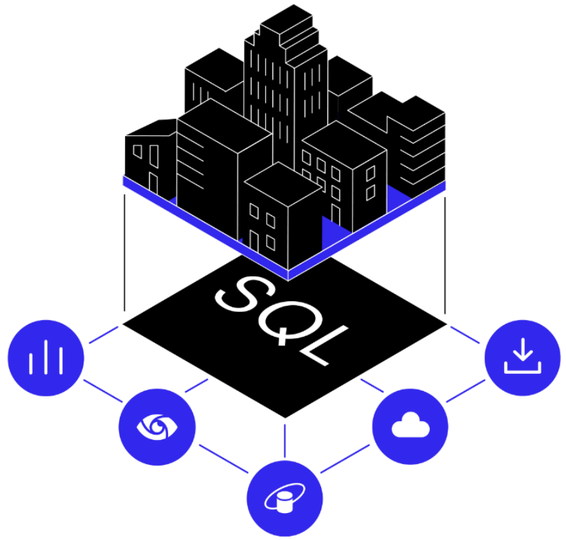
theknowledgeacademy

# 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

theknowledgeacademy

# 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

theknowledgeacademy

# Module 1


## Description

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

# Creating Tables

## Creating Tables in SQL

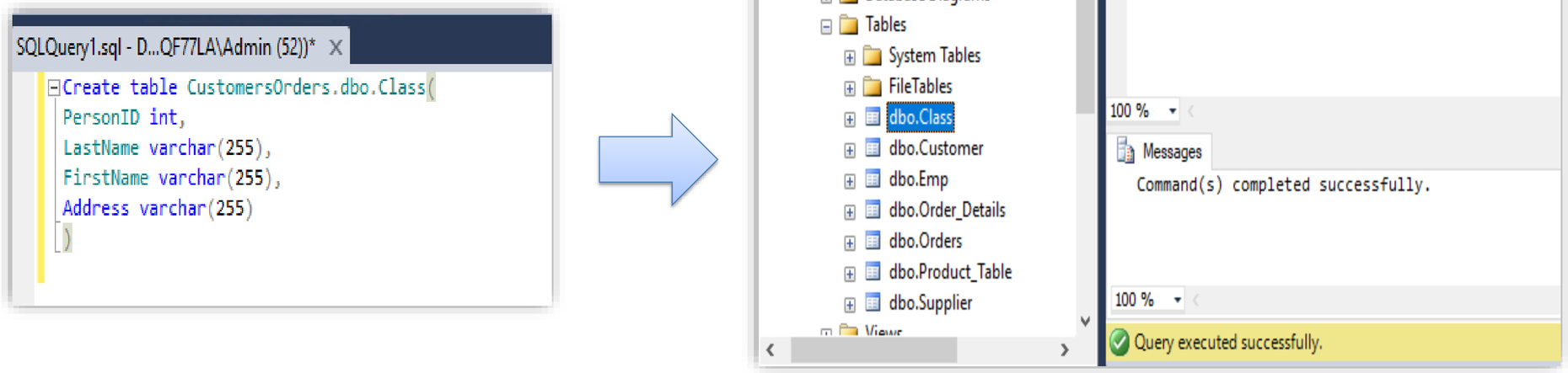
- ✓ The following commands are the syntax for creating table in SQL:

```
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**

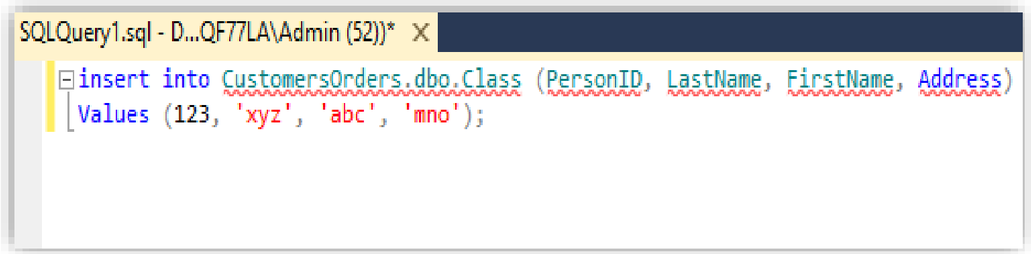
- ✓ The following are the syntax of insert data query in SQL :

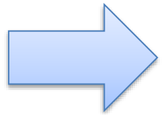
```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

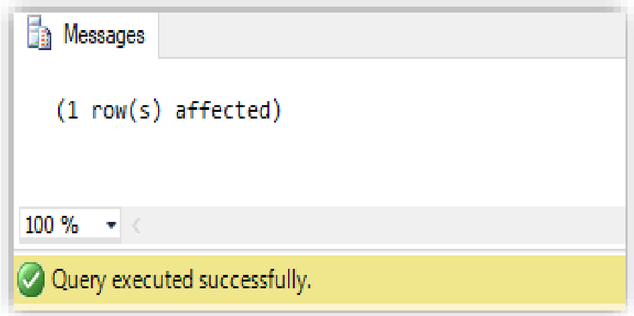
# Creating Tables

## Inserting Data in SQL

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



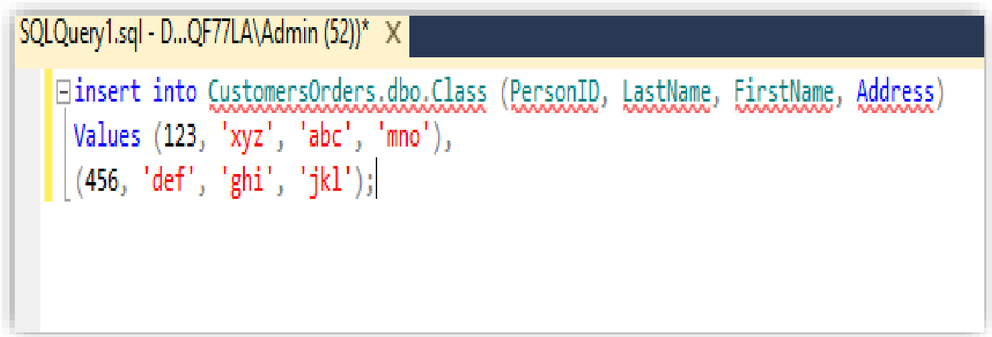



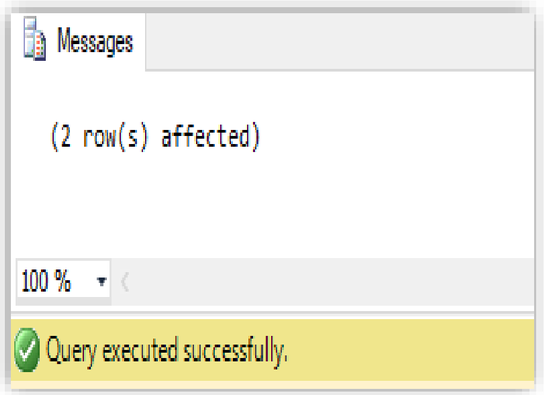


# Creating Tables

## Inserting Multiple Rows in SQL

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

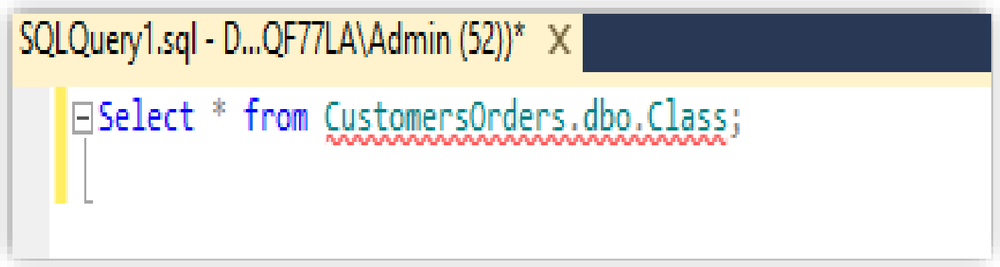


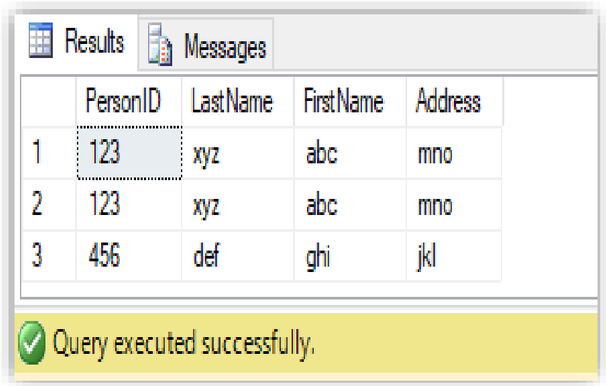



# Creating Tables

## View Table in SQL

- ✓ Execute the following query to view the table:



theknowledgeacademy

# Module 2


## 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
```

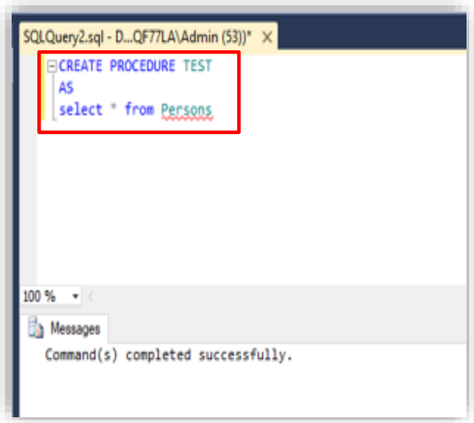
# Stored Procedure Basics

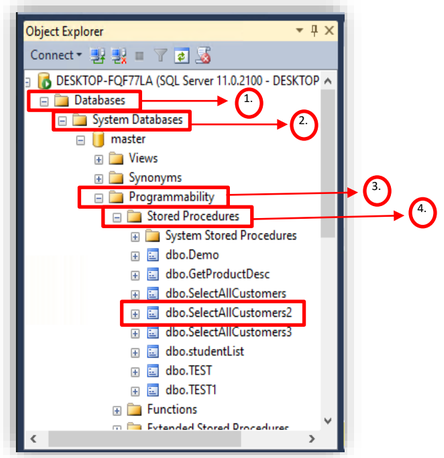
## Creating Stored Procedures

- ✓ 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:

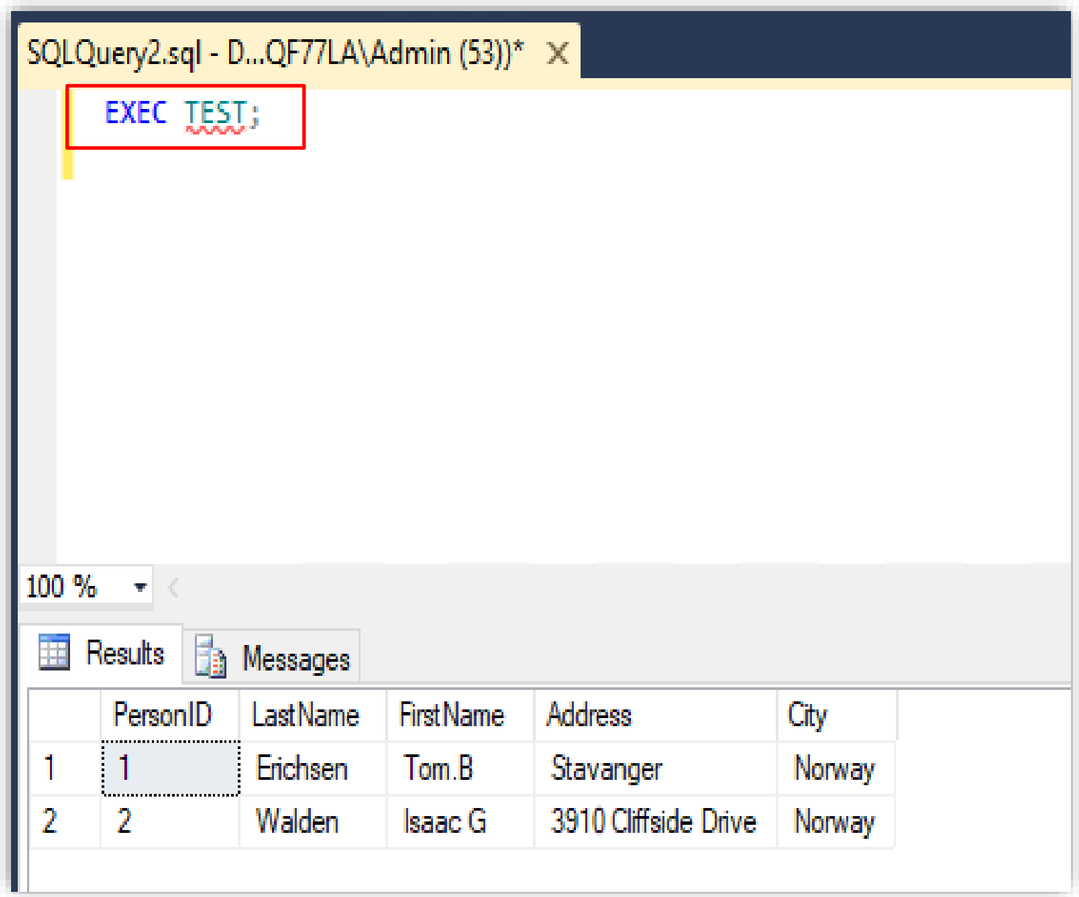


# Stored Procedure Basics

## Creating Stored Procedures

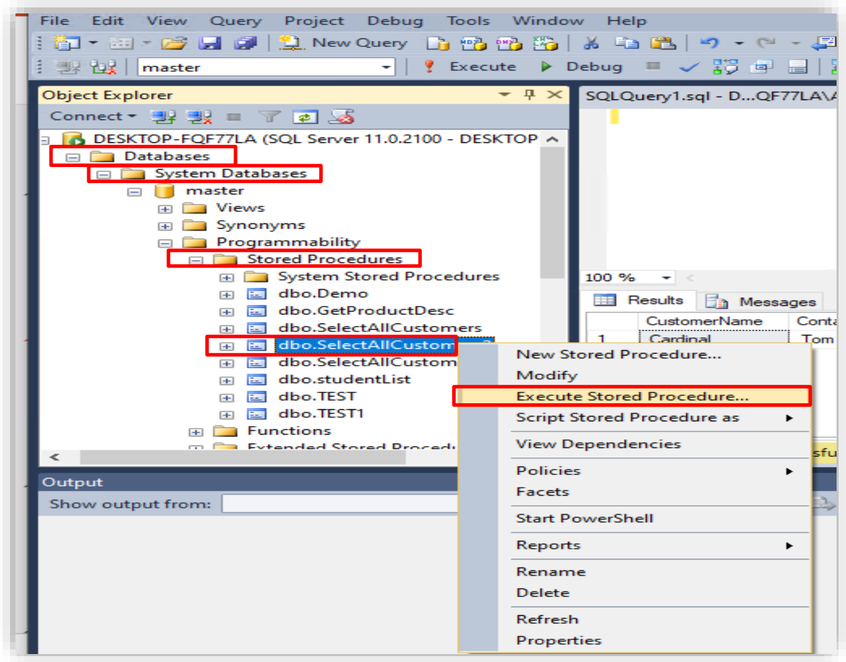
- ✓ Execute the stored procedure:



# Stored Procedure Basics

**Two ways to Execute**

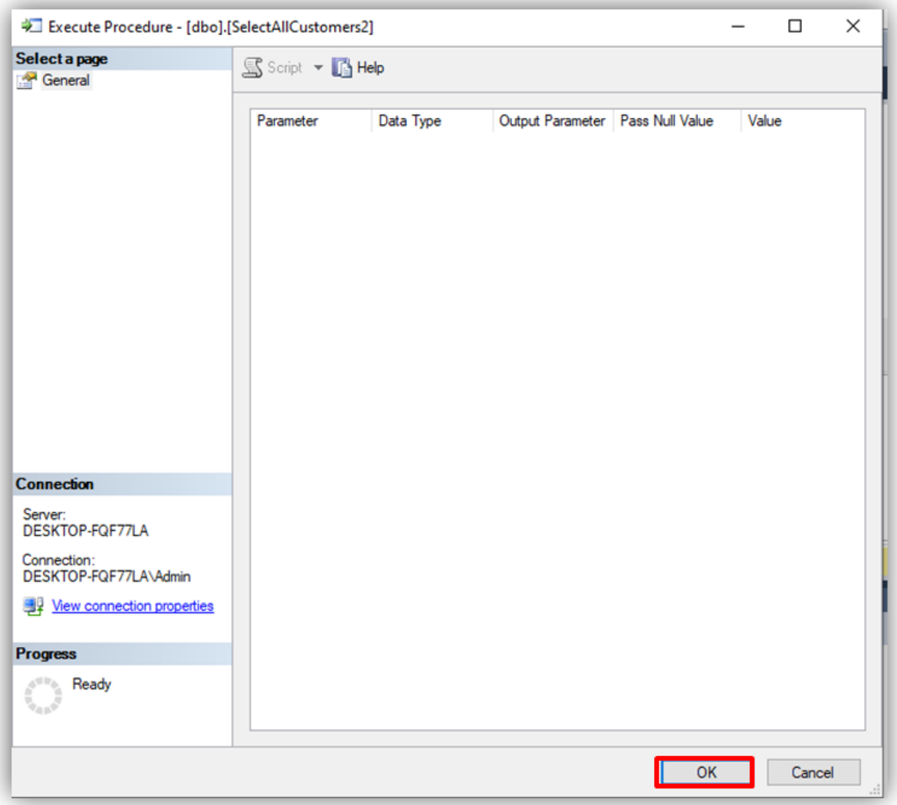
- ✓ There are three ways to execute the stored procedure:
- ✓ Using Object Explorer
- ✓ Step 1: Go to Database > System Database > Programmability > Stored Procedure. Right- click on the procedure, then click on the Execute Stored Procedure…

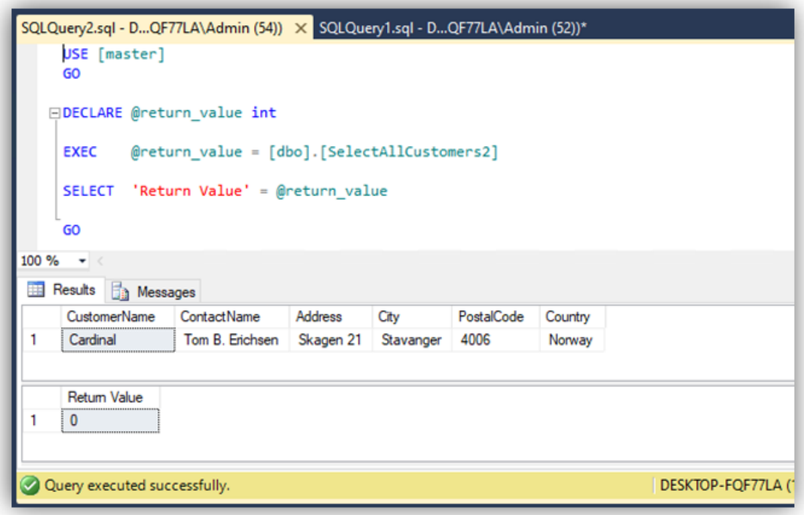


# Stored Procedure Basics

## Two ways to Execute

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

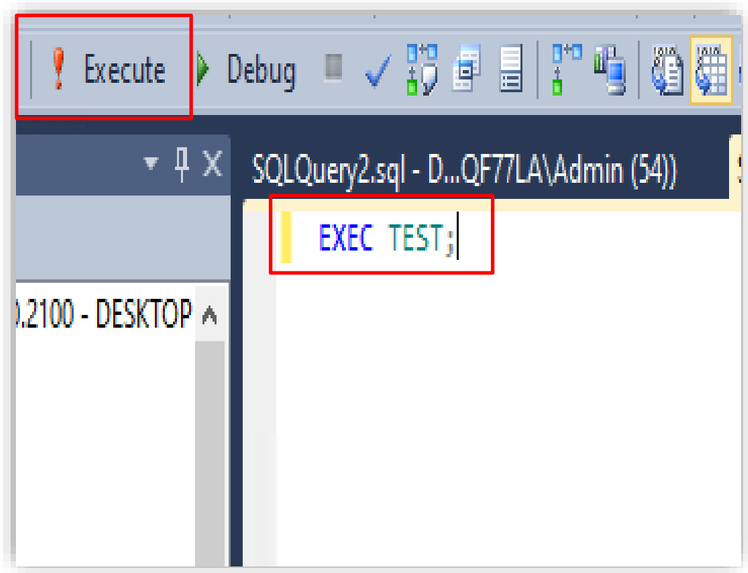


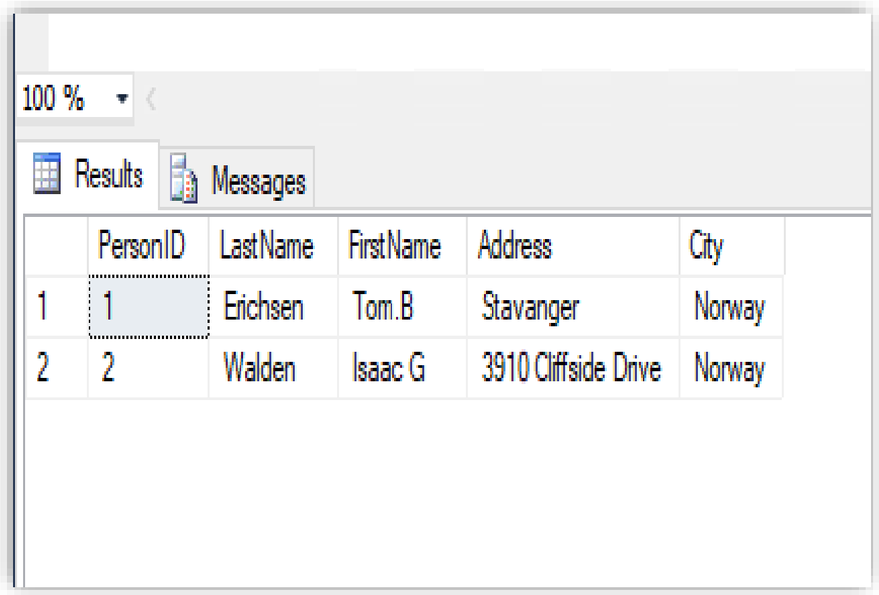



# Stored Procedure Basics

**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

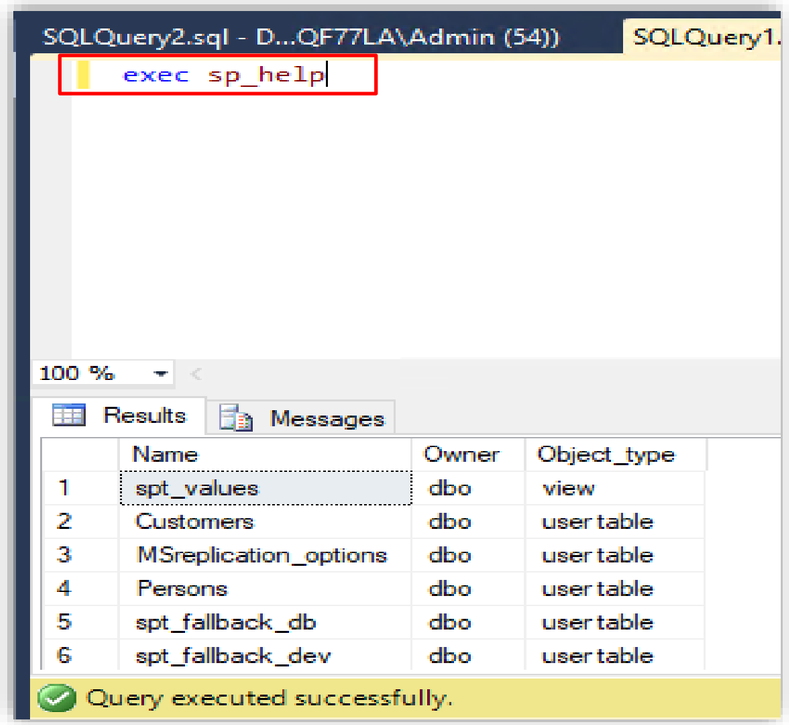
**System Stored Procedures**

- ✓ 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

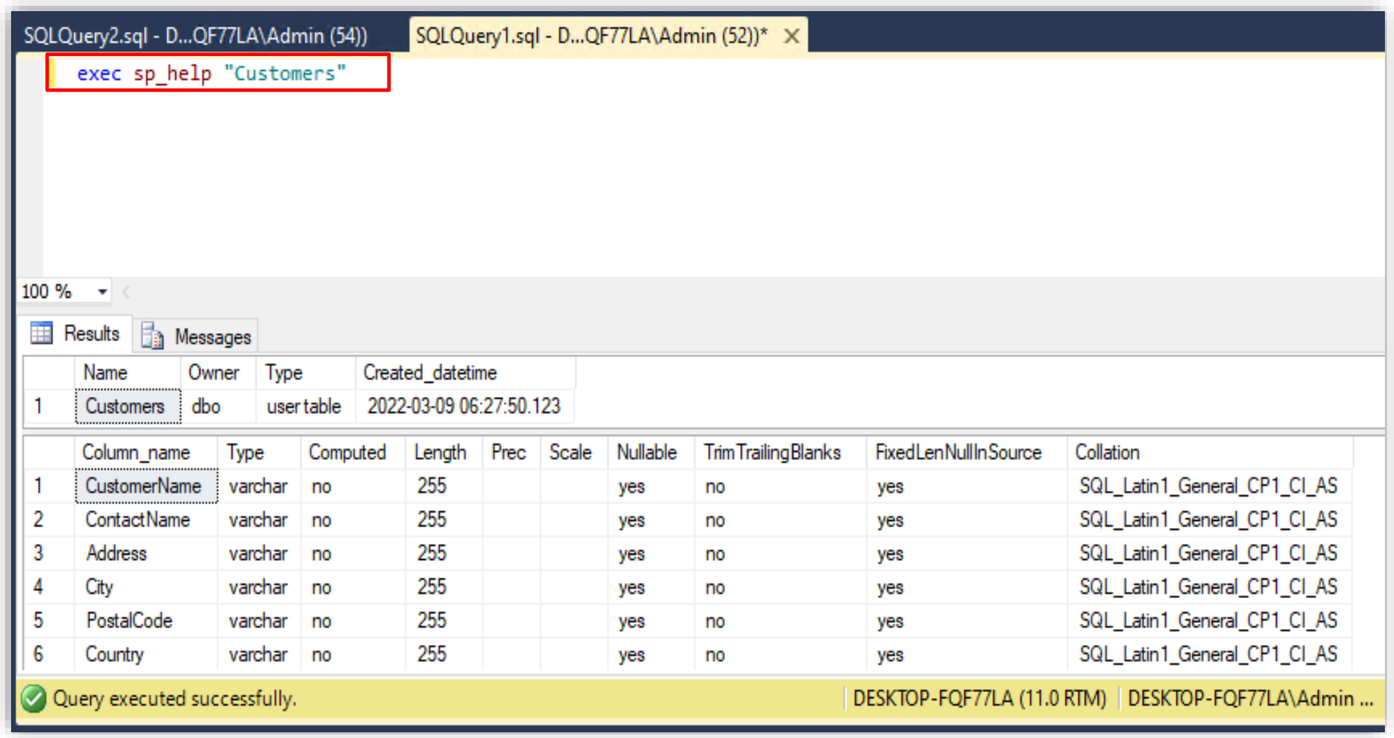
- ✓ Write the below statement to find all database object information:



# Stored Procedure Basics

## System Stored Procedures

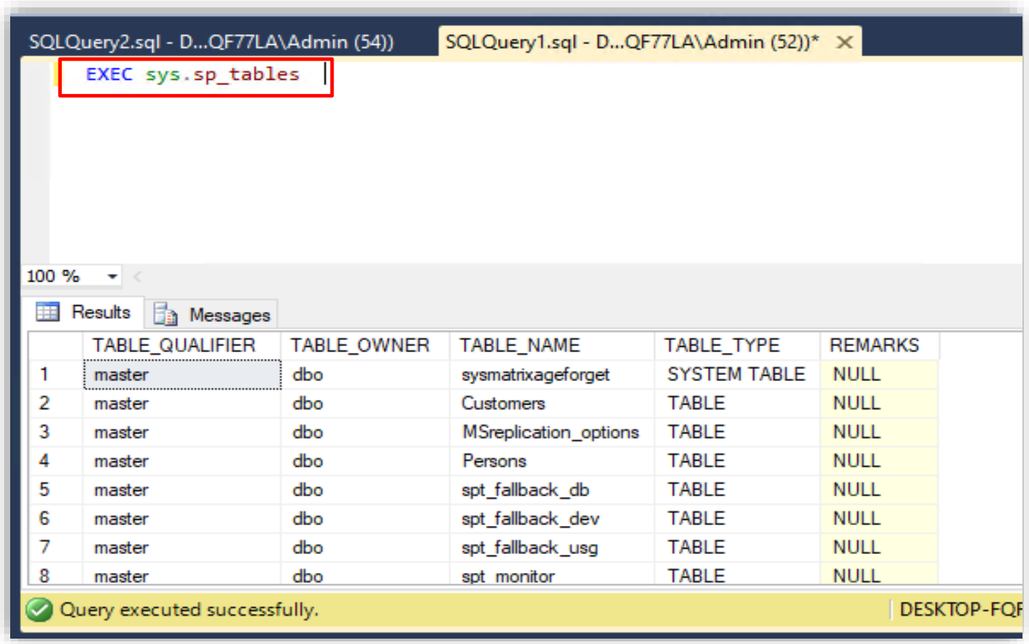
- ✓ To Find Table Information:



# 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

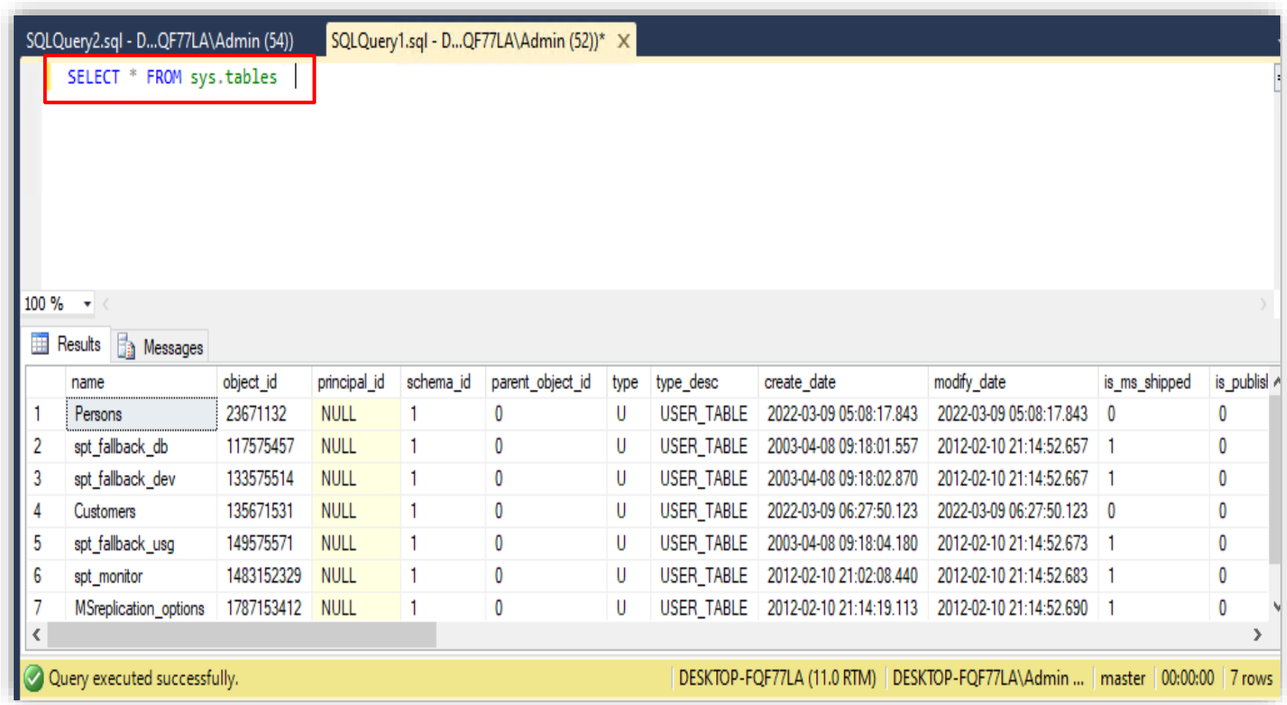



| | TABLE_QUALIFIER | TABLE_OWNER | TABLE_NAME | TABLE_TYPE | REMARKS |
|---|---|---|---|---|---|
| 1 | master | dbo | sysmatrixageforget | SYSTEM TABLE | NULL |
| 2 | master | dbo | Customers | TABLE | NULL |
| 3 | master | dbo | MSreplication_options | TABLE | NULL |
| 4 | master | dbo | Persons | 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 |

# Stored Procedure Basics

## System Stored Procedures

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

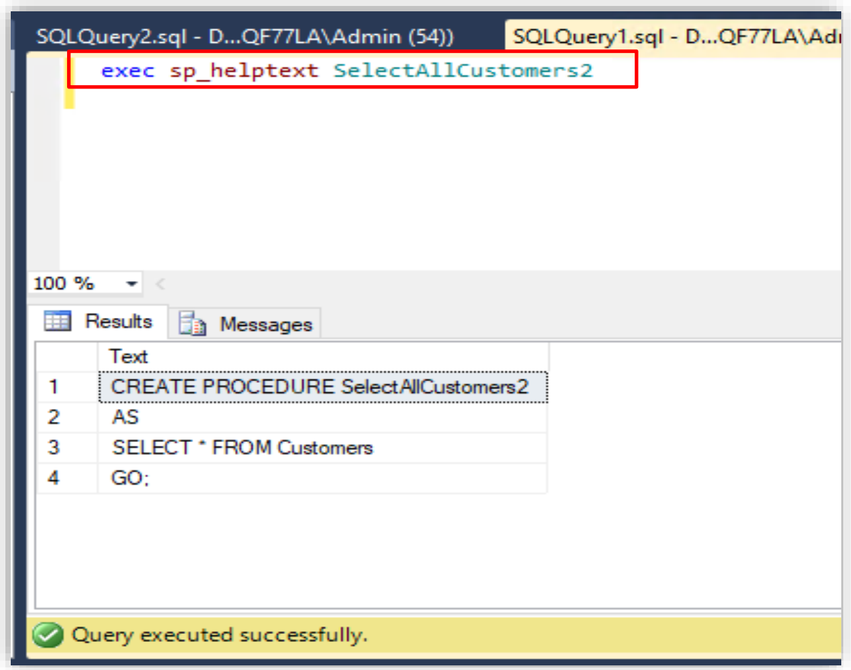



| | name | object_id | principal_id | schema_id | parent_object_id | type | type_desc | create_date | modify_date | is_ms_shipped | is_publisl |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Persons | 23671132 | NULL | 1 | 0 | U | USER_TABLE | 2022-03-09 05:08:17.843 | 2022-03-09 05:08:17.843 | 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 |

# Stored Procedure Basics

**System Stored Procedures**

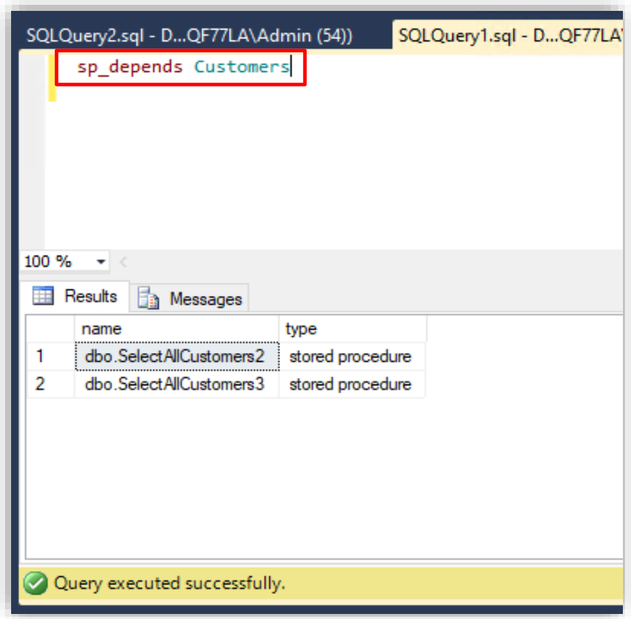
- ✓ **sp_helptext <Object Name>**
- ✓ The sp_helptext displays the definition that is used to generate an object in the multiple rows. The given example displays the definition of the procedure SelectAllCustomers2 in the master database



# 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



theknowledgeacademy

# Module 3


## Description

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

# Variables

**Declaring Variables**

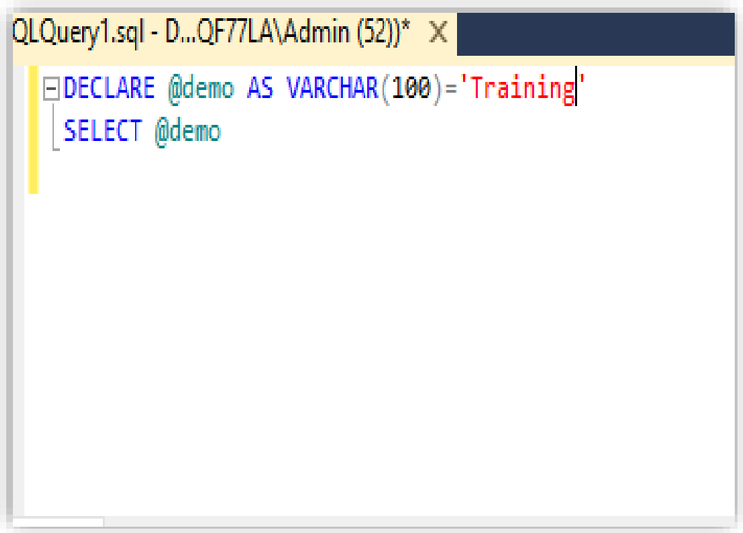
- ✓ During the code execution, a variable permits a programmer to store the data temporarily
- ✓ **Syntax for declaring a variable:**
  - ✓ A user can use DECLARE to create a variable for use. The syntax:

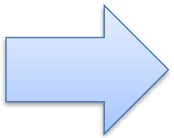
```
DECLARE @VariableName AS datatype
```

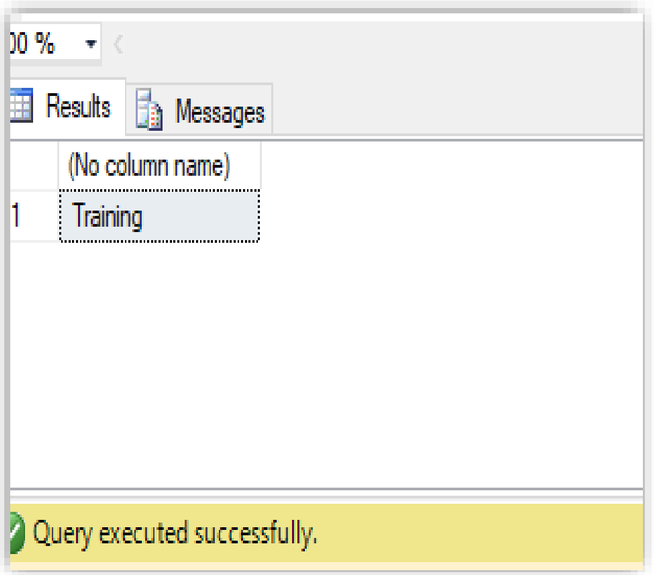
# Variables

## Declaring Variables

- ✓ Example:





# 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<br><br>SET @Index = 1<br>SET @LoopCount = 10<br>SET @InitialValue = 5 | A user can assign values to more than one variable at a time<br><br>SELECT @Index = 1, @LoopCount = 10,<br>@InitialValue = 5 |

# Variables

**SET versus SELECT**

| SET | SELECT |
| --- | --- |
| When assigning from a query and the query returns no result, SET will assign a NULL value to the variable<br><br>DECLARE @CustomerID NCHAR(5)<br>SET @CustomerID = 'XYZ'<br>SET @CustomerID = (SELECT [CustomerID]<br>FROM [dbo].[Customers]<br>WHERE [CustomerID] = 'ABC')<br>SELECT @CustomerID -– Returns NULL | 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<br><br>DECLARE @CustomerID NCHAR(5)<br>SET @CustomerID = 'XYZ'<br>SELECT @CustomerID = [CustomerID]<br>FROM [dbo].[Customers]<br>WHERE [CustomerID] = 'ABC'<br>SELECT @CustomerID –- Returns XYZ |

# Variables

**SET versus SELECT**

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

# 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 on 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 | 0 (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 | • 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<br>• 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 |

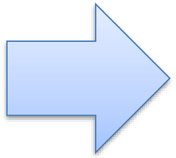
# Variables

**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))*

```
Declare @Name varchar(30)
Begin
Select @Name=CustomerName  from CustomersOrders.dbo.Customer where Customerid=5604
Set @Name= 'Customer Name:'+@Name
Print @Name
End
```



Messages

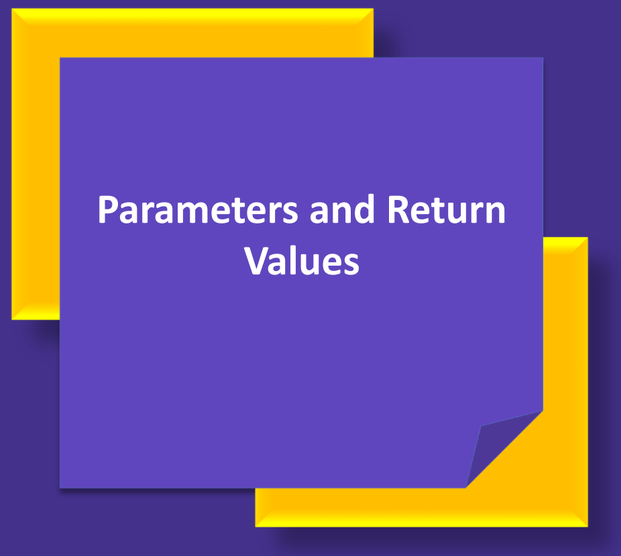
```
Customer Name:Rajan Paul
```

100 %

Query executed successfully.

theknowledgeacademy

# Module 4



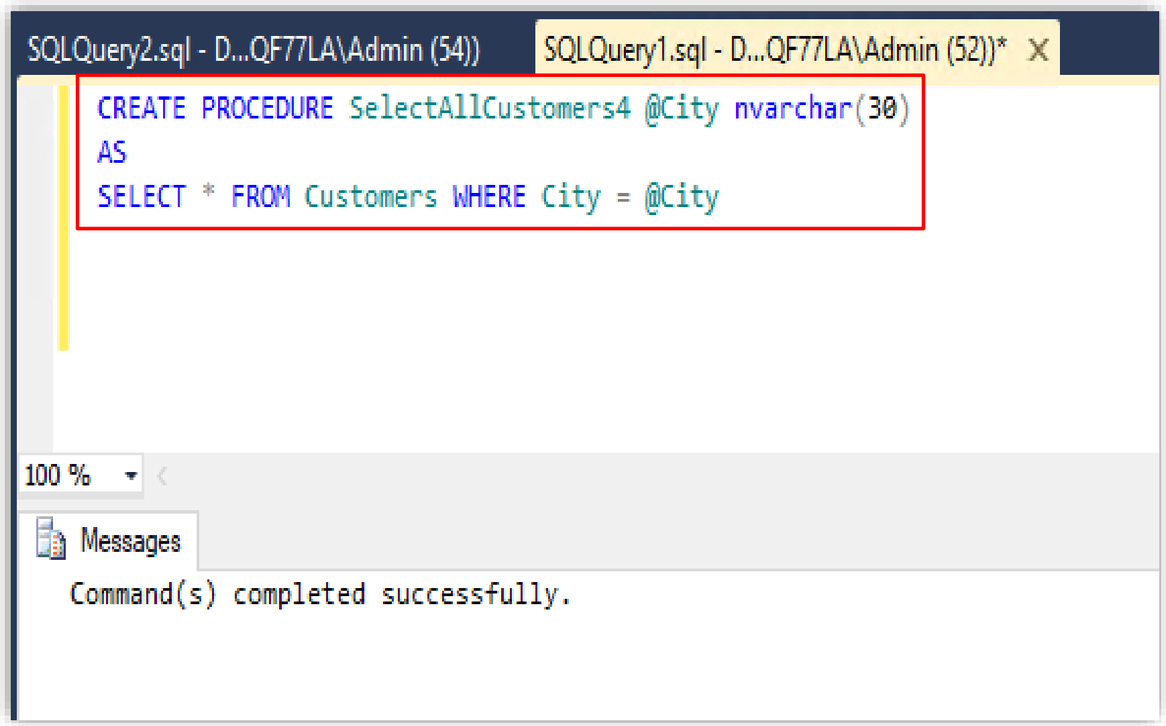

## Description

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

# Parameters and Return Values

**Passing Parameters**

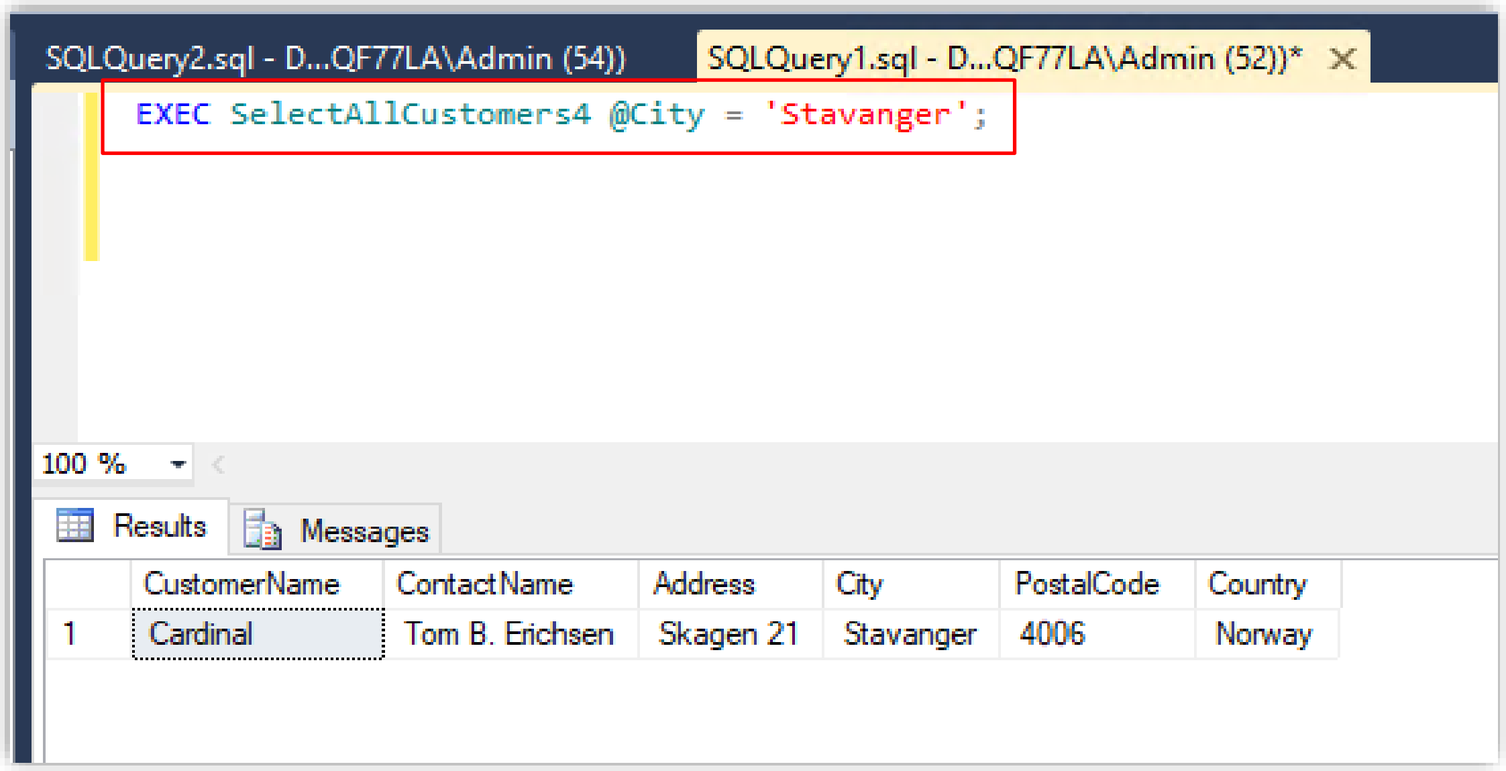
- ✓ Stored Procedure With One Parameter
  - ✓ The below SQL statement creates a stored procedure that selects Customers from a particular City from the "Customers" table:



# Parameters and Return Values

**Passing Parameters**

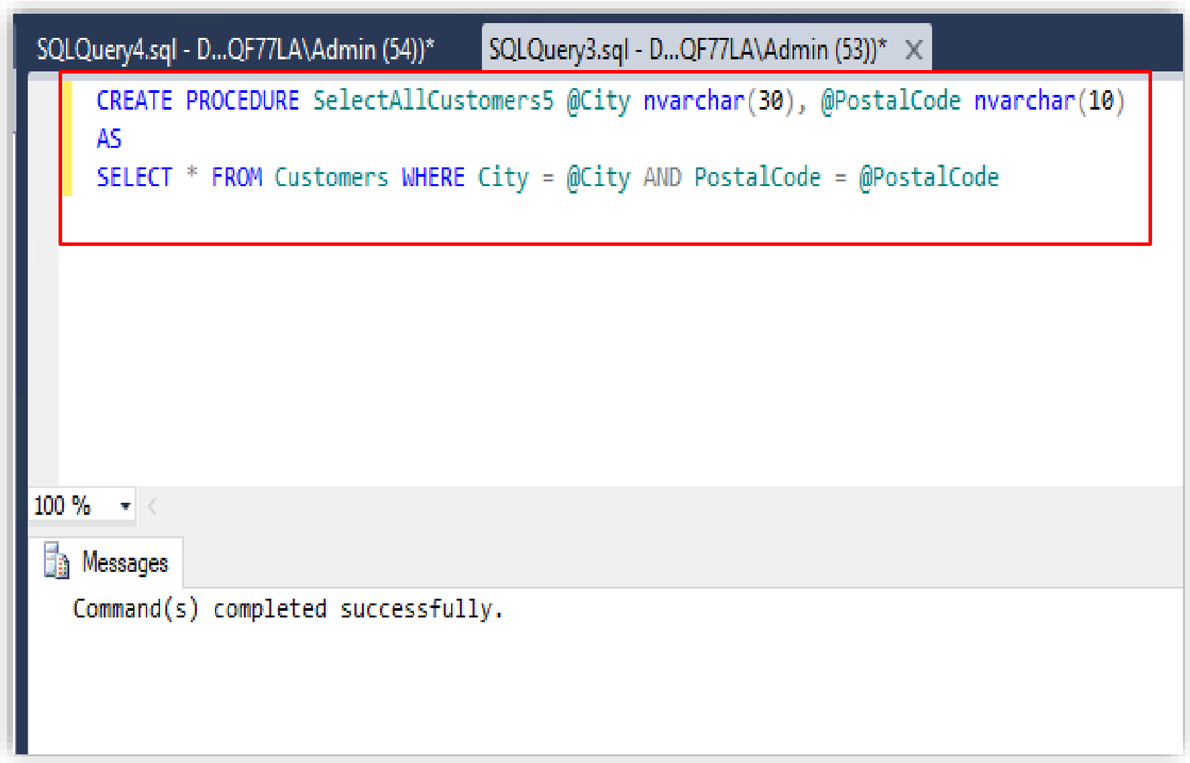
- ✓ Execute the stored procedure:



# Parameters and Return Values

## Passing Parameters

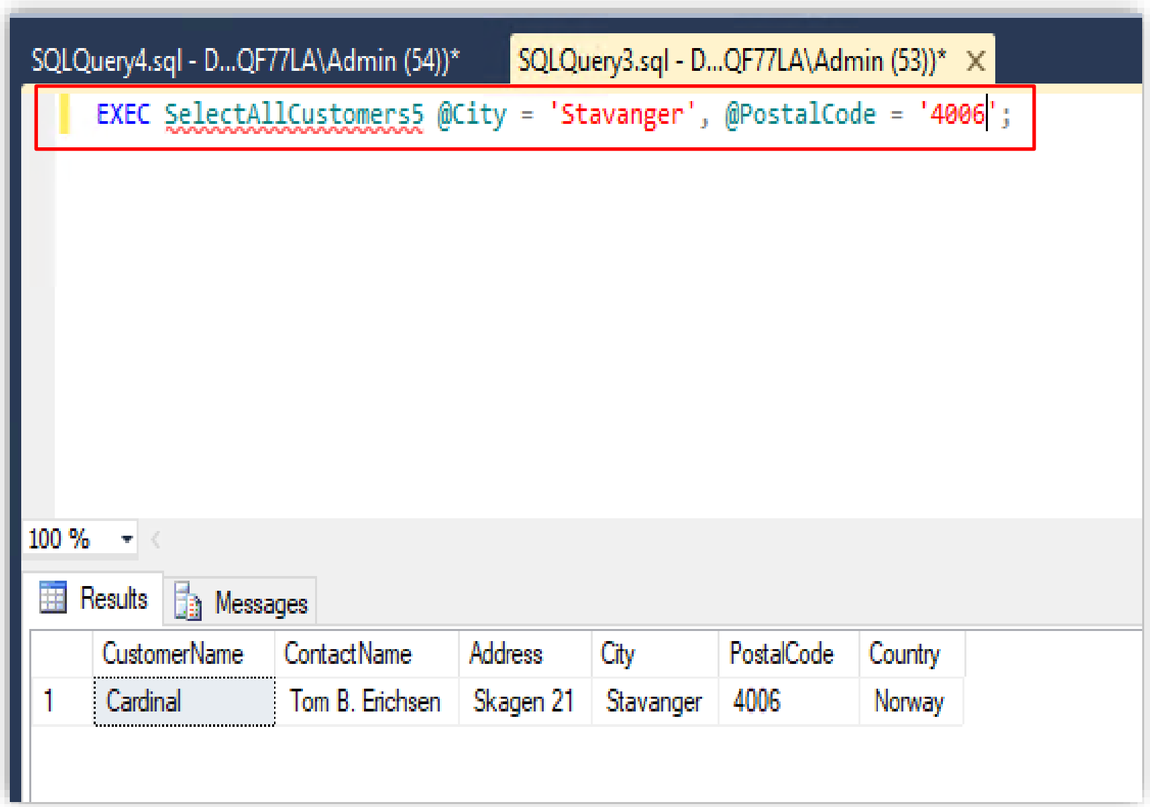
- ✓ It's simple to set up multiple parameters. Simply separate each parameter and data type with a comma, as seen below:



# Parameters and Return Values

## Passing Parameters

- ✓ Execute the stored procedure:

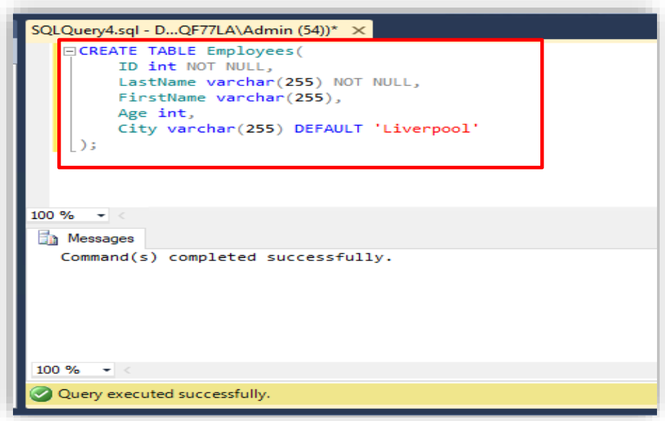


# Parameters and Return Values

## 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

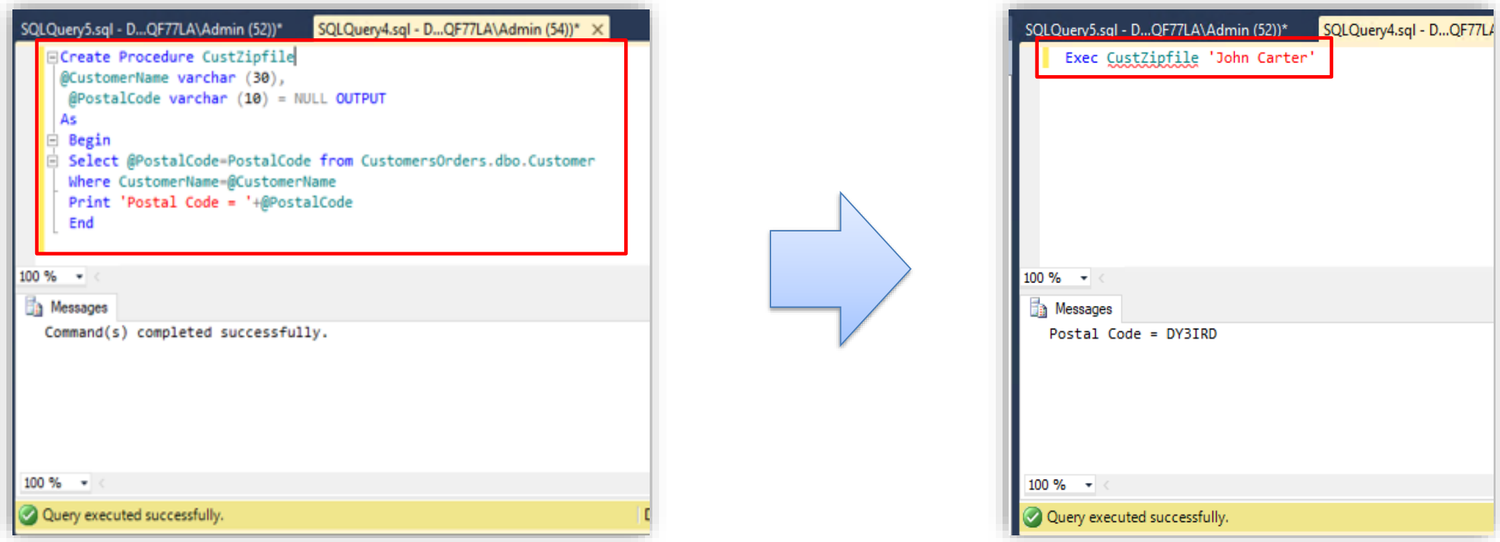
- **SQL DEFAULT on CREATE TABLE**
  - The below SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:



# 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:

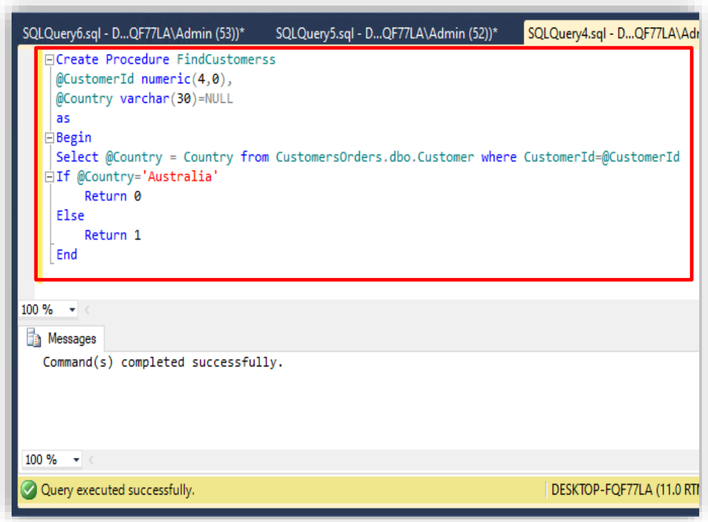


# Parameters and Return Values

## 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:

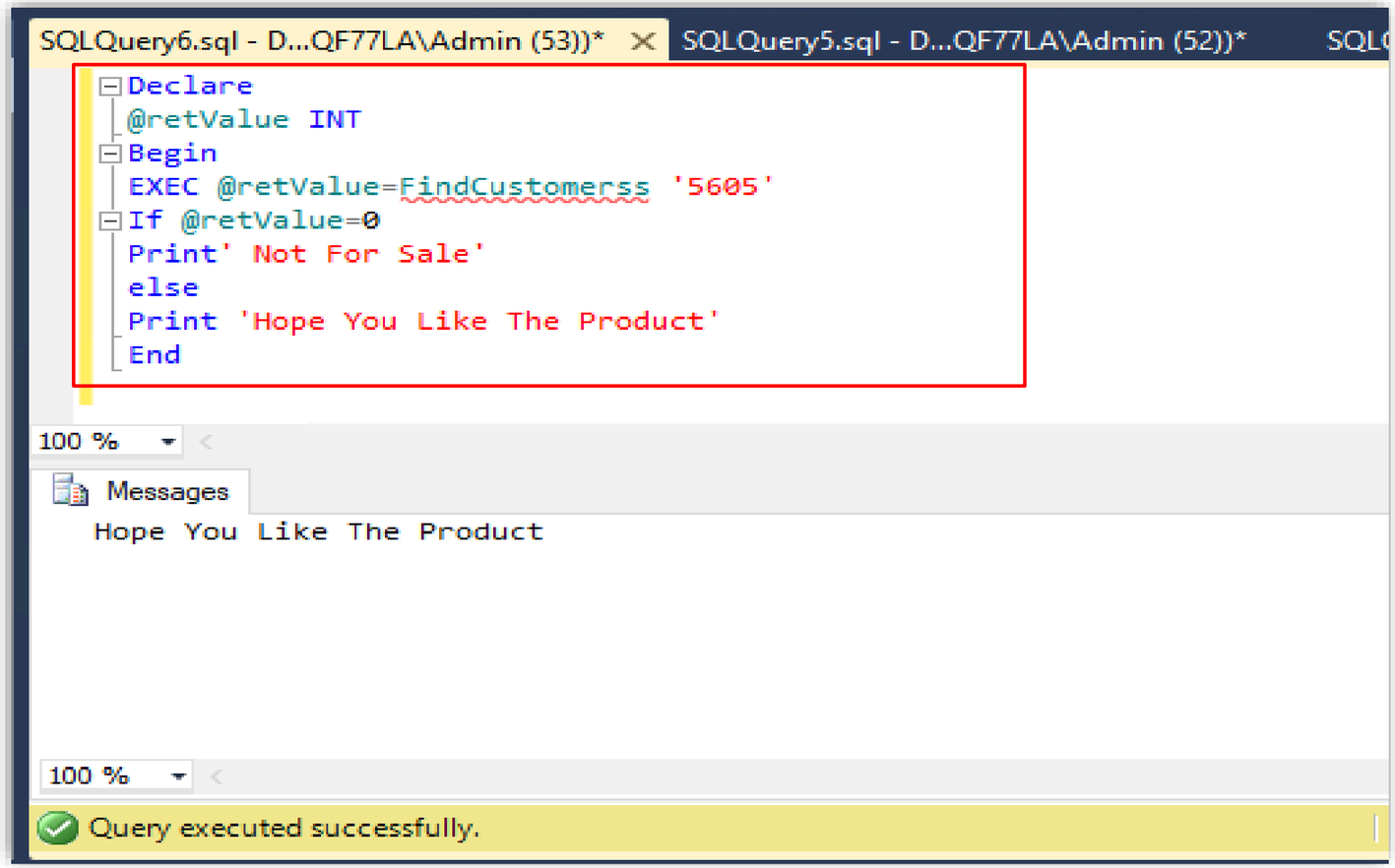
```
RETURN [ integer_expression ]
```



# Parameters and Return Values

## Using Return

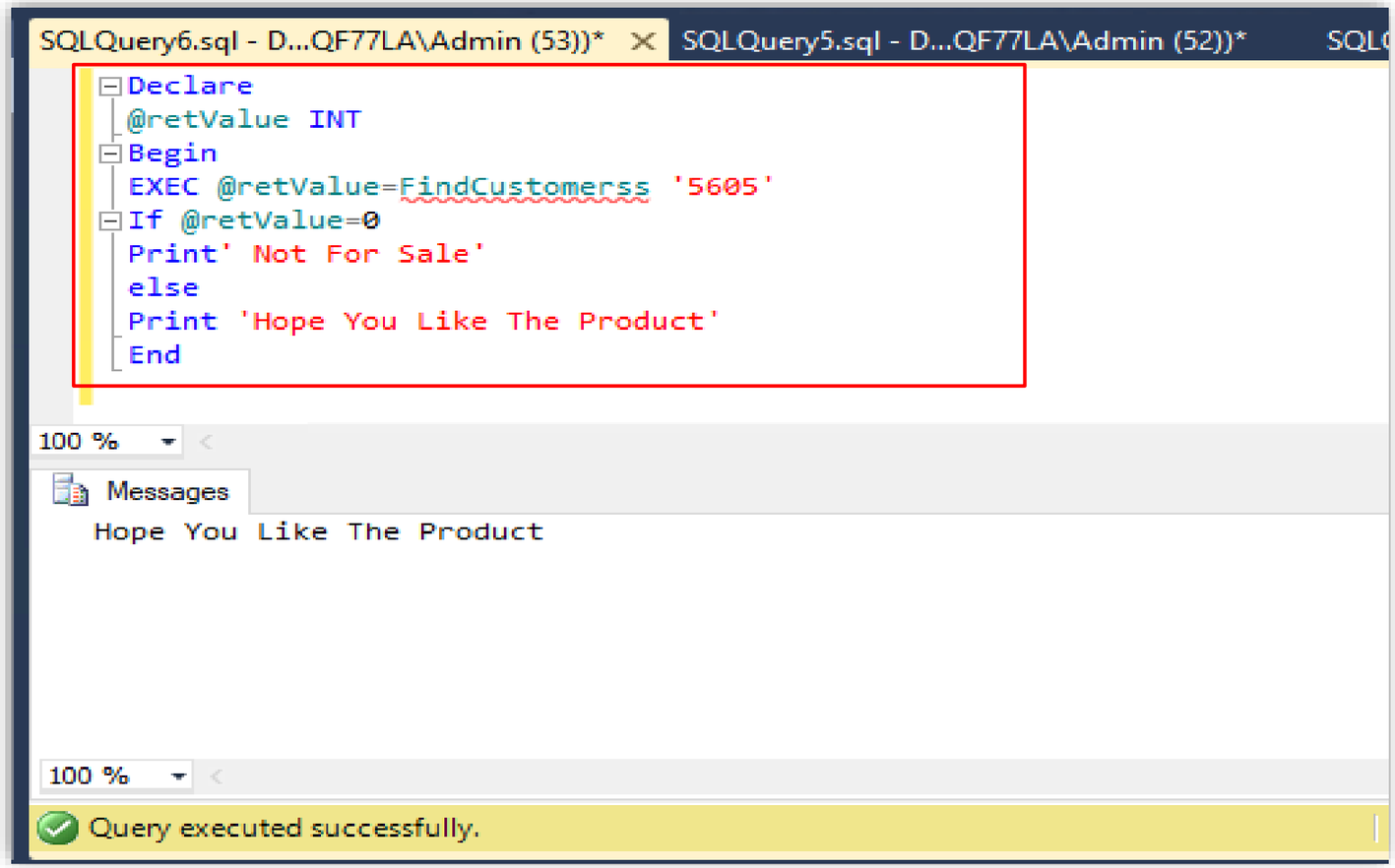
- ✓ Using the Return Value:



# Parameters and Return Values

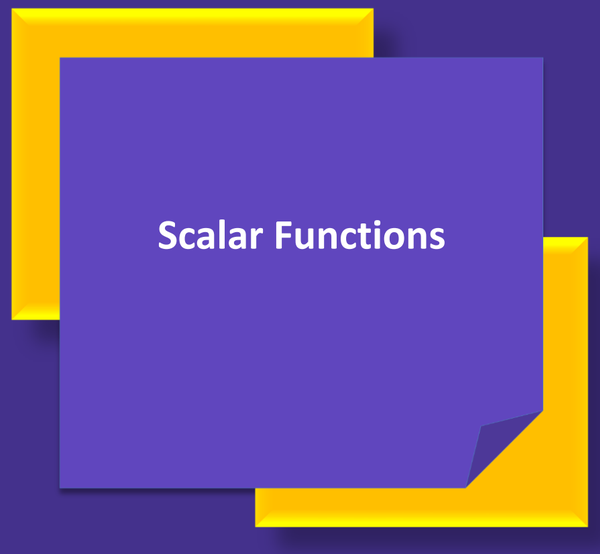
## Using Return

- ✓ Using the Return Value:



theknowledgeacademy

# Module 5




## 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

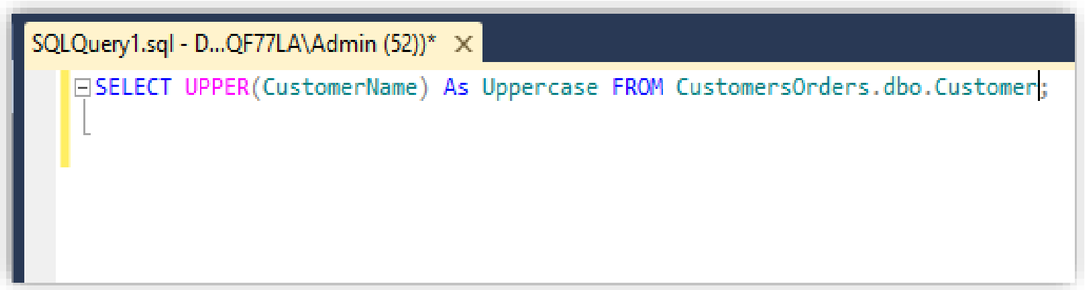
- ✓ Syntax:

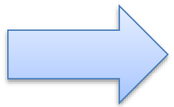
```
SELECT UPPER(column_name) from table-name;
```

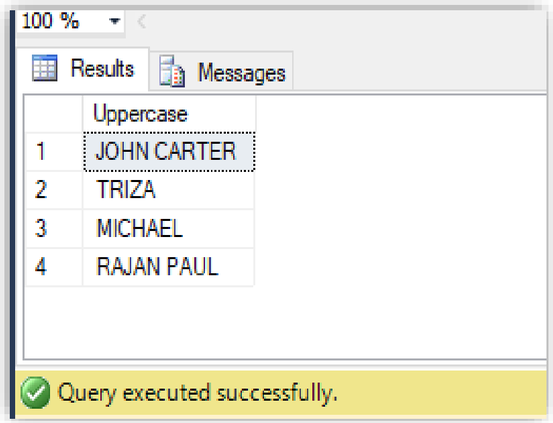
# Scalar Functions

**UCASE()**

- ✓ Using UCASE() function
- ✓ Syntax:





# Scalar Functions

**LCASE() function**

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

- ✓ Syntax:

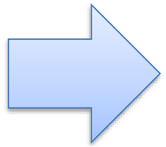
```
SELECT LOWER(column_name) FROM table-name;
```

# Scalar Functions

**LCASE() function**

- ✓ Using LCASE() function
- ✓ Syntax:

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
SELECT LOWER(CustomerName) As Lowercase FROM CustomersOrders.dbo.Customer;
```



100 %

Results | Messages

| | Lowercase |
|---|---|
| 1 | john carter |
| 2 | triza |
| 3 | michael |
| 4 | rajan paul |

Query executed successfully.

# Scalar Functions

**MID() function**

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

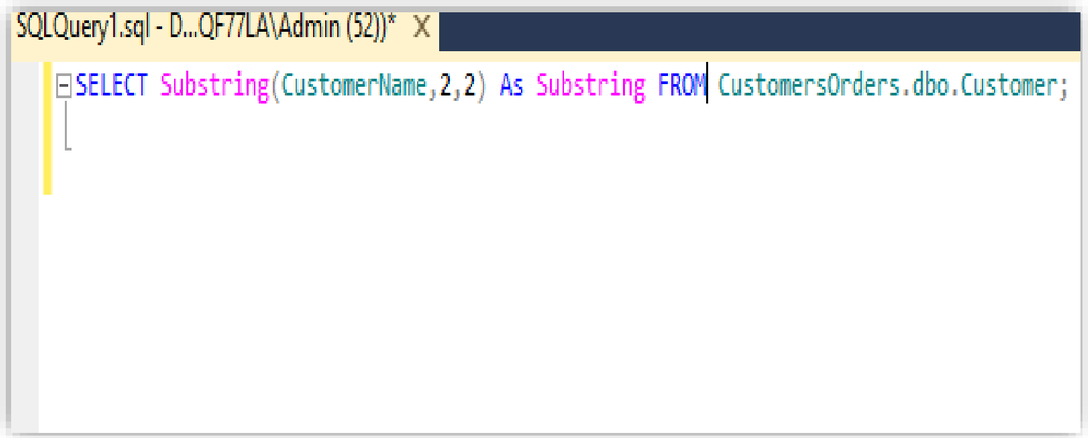
- ✓ Syntax:

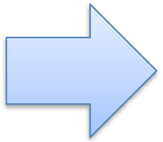
```
SELECT MID(column_name, start, length) from table-name;
```

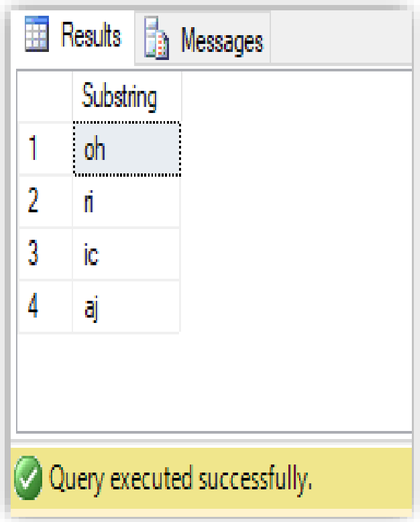
# Scalar Functions

**MID() function**

- ✓ Using MID() function
- ✓ Syntax:





# Scalar Functions

**ROUND() function**

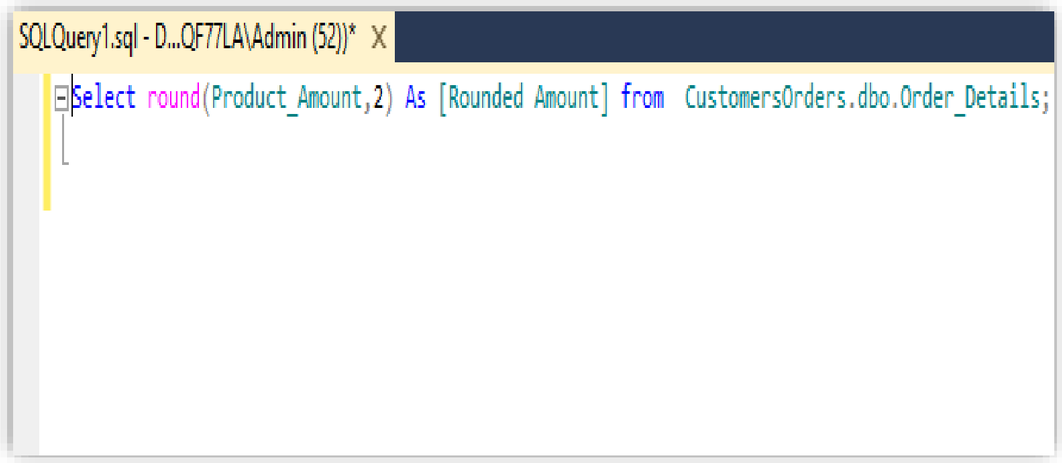
- ✓ This function is used to round a numeric field to the nearest integer
- ✓ It is used on decimal point values
- ✓ Syntax:

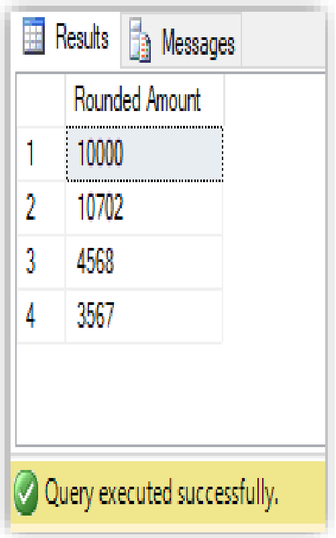
```
SELECT ROUND(column_name, decimals) from table-name;
```

# Scalar Functions

**ROUND() function**

- ✓ Using ROUND() function
- ✓ Syntax:



# Scalar Functions

**LEN() function**

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

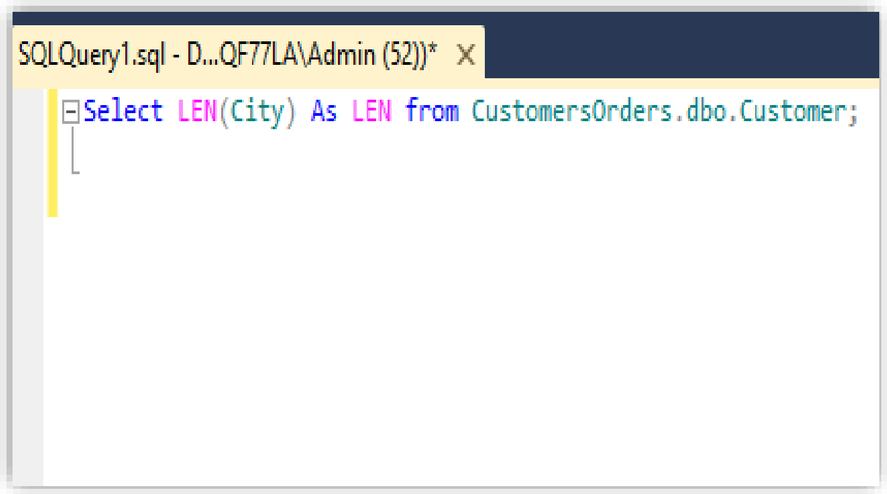
- ✓ Syntax:

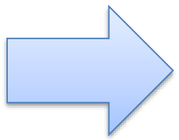
```
SELECT LEN(column_name) FROM table_name;
```

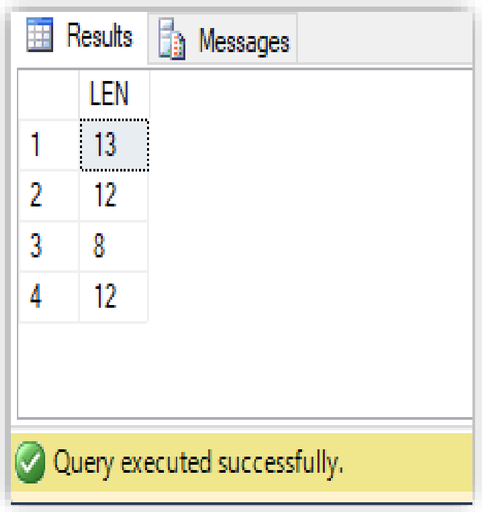
# Scalar Functions

**LEN() function**

- ✓ Using LEN() function
- ✓ Syntax:





# 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))*
Select GETDATE() from CustomersOrders.dbo.Customer;
```

Results | Messages

| | (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:

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
Select FORMAT(OrderDate, 'd', 'en-gb') As Date from  CustomersOrders.dbo.Orders;
```

100 %

Results | Messages

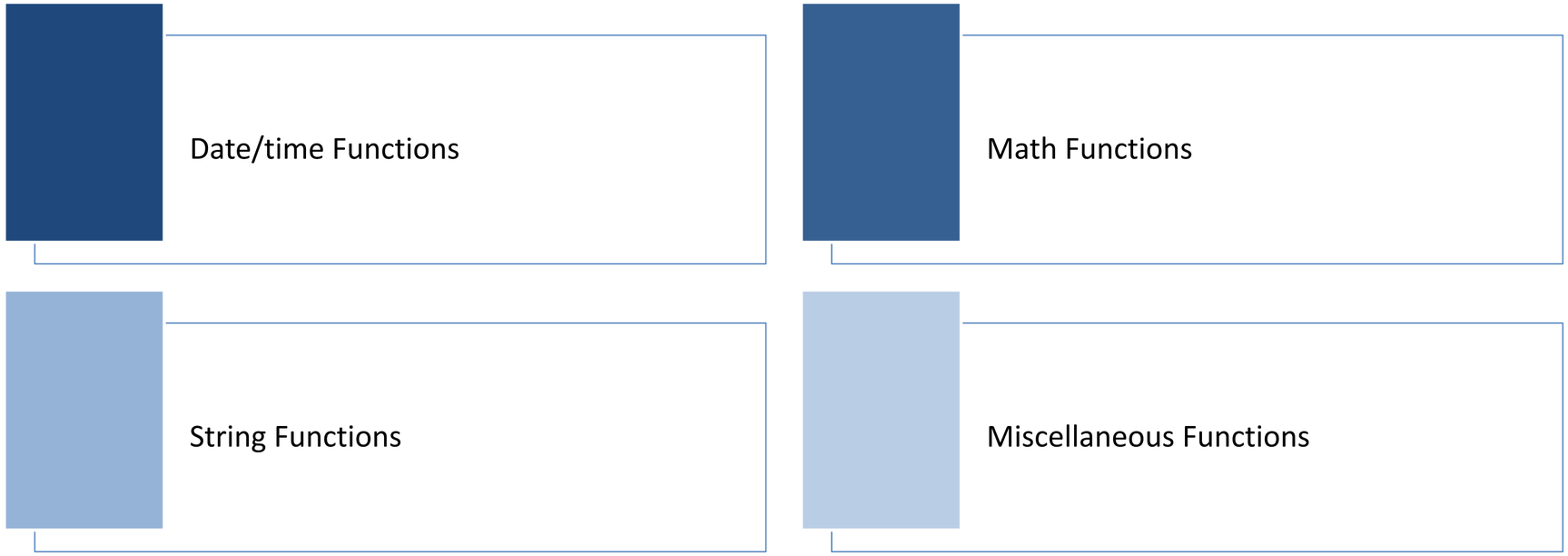
| | Date |
|---|---|
| 1 | 12/12/2017 |
| 2 | 10/01/2018 |
| 3 | 15/12/2017 |
| 4 | 18/01/2018 |

Query executed successfully.

# Scalar Functions

## Advantages of Scalar Functions

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



# 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 | IMESTAMPADD |
| 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 | NEWIDTRING |
| 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

theknowledgeacademy

# Module 6


## 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

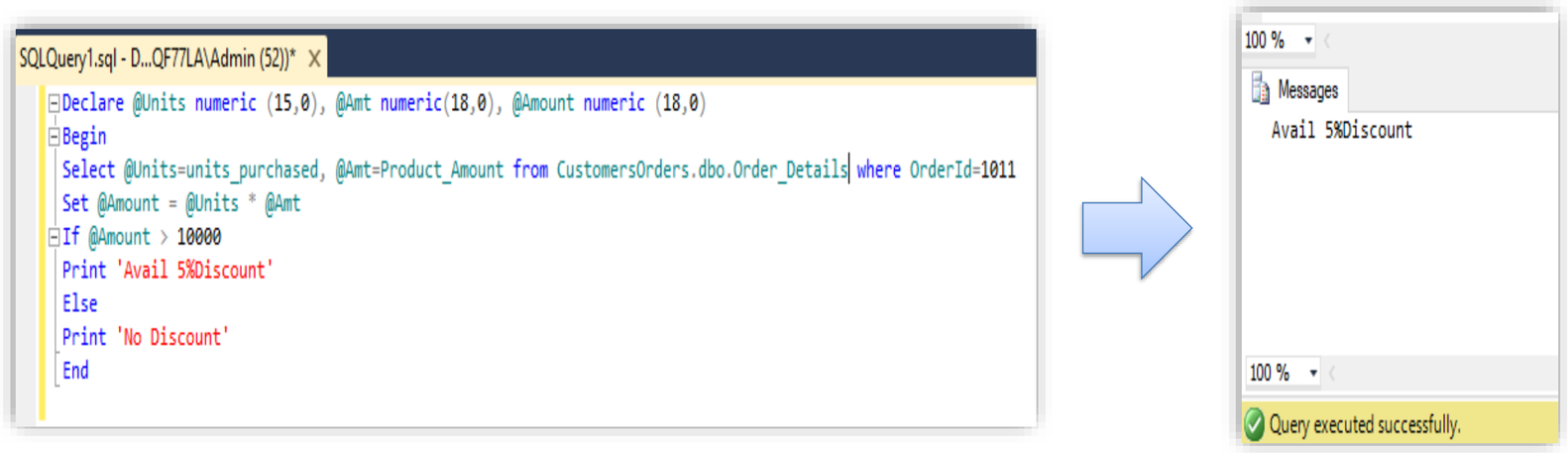
- Syntax:
  - The syntax of IF/ELSE statement in SQL Server is:

```
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



# 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

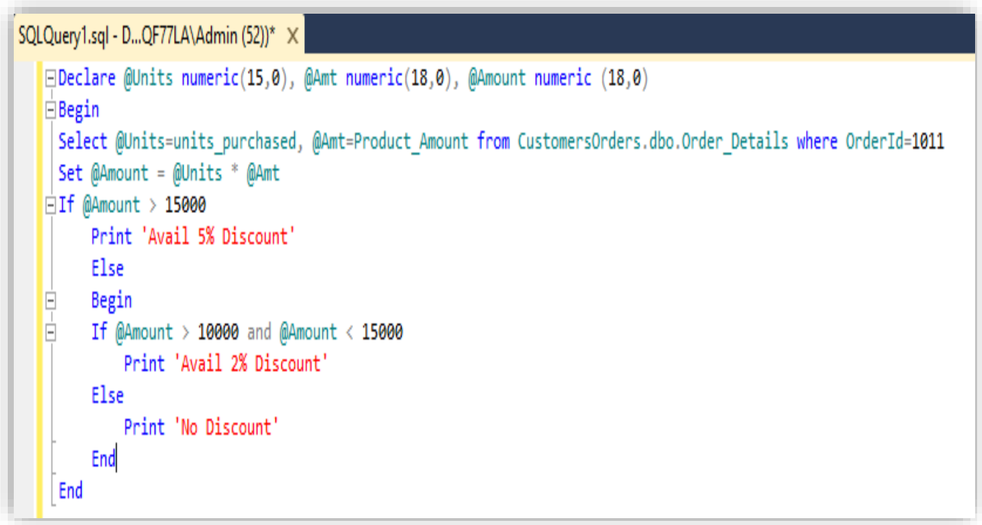
```
SQLQuery1.sql - D...QF77LA\Admin (52))*
Declare @Units numeric (15, 0), @Amt numeric (18, 0), @Amount numeric (18, 0)
Begin
Select @Units=units_purchased, @Amt=Product_Amount from CustomersOrders.dbo.Order_Details where OrderId=1011
Set @Amount = @Units * @Amt
If @Amount > 40000
Print 'Avail 5%Discount'
 End
```

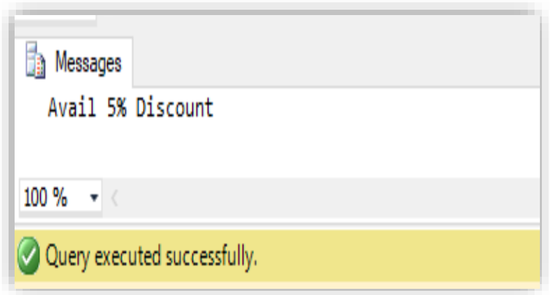
```
100 %
Messages
Command(s) completed successfully.
100 %
Query executed successfully.
```

# 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



# 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

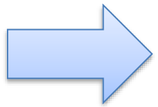
# Testing Conditions

## Using CASE Where Possible

- ✓ Here is a simple example of CASE:

SQLQuery1.sql - D...QF77LA\Admin (52))*

```
Select ProductName,"Availability"=
Case When Qty_Available = 0 Then 'N/A' Else 'Available'
End
From CustomersOrders.dbo.Product_Table
```



Results | Messages

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

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))*
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 |

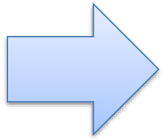
Query executed successfully.

# 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:

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
Select ProductId, Count(*) as TimesSold, AvailDiscount =
Case
When Count(*) > 3 Then 'Discount'
Else
'No Discount'
End
From CustomersOrders.dbo.Product_Table
Group by ProductId
```



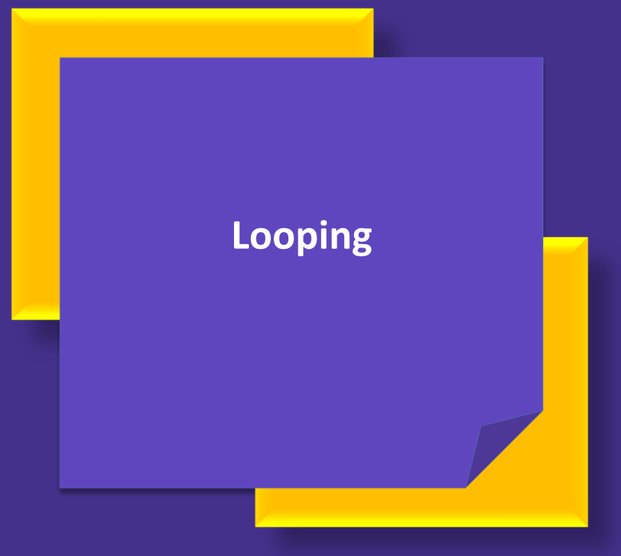
Results | Messages

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

Query executed successfully.

theknowledgeacademy

# 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:

```
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

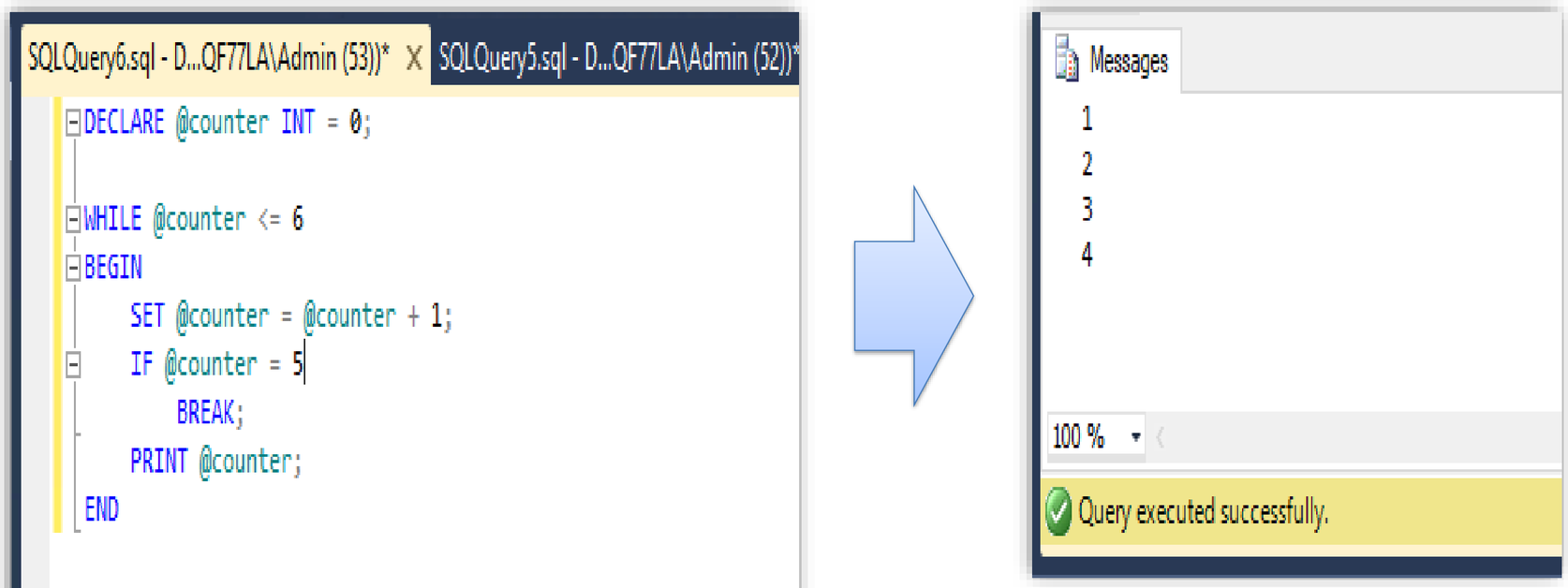
- ✓ Syntax:

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

# Looping

**Breaking out of a Loop**

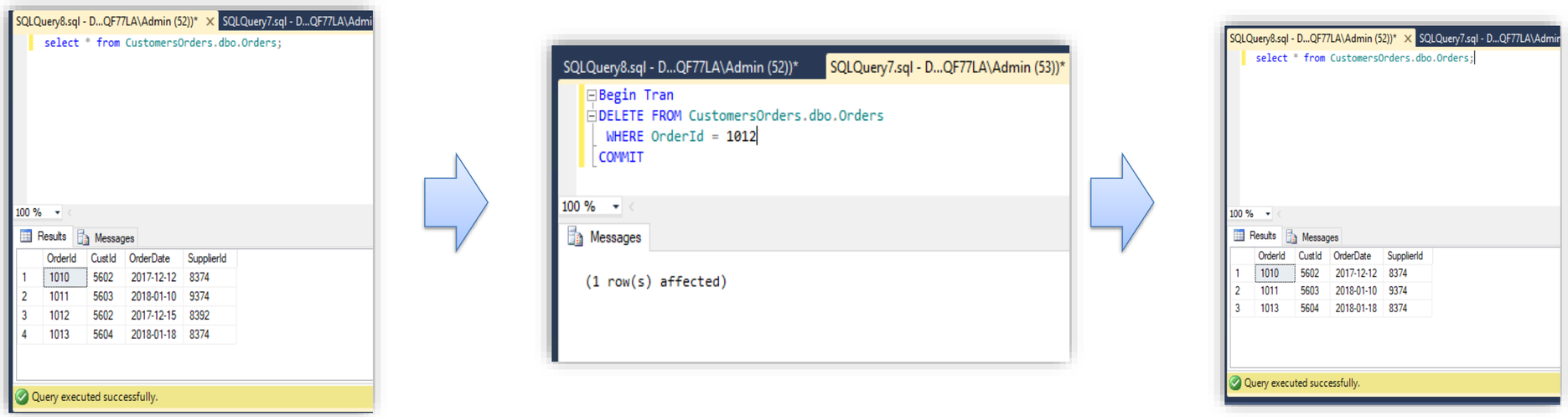
- ✓ Using Break Statement
- ✓ Syntax:



# Looping

## Basic Transactions

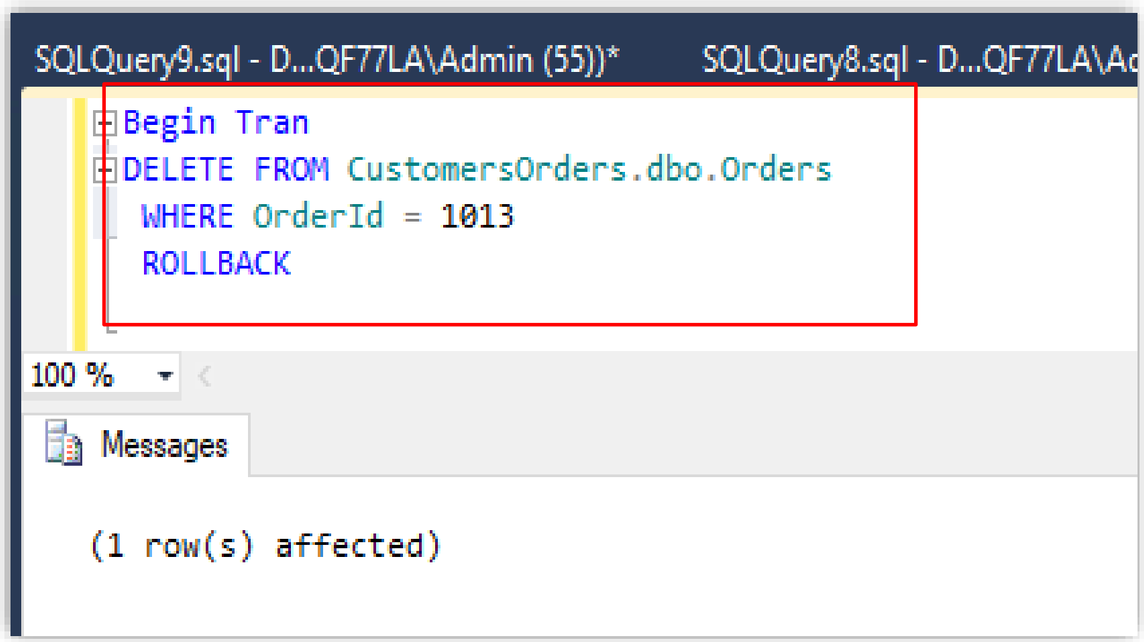
- ✓ The Transaction is the work unit performed opposite the database. Any transaction that reads or writes to a database. The transaction is the spread of one or various modifications to a database
- ✓ The transactions are controlled by the commands. These are the commands listed below:
  - ✓ **COMMIT**: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



# Looping

## Basic Transactions

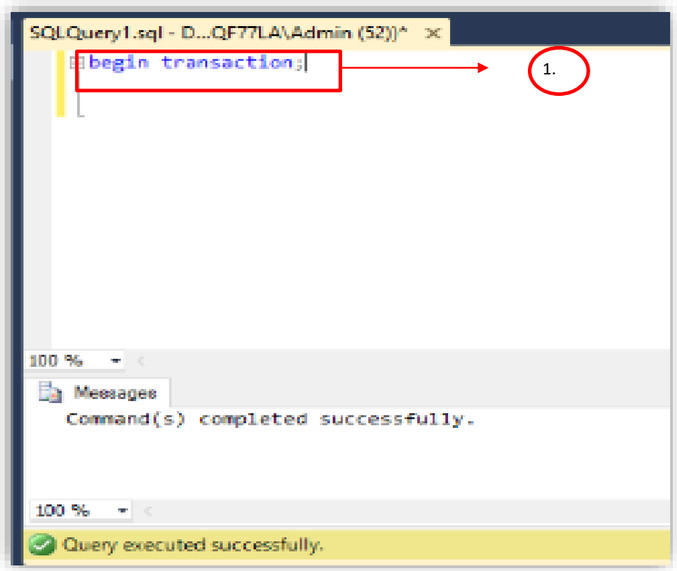
- ✓ **ROLLBACK**: ROLLBACK is used to undo the changes

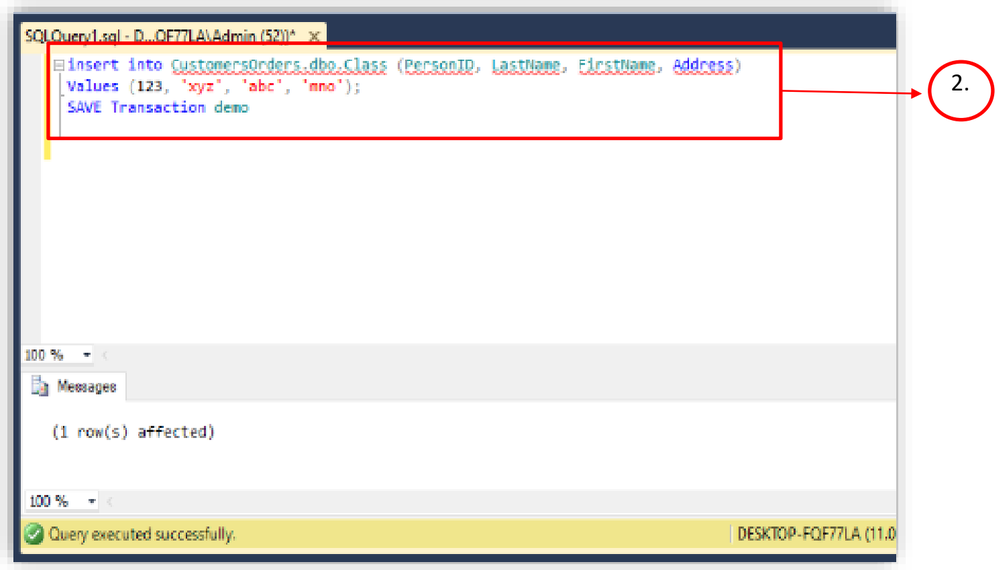


# Looping

## 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.



# Looping

**Using DELETE and UPDATE**

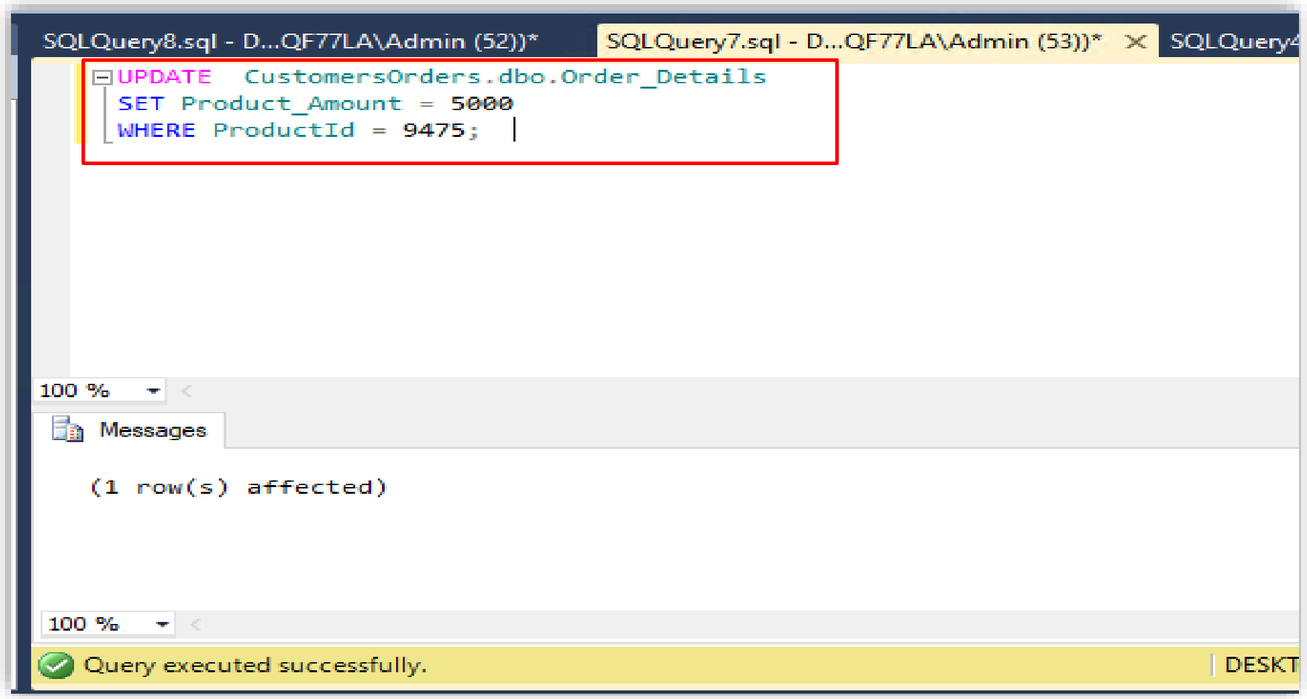
- ✓ **Update Statement**
- ✓ The UPDATE statement is used for modifying the existing records. You can use the WHERE clause with the UPDATE query for updating the specific rows. Otherwise, all rows would be affected
- ✓ **Syntax:**

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

# Looping

## Using DELETE and UPDATE

- ✓ Using Update Statement
- ✓ Example:



# 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 | It is NULL if there is no alternate individual owner.<br>It is NULL if the object type is one of the following:<br>C = CHECK constraint<br>D = DEFAULT (constraint or stand-alone)<br>F = FOREIGN KEY constraint<br>PK = PRIMARY KEY constraint<br>R = Rule (old-style, stand-alone)<br>TA = Assembly (CLR-integration) trigger<br>TR = SQL trigger<br>UQ = UNIQUE constraint |

# Looping

**Sys.Objects**

| Column name | Data type | Description |
|---|---|---|
| schema_id | int | ID of the schema that the object is contained in.<br>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.<br>0 = Not a child object. |
| type | char(2) | Object type:<br>AF = Aggregate function (CLR)<br>C = CHECK constraint<br>D = DEFAULT (constraint or stand-alone)<br>F = FOREIGN KEY constraint |

# Looping

**Sys.Objects**

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

# Looping

**Sys.Objects**

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

# Looping

**Sys.Objects**

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

# Looping

**Sys.Objects**

| Column name | Data type | Description |
|---|---|---|
| type_desc | nvarchar(60) | SERVICE_QUEUE<br>SQL_INLINE_TABLE_VALUED_FUNCTION<br>SQL_SCALAR_FUNCTION<br>SQL_STORED_PROCEDURE<br>SQL_TABLE_VALUED_FUNCTION<br>SQL_TRIGGER<br>SYNONYM<br>SYSTEM_TABLE<br>TABLE_TYPE<br>UNIQUE_CONSTRAINT<br>USER_TABLE<br>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 |

theknowledgeacademy

# Module 8


## 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

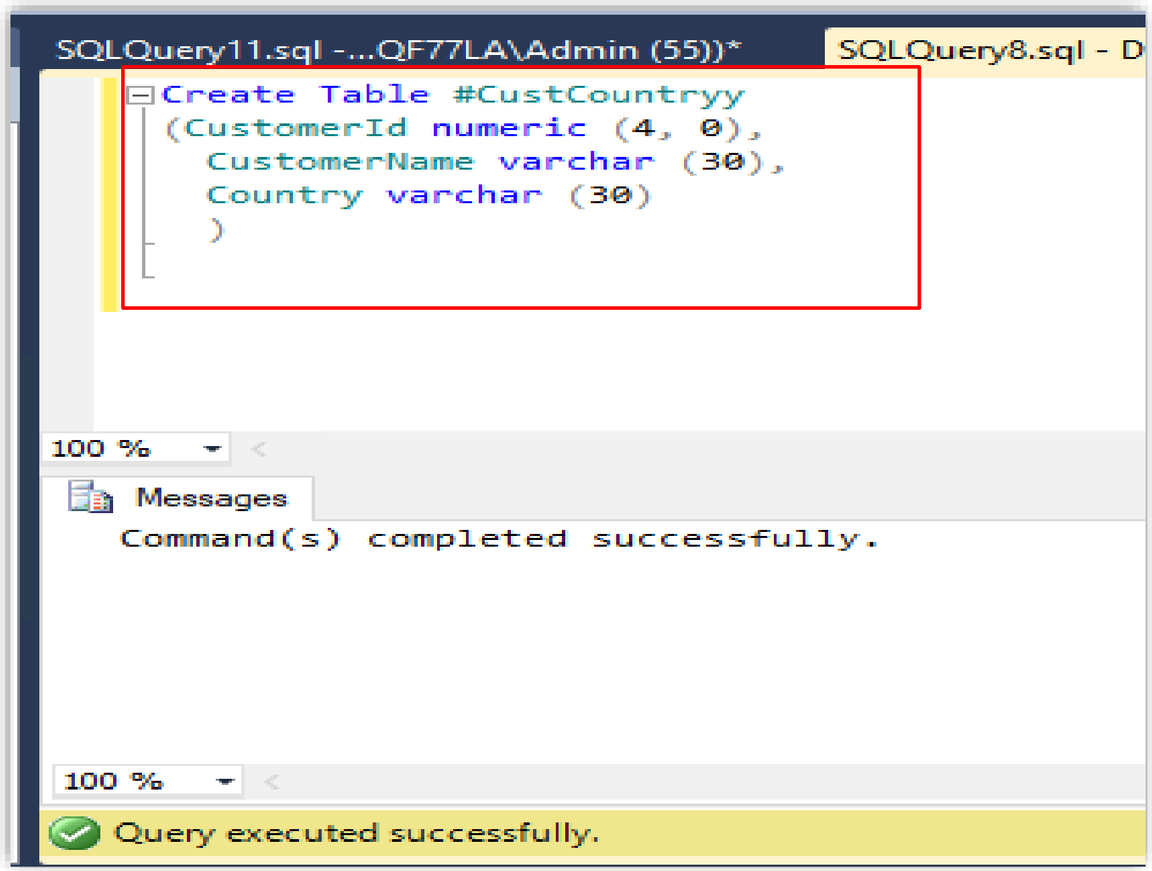
**Using Temporary Tables**

- **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

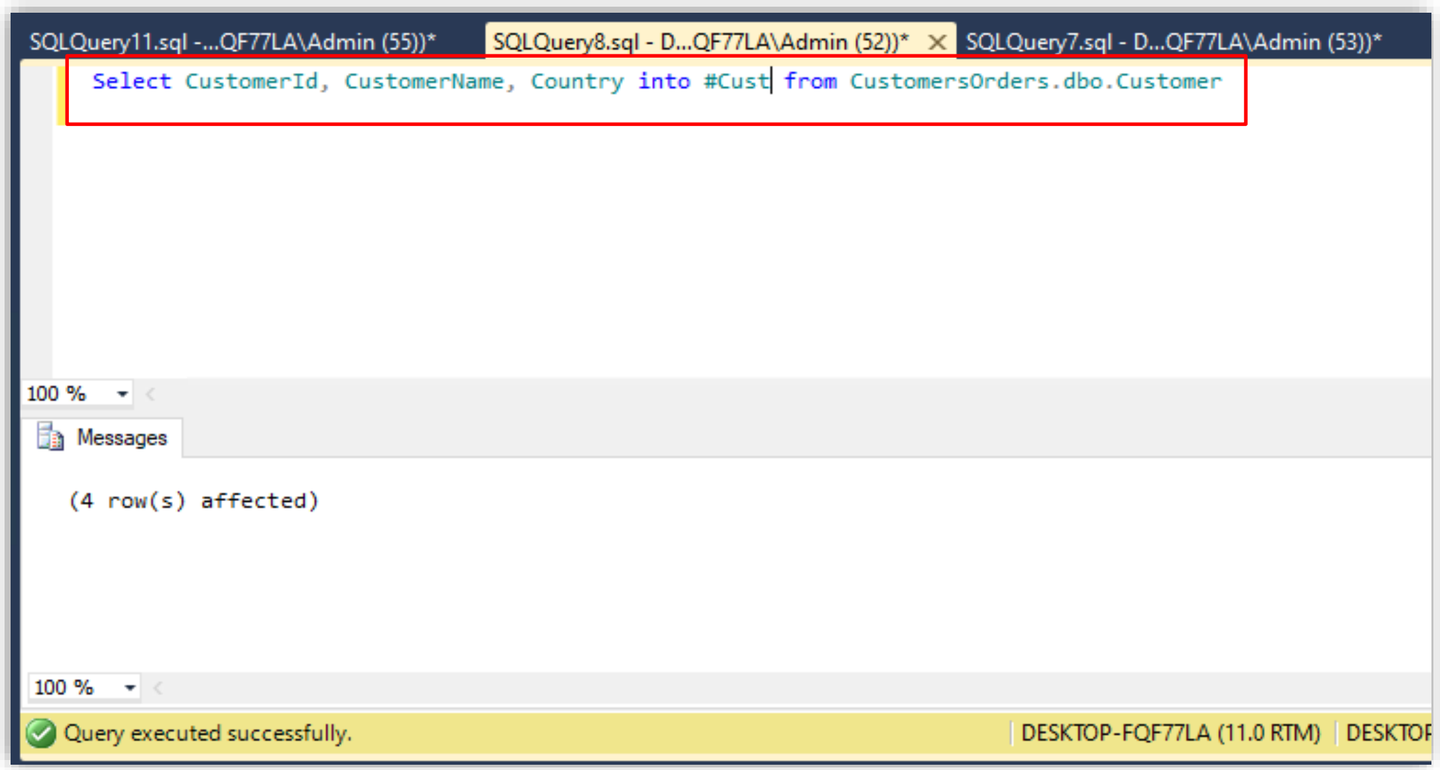
- ✓ **Creating Local TEMPTABLES**



# Temporary Tables and Table Variables

## Using Temporary Tables

- ✓ Inserting into TEMPTABLES



# Temporary Tables and Table Variables

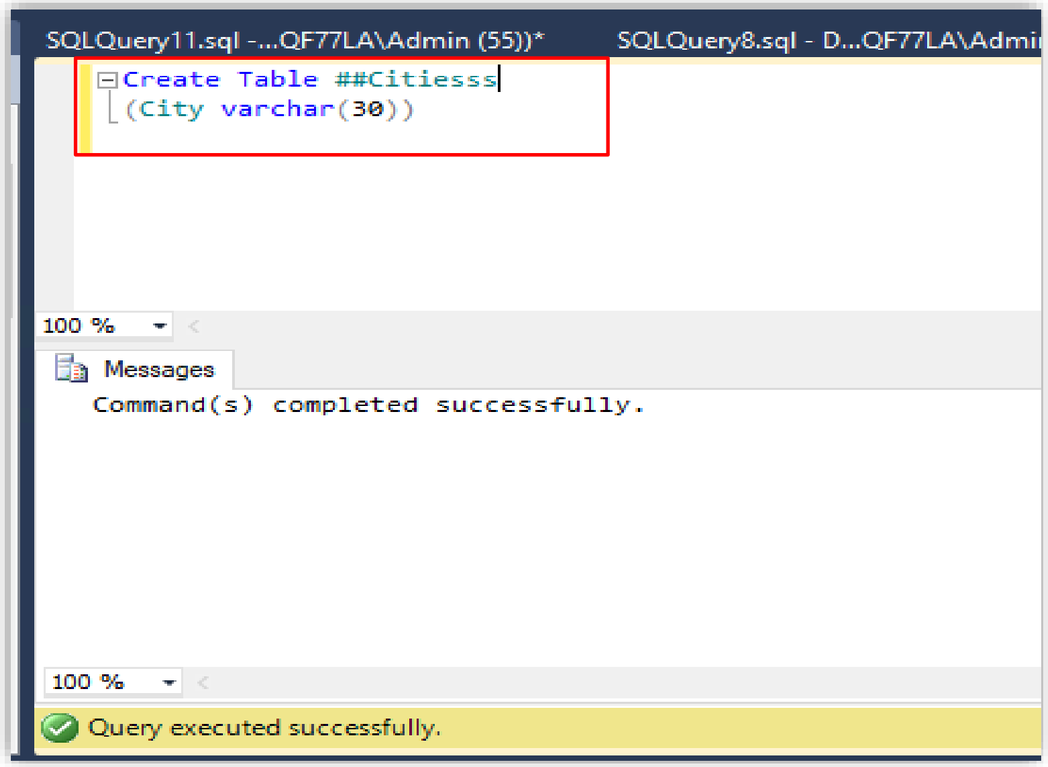
**Using Temporary Tables**

- **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

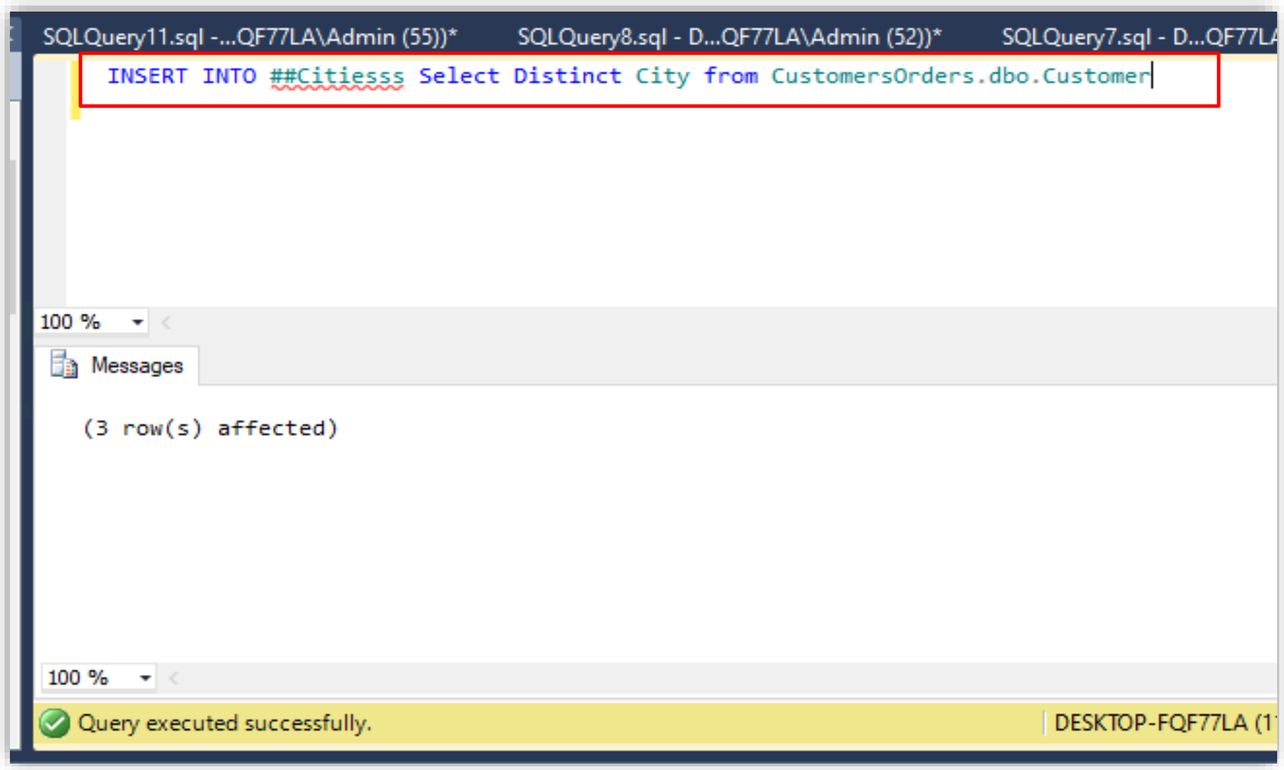
- ✓ **Create Global Temp Table**



# Temporary Tables and Table Variables

## Using Temporary Tables

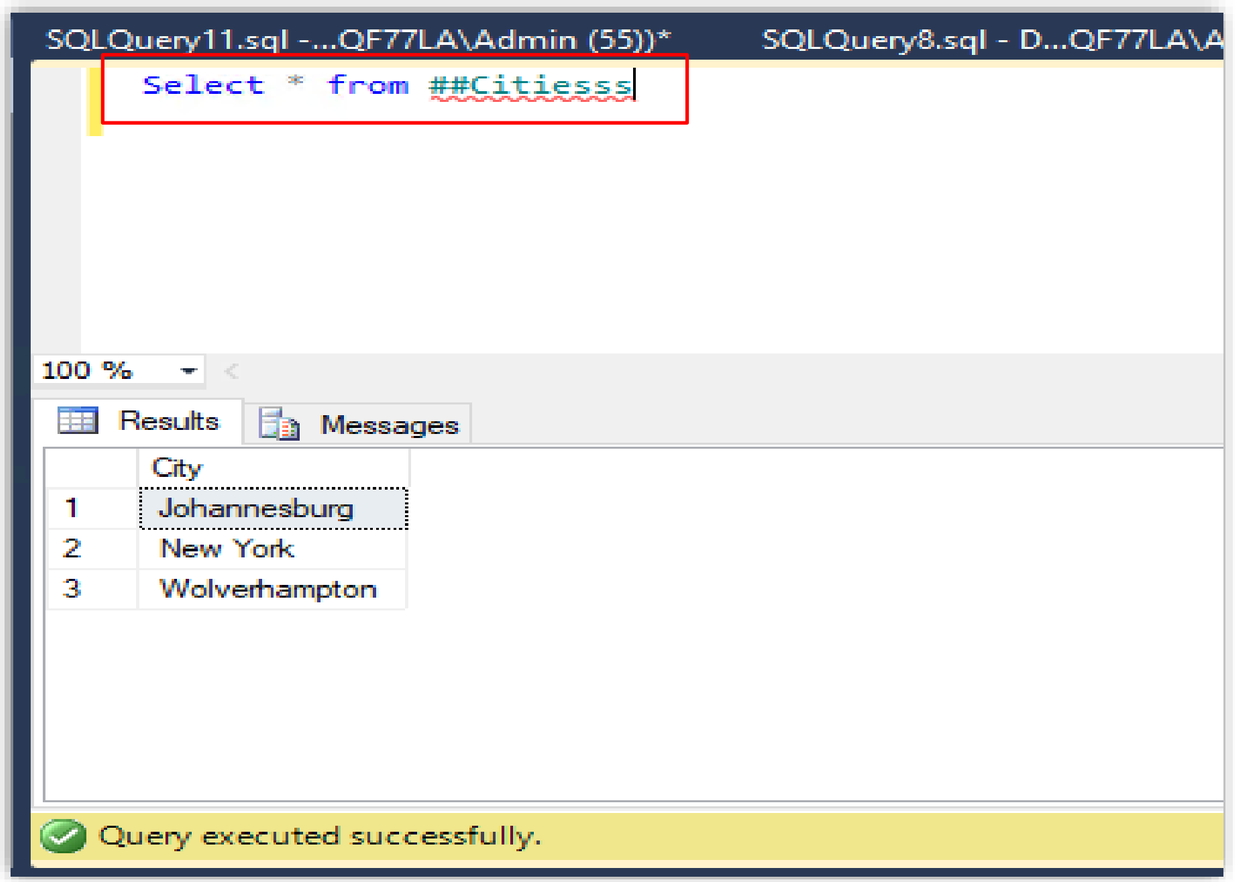
- ✓ **Inserting Rows:**



# Temporary Tables and Table Variables

## Using Temporary Tables

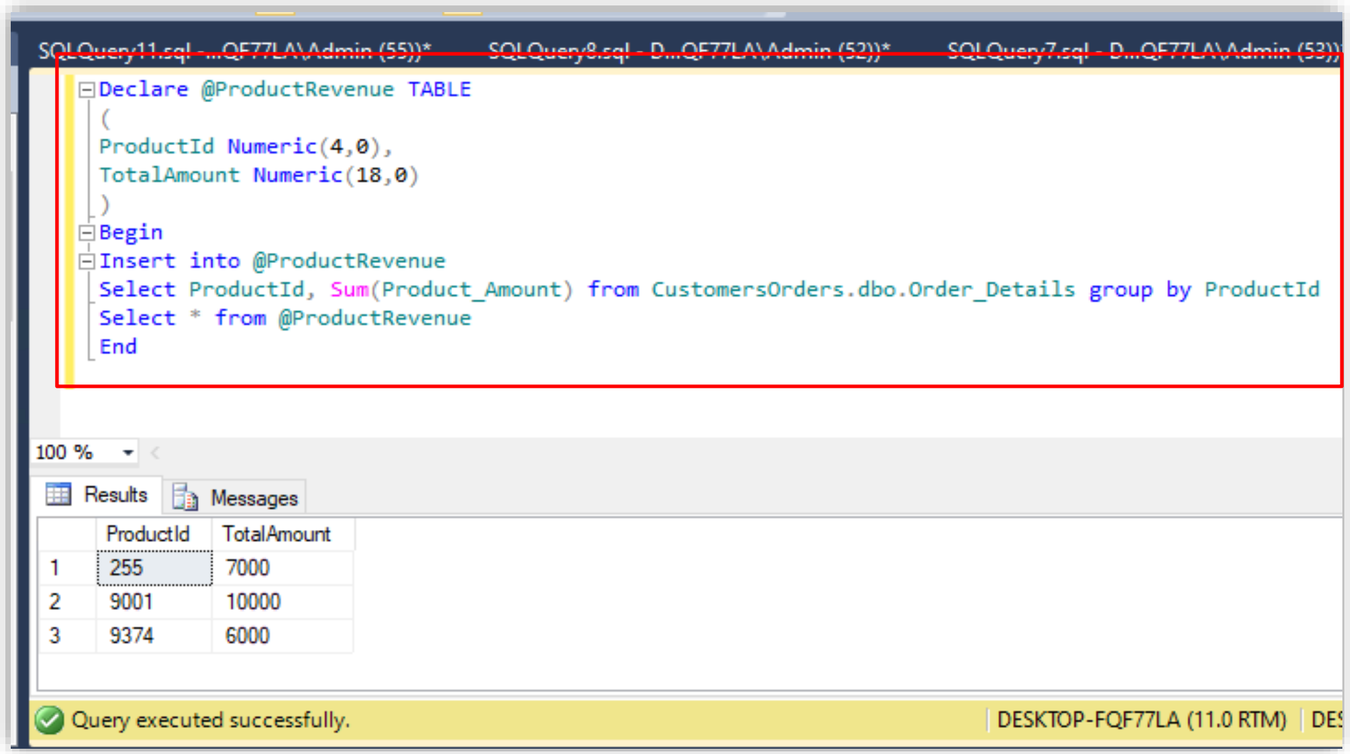
- ✓ **Retrieving Data:**



# Temporary Tables and Table Variables

## Creating Table Variables

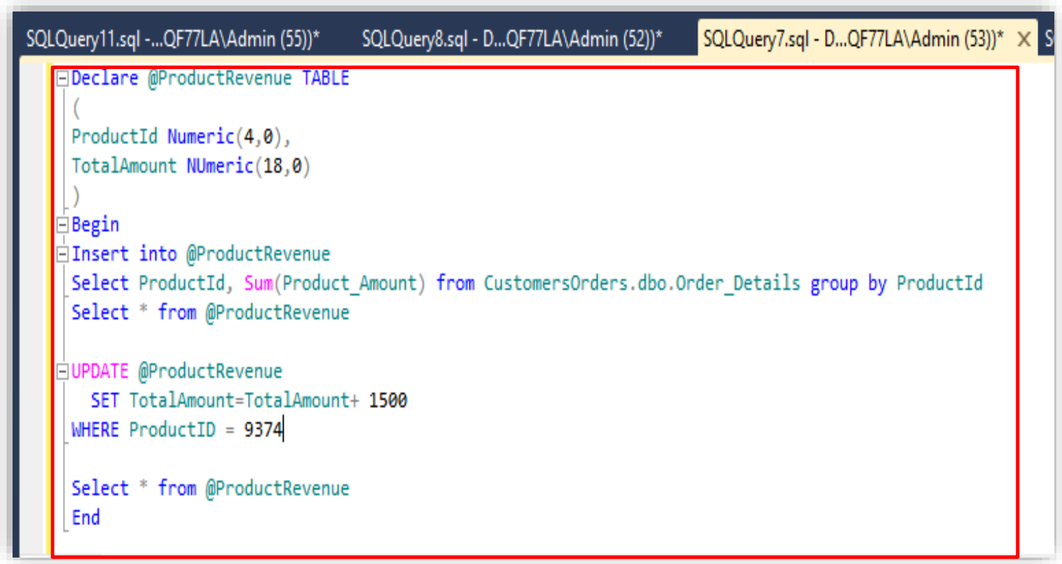
- ✓ Table variables are introduced by Microsoft as an alternative to using temporary tables
- ✓ They store a set of records, so the declaration syntax seems very similar to a CREATE TABLE statement

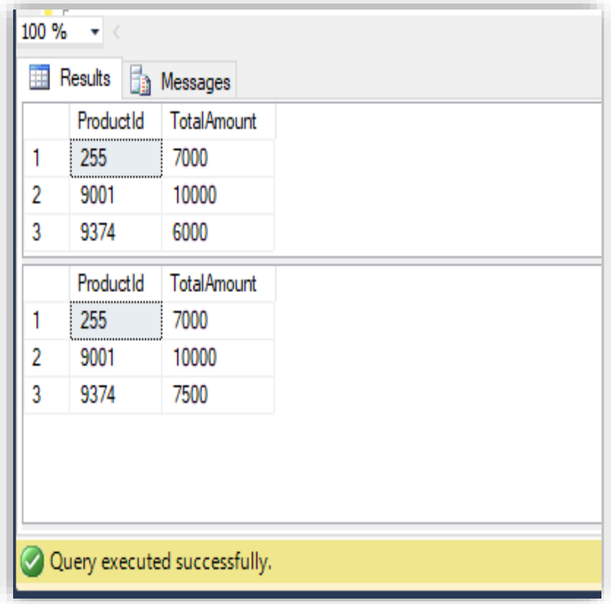


# Temporary Tables and Table Variables

## Creating Table Variables

- ✓ Updating the table variables



# Temporary Tables and Table Variables

## Creating Table Variables

- ✓ Deleting from table variables

```
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
```

100 %

Results | Messages

| | ProductId | TotalAmount |
|---|---|---|
| 1 | 255 | 7000 |
| 2 | 9001 | 10000 |
| 3 | 9374 | 6000 |

| | ProductId | TotalAmount |
|---|---|---|
| 1 | 255 | 7000 |
| 2 | 9001 | 10000 |
| 3 | 9374 | 7500 |

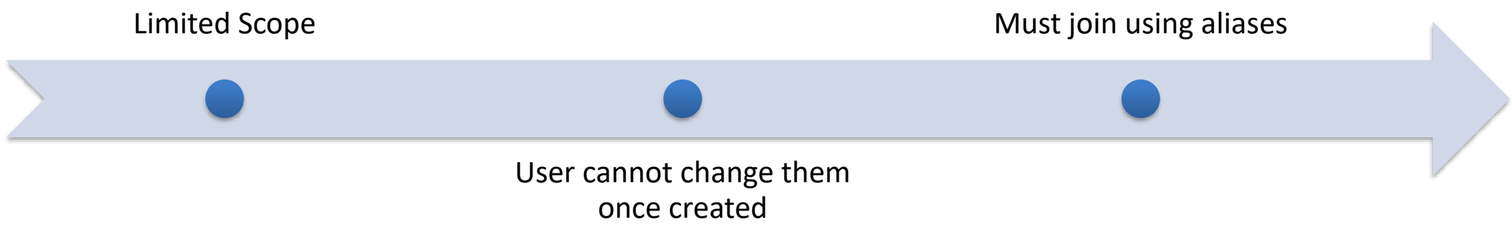
| | ProductId | TotalAmount |
|---|---|---|
| 1 | 255 | 7000 |
| 2 | 9001 | 10000 |

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

theknowledgeacademy

# Module 9


## 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 :

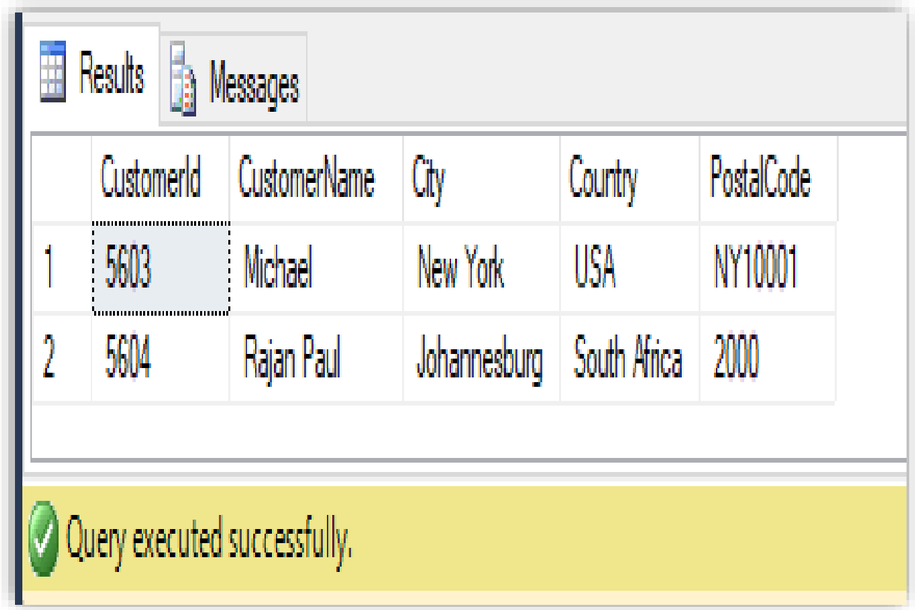
```
SQLQuery1.sql - D...QF77LA\Admin (52))*
Create Function CustFunc()
Returns @tabCust Table
(CustomerId numeric(4,0),
 CustomerName varchar(30),
 City varchar(30),
 Country varchar(30),
 PostalCode Varchar(10)
 )
 As
 Begin
   INSERT INTO @tabCust    SELECT * from CustomersOrders.dbo.Customer
   WHERE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

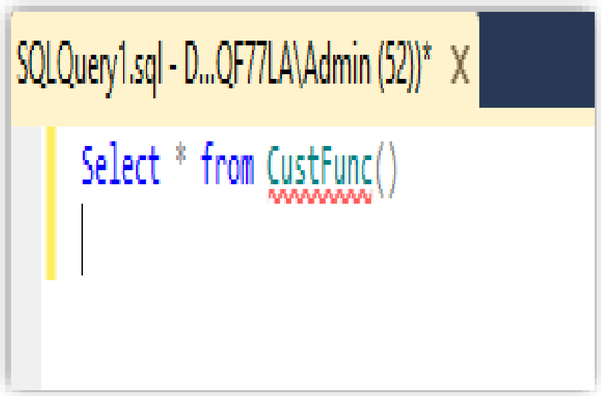
```
Messages
Command(s) completed successfully.

100 %
Query executed successfully.
```

# Table Valued Functions

## In-Line Table-Valued Functions

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



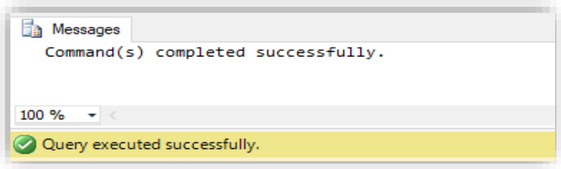



# Table Valued Functions

## 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**

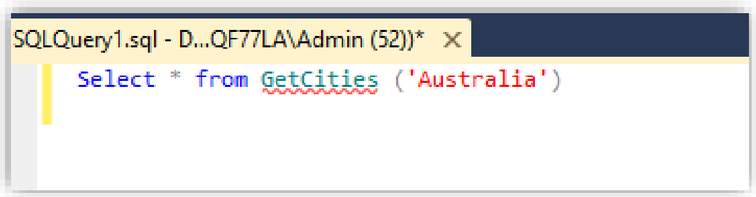
```
SQLQuery1.sql - D...QF77LA\Admin (52))*
Create FUNCTION GetCities(@Country varchar(30))
RETURNS @myRecords TABLE
(
       CustomerlName varchar(30),
       City varchar(30)
)
AS
Begin
    if @Country='Australia'
        INSERT INTO @myRecords    SELECT CustomerName, City from CustomersOrders.dbo.Customer
   WHERE Country ='Australia'

   Return
End
```

```
Messages
Command(s) completed successfully.

100 %
Query executed successfully.
```

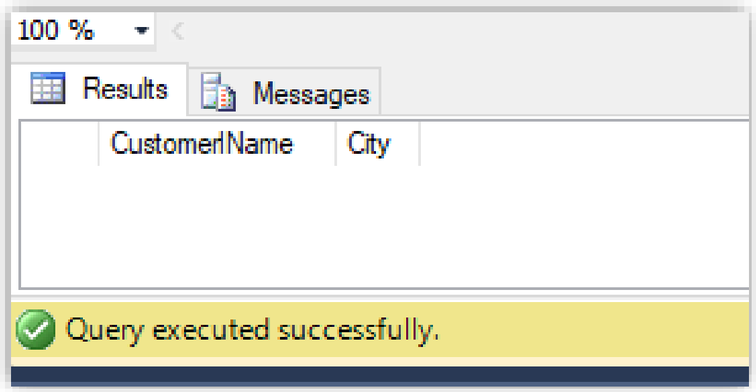
# Table Valued Functions

## Multi-Statement Table-Valued Functions

- ✓ **Retrieving Data:**




- ✓ **Result:**



# 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

theknowledgeacademy

# Module 10


## 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:

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
Select Top 5 CustomerName, City
From
(Select * From CustomersOrders.dbo.Customer) as Cust
Where Country = 'England'
```

Results | Messages

| | CustomerName | City |
|---|---|---|
| 1 | John Carter | Wolverhampton |

Query executed successfully.

# Derived Tables and CTEs

**Common Table Expressions(CTEs)**

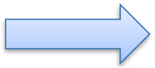
- ✓ It defines a temporary result set which users can use in the SELECT statement
- ✓ It is very helpful in managing complicated queries
- ✓ CTE are specified within the statement using WITH operator
- ✓ A user can define more than one CTE

# 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))*
With
     OrderCust (CustId, 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.CustId
```



Results | Messages

| | 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:

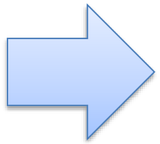
```
WITH expression_name (column_list)
AS
(
   -- Anchor member
   initial_query
   UNION ALL
   -- Recursive member that references expression_name.
   recursive_query
)
-- references expression name
SELECT *
FROM   expression_name
```

# Derived Tables and CTEs

## Recursive CTEs

- ✓ Example:

```
SQLQuery1.sql - DES...n (52)) Executing...*
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)
```



Results | Messages

| | 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 |

theknowledgeacademy

# Module 11


## 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:**

100 %

Results | Messages

| | 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 )
```

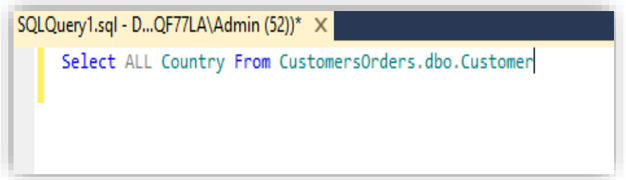
# Subqueries

## Using ALL, ANY and IN

- ALL with select statements

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

- Example:




# Subqueries

## Using ALL, ANY and IN

- ✓ **ALL with HAVING Statement:**

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
SELECT PRODUCTNAME FROM CustomersOrders.dbo.Product_Table WHERE PRODUCTID = ALL(
SELECT PRODUCTID FROM ORDER_DETAILS GROUP BY PRODUCTID HAVING COUNT(*) > 1)
```

100 %

Results | Messages

| | PRODUCTNAME |
|---|---|
| 1 | Polo T-shirts |

Query executed successfully.

# 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))*
SELECT  CustomerName from Customer
where CustomerId  IN (Select CustId from Orders)
```

- ✓ **Output**

Results | Messages

| | 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:**

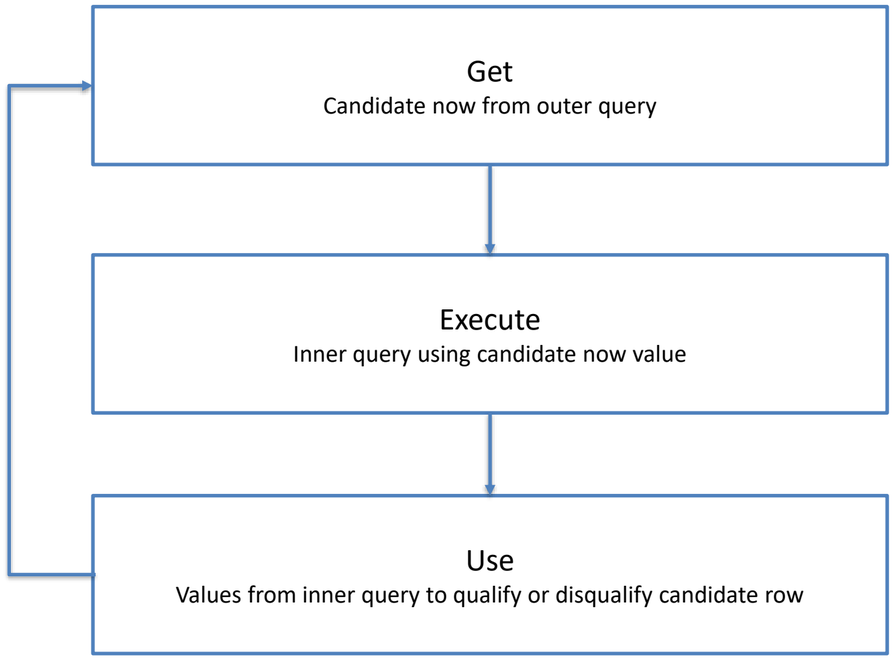
```
SELECT column_name(s)
FROM table_name
WHERE column_name comparison_operator ANY
(SELECT column_name
FROM table_name
WHERE condition(s));
```

# 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

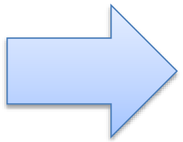
- ✓ The outer query drives the inner query

# Subqueries

## Correlated Subqueries

**Example:**

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
SELECT Customerid, Customername
FROM Customer
WHERE Customerid IN (SELECT CustId
FROM Orders where SupplierId=8374)
```



Results | Messages

| | Customerid | Customername |
|---|---|---|
| 1 | 5602 | Triza |
| 2 | 5604 | Rajan Paul |

Query executed successfully.

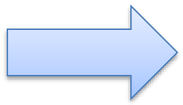
# Subqueries

## Correlated Subqueries

## Co-Related Subquery Example with UPDATE :

SQLQuery1.sql - D...QF77LA\Admin (52))*

```
Update Customer
Set City = 'Johannesburg', Country = 'South Africa', PostalCode = 2000
WHERE Customerid IN (SELECT CustId FROM Orders where SupplierId=8374)
```



Messages

(2 row(s) affected)

100 %

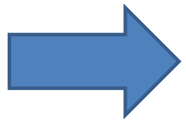
Query executed successfully.

# Subqueries

## Correlated Subqueries

## Co-Related Subquery with DELETE:

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
Delete from Customer
WHERE Customerid NOT IN (SELECT CustId FROM Orders)
```



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

# Subqueries

**Correlated Subqueries**

**Using the EXISTS Operator:**

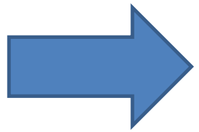
- ✓ The EXISTS operator tests for the existence of rows in the results set of the subquery
- ✓ If the subquery row value is found, then the condition will be flagged TRUE, and the search will not continue in the inner query
- ✓ If the subquery row value is not found, then the condition will be flagged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

# Subqueries

**Correlated Subqueries**

**EXAMPLE of using EXIST operator:**

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
Select Customername from Customer C where
Exists (Select Custid from Orders where C.CustomerId = Orders. CustId)
```



100 %

Results | Messages

| | Customername |
|---|---|
| 1 | Triza |
| 2 | Michael |
| 3 | Rajan Paul |

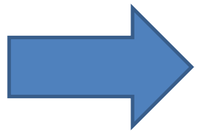
Query executed successfully.

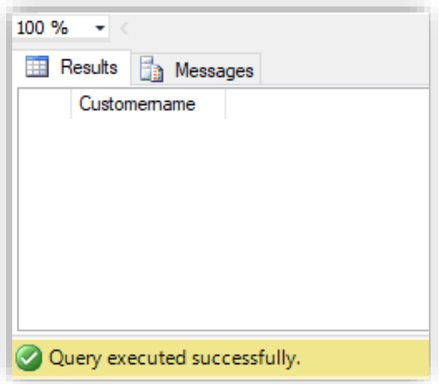
# Subqueries

**Correlated Subqueries**

**EXAMPLE of using NOT EXIST operator:**

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
Select Customername from Customer C where NOT
Exists (Select Custid from Orders where C.CustomerId = Orders. CustId)
```

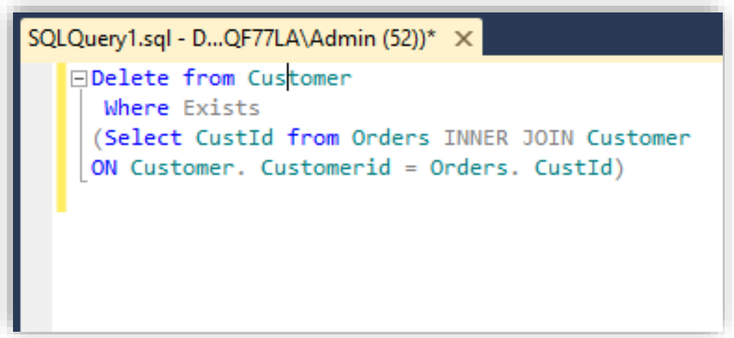


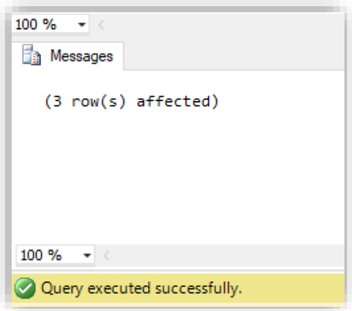


# Subqueries

## Using EXISTS

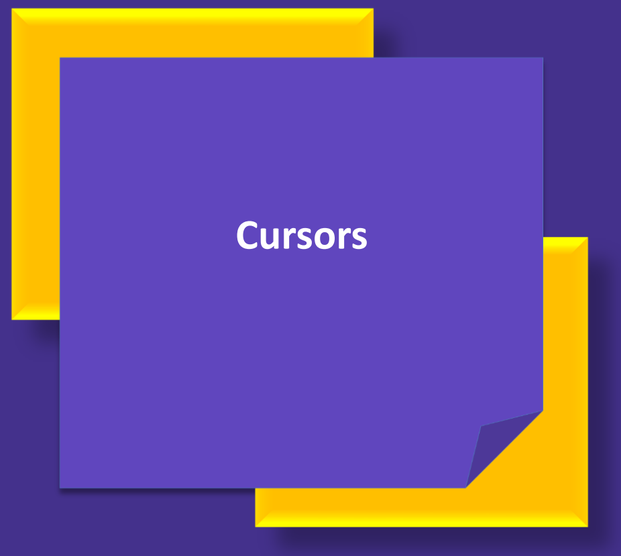
- ✓ Example: Using Exists with DELETE Statement



theknowledgeacademy

# 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:**

```
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
```

```
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

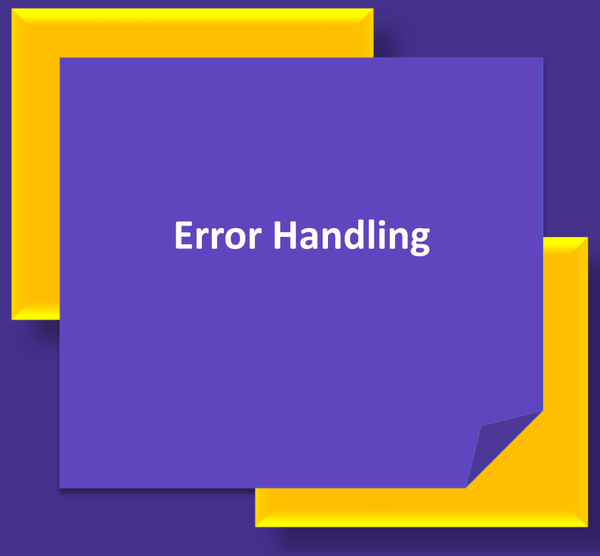
- ✓ Declaring a SCROLL cursor using FETCH

```
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
```

```
100 %
Messages
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
```

theknowledgeacademy

# Module 13




## 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

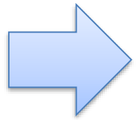
```
BEGIN TRY
 -- T-Sql Statements
END TRY
BEGIN CATCH
 -- T-Sql Statements
 /*Control is passed to CATCH block only if
   there are any exceptions in the TRY
block*/
END CATCH
```

# Error-Handling

## Using TRY/CATCH

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

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
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
```



```
Messages
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

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
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
```

Results | Messages

| | CustomerId | CustomerName | City | Country | PostalCode |
|---|---|---|---|---|---|
| 3 | 5614 | John | London | UK | 144001 |

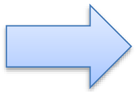
Query executed successfully.

# Error-Handling

## Using TRY/CATCH

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

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
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
```



100 %

Results | Messages

| | CustomerId | CustomerName | City | Country | PostalCode |
|---|---|---|---|---|---|
| 1 | 5612 | Tejinder | Jalandhar | India | 144001 |
| 2 | 5613 | Steve | Melbourne | Australia | 3005 |

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 %
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

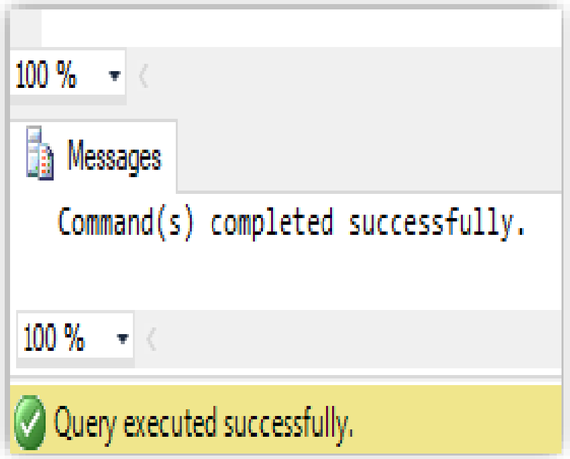
```
USE master
GO

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

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
BEGIN TRY
RAISERROR (50001,1,1) WITH LOG
END TRY
BEGIN CATCH
SELECT ERROR_MESSAGE(), ERROR_NUMBER()
END CATCH
```



# Error-Handling

## Custom Error Messages

- ✓ Example 2

```
SQLQuery1.sql - D...QF77LA\Admin (52))*
BEGIN TRY
RAISERROR (50002,16,1) WITH LOG
END TRY
BEGIN CATCH
SELECT ERROR_MESSAGE(),ERROR_NUMBER()
END CATCH
```

100 %

Results | Messages

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

Query executed successfully.

# Error-Handling

## Custom Error Messages

- ✓ Example 3

```
SQLQuery1.sql - DE...2)) - not connected*
BEGIN TRY
RAISERROR (50003,20,1) WITH LOG
END TRY
BEGIN CATCH
SELECT ERROR_MESSAGE (), ERROR_NUMBER()
END CATCH
```

```
100 %
Messages
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
100 %
```

# 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

# 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

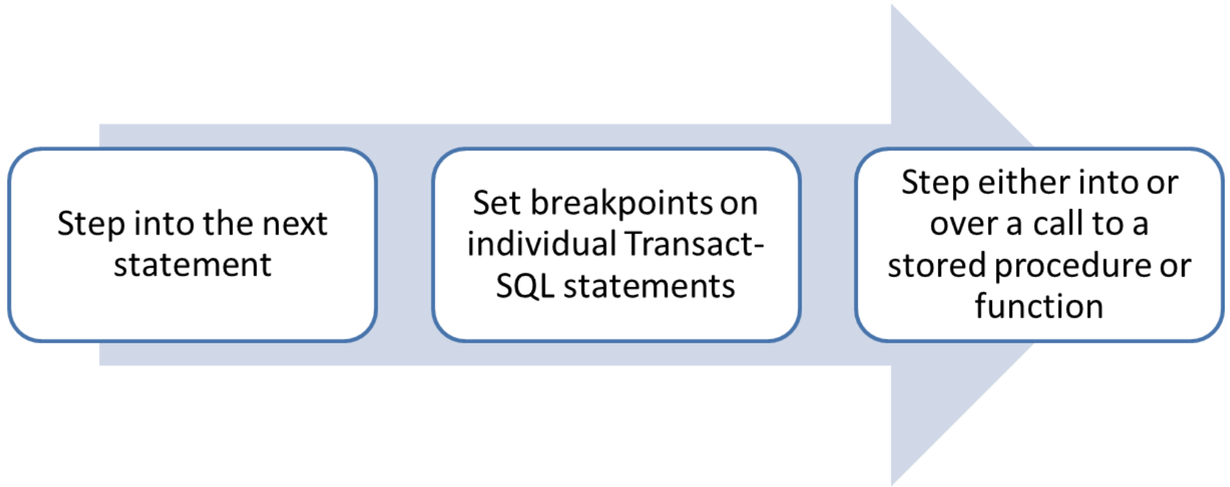
- ✓ After setting the Database Engine Query Editor window to debug mode, we can pause execution on particular lines of code and examine the information and data that is used by or returned by those SQL statements

# 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:

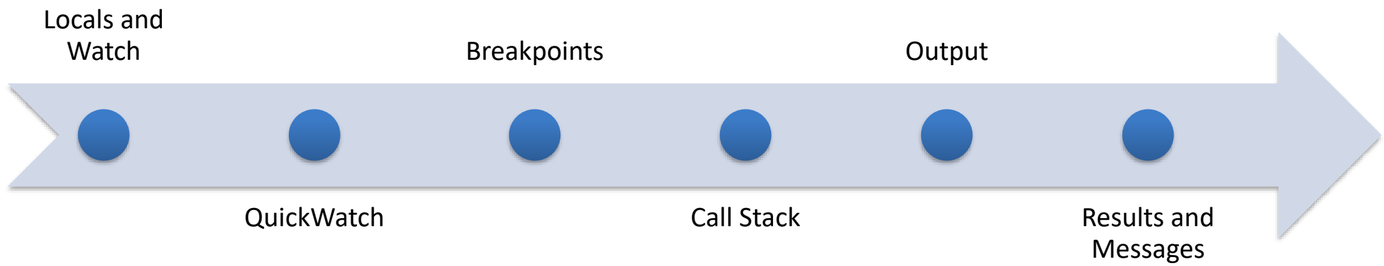


# Error-Handling

**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