Instructor Notes:

Instructor Notes:

Lesson Objectives

- Database programming
- Creating, Executing, Modifying and Dropping Stored Procedures and Functions
- Implementing Exception Handling

Instructor Notes:

Overview

- Introduction to Stored Procedures
- Creating, Executing, Modifying, and Dropping Stored Procedures
- Using Parameters in Stored Procedures
- Executing Extended Stored Procedures
- Handling Error Messages

Instructor Notes:

Definition

- Named Collections of pre compiled Transact-SQL Statements
- Stored procedures can be used by multiple users and client programs leading to reuse of code
- Abstraction of code and better security control
- Reduces network work and better performance
- Can accept parameters and return value or result set

In the simplest terms, a stored procedure is a collection of compiled T-SQL commands that are directly accessible by SQL Server. The commands placed within a stored procedure are executed as one single unit, or batch, of work. In addition to SELECT, UPDATE, or DELETE statements, stored procedures are able to call other stored procedures, use statements that control the flow of execution, and perform aggregate functions or other calculations
Any developer with access rights to create objects within SQL Server can build a stored procedure.
If the stored procedure is on same machine it is called as Local procedure, otherwise if it is stored at different machine then it is called as remote procedure.

Instructor Notes:

Types

- T-SQL supports the following types of procedure
  - System -
    - Procedures pre-built in SQL Server itself
    - Available in master database
    - Name starts with sp_
- Temporary
  - – name starts with #(Local) or ## (Global) and stored in tempdb
  - Available only for that session
- Extended
  - execute routines written in programming languages like C,C++,C# or VB.NET
  - May have names starting with xp_

Types of stored procedure

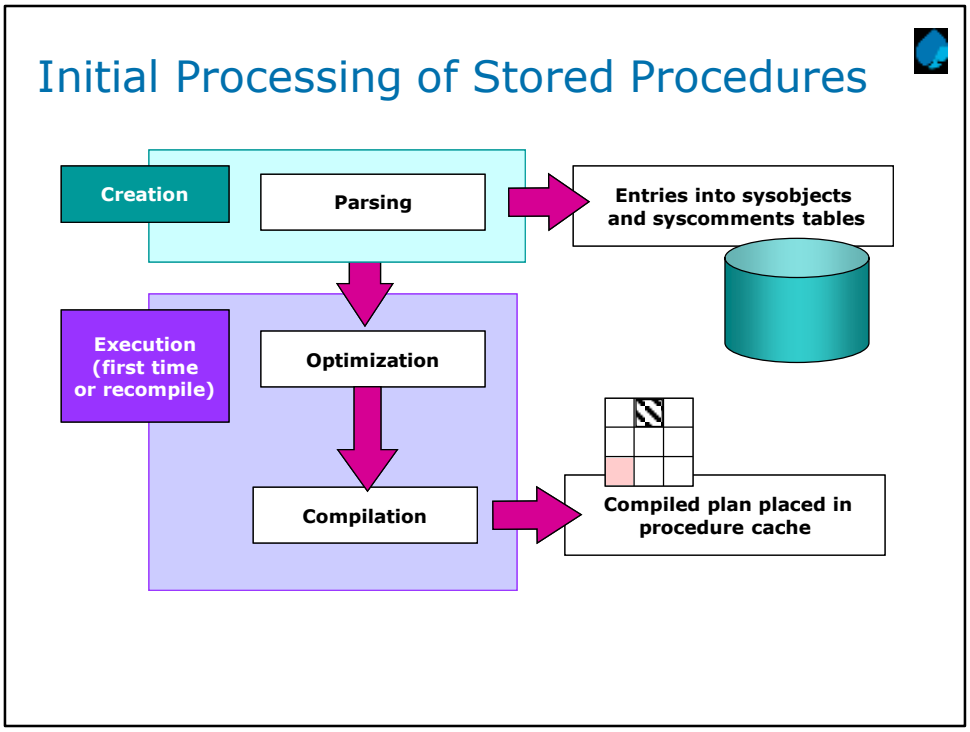
1.System procedure : The name of all system procedure starts with a prefix of sp_, within SQL Server. One cannot modify any system stored procedure that belongs to SQL Server, as this could corrupt not only your database, but also other databases

2.Temporary Procedure : The Database Engine supports two types of temporary procedures: local and global. A local temporary procedure is visible only to the user that created it. A global temporary procedure is available to all currently connected users. Local temporary procedures are automatically dropped at the end of the current session. Global temporary procedures are dropped at the end of the last session using the procedure. To create local temporary procedure name should start with # for global temporary procedure name should start with ##.

3.Extended procedure : Extended stored procedures let you create your own external routines in a programming language such as C,C++,.NET etc. The extended stored procedures appear to users as regular stored procedures and are executed in the same way. Parameters can be passed to extended stored procedures, and extended stored procedures can return results and return status. Extended stored procedures are DLLs that an instance of SQL Server can dynamically load and run. Extended stored procedures run directly in the address space of an instance of SQL Server and are programmed by using the SQL Server Extended Stored Procedure API.

Instructor Notes:

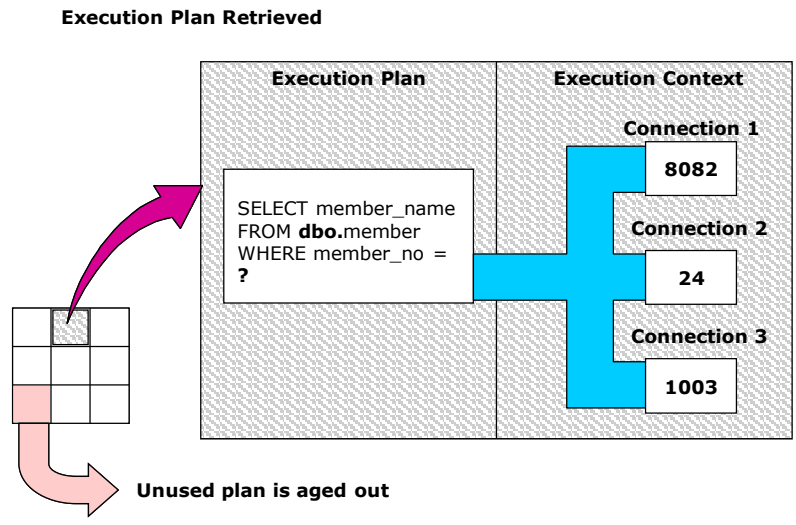
Execute one stored procedure and show execution plan of the procedure in result window.



1. When you create a stored procedure
2. SQL server parses your code and make entries in appropriate system table.
3. SQL server analyzes and optimizes the queries within stored procedure and generates an execution plan. An execution plan holds the instructions to process the query. These instruction include which order to access the table in, which indexes, access methods and join algorithm to use and so on.
4. SQL server generates multiple permutation of execution plan and choose the one with lowest cost
5. After execution of query or procedure we can see the execution plan in result window by clicking on display estimated execution plan button in the tool bar.

Instructor Notes:

Subsequent Processing of Stored Procedures



6. Stored procedures can reuse a previously cached execution plan, which saves the resources involved in generating a new execution plan
7. But if changes have been made to database objects (like adding or dropping column or index) or changes to set option that affect query results or if plan is removed from cache after a while for lack of reuse, causes recompilation and SQL server will generate a new one when the procedure is invoked again.

Instructor Notes:

Advantages

- Share Application Logic across multiple clients
- Shield Database Schema Details (Abstraction)
- Provide Security Mechanisms
- Reduce Network Traffic
- Improve Performance

1. **Share application logic-** Same procedure can be executed n times by multiple clients .
2. **Shield Database schema details -** Even If user does not have access to tables he can use tables through procedure.
3. **Provide security mechanism -** Restricted access can be given to tables and user also does not get to know which tables are used to get the data.
4. **Reduced Network traffic -** Assume that you are using some front end application and calling stored procedure. Then instead of sending SQL statements to server front end, will pass only function call with parameters and call gets executed at server side and result gets transferred to front end application.
5. **Improve performance -** If in case there is any change in procedure (like change in tables or logic changes) . It does not affect front end application if we are using stored procedure. Which improves performance.

Instructor Notes:

CREATE PROCEDURE Statement

- Syntax

```
CREATE { PROC | PROCEDURE } [schema_name.]
procedure_name     [ { @parameter datatype }[ VARYING
] [ = default ] [ [ OUT [ PUT ]     ] [ ,...n ]
 AS
{ <sql_statement> [;][ ...n ] |
<method_specifier> } [;]
Return  <value>
```

CREATE { PROC | PROCEDURE } [schema_name.] procedure_name
[{ @parameter [type_schema_name.] datatype } [[OUT [PUT]] [,...n] { [BEGIN] statements [END] }

schema_name

Is the name of the schema to which the procedure belongs.

procedure_name

Is the name of the new stored procedure. Procedure names must comply with the rules for identifiers and must be unique within the schema.

you should not use the prefix sp_ in the procedure name. This prefix is used by SQL Server to designate system stored procedures.

Local or global temporary procedures can be created by using one number sign (#) before procedure_name (#procedure_name) for local temporary procedures, and two number signs for global temporary procedures (##procedure_name). Temporary names cannot be specified for CLR stored procedures.

The complete name for a stored procedure or a global temporary stored procedure, including ##, cannot exceed 128 characters. The complete name for a local temporary stored procedure, including #, cannot exceed 116 characters.

Instructor Notes:

@ parameter

Is a parameter in the procedure. One or more parameters can be declared in a CREATE PROCEDURE statement. The value of each declared parameter must be supplied by the user when the procedure is called, unless a default for the parameter is defined or the value is set to equal another parameter. A stored procedure can have a maximum of 2,100 parameters.

Specify a parameter name by using an at sign (**@**) as the first character. The parameter name must comply with the rules for identifiers. Parameters are local to the procedure; the same parameter names can be used in other procedures. By default, parameters can take the place only of constant expressions; they cannot be used instead of table names, column names, or the names of other database objects.

OUTPUT

Indicates that the parameter is an output parameter. The value of this option can be returned to the calling EXECUTE statement. Use OUTPUT parameters to return values to the caller of the procedure. **text**, **ntext**, and **image** parameters cannot be used as OUTPUT parameters, unless the procedure is a CLR procedure. An output parameter that uses the OUTPUT keyword can be a cursor placeholder, unless the procedure is a CLR procedure.

The maximum size of a Transact-SQL stored procedure is 128 MB.
A user-defined stored procedure can be created only in the current database. Temporary procedures are an exception to this because they are always created in **tempdb**. If a schema name is not specified, the default schema of the user that is creating the procedure is used.

Ex.

```
CREATE PROC dbo.usp_GetCitiwiseEmployee
                @city           VARCHAR(20)
AS
BEGIN
                SELECT Employee_Name
                FROM Employee
                WHERE City = @city
END
```

Instructor Notes:

Additional notes for instructor

Example

- Code Snippet

```
USE AdventureWorks2012;
GO
 CREATE PROCEDURE
HumanResources.uspGetEmployeesTest2
@LastName nvarchar(50),
 @FirstName nvarchar(50) AS
 SET NOCOUNT ON;
SELECT FirstName, LastName, Department FROM
HumanResources.vEmployeeDepartmentHistory WHERE
FirstName = @FirstName AND LastName = @LastName AND
EndDate IS NULL;
 GO
```

```
USE Northwind
GO
CREATE PROC dbo.OverdueOrders
AS
BEGIN
  SELECT  COUNT(Order_id)
   FROM dbo.Orders
   WHERE RequiredDate < GETDATE() AND ShippedDate IS Null
End
GO
```

Instructor Notes:

Executing Stored Procedures

- Code Snippet

```
EXECUTE HumanResources.uspGetEmployeesTest2 N'Ackerman', N'Pilar';
-- Or
EXEC HumanResources.uspGetEmployeesTest2 @LastName = N'Ackerman',
@FirstName = N'Pilar';
GO
-- Or
EXECUTE HumanResources.uspGetEmployeesTest2 @FirstName = N'Pilar',
@LastName = N'Ackerman';
GO
```

Note : You need to have execute permission for the procedure to execute it

Execute statement:

Executes a command string or character string within a Transact-SQL batch, or one of the following modules: system stored procedure, user-defined stored procedure, or extended stored procedure.

You can use insert with the results of stored procedure or dynamic execute statement taking place of values clause. Execute should return exactly one result set with types that match the table you have set up for it.

Executing a Stored Procedure by Itself

```
EXEC OverdueOrders
```

Executing a Stored Procedure Within an INSERT Statement

```
INSERT INTO Customers
EXEC EmployeeCustomer
```

e.g If you want to store results of executing sp_configure stored procedure in temporary table

```
Create table #config_out
(
Name_col varchar(50),
Minval int,
Maxval int,configval int,
Runval int
)

insert #config_out
Exec sp_configure
```

Instructor Notes:

Altering and Dropping Procedures

- Altering Stored Procedures
  - Include any options in ALTER PROCEDURE
  - Does not affect nested stored procedures

```
ALTER { PROC | PROCEDURE } [schema_name.]
procedure_name [ ; number ]     [ { @parameter [
type_schema_name. ] data_type }         [ VARYING ] [
= default ] [ OUT | OUTPUT ] [READONLY]     ] [ ,...n ] [
WITH <procedure_option> [ ,...n ] ] [ FOR REPLICATION
] AS { [ BEGIN ] sql_statement [;] [ ...n ] [ END ] } [;]
<procedure_option> ::=     [ ENCRYPTION ]     [
RECOMPILE ]     [ EXECUTE AS Clause ]
```

```
DROP PROCEDURE <stored procedure name>;
```

sp_depends
Is system procedure which Displays information about database object dependencies, such as: the views and procedures that depend on a table or view, and the tables and views that are depended on by the view or procedure. References to objects outside the current database are not reported.

I. Listing dependencies on a table
The following example lists the database objects that depend on the Sales.Customer table in the AdventureWorks2012 database. Both the schema name and table name are specified.

```
USE AdventureWorks2012;
GO
EXEC sp_depends @objname = N'Sales.Customer' ;
```

Instructor Notes:

Demo

- Creating Stored Procedures

Instructor Notes:

Stored Procedures Using Parameters

- Stored procedures can take parameters OR arguments and return value
- Parameters can of the following type
- INPUT
  - Default Type
  - IN or INPUT keyword is used to define variables of IN type
  - Used to pass a data value to the stored procedure
- OUTPUT
  - Allow the stored procedure to pass a data value or a back to the caller.
  - OUT keyword is used to identify output parameter

Parameters are used to exchange data between stored procedures that called the stored procedure:
Input parameters allow the caller to pass a data value to the stored procedure.
Output parameter
Output parameters allow the stored procedure to pass a data value or a cursor variable back to the caller.

Parameters to procedure can be passed in the following manner
Passing values by parameter name
If you don't want to send all parameter values in same sequence as they are defined in create or procedure, you can pass them by parameter name Useful when many default values are defined , So required to pass very few parameters.

```
EXECUTE mytables @type='S'
```

Passing values by position
If you want to pass all parameters in the same sequence as they are defined in create or alter procedure. Then use this method.

If a parameter has default values , if default value is assigned to the INPUT parameters . DEFAULT keyword is used during execution to indicate DEFAULT value

```
CREATE PROCEDURE mytables
(
        @type char(2)='U'
)
AS
        SELECT  count( id)   FROM sysobjects      WHERE type = @type
GO

EXECUTE mytables
EXECUTE mytables DEFAULT
EXECUTE mytables 'P'
```

Instructor Notes:

Stored Procedures Using Parameters

```
CREATE PROCEDURE usp_ProductCountByCategory (
    @i_catid  INT ,
    @o_Prodcount INT OUT
  )
AS
BEGIN
    IF @i_catid is NULL OR @i_catid < 0
       return -1
    SELECT  @o_Prodcount=count(ProductID) from Products
     WHERE CategoryID=@i_catid
END
```

- To execute

```
DECLARE  @prodcount  INT
EXEC  usp_ProductCountByCategory  1234, @prodcount  OUT
```

In this example the procedure usp_ProductCountByCategory takes a category id as input and returns the count of products belonging to that category as output

To execute the procedure having OUT variables , one has to declare the
Variable first and then pass that varaible to the procedure using the OUT keyword.

The procedure also handles erroneous situation of invalid inputs by returning -1. More about return statement is discussed later in the session.

As with all Good programming practices , input values must be validated and all variables must be initialized

In case the procedure has default values then the procedure can be defined as

```
CREATE PROCEDURE usp_ProductCountByCategory (
    @i_catid  INT =2345 ,
    @o_Prodcount INT OUT
```

And to execute with default value it would be

```
Exec   usp_ProductCountByCategor DEFAULT, @pcount OUTPUT
```

Example of INPUT and OUTPUT Parameter:-
The following example shows a procedure with an input and an output parameter. The @SalesPerson parameter would receive an input value specified by the calling program. The SELECT statement uses the value passed into the input parameter to obtain the correct SalesYTD value. The SELECT statement also assigns the value to the @SalesYTD output parameter, which returns the value to the calling program when the procedure exits.

```
USE AdventureWorks2012;
GO
IF OBJECT_ID('Sales.uspGetEmployeeSalesYTD', 'P') IS NOT NULL
DROP PROCEDURE Sales.uspGetEmployeeSalesYTD;
GO
CREATE PROCEDURE Sales.uspGetEmployeeSalesYTD @SalesPerson nvarchar(50),
 @SalesYTD money OUTPUT
AS
 SET NOCOUNT ON;
SELECT @SalesYTD = SalesYTD FROM Sales.SalesPerson AS sp JOIN HumanResources.vEmployee AS e ON e.BusinessEntityID = sp.BusinessEntityID WHERE LastName = @SalesPerson;
 RETURN
GO
```

The following example calls the procedure created in the first example and saves the output value returned from the called procedure in the @SalesYTD variable, which is local to the calling program.

```
-- Declare the variable to receive the output value of the procedure.
DECLARE @SalesYTDBySalesPerson money;
-- Execute the procedure specifying a last name for the input parameter
 -- and saving the output value in the variable @SalesYTDBySalesPerson
 EXECUTE Sales.uspGetEmployeeSalesYTD N'Blythe', @SalesYTD = @SalesYTDBySalesPerson OUTPUT;
 -- Display the value returned by the procedure.
PRINT 'Year-to-date sales for this employee is ' + convert(varchar(10),@SalesYTDBySalesPerson);
 GO
```

Instructor Notes:

Returning a Value from Stored Procedures

- Values can be returned from stored procedure using the following options
  - OUTPUT parameter
    - More than 1 parameter can be of type OUTPUT
  - Return statement
    - Used to provide the execution status of the procedure to the calling program
    - Only one value can be returned
    - to -99 are reserved for internal usage , one can return customized values also
- Return value can be processed by the calling program as

  exec @return_value = <storedprocname>

The above example checks the state for the ID of a specified contact. If the state is Washington (WA), a status of 1 is returned. Otherwise, 2 is returned for any other condition (a value other than WA for StateProvince or ContactID that did not match a row).

Output parameters: Scalar data can be returned from a stored procedure with output variables.
RETURN: A single integer value can be returned from a stored procedure with a RETURN statement.
Result sets: A stored procedure can return data via one or more SELECT statements.
RAISERROR **or** THROW: Informational or error messages can be returned to the calling application via RAISERROR or THROW.

```
Example 1:
CREATE  Procedure usp_updateprodprice
     @i_vcategory  int,
 As
BEGIN
         if @i_vcategory  is NULL or @i_vcategory <=0
          begin
            raiserror (50001, 1,1)
           return  -1
         end
         Update Products  set ProductPrice = ProductPrice*1.1
          WHERE  CategoryID=  @i_vcategory
           return 0
END

DECLARE @return_value int
Exec @return_value = usp_updateprodprice  7

Example 2
USE AdventureWorks2012;
              GO
              CREATE PROCEDURE checkstate @param varchar(11)
              AS
              IF (SELECT StateProvince FROM Person.vAdditionalContactInfo WHERE ContactID = @param) =
              'WA'
                 RETURN 1
              ELSE
                 RETURN 2;
              GO
DECLARE @return_status int;
EXEC @return_status = checkstate '2';
SELECT 'Return Status' = @return_status;
GO
```

Instructor Notes:

WITH RESULT SETS

- In earlier versions of SQL server when we wished to change a column name or datatype in the resultset of a stored procedure, all the references needed to be changed. There was no simple way to dump the output of a stored procedure without worrying about the column names and data types.
- The EXECUTE statement has been extended in SQL Server 2012 to include the WITH RESULT SETS option. This allows you to change the column names and data types of the result set returned in the execution of a stored procedure

Example 1:

```
Use AdventureWorks2012
        CREATE PROC spGet_Employees
        AS
        SELECT BusinessEntityID, JobTitle,OrganizationLevel
        FROM  HumanResources.Employee
        ORDER BY BusinessEntityID

To execute the stored procedure
        EXEC spGet_Employees
        WITH RESULT SETS
        (
        (
        BEID int,
        Title varchar(30),
        OL int
        )
        )
```

Example :-2

```
use northwind
CREATE PROCEDURE GetData @pid int
AS
BEGIN
SELECT ProductID,ProductName,UnitPrice  FROM products
WHERE productid= @pid
END

CALLING PROCEDURE WITH RESULT SETS:

exec GetData 5
with result sets((PID INT,PNAME VARCHAR(10),UPrice money))
```

Instructor Notes:

Since recompilation is more of an admin task , an average developer may not have the opportunity to implement this unless he is playing that role. Therefore it is advisable not to go too deep on the same

Recompiling Stored Procedures

- Stored Procedures are recompiled to optimize the queries which makes up that Stored Procedure
- Stored Procedure needs recompilation when
  - Data in underlying tables are changed
  - Indexes are added /removed in tables
- Recompilation can be done by Using
  - CREATE PROCEDURE [WITH RECOMPILE]
  - EXECUTE [procedure]WITH RECOMPILE]
  - sp_recompile [procedure]

Example of Procedure with Recompile

```
USE AdventureWorks2012;
CREATE PROCEDURE dbo.uspProductByVendor @Name varchar(30) = '%'
WITH RECOMPILE
 AS
SET NOCOUNT ON;
SELECT v.Name AS 'Vendor name', p.Name AS 'Product name' FROM
Purchasing.Vendor AS v JOIN Purchasing.ProductVendor AS pv ON
v.BusinessEntityID = pv.BusinessEntityID JOIN Production.Product AS p ON
pv.ProductID = p.ProductID WHERE v.Name LIKE @Name;
```

Example of Execute With Recompile

```
USE AdventureWorks2012;
 GO
 EXECUTE HumanResources.uspGetAllEmployees WITH RECOMPILE;
GO
```

Example of sp_recompile

```
USE AdventureWorks2012;
GO
EXEC sp_recompile N'HumanResources.uspGetAllEmployees';
GO
```

Instructor Notes:

To View the Definition of Stored Procedure

- To view the definition of a procedure in Query Editor
  - EXEC sp_helptext N'AdventureWorks2012.dbo.uspLogError';
- To view the definition of a procedure with System Function: OBJECT_DEFINITION
  - SELECT OBJECT_DEFINITION(OBJECT_ID(N'AdventureWorks2012.dbo.uspLogError'));
  - Change the database name and stored procedure name to reference the database and stored procedure that you want.

```
CREATE PROC usp_GetStudentNameInOutputVariable
                @studentid INT, --Input parameter
                @studentname VARCHAR(200) OUT -- Out parameter
AS
BEGIN
                SELECT @studentname= Firstname+' '+Lastname
                FROM Students
                WHERE studentid=@studentid
END
```

To execute :

```
DECLARE @name VARCHAR(30)
EXEC usp_GetStudentNameInOutputVariable 1012, @name OUT
SELECT @name
```

To view definition of stored procedure in Query Editor

```
EXEC sp_helptext 'usp_GetStudentNameInOutputVariable'
```

To view definition of stored procedure with System Functions

```
SELECT
OBJECT_DEFINITION(
                OBJECT_ID
                ('usp_GetStudentNameInOutputVariable'));
```

Instructor Notes:

Guidelines

- One Stored Procedure for One Task
- Create, Test, and Troubleshoot
- Avoid sp_ Prefix in Stored Procedure Names
- Use Same Connection Settings for All Stored Procedures
- Minimize Use of Temporary Stored Procedures

Instructor Notes:

Error Handling in Procedures

- SQL Server 2005 onwards error handling can be done with
  - TRY .. CATCH blocks
  - @@ERROR global variable
- If a statements inside a TRY block raises an exception then processing of TRY blocks stops and is then picked up in the CATCH block
- The syntax of the TRY CATCH is

```
BEGIN TRY
    --- statements
END TRY
BEGIN CATCH
  --- statements
END CATCH
```

It is a known thing that during an application development one of the most common things we need to take care is Exception Handling . The same point holds good when we are building our databases also .

Typical scenarios where we need to handle errors in DB
1. When we perform some DML operations and need to check the output
2. When a transaction fails – we need to rollback or undo the changes done to the data

Error handling can be done in two ways in SQL Server
Using @@ERROR
Using TRY..CATCH BLOCK
Earlier version of SQL Server like (2000 or 7.x) did not support this TRY ..CATCH construct , therefore error handling was done only using @@ERROR global variable . Whenever an error occurs the variable automatically populates the error message , which needs to be traced in the next line for example
Insert into sometable values(….)
Select @@error
This will show the errorcode generated in the last SQL statement

Instructor Notes:

The instructor shows the demo of both constructs . The same code is also attached with the ppts

Error Handling

Using @@Error

```
DECLARE  @v_deptcode int
DECLARE @v_deptname varchar(10)
DECLARE @errorcode int

set @v_deptcode=10
set @v_deptname='Pre sales'

insert into dept
values(@v_deptcode,'Pre sales')

set  @errorcode = @@ERROR
if @errorcode > 0
begin
        print 'error'
        print @errorcode
end
else
    print 'added successfully'
```

Using TRY ..CATCH

```
DECLARE  @v_deptcode int
DECLARE @v_deptname varchar(10)
DECLARE @errorcode int

set @v_deptcode=10
set @v_deptname='Pre sales'

BEGIN TRY
    insert into dept
values(@v_deptcode,'Pre sales')
END TRY

BEGIN CATCH
   PRINT 'An error occurred while
inserting
   PRINT ERROR_NUMBER()

END CATCH
```

TRY ..CATCH blocks are standard approach to exception handling in any modern language (Java, VB, C++ , C# etc) . The syntax of TRY ..CATCH block is almost similar to that of the programming languages . Nested Try catch is also possible

The general syntax of the TRY..CATCH is as follows

```
--sql statements
BEGIN TRY
   sql statements
    sql statements
END TRY
BEGIN CATCh
--sql statemnts
END CATCH
```

A set of system functions has been provided by SQL server to handle errors

ERROR_MESSAGE()	Returns the complete description of the **error** message
ERROR_NUMBER()	Returns the number of the **error**
ERROR_SEVERITY()	Returns the number of the Severity
ERROR_STATE()	Returns the **error** state number
ERROR_PROCEDURE()	Returns the name of the stored procedure where the **error** occurred
ERROR_LINE()	Returns the line number that caused the **error**

Instructor Notes:

Explain the use of RAISEERROR statement
Important points :
1.Error numbers for user-defined error messages should be greater than 50000.
2.Severity levels from 0 through 18 can be specified by any user.

Error Handling using RAISEERROR

- RAISERROR can be used to
  - Return user defined or system messages back to the application
  - Assign a specific error number , severity and state to a message
- Can be associated to a Query or a Procedure
- Has the following syntax

  RAISERROR (message ID | message string ,severity, state)
- Message ID has to be a number greater than 50,000
- Can be used along with TRY ..CATCH /other error handling mechanisms

Every stored procedure returns an integer return code to the caller. If the stored procedure does not explicitly set a value for the return code, the return code is 0.

RAISERROR statement
RAISERROR ({ msg_id | msg_str | @local_variable } { ,severity ,state } [,argument [,...n]])

msg_id
Is a user-defined error message number stored in the **sys.messages** catalog view using **sp_addmessage**. Error numbers for user-defined error messages should be greater than 50000. When msg_id is not specified, RAISERROR raises an error message with an error number of 50000.

msg_str
Is a user-defined message with formatting similar to the **printf** function in the C standard library. The error message can have a maximum of 2,047 characters. If the message contains 2,048 or more characters, only the first 2,044 are displayed and an ellipsis is added to indicate that the message has been truncated. Note that substitution parameters consume more characters than the output shows because of internal storage behavior. For example, the substitution parameter of %d with an assigned value of 2 actually produces one character in the message string but also internally takes up three additional characters of storage. This storage requirement decreases the number of available characters for message output.
When msg_str is specified, RAISERROR raises an error message with an error number of 5000.

Instructor Notes:

@local_variable

Is a variable of any valid character data type that contains a string formatted in the same manner as *msg_str*. *@local_variable* must be char or varchar, or be able to be implicitly converted to these data types.

severity

Is the user-defined severity level associated with this message. When using *msg_id* to raise a user-defined message created using sp_addmessage, the severity specified on RAISERROR overrides the severity specified in sp_addmessage.

Severity levels from 0 through 18 can be specified by any user. Severity levels from 19 through 25 can only be specified by members of the sysadmin fixed server role or users with ALTER TRACE permissions. For severity levels from 19 through 25, the WITH LOG option is required.

RAISERROR is used to return messages back to applications using the same format as a system error or warning message generated by the SQL Server Database Engine.

RAISERROR can return either:

•A user-defined error message that has been created using the sp_addmessage system stored procedure. These are messages with a message number greater than 50000 that can be viewed in the sys.messages catalog view.

•A message string specified in the RAISERROR statement.

A RAISERROR severity of 11 to 19 executed in the TRY block of a TRY…CATCH construct causes control to transfer to the associated CATCH block. Specify a severity of 10 or lower to return messages using RAISERROR without invoking a CATCH block. PRINT does not transfer control to a CATCH block.

When RAISERROR is used with the *msg_id* of a user-defined message in sys.messages, *msg_id* is returned as the SQL Server error number, or native error code. When RAISERROR is used with a *msg_str* instead of a *msg_id*, the SQL Server error number and native error number returned is 50000.
When you use RAISERROR to return a user-defined error message, use a different state number in each RAISERROR that references that error. This can help in diagnosing the errors when they are raised.
Use RAISERROR to:

- Help in troubleshooting Transact-SQL code.
- Check the values of data.
- Return messages that contain variable text.
- Cause execution to jump from a TRY block to the associated CATCH block.
- Return error information from the CATCH block to the calling batch or application

Instructor Notes:

Example of Raiserror with TRY ..CATCH

```
CREATE  Procedure usp_updateprodprice
     @i_vcategory  int,
      @i_vpriceinc   money
As
BEGIN
        if @i_vcategory  is NULL or @i_vcategory <=0
         begin
            raiserror (50001, 1,1)
           return
         end
        if  @i_vpriceinc  <= 0
         begin
            raiserror  (50002, 1,1)
           return
         end
```

The structure of the new table is as follows

```
create table revised_product
   (ProductID  int not null,
    ProductName  nvarchar(80),
    unitPrice money,
    CategoryID  int,
    revisedprice  money
)
```

The procedure takes a CategoryID and price increase percentage and updates the revised product table with product details, old price and revised price . The procedure can be executed as follows

All the error codes and messages has been added in the sysmessages table using the procedure sp_addmessage

```
sp_addmessage 50001,1,'invalid category‘
```

When the procedure is executed , it will display the message associated with 50001

```
BEGIN TRY
  exec usp_updateprodprice -7,.2
END TRY
BEGIN CATCH
 select ERROR_MESSAGE()
END CATCH
```

Instructor Notes:

Example of Raisererror with TRY ..CATCH

```
if  not exists( SELECT 'a' FROM Categories
          WHERE  CategoryID = @i_vcategory)
        begin
          raiserror  (50003,1,1)
            return
        end
       BEGIN TRY
            insert into revised_product
           select ProductID,ProductName,
unitPrice,@i_vcategory,unitPrice+unitPrice*@i_vpriceinc
              FROM Products where CategoryID=@i_vcategory

         return
      END TRY

      BEGIN CATCH
         raiserror  (50004,1,1)
         rollback tran
        -- return -1
     END CATCH
END
```

The structure of the Departments and Employee table is as follows
CREATE TABLE Departments (DeptID INT PRIMARY KEY, DeptName VARCHAR(20));

CREATE TABLE Employee (EmployeeID INT PRIMARY KEY, EmployeeName VARCHAR(30), DeptID INT FOREIGN KEY REFRENCES Departments(DeptID))

The procedure takes a DeptID and gives the number of employees from the department if DeptID exists otherwise raises an error. The procedure can be executed as follows

```
CREATE PROC dbo.usp_DepartmentwiseEmployee
                @DeptIDINT
AS
BEGIN
                IF EXISTS (SELECT DeptID FROM Departments
                                WHERE DeptID = @DeptID)
                BEGIN
                                SELECT COUNT(EmployeeID)
                                FROM Employee
                                WHERE DeptID = @DeptID
                END
                ELSE
                BEGIN
                                RAISERROR('Department ID Does Not Exist', 1, 1)
                END
END

EXEC usp_DepartmentwiseEmployee 13
```

If Department ID 13 not exist then it will raise error message

Instructor Notes:

THROW Statement

- Exception handling is now made easier with the introduction of the THROW statement in SQL Server 2012.
- In previous versions, RAISERROR was used to show an error message.

Drawbacks of RAISERROR:
It requires a proper message number to be shown when raising any error.
The message number should exist in sys.messages.
RAISERROR cannot be used to re-throw an exception raised in a TRY..CATCH block.
RAISERROR has been considered as deprecated features

Unlike RAISERROR, THROW does not require that an error number to exist in sys.messages (although it has to be between 50000 and 2147483647).
You can throw an error using Throw as below:

```
THROW 50001, 'Error message', 1;
```

This will return an error message:

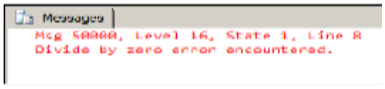
```
Msg 50001, Level 16, State 1, Line 1 Error message
```

Instructor Notes:

Difference Between RaiseError and Throw

```
BEGIN TRY
DECLARE @MyInt int
SET @MyInt = 1 / 0
END TRY
BEGIN CATCH
DECLARE @ErrorMessage nvarchar(4000),
@ErrorSeverity int
SELECT @ErrorMessage = ERROR_MESSAGE(),
@ErrorSeverity = ERROR_SEVERITY()
RAISERROR (@ErrorMessage, @ErrorSeverity, 1)
END CATCH
```

```
BEGIN TRY
DECLARE @MyInt int
SET @MyInt = 1/0
END TRY
BEGIN CATCH
-- throw out the error
THROW
END CATCH
```



Note: All exceptions being raised by THROW have a severity of 16.

RAISERROR statement THROW statement
If a msg_id is passed to RAISERROR, the ID must be defined in sys.messages.
The error_number parameter does not have to be defined in sys.messages.

RAISERROR statement THROW statement
The msg_str parameter can contain printf formatting styles.
The message parameter does not accept printf style formatting.

RAISERROR statement THROW statement
The severity parameter specifies the severity of the exception.
There is no severity parameter. The exception severity is always set to 16.

```
BEGIN TRY
            SELECT 1/0
END TRY

BEGIN CATCH
            THROW
END CATCH
```

```
USE tempdb;
GO
CREATE TABLE dbo.TestRethrow (  ID INT PRIMARY KEY );
BEGIN TRY
INSERT dbo.TestRethrow(ID) VALUES(1);
-- Force error 2627, Violation of PRIMARY KEY constraint to be raised.
INSERT dbo.TestRethrow(ID) VALUES(1);
END TRY
BEGIN CATCH
PRINT 'In catch block.';
THROW;
END CATCH;
```

Instructor Notes:

Advantages of THROW:

- THROW has now made the developer's life much easier and developers can now code independent of the Tester's input on the exception message.
- It can be used in a TRY..CATCH block.
- No restrictions on error message number to exist in sys.messages.

Using Throw and Catch statements allows for clear error handling and showing error messages.
SQL Server 2005 added TRY/CATCH, which was borrowed from .NET's Try catch model
It brought vast improvements over using @@ERROR
But still we were using RAISERROR for generating errors.
SQL Server 2012 adds THROW
It is a recommended alternative way to generate your own errors
Two usages for THROW:

- With error code, description, and state parameters (like RAISERROR)
- Inside a CATCH block with no parameters (re-throw)

Instructor Notes:

Best Practices

- Verify Input Parameters
- Design Each Stored Procedure to Accomplish a Single Task
- Validate Data Before You Begin Transactions
- Use the Same Connection Settings for All Stored Procedures
- Use WITH ENCRYPTION to Hide Text of Stored Procedures

Example of WITH ENCRYPTION

```
Use AdventureWorks2012
CREATE PROCEDURE HumanResources.uspEncrypthis
 WITH ENCRYPTION
AS
SET NOCOUNT ON;
 SELECT BusinessEntityID, JobTitle, NationalIDNumber, VacationHours, SickLeaveHours FROM HumanResources.Employee;
GO
```

The WITH ENCRYPTION option obfuscates the definition of the procedure when querying the system catalog or using metadata functions, as shown by the following examples.

Run sp_helptext:

```
EXEC sp_helptext 'HumanResources.uspEncryptThis';
```

Here is the result set.

The text for object 'HumanResources.uspEncryptThis' is encrypted.

Instructor Notes:

Demo

- Stored Procedures

Instructor Notes:

Functions

- Named Collections of Transact-SQL Statements
- Takes parameter and returns a single value
- Can be used as a part of expression
- Note : Table data types are also singular value

What is user defined function?
You've seen parameterized functions such as CONVERT, RTRIM, and ISNULL as well as parameterless function such as getdate(). SQL Server 2008 lets you create functions of your own. You should think of the built-in functions as almost like built-in commands in the SQL language; they are not listed in any system table, and there is no way for you to see how they are defined.

Table Variables
To make full use of user-defined functions, it's useful to understand a special type of variable that SQL Server 2008 provides. You can declare a local variable as a table or use a table value as the result set of a user-defined function. You can think of table as a special kind of datatype. Note that you cannot use table variables as stored procedure or function parameters, nor can a column in a table be of type table.
Table variables have a well-defined scope, which is limited to the procedure, function, or batch in which they are declared. Here's a simple example:

```
DECLARE @pricelist TABLE(tid varchar(6), price money)
INSERT @pricelist SELECT title_id, price FROM titles SELECT * FROM @pricelist
```

The definition of a table variable looks almost like the definition of a normal table, except that you use the word DECLARE instead of CREATE, and the name of the table variable comes before the word TABLE.
The definition of a table variable can include:
A column list defining the datatypes for each column and specifying the NULL or NOT NULL property
PRIMARY KEY, UNIQUE, CHECK, or DEFAULT constraints
The definition of a table variable cannot include:
Foreign key references to other tables
Columns referenced by a FOREIGN KEY constraint in another table

Instructor Notes:

Difference between Stored Procedure and Function

Procedure	Function
Return single integer value represents return status	Return single value of any scalar data type supported by SQL server or Table type
Use execute statement to execute stored procedure	Can be called through select statement if it returns scalar value otherwise can be called through from statement if it returns table.
Use output parameter to pass values to caller	Use return statement to pass values to caller

Instructor Notes:

Types of User-Defined Function

- Scalar Functions
  - Similar to a built-in function
- Multi-Statement Table-valued Functions
  - returns a defined table as a result of operations
- In-Line Table-valued Functions
  - Returns a table value as the result of single SELECT statement

Types of function
Scalar function - Specifies the scalar value that the scalar function returns.
Inline table-valued - In inline table-valued functions, the TABLE return value is defined through a single SELECT statement. Inline functions do not have associated return variables

```
CREATE FUNCTION dbo.udf_AuthorsByLetter
(
        @Letter CHAR(1)
)
RETURNS TABLE
AS RETURN
    SELECT * FROM pubs.Authors
      WHERE LEFT(au_lname, 1) = @Letter
```

Multistatement table-valued - In multistatement table-valued functions, @return_variable is a TABLE variable, used to store and accumulate the rows that should be returned as the value of the function.

```
CREATE FUNCTION dbo.udf_FactorialsTAB (@Number int )
  RETURNS @Factorials TABLE (Number INT, Factorial INT)
  AS BEGIN
            DECLARE @I INT, @F INT
            SELECT @I = 1, @F = 1
            WHILE @I < = @N  BEGIN
            SET @F = @I * @F
            INSERT INTO @Factorials VALUES (@I, @F)
            SET @I = @I + 1
            END -- WHILE
  RETURN
END
```

Instructor Notes:

Creating User-Defined Function

```
CREATE Function  udf_GetProductcategory (@i_prodID  INT)
RETURNS  nvarchar(40)
as
 BEGIN
    declare @retvalue  nvarchar(40)
     if  @i_prodID is NULL or @i_prodID <= 0
        return null
    SELECT @retvalue=CategoryName  From
     PRODUCTS , CATEGORIES
     WHERE PRODUCTS.CategoryID = CATEGORIES.CategoryID
      AND  PRODUCTS.ProductID=@i_prodID
       return  @retvalue
END
```

CREATE FUNCTION [schema_name.] function_name
([{ @parameter_name [AS] parameter_data_type
[= default] } [,...n]]) RETURNS return_data_type
 [WITH <function_option> [,...n]]
[AS]
 BEGIN
  function_body
  RETURN scalar_expression
END [;]

schema_name
Is the name of the schema to which the user-defined function belongs.

function_name
Is the name of the user-defined function. Function names must comply with the rules for identifiers and must be unique within the database and to its schema.

@parameter_name
Is a parameter in the user-defined function. One or more parameters can be declared.
A function can have a maximum of 1,024 parameters. The value of each declared parameter must be supplied by the user when the function is executed, unless a default for the parameter is defined. Specify a parameter name by using an at sign (@) as the first character. The parameter name must comply with the rules for identifiers. Parameters are local to the function; the same parameter names can be used in other functions. Parameters can take the place only of constants; they cannot be used instead of table names, column names, or the names of other database objects.

Instructor Notes:

Restrictions on Functions

- Restrictions on Functions
  - A function can return only single value at a time
  - The SQL statements within a function cannot include any nondeterministic system functions.
    - E.g getdate() function is nondeterministic hence cannot be used inside function but can be pass as argument

return_data_type

Is the return value of a scalar user-defined function. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except the **timestamp** data type. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, and **timestamp** data types. The nonscalar types **cursor** and **table** cannot be specified as a return data type in either Transact-SQL or CLR functions.

Restrictions on function

•A function can return only single value at a time

•The SQL statements within a function cannot include any nondeterministic system functions.

•E.g getdate() function is nondeterministic hence cannot be used inside function but can be pass as argument.

Instructor Notes:

Creating a Function with Schema Binding

- Schema binding prevents the altering or dropping of any object on which the function depends.
- If a schema-bound function references TableA, then columns may be added to TableA, but no existing columns can be altered or dropped, and neither can the table itself.
- Schema binding not only alerts the developer that the change may affect an object, it also prevents the change.
- To remove schema binding so that changes can be made, ALTER the function so that schema binding is no longer included.

```
Use with schemabinding
CREATE FUNCTION FunctionName (Input Parameters)
        RETURNS DataType
        WITH SCHEMA BINDING
        AS
        BEGIN;
        Code;
        RETURNS Expression;
        END;
```

SCHEMABINDING
Specifies that the function is bound to the database objects that it references. This condition will prevent changes to the function if other schema-bound objects are referencing it.
The binding of the function to the objects it references is removed only when one of the following actions occurs:

- The function is dropped.
- The function is modified by using the ALTER statement with the SCHEMABINDING option not specified.

A function can be schema bound only if the following conditions are true:

- The function is a Transact-SQL function.
- The user-defined functions and views referenced by the function are also schema-bound.
- The objects referenced by the function are referenced using a two-part name.
- The function and the objects it references belong to the same database.
- The user who executed the CREATE FUNCTION statement has REFERENCES permission on the database objects that the function references.

. Although it is true that stored procedures cannot be schema bound, there is a new feature in SQL Server 2012 called Result Sets that can guarantee the structure of the returned results at run time.

Instructor Notes:

Altering and Dropping User-defined Functions

- Altering Functions
  - Retains assigned permissions
  - Causes the new function definition to replace existing definition

```
ALTER FUNCTION dbo.fn_NewRegion
<New function content>
```

- Dropping Functions
  - DROP FUNCTION dbo.fn_NewRegion

ALTER FUNCTION
Alters an existing Transact-SQL or CLR function that was previously created by executing the CREATE FUNCTION statement, without changing permissions and without affecting any dependent functions, stored procedures, or triggers.
Syntax
Scalar Functions

```
ALTER FUNCTION [ schema_name. ] function_name
 ( [ { @parameter_name [ AS ]  parameter_data_type     [ = default ] }     [ ,...n
]   ] ) RETURNS return_data_type
[ AS ]
 BEGIN
       function_body
       RETURN scalar_expression
END [ ; ]
```

All arguments have same meaning as it is in create function

DROP FUNCTION
Removes one or more user-defined functions from the current database. User-defined functions are created by using create function and modified by using alter function.

Instructor Notes:

Using Scalar User-Defined Function

- RETURNS Clause Specifies Data Type
- Function Is Defined Within a BEGIN and END Block
- Return Type Is Any Data Type Except text, ntext, image, cursor, or timestamp

```
Use AdventureWorks2012
        GO
        CREATE FUNCTION dbo.ufnGetOrderTotalByProduct(@ProductID int)
        RETURNS int
        AS
        BEGIN
        DECLARE @OrderTotal int;
        SELECT @OrderTotal = sum(sod.OrderQty)
        FROM Production.Product p
        JOIN Sales.SalesOrderDetail sod
        ON p.ProductID = sod.ProductID
        WHERE p.ProductID = @ProductID
        GROUP BY p.ProductID;
        RETURN @OrderTotal;
        END;
        GO
Call the function with select statement
      USE AdventureWorks2012;
        GO
        SELECT p.Name, dbo.ufnGetOrderTotalByProduct(p.ProductID) as OrderTotal
        FROM Production.Product p
        ORDER BY OrderTotal DESC;
      Result of the Query

        Name          OrderTotal
        ---------------------------------------------------- -----------
        AWC Logo Cap                  8311
        Water Bottle - 30 oz.         6815
        Sport-100 Helmet, Blue        6743
        Long-Sleeve Logo Jersey, L    6592
        Sport-100 Helmet, Black       6532
        Sport-100 Helmet, Red         6266
        Classic Vest, S               4247
```

Instructor Notes:

Using Scalar User-Defined Function

Scalar functions may be used anywhere within any expression that accepts a single value. User-defined scalar functions must always be called by means of at least a two-part name(owner.name). The above script demonstrates calling the ufnGetOrderTotalByProduct function within AdventureWorks2012. In this case, you ask for the order total for each ProductID in the Production.Product table. You could pass a single value into the scalar function as in the previous example, but the scalar function enables you to do something a little more complex. You can use the ProductID from the table you're querying as the parameter value for the UDF. This means that the UDF will be called once per ProductID returned by the query.

Example 1:

```
USE Northwind
CREATE FUNCTION udf_DateFormat
  (@indate datetime, @separator char(1))
RETURNS Nchar(20)
AS
BEGIN
  RETURN
  CONVERT(Nvarchar(20), datepart(mm,@indate))
  + @separator
  + CONVERT(Nvarchar(20), datepart(dd, @indate))
  + @separator
  + CONVERT(Nvarchar(20), datepart(yy, @indate))
END
--Call the function

--Function is called from select statement

SELECT dbo.fn_DateFormat(GETDATE(), ':')
```

Example 2:

```
-Write getordercount function which return no of orders placed by the customer.
use northwind
create function getordercount
(@custcode varchar(10))  returns int
as
begin
declare @cnt int
select @cnt = count(*) from orders
where customerid = @custcode
return @cnt
end
-- Call the function
select companyname,city,dbo.getordercount(customerid)
from customers
```

Example 3:

The following user-defined scalar function performs a simple mathematical function. The second parameter includes a default value:

```
CREATE FUNCTION dbo.ufnCalculateQuotient
(@Numerator numeric(5,2),
@Denominator numeric(5,2)= 1.0)
RETURNS numeric(5,2)
AS
BEGIN;
RETURN @Numerator/@Denominator;
END;
GO
SELECT dbo.ufnCalculateQuotient(12.1,7.45),
dbo.ufnCalculateQuotient (7.0,DEFAULT);
Result:
---------- ----------
```

Instru 1.62 7.00

Instructor Notes:

In-Line Table Valued Function

- An inline table-valued user-defi ned function retains the benefi ts of a view, and adds parameters.
- The inline table-valued user-defi ned function has no BEGIN/END body.
-

```
CREATE FUNCTION FunctionName (InputParameters)
RETURNS Table
AS
RETURN (Select Statement);.
```

As with a view, if the SELECT statement is updatable, then the function is also updatable.

```
Use AdventureWorks2012
    CREATE FUNCTION
    dbo.ufnGetOrderTotalByProductCategory(@ProductCategoryID int)
    RETURNS TABLE
    AS
    RETURN
    (
    SELECT p.ProductID, p.Name, sum(sod.OrderQty) as TotalOrders
    FROM Production.Product p
    JOIN Sales.SalesOrderDetail sod
    ON p.ProductID = sod.ProductID
    JOIN Production.ProductSubcategory s
    ON p.ProductSubcategoryID = s.ProductSubcategoryID
    JOIN Production.ProductCategory c
    ON s.ProductCategoryID = c.ProductCategoryID
    WHERE c.ProductCategoryID = @ProductCategoryID
    GROUP BY p.ProductID, p.Name
    );
    GO
```

SELECT statement is returned as a virtual table:
To call the function

```
SELECT ProductID, Name, TotalOrders FROM dbo.ufnGetOrderTotalByProductCategory(1)
ORDER BY TotalOrders DESC;
```

Instructor Notes:

Multi-Statement Table Valued Function

- The multi-statement table-valued, user-defined function combines the scalar function's capability to contain complex code with the inline table-valued function's capability to return a result set.
- This type of function creates a table variable and then populates it within code.
- The table is then passed back from the function so that it may be used within SELECT statements.

The primary benefit of the multi-statement table-valued, user-defined function is that complex result sets may be generated within code and then easily used with a SELECT statement. This enables you to build complex logic into a query and solve problems that would otherwise be difficult to solve without a cursor.

Example of Multi-Statement Table Valued Function
The following process builds a multistatement table-valued, user-defi ned function that returns a basic result set:
1. The function fi rst creates a table variable called @ProductList within the CREATE FUNCTION header.
2. Within the body of the function, two INSERT statements populate the @ProductList table variable.
3. When the function completes execution, the @ProductList table variable is passed back as the output of the function.
The dbo.ufnGetProductsAndOrderTotals function returns every product in the Production.Product table and the order total for each product:

```
USE AdventureWorks2012;
GO
CREATE FUNCTION dbo.ufnGetProductsAndOrderTotals()
RETURNS @ProductList TABLE
(ProductID int,
ProductName nvarchar(100),
TotalOrders int)
AS
BEGIN;
INSERT @ProductList(ProductID, ProductName)
SELECT ProductID, Name
FROM Production.Product;
UPDATE pl
SET TotalOrders =
(SELECT sum(sod.OrderQty)
FROM @ProductList ipl
JOIN Sales.SalesOrderDetail sod
ON ipl.ProductID = sod.ProductID
WHERE ipl.ProductID = pl.ProductID)
FROM @ProductList pl;
RETURN;
END;
```

Calling the Function
To execute the function, refer to it within the FROM portion of a SELECT statement. The following code retrieves the result from the dbo.ufnGetProductsAndOrderTotals function:

```
SELECT ProductID, ProductName, TotalOrders
FROM dbo.ufnGetProductsAndOrderTotals()
ORDER BY TotalOrders DESC;
```

Instructor Notes:

Example

```
CREATE FUNCTION FunctionName (InputParamenters)
RETURNS @TableName TABLE (Columns)
AS
BEGIN;
Code to populate table variable
RETURN;
END;
```

Instructor Notes:

View Definition of Function

```
SELECT definition, type
FROM sys.sql_modules AS m
JOIN sys.objects AS o ON m.object_id = o.object_id
AND type IN ('FN', 'IF', 'TF');
GO
```

Instructor Notes:

Types of Functions

- Scalar user-defi ned functions return a single value and must be deterministic.
- Inline table-valued user-defi ned functions are similar to views and return the results of a single SELECT statement.
- Multistatement, table-valued, user-defi ned functions use code to populate a table variable, which is then returned.

Instructor Notes:

Best Practices

- Choose inline table-valued functions over multistatement table-valued functions whenever possible.
- Even if it looks like you need a scalar function, write it as an inline table-valued function avoid scalar functions wherever possible.
- If you need a multistatement table-valued function, check to see if a stored procedure might be the appropriate solution. This might require a broader look at query structure, but it's worth taking the time to do it.

Instructor Notes:

Demo

- Creating User-Defined Functions

Instructor Notes:

The structure of the Category and Product table is as follows

```
CREATE TABLE Category
(
        CategoryID INT PRIMARY KEY,
        CategoryName VARCHAR(20)
)


CREATE TABLE Product
(
        ProductID INT PRIMARY KEY,
        ProductName VARCHAR(30),
        UnitPrice   MONEY,
        Quantity INT,
        CategoryID INT
        FOREIGN KEY REFERENCES Category(CategoryID)
)
```

Instructor Notes:

Ex. 1 : Insert record in Category table using Stored Procedure

```
CREATE PROC usp_InsertCategory
(
	@CatID   INT,
	@CatName                VARCHAR(20)
)
AS
BEGIN
	IF(@CatID IS NULL OR @CatID < 0)
	BEGIN
		RAISERROR('Category ID cannot be null or negative', 1, 1)
	END
	ELSE
	BEGIN
		IF EXISTS (SELECT CategoryID FROM Category WHERE
CategoryID = @CatID)
		BEGIN
			RAISERROR('Category ID already exists', 1, 1)
		END
		ELSE
		BEGIN
			INSERT INTO Category
			(CategoryID, CategoryName)
			VALUES
			(@CatID, @CatName)
		END
	END
END
```

Select the procedure and press F5, to create the procedure.
Execute procedure as follows :

```
EXEC usp_InsertCategory 1, 'Bikes'
```

OR

```
EXEC usp_InsertCategory @CatName = 'Electronics', @CatID = 2
```

Instructor Notes:

Ex. 2 : Insert record in Product table using Stored Procedure

```
CREATE PROC usp_InsertProduct
(
	@ProdID INT,
	@ProdName VARCHAR(20),
	@Price MONEY,
	@Qty INT,
	@CatID INT
)
AS
BEGIN
	IF(@ProdID IS NULL OR @ProdID < 0)
	BEGIN
		RAISERROR('Product ID cannot be null or negative', 1, 1)
	END
	ELSE
	BEGIN
		IF EXISTS (SELECT ProductName FROM Product WHERE
ProductID = @ProdID)
		BEGIN
			RAISERROR('Product ID already exists', 1, 1)
		END
		ELSE
		BEGIN
			IF EXISTS (SELECT CategoryName FROM Category
WHERE CategoryID = @CatID OR @CatID IS NULL)
			BEGIN
				IF (@Price <= 0 OR @Qty <=0)
				BEGIN
					RAISERROR('Unit Price or Quantity cannot be
negative or zero', 1, 1)
				END
				ELSE
				BEGIN
					INSERT INTO Product
					(ProductID, ProductName, UnitPrice, Quantity,
CategoryID)
					VALUES
					(@ProdID, @ProdName, @Price, @Qty,
@CatID)
				END
			END
			ELSE
			BEGIN
				RAISERROR('Category ID does not exists', 1, 1)
			END
		END
	END
END
```

Select the procedure and press F5, to create the procedure.
Execute procedure as follows :

```
EXEC usp_InsertProduct 101, 'Cover', 400, 5, 1
```

Instructor Notes:

Ex. 3 : Update record of Category table using Stored Procedure

```
CREATE PROC usp_UpdateCategory
(
        @CatID   INT,
        @CatName                VARCHAR(20)
)
AS
BEGIN
        IF(@CatID IS NULL OR @CatID < 0)
        BEGIN
                RAISERROR('Category ID cannot be null or negative', 1, 1)
        END
        ELSE
        BEGIN
                IF EXISTS (SELECT CategoryID FROM Category WHERE
CategoryID = @CatID)
                BEGIN
                        UPDATE Category
                        SET CategoryName = @CatName
                        WHERE CategoryID = @CatID
                END
                ELSE
                BEGIN
                        RAISERROR('Category ID not exists', 1, 1)
                END
        END
END
```

Select the procedure and press F5, to create the procedure.
Execute procedure as follows :

```
EXEC usp_UpdateCategory 1, 'Bikes Accessories'
```

Instructor Notes:

Ex. 4 : Update record of Product table using Stored Procedure

```
CREATE PROC usp_UpdateProduct
(
	@ProdID INT,
	@ProdName VARCHAR(20),
	@Price MONEY,
	@Qty INT,
	@CatID INT
)
AS
BEGIN
	IF(@ProdID IS NULL OR @ProdID < 0)
	BEGIN
		RAISERROR('Product ID cannot be null or negative', 1, 1)
	END
	ELSE
	BEGIN
		IF EXISTS (SELECT ProductName FROM Product WHERE
ProductID = @ProdID)
		BEGIN
			IF EXISTS (SELECT CategoryName FROM Category
WHERE CategoryID = @CatID OR @CatID IS NULL)
			BEGIN
				IF (@Price <= 0 OR @Qty <=0)
				BEGIN
					RAISERROR('Unit Price or Quantity cannot be
negative or zero', 1, 1)
				END
				ELSE
				BEGIN
					UPDATE Product SET
					ProductName = @ProdName,
					UnitPrice = @Price,
					Quantity = @Qty,
					CategoryID = @CatID
					WHERE ProductID = @ProdID
				END
			END
			ELSE
			BEGIN
				RAISERROR('Category ID does not exists', 1, 1)
			END
		END
		ELSE
		BEGIN
			RAISERROR('Product ID does not exists', 1, 1)
		END
	END
END
```

Select the procedure and press F5, to create the procedure.
Execute procedure as follows :

```
EXEC usp_UpdateProduct 101, 'Cover', 400, 4, 1
```

Instructor Notes:

Ex. 5 : Delete record from Category table using Stored Procedure

```
CREATE PROC usp_DeleteCategory
(
	@CatID   INT
)
AS
BEGIN
	IF(@CatID IS NULL OR @CatID < 0)
	BEGIN
		RAISERROR('Category ID cannot be null or negative', 1, 1)
	END
	ELSE
	BEGIN
		IF EXISTS (SELECT CategoryID FROM Category WHERE
CategoryID = @CatID)
		BEGIN
			DELETE FROM Category WHERE CategoryID = @CatID
		END
		ELSE
		BEGIN
			RAISERROR('Category ID not exists', 1, 1)
		END
	END
END

EXEC usp_DeleteCategory 2
```

Ex. 6 : Delete record from Product table using Stored Procedure

```
CREATE PROC usp_DeleteProduct
(
	@ProdID INT
)
AS
BEGIN
	IF(@ProdID IS NULL OR @ProdID < 0)
	BEGIN
		RAISERROR('Product ID cannot be null or negative', 1, 1)
	END
	ELSE
	BEGIN
		IF EXISTS (SELECT ProductName FROM Product WHERE
ProductID = @ProdID)
		BEGIN
			DELETE FROM Product WHERE ProductID = @ProdID
		 END
		 ELSE
		BEGIN
			RAISERROR('Product ID does not exists', 1, 1)
		END
	END
END
```

Instructor Notes:

Ex. 7 : Stored procedure to display all products information with category name

```
CREATE PROC usp_DisplayAllProducts
AS
BEGIN
	SELECT p.ProductID, p.ProductName, p.UnitPrice, p.Quantity,
c.CategoryName
	FROM Product  p INNER JOIN Category c
	ON p.CategoryID = c.CategoryID
END
```

Execute the stored procedure by using record sets as follows :

```
EXEC usp_DisplayAllProducts WITH RESULT SETS
((PID INT,
PName VARCHAR(30),
Price MONEY,
Quantity INT,
Category VARCHAR(20)))
```

Ex. 8 : Stored procedure to display all products as per category

```
CREATE PROC usp_DisplayProductCategoryWise
(
	@CatID    INT
)
AS
BEGIN
	IF EXISTS (SELECT CategoryName FROM Category WHERE
CategoryID = @CatID)
	BEGIN
		SELECT ProductID, ProductName, UnitPrice, Quantity
		FROM Product
		WHERE CategoryID = @CatID
	END
	ELSE
	BEGIN
		RAISERROR('Category ID does not exists', 1, 1)
	END
END

EXEC usp_DisplayProductCategoryWise 1
```

Instructor Notes:

Ex. 9 : Stored procedure to search product

```
CREATE PROC usp_SearchProduct
(
	@ProdID   INT
)
AS
BEGIN
	IF EXISTS (SELECT ProductName FROM Product WHERE ProductID =
@ProdID)
	BEGIN
		SELECT ProductID, ProductName, UnitPrice, Quantity,
CategoryName
		FROM Product INNER JOIN Category
		ON Product.CategoryID = Category.CategoryID
		WHERE ProductID = @ProdID
	END
	ELSE
	BEGIN
		RAISERROR('Product ID does not exists', 1,1)
	END
END

EXEC usp_SearchProduct 101
```

Instructor Notes:

Summary

- In this lesson, you have learnt:
- Creating, Executing, Modifying, and Dropping Stored Procedures
- Using Parameters in Stored Procedures
- Using User defined Functions
  - Scalar User-defined Function
  - Multi-Statement Table-valued Function
  - In-Line Table-valued Function

Instructor Notes:

Answers for the Review Questions:

Answer 1: True
Answer 2: Extended

Review Question

- Question 1: A stored procedure can return a single integer value
  - True
  - False
- Question 2: ---------- stored procedures call subroutines written in languages like c, c++, .NET
- Question 3: ---------- function includes only one select statement