHERE [condition];
```

Looping

Sys.Objects

- ✓ It comprises a row for every user-defined, schema-scoped object that is generated within the database

Column name	Data type	Description
name	sysname	Object name
object_id	int	Object identification number. It is unique within the database.
principal_id	int	ID of the individual owner, if different from the schema owner. By default, schema-contained objects are owned by the schema owner.

Looping

Sys.Objects

Column name	Data type	Description
principal_id	int	<p>It is NULL if there is no alternate individual owner.</p> <p>It is NULL if the object type is one of the following:</p> <ul style="list-style-type: none">C = CHECK constraintD = DEFAULT (constraint or stand-alone)F = FOREIGN KEY constraintPK = PRIMARY KEY constraintR = Rule (old-style, stand-alone)TA = Assembly (CLR-integration) triggerTR = SQL triggerUQ = UNIQUE constraint

Looping

Sys.Objects

Column name	Data type	Description
schema_id	int	ID of the schema that the object is contained in. Schema-scoped system objects are always contained in the sys or INFORMATION_SCHEMA schemas.
parent_object_id	int	ID of the object to which this object belongs. 0 = Not a child object.
type	char(2)	Object type: AF = Aggregate function (CLR) C = CHECK constraint D = DEFAULT (constraint or stand-alone) F = FOREIGN KEY constraint

Looping

Sys.Objects

Column name	Data type	Description
type	char(2)	FN = SQL scalar function FS = Assembly (CLR) scalar-function FT = Assembly (CLR) table-valued function IF = SQL inline table-valued function IT = Internal table P = SQL Stored Procedure PC = Assembly (CLR) stored-procedure PG = Plan guide PK = PRIMARY KEY constraint R = Rule (old-style, stand-alone) RF = Replication-filter-procedure S = System base table SN = Synonym SO = Sequence object

Looping

Sys.Objects

Column name	Data type	Description
type	char(2)	U = Table (user-defined) V = View SQ = Service queue TA = Assembly (CLR) DML trigger TF = SQL table-valued-function TR = SQL DML trigger TT = Table type UQ = UNIQUE constraint X = Extended stored procedure ET = External Table
type_desc	nvarchar(60)	Description of the object type: AGGREGATE_FUNCTION CHECK_CONSTRAINT

Looping

Sys.Objects

Column name	Data type	Description
type_desc	nvarchar(60)	CLR_SCALAR_FUNCTION CLR_STORED_PROCEDURE CLR_TABLE_VALUED_FUNCTION CLR_TRIGGER DEFAULT_CONSTRAINT EXTENDED_STORED_PROCEDURE FOREIGN_KEY_CONSTRAINT INTERNAL_TABLE PLAN_GUIDE PRIMARY_KEY_CONSTRAINT REPLICATION_FILTER_PROCEDURE RULE SEQUENCE_OBJECT

Looping

Sys.Objects

Column name	Data type	Description
type_desc	nvarchar(60)	SERVICE_QUEUE SQL_INLINE_TABLE_VALUED_FUNCTION SQL_SCALAR_FUNCTION SQL_STORED_PROCEDURE SQL_TABLE_VALUED_FUNCTION SQL_TRIGGER SYNONYM SYSTEM_TABLE TABLE_TYPE UNIQUE_CONSTRAINT USER_TABLE VIEW

Looping

Sys.Objects

Column name	Data type	Description
create_date	datetime	Date object was created.
modify_date	datetime	Date object was last modified by using an ALTER statement. If the object is a table or a view, modify_date also changes when a clustered index on the table or view is created or altered.
is_ms_shipped	bit	Object is created by an internal SQL Server component
is_published	bit	Object is published
is_schema_published	bit	Only the schema of the object is published

Module 8

Temporary Tables and
Table Variables

Description

1

Using Temporary Tables

2

Creating Table Variables

3

Pros and Cons of Each Approach

Temporary Tables and Table Variables

Using Temporary Tables

- ✓ A table that exists temporarily in the database server

- ✓ It stores a subset of data from an ordinary table for a specified period

- ✓ They are especially beneficial when a user has a large number of records in the table, and frequently needs to interact with the small subset of those records

- ✓ In such cases, rather than filtering the data repeatedly to fetch the subset, a user can filter the data once and store it in a temporary table

- ✓ These tables are stored inside the “tempdb”, which is a system database

Temporary Tables and Table Variables

Using Temporary Tables

- ✓ **Types of temporary table**
 - ✓ 1. Local temporary tables
 - ✓ These tables are only accessible in the current connection to the database for the current user
 - ✓ When the user disconnects from instances, they are automatically deleted
 - ✓ The local temporary table name is started with the single hash ("#") sign

Temporary Tables and Table Variables

Using Temporary Tables

✓ Creating Local TEMP TABLES

The screenshot shows a SQL Server Management Studio window with two panes. The top pane, titled 'SQLQuery11.sql - ...QF77LA\Admin (55)*', contains the SQL code for creating a local temporary table:

```
Create Table #CustCountryy  
  (CustomerId numeric (4, 0),  
   CustomerName varchar (30),  
   Country varchar (30)  
 )
```

The bottom pane, titled 'Messages', displays the results of the execution:

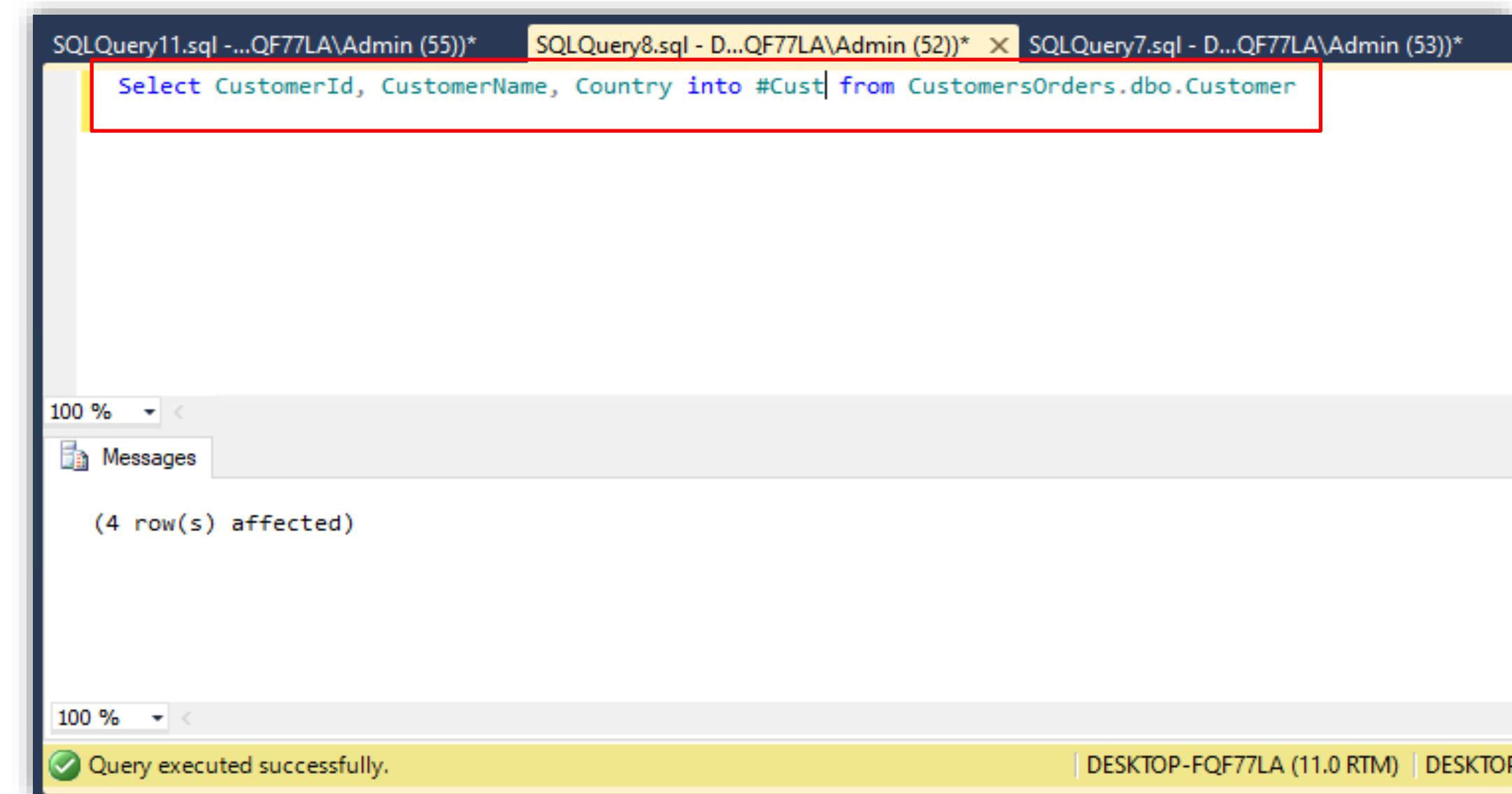
100 % < Messages
Command(s) completed successfully.

100 % < Query executed successfully.

Temporary Tables and Table Variables

Using Temporary Tables

- ✓ Inserting into TEMP TABLES



The screenshot shows a SQL Server Management Studio (SSMS) interface with three tabs at the top: "SQLQuery11.sql - ...QF77LA\Admin (55)*", "SQLQuery8.sql - D...QF77LA\Admin (52)*" (highlighted in yellow), and "SQLQuery7.sql - D...QF77LA\Admin (53)*". The main query window displays the following T-SQL code:

```
Select CustomerId, CustomerName, Country into #Cust from CustomersOrders.dbo.Customer
```

The code is highlighted with a red box. Below the query window, the "Messages" pane shows the output: "(4 row(s) affected)". At the bottom of the screen, a yellow status bar indicates "Query executed successfully." and shows the system information "DESKTOP-FQF77LA (11.0 RTM) | DESKTOP-FQF77LA".

Temporary Tables and Table Variables

Using Temporary Tables

✓ 2. Global temporary tables

- ✓ These tables are accessible to any connection once generated. These tables are dropped when the current user disconnects

- ✓ The name of these tables starts with a double hash ("##") sign

- ✓ Global temporary table are also generated in the "tempdb" and is then accessible to any user by any connection

- ✓ Global temporary table can only be deleted when all the connections have been closed

Temporary Tables and Table Variables

Using Temporary Tables

- ✓ Create Global Temp Table

The screenshot shows a dual-pane interface of SQL Server Management Studio. The left pane displays a query window titled 'SQLQuery11.sql - ...QF77LA\Admin (55)*' containing the SQL command:

```
Create Table ##Citiesss  
[City varchar(30))]
```

This command is highlighted with a red rectangular box. The right pane displays a 'Messages' window with the status 'Command(s) completed successfully.' A yellow banner at the bottom of the screen also indicates 'Query executed successfully.'

Temporary Tables and Table Variables

Using Temporary Tables

- ✓ Inserting Rows:

The screenshot shows a SQL Server Management Studio window with three tabs at the top: 'SQLQuery11.sql - ...QF77LA\Admin (55)*', 'SQLQuery8.sql - D...QF77LA\Admin (52)*', and 'SQLQuery7.sql - D...QF77LA'. The 'SQLQuery7.sql' tab is active, displaying the following SQL code:

```
INSERT INTO ##Citiessss Select Distinct City from CustomersOrders.dbo.Customer
```

The code is highlighted with a red box. Below the code, the 'Messages' pane shows the output: '(3 row(s) affected)'. At the bottom of the window, a yellow status bar displays the message 'Query executed successfully.' and the computer name 'DESKTOP-FQF77LA (1)'.

Temporary Tables and Table Variables

Using Temporary Tables

- ✓ Retrieving Data:

The screenshot shows a SQL Server Management Studio interface with two query panes. The left pane is titled 'SQLQuery11.sql - ...QF77LA\Admin (55)*' and contains the SQL query: 'Select * from ##Citiessss'. The right pane is titled 'SQLQuery8.sql - D...QF77LA\A' and displays the results of the query in a table format. The table has one column named 'City' with three rows: 'Johannesburg', 'New York', and 'Wolverhampton'. A red box highlights the query in the left pane. A yellow box highlights the 'Results' tab in the bottom navigation bar. A green success message at the bottom states 'Query executed successfully.'

City
Johannesburg
New York
Wolverhampton

Query executed successfully.

Temporary Tables and Table Variables

Creating Table Variables

- ✓ Table variables are introduced by Microsoft as an alternative to using temporary tables
- ✓ They store a set of records, so the declaration syntax seems very similar to a CREATE TABLE statement

The screenshot shows a SQL Server Management Studio (SSMS) window with three tabs at the top: 'SQLQuery11.sql' (QF77LA\Admin (55)), 'SQLQuery8.sql' (D...QF77LA\Admin (52)), and 'SQLQuery7.sql' (D...QF77LA\Admin (53)). The code in the main pane is:

```
Declare @ProductRevenue TABLE
(
    ProductId Numeric(4,0),
    TotalAmount Numeric(18,0)
)
Begin
    Insert into @ProductRevenue
    Select ProductId, Sum(Product_Amount) from CustomersOrders.dbo.Order_Details group by ProductId
    Select * from @ProductRevenue
End
```

The results pane below shows a table with three rows:

	ProductId	TotalAmount
1	255	7000
2	9001	10000
3	9374	6000

A red box highlights the entire code block in the top pane. A green checkmark icon in the bottom status bar indicates "Query executed successfully."

Temporary Tables and Table Variables

Creating Table Variables

- ✓ Updating the table variables

The screenshot shows a SQL Server Management Studio interface with three tabs: SQLQuery11.sql, SQLQuery8.sql, and SQLQuery7.sql. The SQLQuery7.sql tab contains the following T-SQL code:

```
Declare @ProductRevenue TABLE
(
    ProductId Numeric(4,0),
    TotalAmount NUmeric(18,0)
)
Begin
    Insert into @ProductRevenue
    Select ProductId, Sum(Product_Amount) from CustomersOrders.dbo.Order_Details group by ProductId
    Select * from @ProductRevenue

    UPDATE @ProductRevenue
    SET TotalAmount=TotalAmount+ 1500
    WHERE ProductID = 9374

    Select * from @ProductRevenue
End
```

A large red rectangle highlights the entire code block. A blue arrow points from the code area to the results grid. The results grid shows two sets of data. The first set, labeled "Results", shows the initial data from the table variable:

ProductId	TotalAmount
1	255
2	9001
3	9374

The second set, labeled "Messages", shows the result of the UPDATE statement:

ProductId	TotalAmount
1	255
2	9001
3	9374

At the bottom of the results grid, a yellow bar indicates: "Query executed successfully."

Temporary Tables and Table Variables

Creating Table Variables

- ✓ Deleting from table variables

The diagram illustrates the execution flow of a SQL script. On the left, a code editor window displays T-SQL code. A red box highlights the declaration of a table variable (@ProductRevenue) and its initial population via an INSERT statement. A blue arrow points from this code to the right, leading to a SQL Server Management Studio (SSMS) results grid. The first row of the grid shows the initial data: ProductId 255 with TotalAmount 7000, ProductId 9001 with TotalAmount 10000, and ProductId 9374 with TotalAmount 6000. In the second row, the TotalAmount for ProductId 9374 is updated to 7500 via an UPDATE statement. In the third row, a DELETE statement removes the entry for ProductId 9374, resulting in only two rows of data: ProductId 255 with TotalAmount 7000 and ProductId 9001 with TotalAmount 10000. The status bar at the bottom of the SSMS window indicates "Query executed successfully."

```
Declare @ProductRevenue TABLE
(
    ProductId Numeric(4,0),
    TotalAmount NUmeric(18,0)
)
Begin
    Insert into @ProductRevenue
    Select ProductId, Sum(Product_Amount) from CustomersOrders.dbo.Order_Details group by ProductId
    Select * from @ProductRevenue
    UPDATE @ProductRevenue
        SET TotalAmount=TotalAmount+ 1500
    WHERE ProductID = 9374
    Select * from @ProductRevenue

    Delete from @ProductRevenue where ProductId=9374
    Select * from @ProductRevenue
End
```

ProductId	TotalAmount
1 255	7000
2 9001	10000
3 9374	6000

ProductId	TotalAmount
1 255	7000
2 9001	10000
3 9374	7500

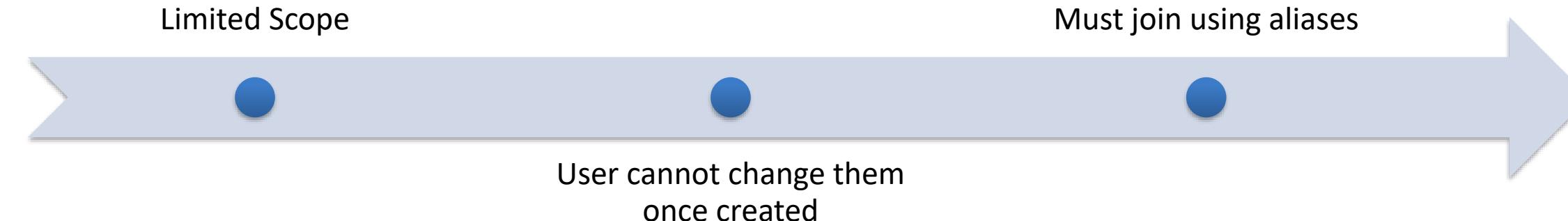
ProductId	TotalAmount
1 255	7000
2 9001	10000

Query executed successfully.

Temporary Tables and Table Variables

Pros and Cons of Each Approach

- ✓ **Advantages of table variables**
- ✓ Speed: Table variables do not require locking and logging resources, nor do they have to be stored in the database
- ✓ Work with UDFs: A user can insert, update, and delete records in table variables within the user-defined functions
- ✓ **Disadvantages of table variables**



Temporary Tables and Table Variables

Pros and Cons of Each Approach

- ✓ **Advantages of temporary tables**
- ✓ **Simplicity of coding:** Temporary tables behave like ordinary tables. A user can sort, filter, and join them as if they were permanent tables
- ✓ **Access rights/security:** A user can generate a temporary table and has access to insert, delete, and update its records without worrying about anything
- ✓ **Speed:** SQL Server has less locking and logging overheads for temporary tables, so they execute more promptly

Temporary Tables and Table Variables

Pros and Cons of Each Approach

- ✓ **Disadvantages of temporary tables**
- ✓ **Not as fast as table variables:** Using temporary tables are quicker than using permanent ones, but they are not faster than the temporary variables
- ✓ **Can not update in functions:** A user can not use INSERT, UPDATE, or DELETE statements against the temporary tables in UDFs

Module 9



Table Valued Functions

Description

1

In-Line Table-valued Functions

2

Multi-Statement Table-Valued Functions

3

Limitations of User-Defined Functions

Table Valued Functions

In-Line Table-Valued Functions

- ✓ In SQL Server, table valued functions are great for writing DRY SQL code by encapsulating commonly utilised snippets of database logic

- ✓ Table-valued functions return the data of table types

- ✓ The example given on the next slide shows the creation of an inline table-valued function that will retrieve the records of all the customers

Table Valued Functions

In-Line Table-Valued Functions

- ✓ Example :

```
SQLQuery1.sql - D...QF77LA\Admin (52)* ×
Create Function CustFunc()
Returns @tabCust Table
(CustomerId numeric(4,0),
CustomerName varchar(30),
City varchar(30),
Country varchar(30),
PostalCode Varchar(10))
As
Begin
    INSERT INTO @tabCust    SELECT * from CustomersOrders.dbo.Customer
    WHERE CustomerId > 5602
Return
End
```

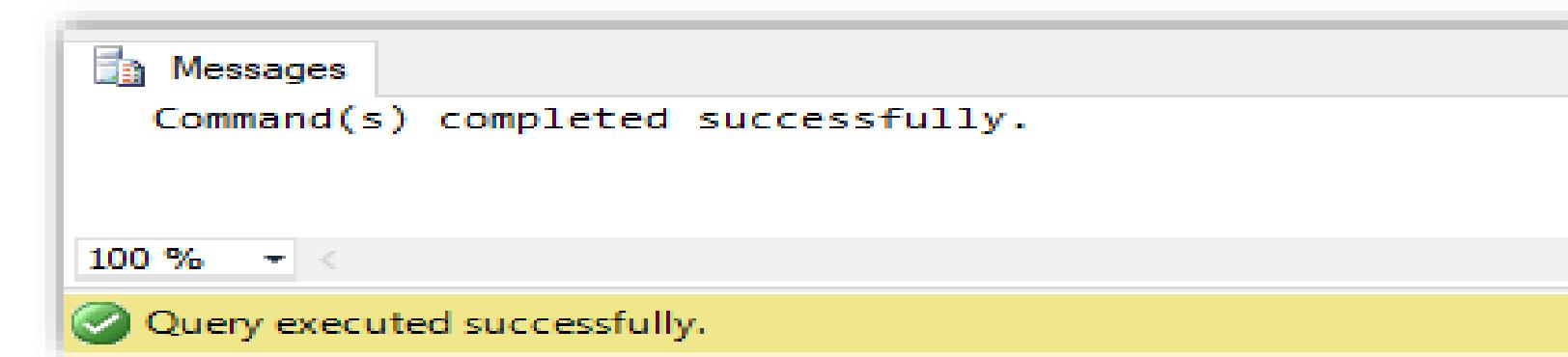


Table Valued Functions

In-Line Table-Valued Functions

- ✓ Example: Retrieving Data from in-line table valued functions

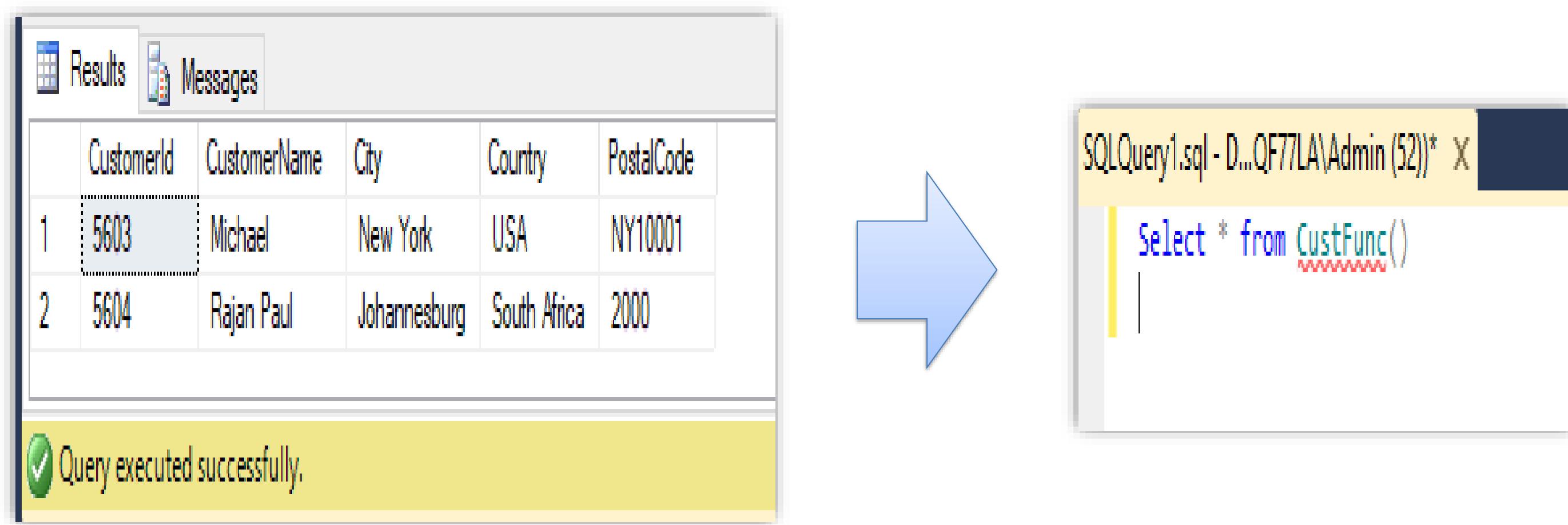
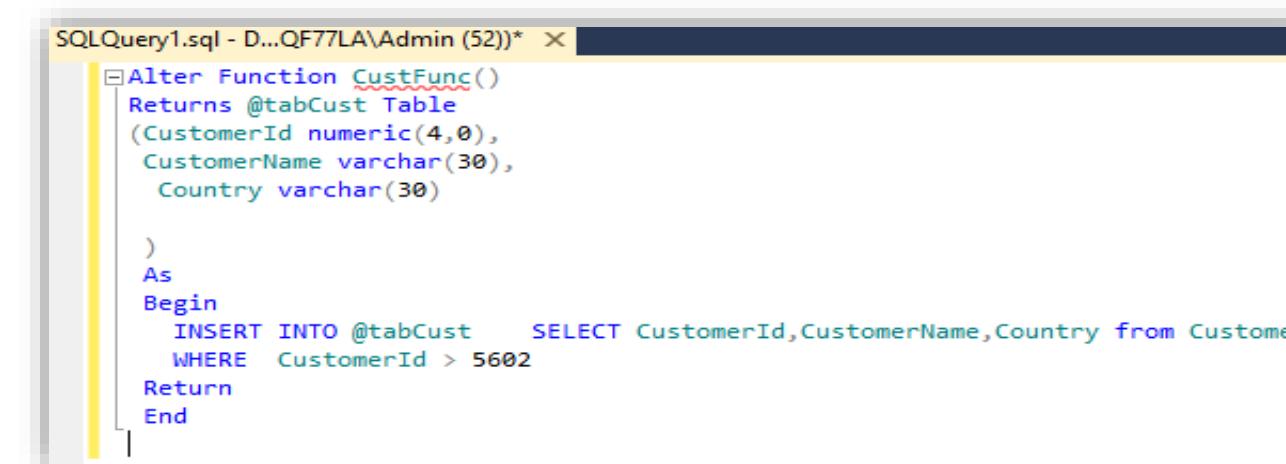


Table Valued Functions

In-Line Table-Valued Functions

- ✓ **Modifying an Inline Table-Valued Function**
- ✓ Use the ALTER keyword to modify an existing function
- ✓ Here we are modifying the "BornBefore" function so that it takes two datetime type parameters and returns student records for students whose DOB value lies between the values passed by the two parameters



```
SQLQuery1.sql - D...QF77LA\Admin (52)*
ALTER Function CustFunc()
Returns @tabCust Table
(CustomerId numeric(4,0),
CustomerName varchar(30),
Country varchar(30))

)
As
Begin
    INSERT INTO @tabCust    SELECT CustomerId,CustomerName,Country from Customer
    WHERE CustomerId > 5602
Return
End
```

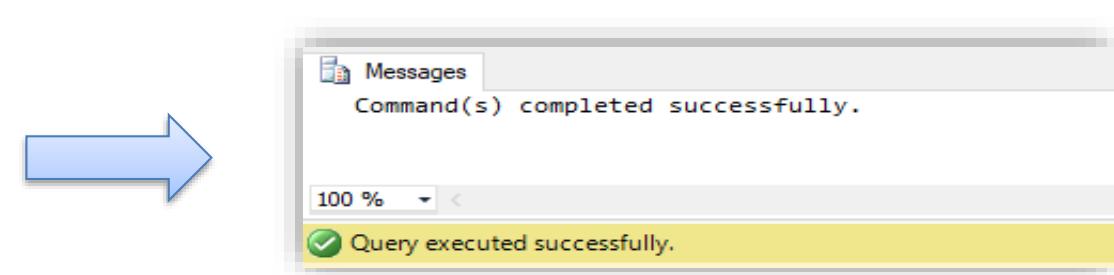
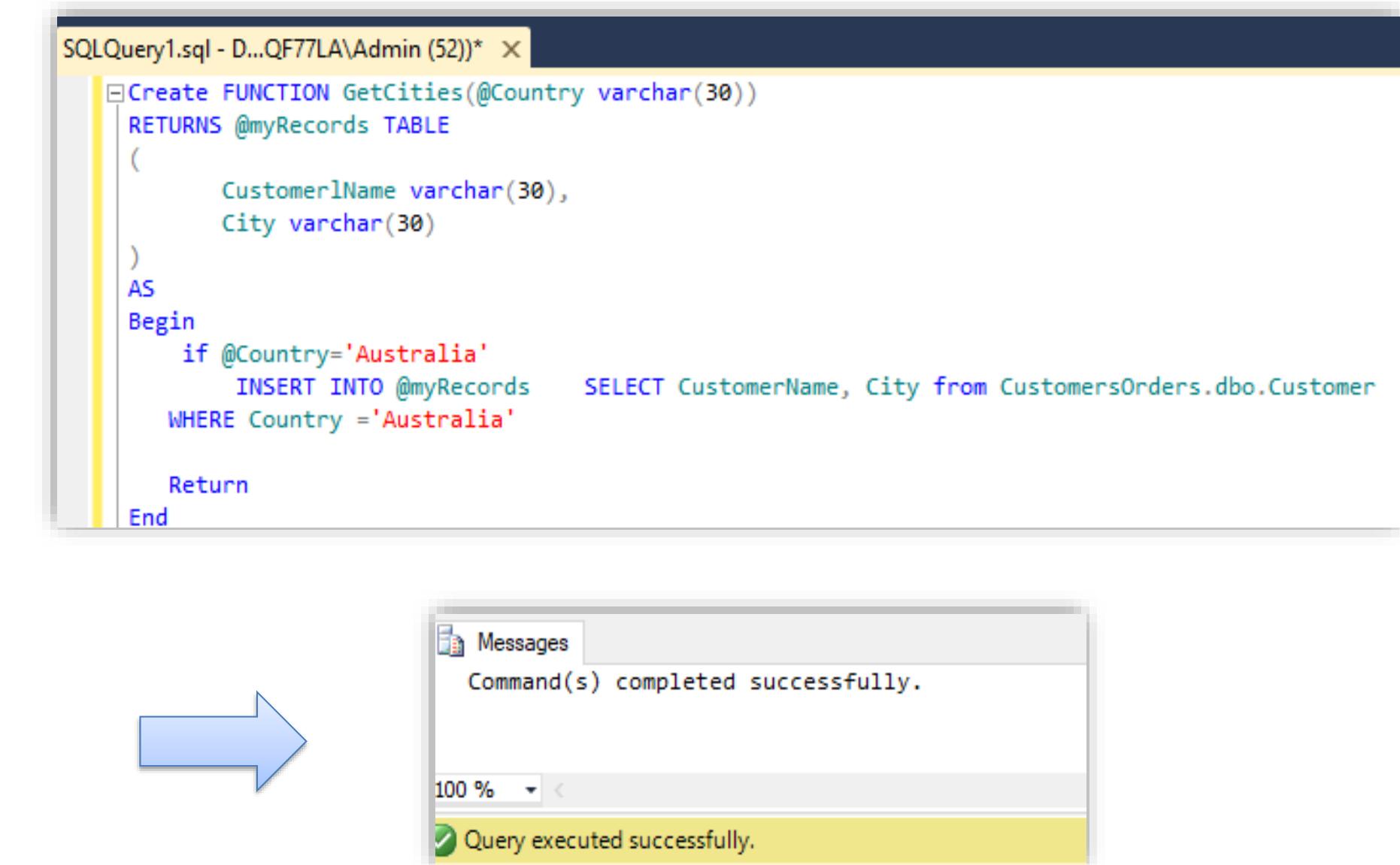


Table Valued Functions

Multi-Statement Table-Valued Functions

- ✓ Creating Multi-Statement table valued function



The screenshot shows a SQL query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*". The code creates a function "GetCities" that takes a parameter "@Country" and returns a table "@myRecords" containing "CustomerName" and "City" for customers in the specified country. An IF statement checks if the country is 'Australia' and inserts rows into the table. A blue arrow points from the query window to a "Messages" window below.

```
Create FUNCTION GetCities(@Country varchar(30))
RETURNS @myRecords TABLE
(
    CustomerName varchar(30),
    City varchar(30)
)
AS
Begin
    if @Country='Australia'
        INSERT INTO @myRecords    SELECT CustomerName, City from CustomersOrders.dbo.Customer
        WHERE Country = 'Australia'

    Return
End
```

Messages

Command(s) completed successfully.

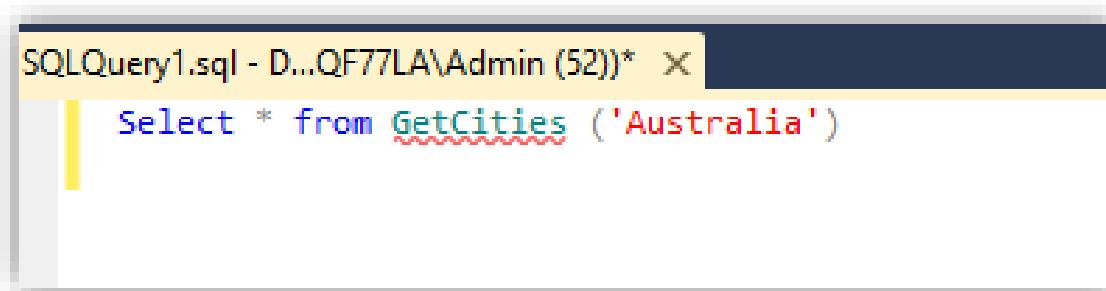
100 % <

Query executed successfully.

Table Valued Functions

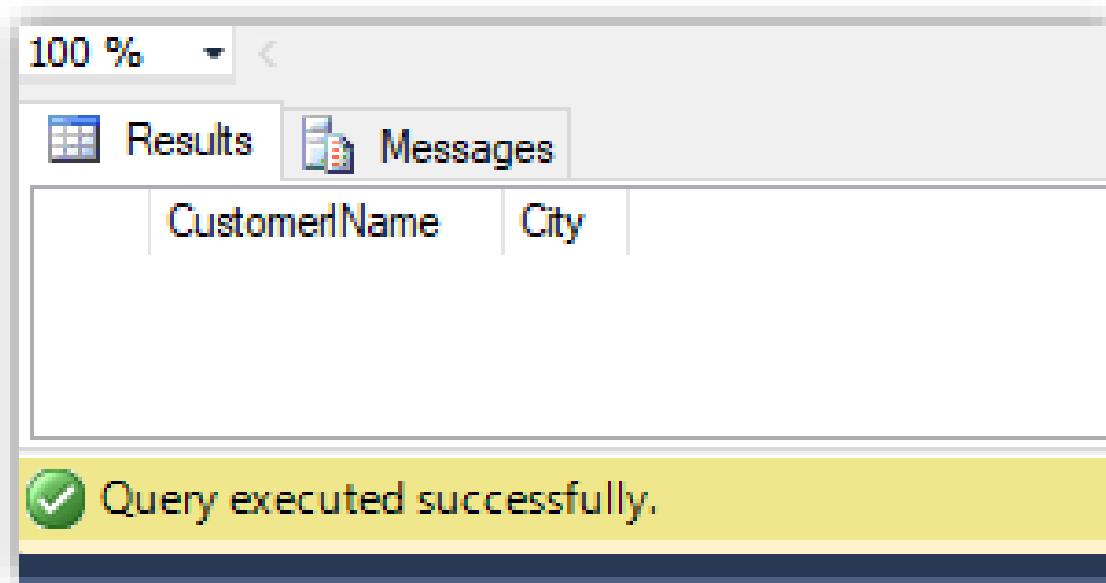
Multi-Statement Table-Valued Functions

- ✓ Retrieving Data:



```
SQLQuery1.sql - D...QF77LA\Admin (52)* X
Select * from GetCities ('Australia')
```

- ✓ Result:



CustomerName	City

Query executed successfully.

Table Valued Functions

Limitations of User-Defined Functions

- ✓ The following are some limitations of user-defined functions:
 - ✓ User Defined Functions cannot use non-deterministic built-in functions
 - ✓ They can return only one rowset to the user
 - ✓ UDFs cannot call stored procedures
 - ✓ They also cannot perform dynamically constructed SQL statements
 - ✓ They cannot return XML string as output
 - ✓ Within a UDF, RAISERROR statement cannot be used
 - ✓ They can accept a smaller number of parameters than stored procedures
 - ✓ UDFs cannot alter the data in permanent tables, they can only read the data
 - ✓ They cannot make use of temporary tables

Module 10

Derived Tables and CTEs

Description

1

Using Derived Tables

2

Common Table Expressions(CTEs)

3

Recursive CTEs

Derived Tables and CTEs

Introduction

- ✓ A derived table is a way to create a temporary set of records which can be used within another query in SQL

- ✓ A user can use the derived tables to compress the long queries, or even just to break a complex process into logical steps

- ✓ Derived tables is also known as subqueries

- ✓ It is defined in parenthesis followed by an AS clause to stipulate the derived table name

Derived Tables and CTEs

Introduction

- ✓ Basic Syntax of a Derived Table with an example:

The diagram illustrates the execution flow of a SQL query. On the left, a screenshot of the SQL Server Management Studio (SSMS) interface shows a query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)* X". The query itself is:

```
Select Top 5 CustomerName, City  
From  
(Select * From CustomersOrders.dbo.Customer) as Cust  
Where Country = 'England'
```

An orange arrow points from the query window to a second screenshot on the right, which shows the "Results" tab of the SSMS interface. The results are displayed in a table:

	CustomerName	City
1	John Carter	Wolverhampton

Below the table, a green message bar indicates: "Query executed successfully."

Derived Tables and CTEs

Common Table Expressions(CTEs)

- ✓ It defines a temporary result set which users can use in the SELECT statement
- ✓ It is very helpful in managing complicated queries
- ✓ CTE are specified within the statement using WITH operator
- ✓ A user can define more than one CTE

Derived Tables and CTEs

Common Table Expressions(CTEs)

✓ Creating a Nonrecursive CTE

- ✓ A nonrecursive CTE does not reference itself within the CTE
- ✓ Nonrecursive CTEs tend to be simpler than recursive CTEs
- ✓ Here we are creating a CTE named OrderCust:

```
SQLQuery1.sql - D...QF77LA\Admin (52)* X
With
    OrderCust (CustomerId, Ord_Date)
As
    (Select CustId, OrderDate from CustomersOrders.dbo.Orders)
    Select Customername, OC.Ord_Date
    From CustomersOrders.dbo.Customer
    INNER JOIN OrderCust as OC
    On Customer.CustomerId = OC.CustomerId
```



	Customername	Ord_Date
1	Triza	2017-12-12
2	Michael	2018-01-10
3	Triza	2017-12-15
4	Rajan Paul	2018-01-18

Query executed successfully.

Derived Tables and CTEs

Recursive CTEs

✓ Creating a Recursive Common Table Expression

- ✓ Recursive CTE references itself within the CTE
- ✓ It is beneficial while working with hierarchical data because the CTE continues to perform until the query returns the complete hierarchy. The following is the Syntax:

```
WITH expression_name (column_list)
AS
(
    -- Anchor member
    initial_query
    UNION ALL
    -- Recursive member that references expression_name.
    recursive_query
)
-- references expression name
SELECT *
FROM expression_name
```

Derived Tables and CTEs

Recursive CTEs

- ✓ Example:

The screenshot shows a SQL query window titled "SQLQuery1.sql - DES...n (52) Executing..." containing the following T-SQL code:

```
WITH UserCTE AS (
    SELECT CustomerId, CustomerName, City, 0 as steps
    From CustomersOrders.dbo.Customer
    Where CustomerId < 5604
    UNION ALL
    Select ng.CustomerId, ng.CustomerName, ng.city, us.steps +1 AS steps
    From UserCTE us
    INNER JOIN CustomersOrders.dbo.Customer ng
        ON us.CustomerId = ng.CustomerId)
Select * from UserCTE OPTION (MAXRECURSION 0)
```

A large blue arrow points from the query window to a "Results" grid on the right. The results grid has columns: CustomerId, CustomerName, City, and steps. The data is as follows:

	CustomerId	CustomerName	City	steps
1	5601	John Carter	Wolverhampton	0
2	5602	Triza	Johannesburg	0
3	5603	Michael	New York	0
4	5603	Michael	New York	1
5	5603	Michael	New York	2

Module 11

Subqueries

Description

1	Subquery
2	Using ALL, ANY and IN
3	Correlated Subqueries
4	Using EXISTS

Subqueries

Introduction

- ✓ A subquery can be defined as a query within a query. A user can generate subqueries within the SQL statements
 - ✓ These subqueries can reside in the FROM clause, in the WHERE clause, or the SELECT clause
-
- ✓ **WHERE clause**
 - ✓ Subqueries will most frequently be found in the WHERE clause
 - ✓ These subqueries are also known as nested subqueries

Subqueries

Introduction

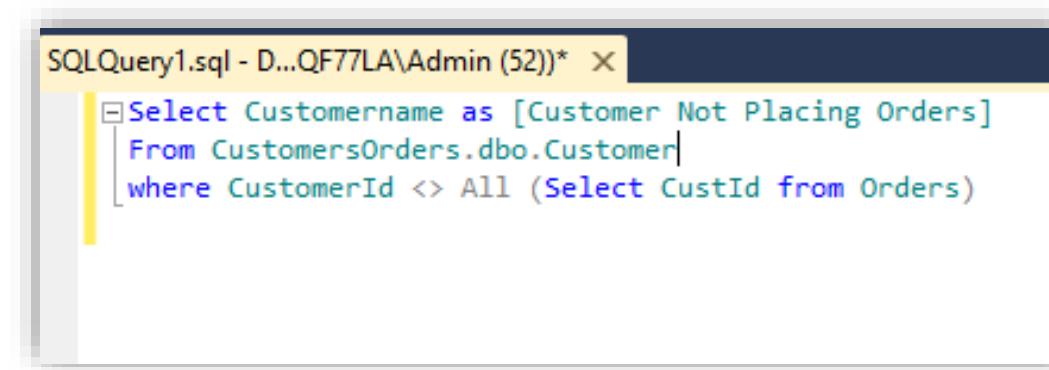
- ✓ **FROM clause**
- ✓ Subqueries can also be found in the FROM clause, these are known as inline views

- ✓ **SELECT clause**
- ✓ A subquery within a SELECT clause are generally used to retrieve a calculation by using an aggregate function, e.g. SUM, COUNT, MIN, or MAX function

Subqueries

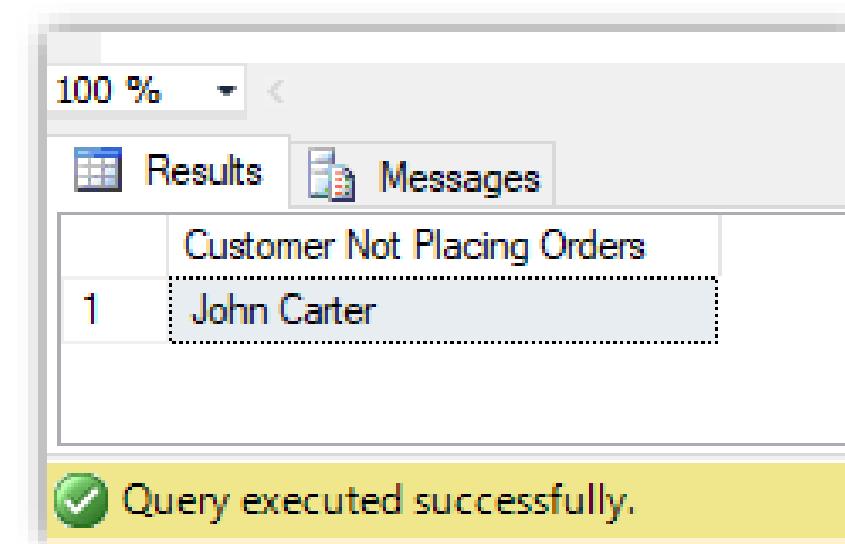
Introduction

- ✓ **Example:** Find the name of the Customer who is not placing an Order



```
SQLQuery1.sql - D...QF77LA\Admin (52)*
Select Customername as [Customer Not Placing Orders]
From CustomersOrders.dbo.Customer
where CustomerId <> All (Select CustId from Orders)
```

- ✓ **Output:**



	Customer Not Placing Orders
1	John Carter

Query executed successfully.

Subqueries

Using ALL, ANY and IN

- ✓ ALL
 - ✓ This operator compares the scalar value with a single-column set of values
 - ✓ ALL is used with SELECT, WHERE, HAVING statement
 - ✓ Syntax:

```
scalar_expression { = | <> | != | > | >= | !=> | < | <= | !< } ALL ( subquery )
```

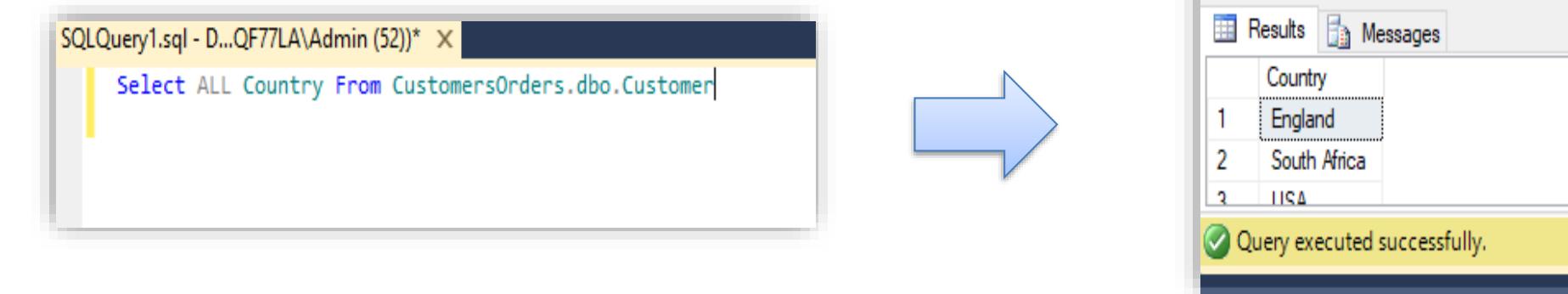
Subqueries

Using ALL, ANY and IN

- ✓ ALL with select statements

Syntax:
SELECT ALL field_name
FROM table_name
WHERE condition(s);

- ✓ Example:



The image shows two windows from SQL Server Management Studio. The left window is titled 'SQLQuery1.sql - D...QF77LA\Admin (52)*' and contains the SQL query: 'Select ALL Country From CustomersOrders.dbo.Customer'. The right window is titled 'Results' and displays a table with three rows of data: 'Country' (1) 'England', (2) 'South Africa', and (3) 'USA'. Below the table, a message says 'Query executed successfully.'

Country
1 England
2 South Africa
3 USA

Query executed successfully.

Subqueries

Using ALL, ANY and IN

- ✓ ALL with HAVING Statement:

The image shows a screenshot of SQL Server Management Studio. At the top, there is a title bar with the text "SQLQuery1.sql - D...QF77LA\Admin (52)* X". Below the title bar is a query editor window containing the following SQL code:

```
SELECT PRODUCTNAME FROM CustomersOrders.dbo.Product_Table WHERE PRODUCTID = ALL(
    SELECT PRODUCTID FROM ORDER_DETAILS GROUP BY PRODUCTID HAVING COUNT(*) > 1)
```

Below the query editor is a results window titled "Results". It displays a single row of data:

PRODUCTNAME
Polo T-shirts

At the bottom of the results window, there is a message: "Query executed successfully." A large blue arrow points from the query editor towards the results window.

Subqueries

Using ALL, ANY and IN

✓ IN

- ✓ The IN operator enables us to define several values in a WHERE clause
- ✓ It is a shorthand for multiple OR conditions
- ✓ **Syntax:**

```
SELECT column_name(s)  
FROM table_name  
WHERE column_name IN (value1,  
value2, ...);
```

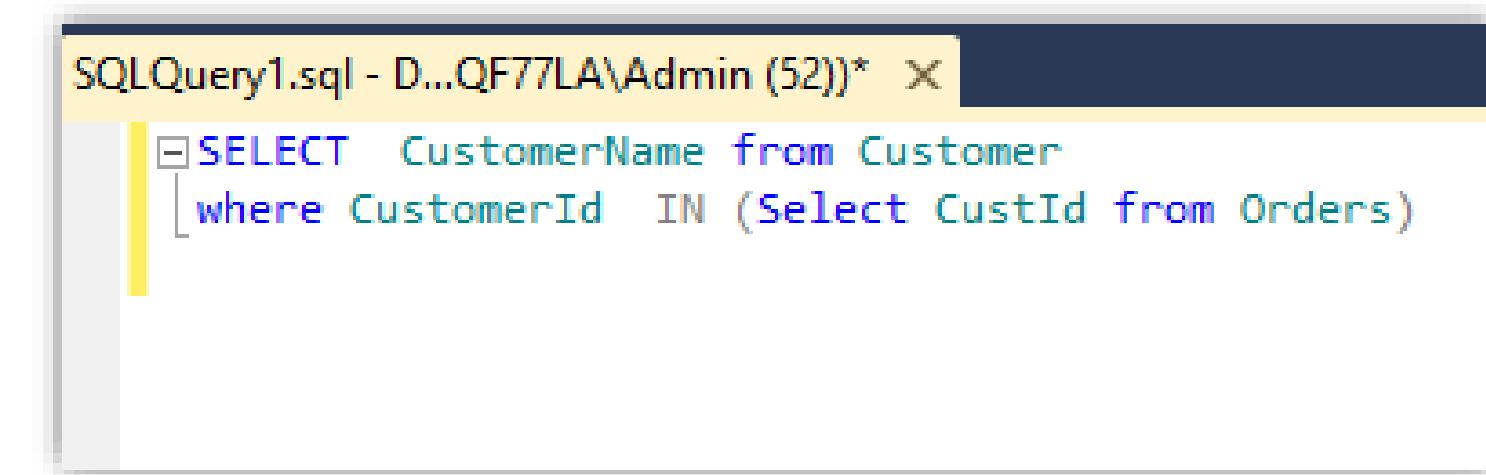
OR

```
SELECT column_name(s)  
FROM table_name  
WHERE column_name IN (SELECT  
STATEMENT);
```

Subqueries

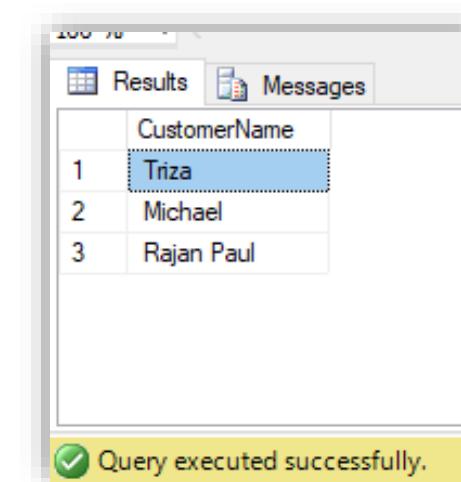
Using ALL, ANY and IN

✓ Example



```
SQLQuery1.sql - D...QF77LA\Admin (52)* X
SELECT CustomerName from Customer
where CustomerId IN (Select CustId from Orders)
```

✓ Output



CustomerName
1 Triza
2 Michael
3 Rajan Paul

Query executed successfully.

Subqueries

Using ALL, ANY and IN

- ✓ **ANY**
- ✓ ANY compares the value to each value in the list or results from a query and measures true if the result of an inner query comprises at least one row
- ✓ It must be preceded by comparison operators
- ✓ This operator returns a true statement if any of the subquery values satisfy the condition

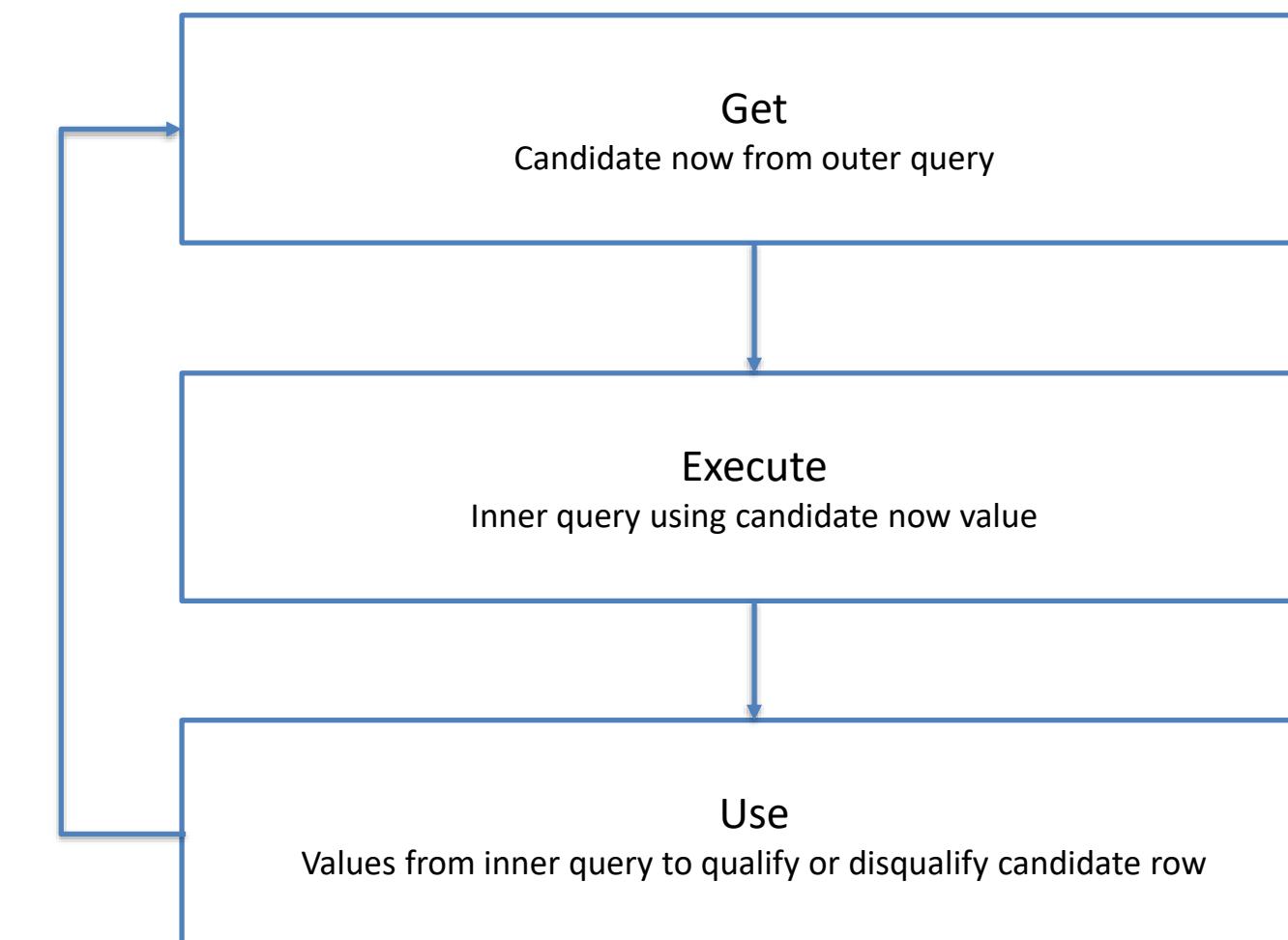
Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name comparison_operator ANY
(SELECT column_name
FROM table_name
WHERE condition(s));
```

Subqueries

Correlated Subqueries

- ✓ These subqueries are utilised for row-by-row processing
- ✓ Each subquery is performed once for every row of the outer query



Subqueries

Correlated Subqueries

Nested Subqueries Versus Correlated Subqueries:

- ✓ With the standard nested subquery, the inner SELECT query runs first and performs once, returning values to be used by the central query

- ✓ It performs once for every candidate row considered by the outer query

- ✓ The outer query drives the inner query

Subqueries

Correlated Subqueries

Example:

The image shows a screenshot of SQL Server Management Studio (SSMS) illustrating the execution of a correlated subquery. On the left, a query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*" contains the following T-SQL code:

```
SELECT Customerid, Customername
FROM Customer
WHERE Customerid IN (SELECT CustId
FROM Orders where SupplierId=8374)
```

A large blue arrow points from the query window to the results window on the right. The results window has tabs for "Results" and "Messages". The "Results" tab displays a table with two rows:

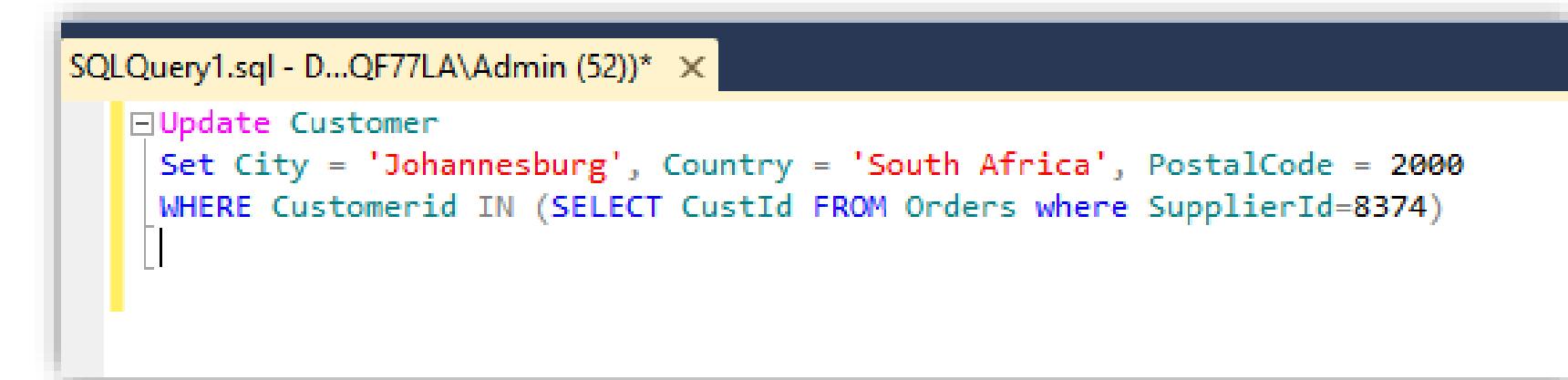
	Customerid	Customername
1	5602	Triza
2	5604	Rajan Paul

The "Messages" tab at the bottom of the results window shows a green checkmark icon and the text "Query executed successfully."

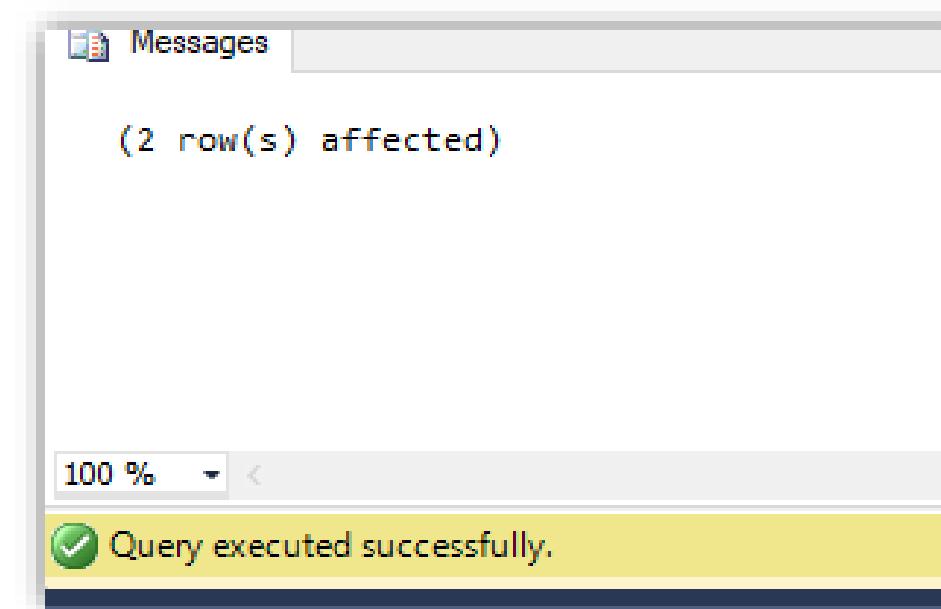
Subqueries

Correlated Subqueries

Co-Related Subquery Example with UPDATE :



```
SQLQuery1.sql - D...QF77LA\Admin (52)* X
Update Customer
Set City = 'Johannesburg', Country = 'South Africa', PostalCode = 2000
WHERE Customerid IN (SELECT CustId FROM Orders where SupplierId=8374)
```



Subqueries

Correlated Subqueries

Co-Related Subquery with DELETE:

The diagram illustrates the execution of a SQL query. On the left, a screenshot of a SQL Server Management Studio (SSMS) window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*" shows the following T-SQL code:

```
DELETE FROM Customer  
WHERE CustomerId NOT IN (SELECT CustId FROM Orders)
```

A large blue arrow points from the query window to the right, leading to another screenshot of the SSMS window. This second window is titled "Messages" and displays the following output:

(1 row(s) affected)

Query executed successfully.

Subqueries

Correlated Subqueries

Using the EXISTS Operator:

- ✓ The EXISTS operator tests for the existence of rows in the results set of the subquery
- ✓ If the subquery row value is found, then the condition will be flagged TRUE, and the search will not continue in the inner query
- ✓ If the subquery row value is not found, then the condition will be flagged FALSE, and the search will continue in the inner query

Subqueries

Correlated Subqueries

EXAMPLE of using EXIST operator:

The diagram illustrates the execution of a correlated subquery. On the left, a screenshot of a SQL Server Management Studio (SSMS) window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*" shows the following T-SQL code:

```
Select Customername from Customer C where  
Exists (Select Custid from Orders where C.CustomerId = Orders.CustId)
```

A large blue arrow points from the query window to the results window on the right. The results window, titled "Results", displays a table with the column "Customename". The data is as follows:

	Customename
1	Triza
2	Michael
3	Rajan Paul

At the bottom of the results window, a yellow bar indicates the query was executed successfully.

Subqueries

Correlated Subqueries

EXAMPLE of using NOT EXIST operator:

The diagram illustrates the execution of a SQL query. On the left, a screenshot of a SQL Server Management Studio (SSMS) query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*" shows the following T-SQL code:

```
SELECT Customername FROM Customer C WHERE NOT EXISTS (SELECT Custid FROM Orders WHERE C.CustomerId = Orders.CustId)
```

A large blue arrow points from the query window to the right, indicating the flow of execution. On the right, another screenshot of the SSMS results window shows the output of the query. The results grid has one column labeled "Customername" and contains a single row with the value "Customername". Below the grid, a yellow status bar displays the message "Query executed successfully."

Subqueries

Using EXISTS

- ✓ Example: Using Exists with DELETE Statement

The diagram illustrates the execution of a SQL query. On the left, a screenshot of the SQL Server Management Studio (SSMS) interface shows a query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*". The query itself is:

```
Delete from Customer
Where Exists
(Select CustId from Orders INNER JOIN Customer
ON Customer.Customerid = Orders.CustId)
```

A large blue arrow points from the query window to the right, indicating the flow of execution. To the right is another screenshot of the SSMS interface, specifically the "Messages" window. This window displays the results of the query execution:

100 % < Messages
(3 row(s) affected)
100 % <
Query executed successfully.

Module 12



Description

1	Cursors
2	Syntax of Fetching Rows

Cursors

Introduction

- ✓ A Cursor is a Virtual table or SQL Object that retrieves the data from the table one row at a time
- ✓ To update the records in a database table row by row, we cause cursors

✓ Life Cycle of cursor

- ✓ Declare
- ✓ Open
- ✓ Fetch
- ✓ Close
- ✓ Deallocate

Cursors

Types of Cursors

- ✓ Forward Only Cursor

- ✓ Scroll Cursor

- ✓ Static Cursor

- ✓ Dynamic Cursor

- ✓ Keyset Driven Cursor

Cursors

Forward Only Cursor

- ✓ In forward only cursor, we can only fetch the next record. We cant fetch the first, last, and a specific record
- ✓ Example:

The diagram illustrates a SQL query being run in SSMS. On the left, the SQL code is displayed in the Query Editor:

```
SQLQuery4.sql - D...QF77LA\Admin (54)*      SQLQuery12.sql -...QF77LA\Admin (56)*
└─ Declare @CustName varchar(30)
└─ Declare @City varchar(30)
└─ Declare CustCur Cursor
  For
    Select Customername, City from CustomersOrders.dbo.Customer
  └─ Begin
    Open CustCur
    Fetch CustCur Into @CustName, @City
    Print 'Customer Name:' +@CustName
    Print 'City :'+@City
    Close CustCur
  └─ End
  └─ Deallocate CustCur
```

A large blue arrow points from the Query Editor to the Messages window on the right, which displays the results of the execution:

100 % < Messages
Customer Name:John Carter
City :Wolverhampton

Cursors

Scroll Cursor

- ✓ With the help of scroll cursor, a user can fetch any record as first, last, prior, and specific record from the table
- ✓ Declaration of Scroll cursor

```
declare scroll_cursor cursorname  
scroll for  
select * from table
```

Cursors

Syntax of Fetching rows

- ✓ Retrieves a specific row from a server cursor

```
FETCH
  [ [ NEXT | PRIOR | FIRST | LAST
      | ABSOLUTE { n | @nvar }
      | RELATIVE { n | @nvar }
    ]
  FROM
  ]
{ { [ GLOBAL ] cursor_name } | @cursor_variable_name }
[ INTO @variable_name [ ,...n ] ]
```

Cursors

Syntax of Fetching rows

- ✓ Declaring a SCROLL cursor using FETCH

The image shows a screenshot of SQL Server Management Studio (SSMS) illustrating the use of a cursor. On the left, the 'SQLQuery12.sql' window displays the following T-SQL code:

```
SQLQuery4.sql - D...QF77LA\Admin (54)*      SQLQuery12.sql - ...QF77LA\Admin (56)*
Declare @CustName varchar(30)
Declare @City varchar(30)
Declare CustCur Cursor
for
Select Customername, City from CustomersOrders.dbo.Customer
Begin
Open CustCur
While @@FETCH_STATUS=0
Begin
Fetch CustCur Into @CustName, @City
Print 'Customer Name:' +@CustName
Print 'City :'+@City
End
Close CustCur
End
Deallocate CustCur
```

A large blue arrow points from the code window to the 'Messages' window on the right, which displays the output of the executed query:

Customer Name:John Carter
City :Wolverhampton
Customer Name:Triza
City :Johannesburg
Customer Name:Michael
City :New York
Customer Name:Rajan Paul
City :Johannesburg
Customer Name:Rajan Paul
City :Johannesburg

Module 13

Error Handling

Description

1	Using TRY / CATCH
2	System Error Functions
3	Custom Error Messages
4	The Obsolete @@ Error Function
5	The SQL Server Debugger

Error-Handling

Using TRY/CATCH

- ✓ TRY/CATCH is the Structured Error handling construct
- ✓ If any error is raised in the TRY block then the control is instantly passed to the CATCH block
- ✓ CATCH block will not be executed if none of the statements in the TRY block raises any exception

```
BEGIN TRY  
-- T-Sql Statements  
END TRY  
BEGIN CATCH  
-- T-Sql Statements  
/*Control is passed to CATCH block only if  
there are any exceptions in the TRY  
block*/  
END CATCH
```

Error-Handling

Using TRY/CATCH

- ✓ Example: If none of the statements in the TRY block raises any exception

The screenshot shows the SQL Server Management Studio interface. On the left, a query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*" displays the following T-SQL code:

```
PRINT 'BEFORE TRY'
BEGIN TRY
    PRINT 'First Statement in the Try block'
    INSERT INTO Customer (CustomerId, CustomerName, City, Country, PostalCode)
    VALUES (5613, 'Steve', 'Melbourne', 'Australia', 3005)
    PRINT 'Last Statement in the TRY block'
END TRY
BEGIN CATCH
    PRINT 'In CATCH Block'
END CATCH
PRINT 'After END CATCH'
GO
```

A large blue arrow points from the query window to the right, where a "Messages" window is shown. This window contains the output of the executed code:

```
BEFORE TRY
First Statement in the Try block
(1 row(s) affected)
Last Statement in the TRY block
After END CATCH
100 %
Query executed successfully.
```

Error-Handling

Using TRY/CATCH

- ✓ In this example, the INSERT statement results in a statement Terminating Primary Key Violation error

The diagram illustrates the execution flow of a SQL query. On the left, a screenshot of a SQL Server Management Studio (SSMS) window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*" shows the following T-SQL code:

```
Begin
    PRINT 'BEFORE TRY'
    BEGIN TRY
        PRINT 'First Statement in the TRY block'

        INSERT INTO dbo.Customer
        VALUES (5614, 'John', 'London', 'UK', '144001')
        PRINT 'Last Statement in the TRY block'
    END TRY
    BEGIN CATCH
        PRINT 'In CATCH Block'
    END CATCH
    PRINT 'After END CATCH'
    SELECT * FROM dbo.Customer WITH (NOLOCK)
End
```

A large blue arrow points from the SSMS window to the right, indicating the flow of execution. On the right, a screenshot of the SSMS "Results" tab shows the output of the query. The "Results" pane displays a table with one row of data:

	CustomerId	CustomerName	City	Country	PostalCode
3	5614	John	London	UK	144001

The "Messages" pane at the bottom of the results window shows a green checkmark icon and the message "Query executed successfully."

Error-Handling

Using TRY/CATCH

- ✓ In this instance, the second statement results in a Batch Abortion CONVERSION/CAST error:

The diagram illustrates the use of TRY/CATCH in SQL Server. On the left, a screenshot of the SSMS 'SQLQuery1.sql' window shows a script. It starts with a PRINT statement, followed by a BEGIN TRY block containing two INSERT statements. The first INSERT statement succeeds, but the second one fails due to a conversion error (attempting to cast 'TDF' to a numeric value). A blue arrow points from the SSMS window to the results window on the right. The results window shows the successful insertion of the first record (CustomerID 5612, CustomerName 'Tejinder', City 'Jalandhar', Country 'India', PostalCode '144001') and the failure of the second record (CustomerID 5613, CustomerName 'Steve', City 'Melbourne', Country 'Australia', PostalCode '2005'). A message at the bottom of the results window states 'Query executed successfully.'

```
PRINT 'BEFORE TRY'
BEGIN TRY
    PRINT 'First Statement'
    INSERT INTO Supplier (SupplierId, SupplierName, ContactNo)
    VALUES (8947, 'KMY', 8565698423)
    UPDATE Supplier
    SET ContactNo = 9857154623
    WHERE SupplierId = 8947
    INSERT INTO Supplier (SupplierId, SupplierName, ContactNo)
    VALUES (9865, 'TDF', 7841589145)
    PRINT 'Last Statement'
END TRY
BEGIN CATCH
    PRINT 'In CATCH Block'
END CATCH
```

CustomerID	CustomerName	City	Country	PostalCode
1	5612	Tejinder	Jalandhar	India 144001
2	5613	Steve	Melbourne	Australia 2005

Query executed successfully.

Error-Handling

System Error Functions

- ✓ **Error_Message**
- ✓ This function returns the text message of the error that caused the CATCH block

- ✓ **Syntax:**

`ERROR_MESSAGE()`

- ✓ **Return Types:**

`nvarchar(4000)`

Error-Handling

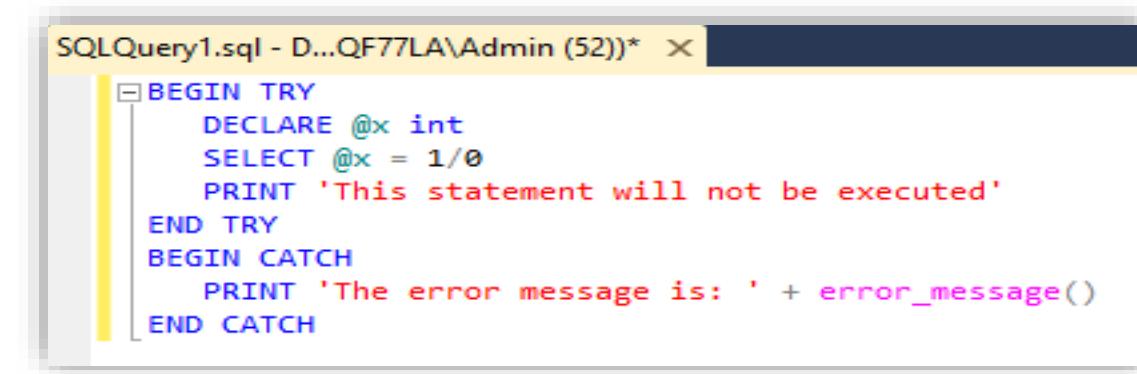
System Error Functions

- ✓ **Return Value**
- ✓ When called in a CATCH block, ERROR_MESSAGE returns the error message text that caused the CATCH block to run
- ✓ The text comprises the values supplied for any substitutable parameters
- ✓ For instance, object names, lengths, or times
- ✓ When called outside the scope of the CATCH block, ERROR_MESSAGE returns NULL

Error-Handling

System Error Functions

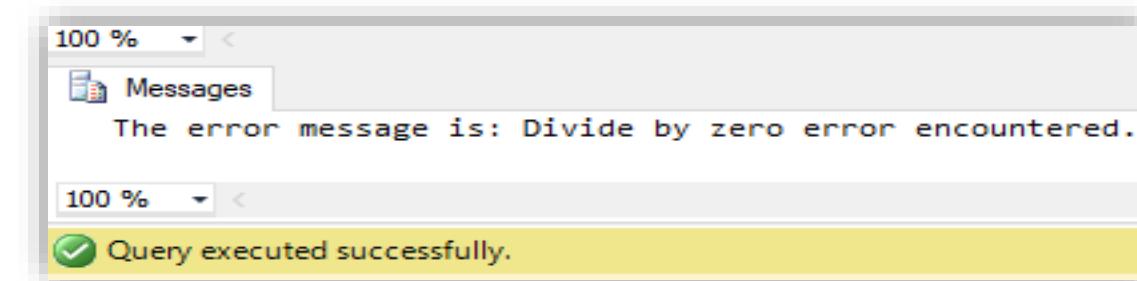
- ✓ Using **ERROR_MESSAGE** in a **CATCH** block
- ✓ The provided example shows a SELECT statement that generates a divide-by-zero error
- ✓ The error message returns by the CATCH block



```
SQLQuery1.sql - D...QF77LA\Admin (52)* ×
BEGIN TRY
    DECLARE @x int
    SELECT @x = 1/0
    PRINT 'This statement will not be executed'
END TRY
BEGIN CATCH
    PRINT 'The error message is: ' + error_message()
END CATCH
```



Output:



100 % < Messages
The error message is: Divide by zero error encountered.

100 % < Status
Query executed successfully.

Error-Handling

Custom Error Messages

- ✓ Custom error messages enable us to design: business-specific messages, the methods to manage these scenarios, and the advanced logging systems for error review

- ✓ Every custom error message has a severity assignment, which determines the significance of the error and recognises how it should be handled

- ✓ Few of the error messages are merely informational and are not even captured by the error handling

- ✓ Other error messages are very critical and instantly kill the process on which the statement was performed

Error-Handling

Custom Error Messages

- ✓ **Defining Custom Error Messages**
- ✓ We can use the stored procedure `sp_addmessage` to define a custom error message, which adds a record to the `sys.messages` system view
- ✓ We need to provide a severity level, an error number, and the error message to execute this stored procedure

Error-Handling

Custom Error Messages

- ✓ **Example:** The given code snippet is defining three types of custom error messages

```
USE master  
GO
```

```
EXEC sp_addmessage 50001, 1, N'This message is not that big of a deal. This is  
not caught by error handling, and prints this message to the screen.';  
EXEC sp_addmessage 50002, 16, N'This actually causes an error, and is caught by  
error-handling';  
EXEC sp_addmessage 50003, 20, N'This causes an error, and stops any further  
processing. This is not caught by error handling.';
```

Error-Handling

Custom Error Messages

- ✓ **Using the Custom Error Messages**
- ✓ The following snippet uses RAISERROR inside of a TRY CATCH construct

The image shows a screenshot of SQL Server Management Studio. On the left, a query window titled "SQLQuery1.sql - D...QF77LA\Admin (52)*" contains the following T-SQL code:

```
BEGIN TRY
    RAISERROR (50001,1,1) WITH LOG
END TRY
BEGIN CATCH
    SELECT ERROR_MESSAGE(), ERROR_NUMBER()
END CATCH
```

A large blue arrow points from the query window to the right. On the right, a "Messages" window displays the results of the execution:

- The top message area shows "Command(s) completed successfully."
- The bottom status bar shows "Query executed successfully."

Error-Handling

Custom Error Messages

- ✓ Example 2

The screenshot shows two windows from SQL Server Management Studio. The top window is titled "SQLQuery1.sql - D...QF77LA\Admin (52)*" and contains the following T-SQL code:

```
BEGIN TRY
    RAISERROR (50002,16,1) WITH LOG
END TRY
BEGIN CATCH
    SELECT ERROR_MESSAGE(),ERROR_NUMBER()
END CATCH
```

The bottom window is a results grid with two tabs: "Results" and "Messages". The "Results" tab is selected and shows the following output:

	(No column name)	(No column name)
1	Error 50002, severity 16, state 1 was raised, b...	18054

A yellow status bar at the bottom of the results grid displays the message "Query executed successfully."

Error-Handling

Custom Error Messages

✓ Example 3

The screenshot shows two windows from SQL Server Management Studio. The top window is titled "SQLQuery1.sql - DE...2) - not connected*" and contains the following T-SQL code:

```
BEGIN TRY
    RAISERROR (50003,20,1) WITH LOG
END TRY
BEGIN CATCH
    SELECT ERROR_MESSAGE (), ERROR_NUMBER()
END CATCH
```

The bottom window is titled "Messages" and displays the following error message:

```
Msg 18054, Level 16, State 1, Line 2
Error 50003, severity 20, state 1 was raised, but no message with that error number was found in sys.messages. If error is larger than 50000, make
```

Error-Handling

The Obsolete @@Error Function

- ✓ It returns the error number for the last SQL statement executed
- ✓ Syntax:

@@ERROR

- ✓ Return Types:

integer

Error-Handling

The Obsolete @@Error Function

- ✓ It will return 0 if the previous SQL statement encountered no errors
- ✓ If the previous statement encountered an error, then it will return an error number
- ✓ If the failure was one of the errors in the sys.messages catalog view, then @@ERROR comprises the value from the sys.messages.message_id column for that error
- ✓ A user can view the text related to an @@ERROR error number in sys.messages, because @@ERROR is cleared and reset on every statement executed

Error-Handling

The Obsolete @@Error Function

- ✓ Example: Using @@ERROR to identify a specific error
- ✓ In the example, we are using @@ERROR to check for a check constraint violation (error #547) in an UPDATE statement

The screenshot shows two windows from SQL Server Management Studio. The left window is titled 'SQLQuery1.sql - D...QF77LA\Admin (52)*' and contains the following T-SQL code:

```
UPDATE Customer
SET CustomerId = '5613'
WHERE CustomerName = 'Steve'

IF @@ERROR = 547
PRINT 'A Check constraint violation occurred.'
```

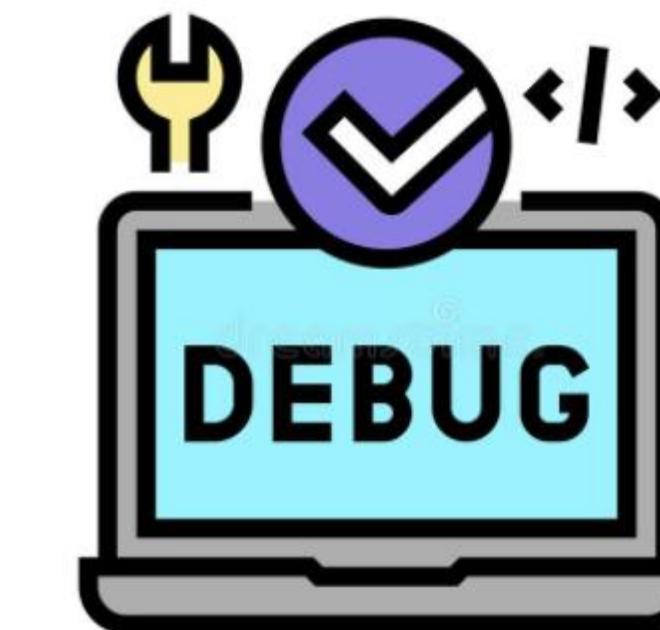
A large blue arrow points from the left window to the right window. The right window is titled 'Messages' and displays the following output:

```
100 % < 
Messages
(1 row(s) affected)
100 % <
Query executed successfully.
```

Error-Handling

The SQL Server Debugger

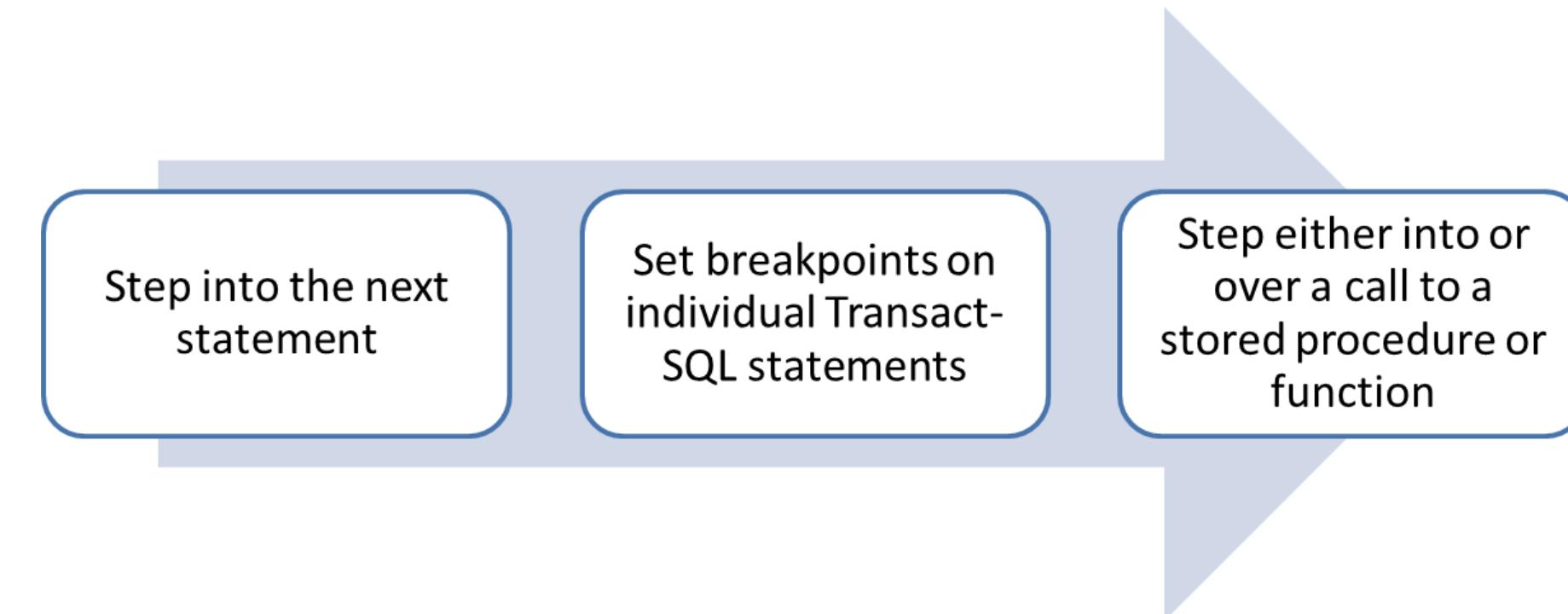
- ✓ The SQL debugger enables us to find the errors in SQL code by examining the run-time behaviour of the code
- ✓ After setting the Database Engine Query Editor window to debug mode, we can pause execution on particular lines of code and examine the information and data that is used by or returned by those SQL statements



Error-Handling

The SQL Server Debugger

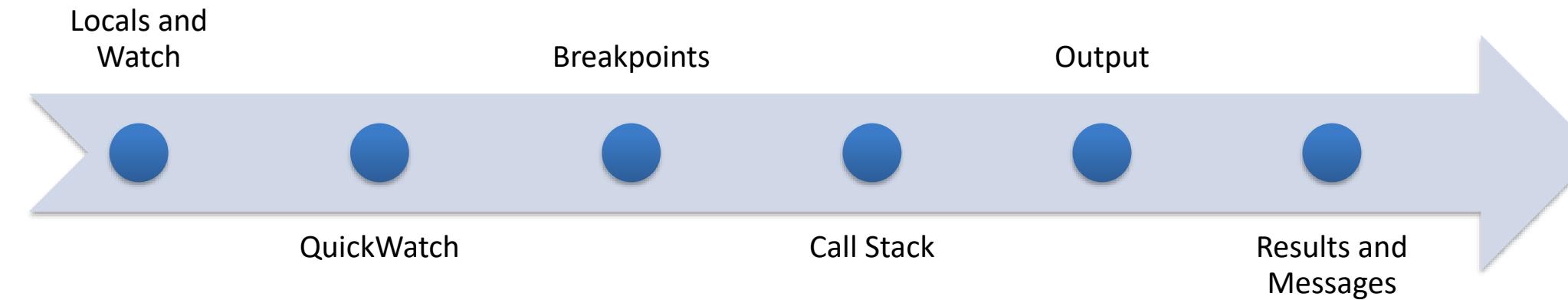
- ✓ Stepping Through Transact-SQL Code
- ✓ SQL debugger renders the some options (given-below) that can be utilised to navigate through Transact-SQL code when the Database Engine Query Editor window is in debug mode:



Error-Handling

The SQL Server Debugger

- ✓ Viewing Debugger Information
- ✓ The debugger will pause the execution on a particular SQL statement. To view the current execution state, use the stated debugger windows:





Congratulations

Congratulations on completing this module!

Contact Us

info@theknowledgeacademy.com

www.theknowledgeacademy.com/tickets

<https://uk.trustpilot.com/review/theknowledgeacademy.com>

theknowledgeacademy