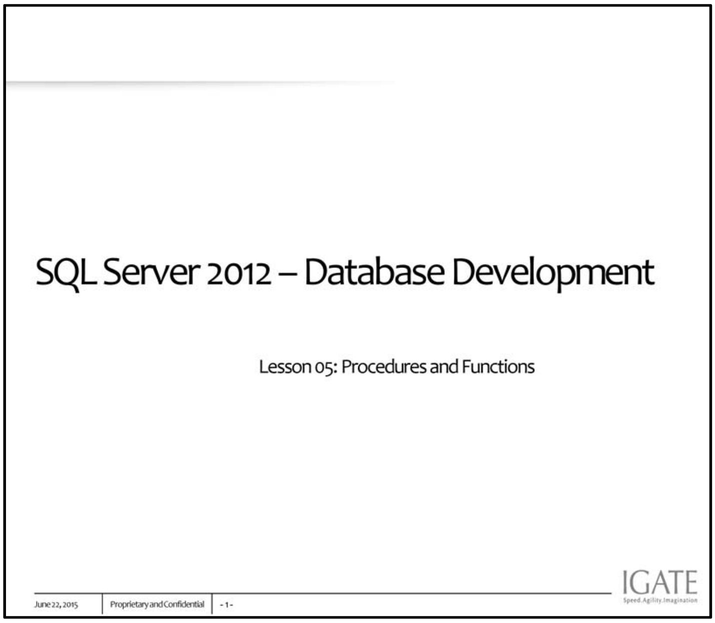
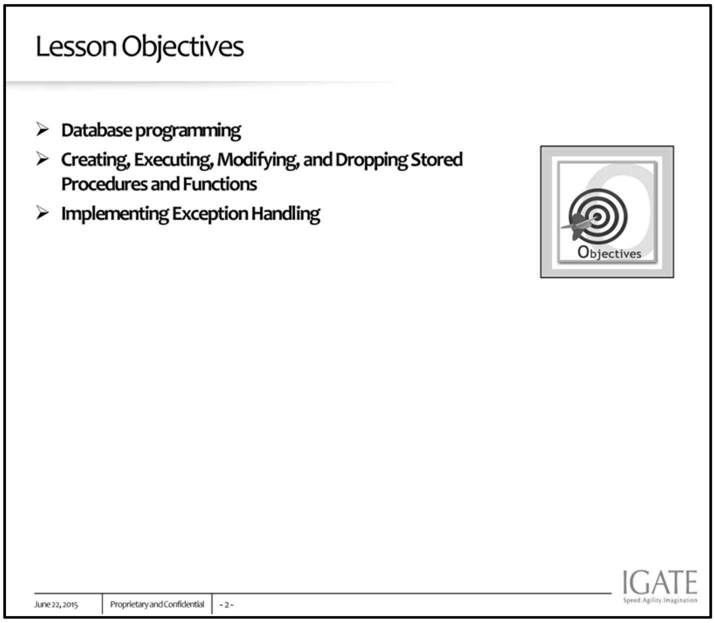


SQL Server 2012 – Database Development

Lesson 05: Procedures and Functions

June 22, 2015 | Proprietary and Confidential | - 1 -

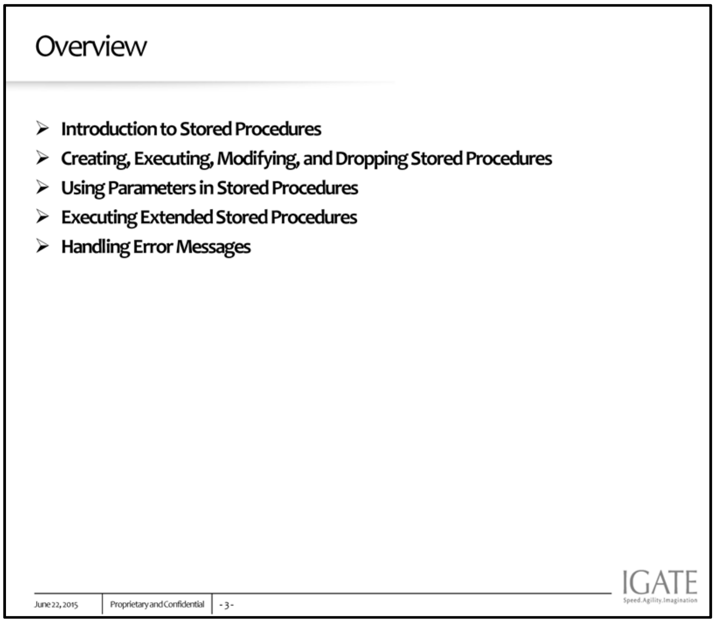


Lesson Objectives

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

June 22, 2015 | Proprietary and Confidential | - 2 -

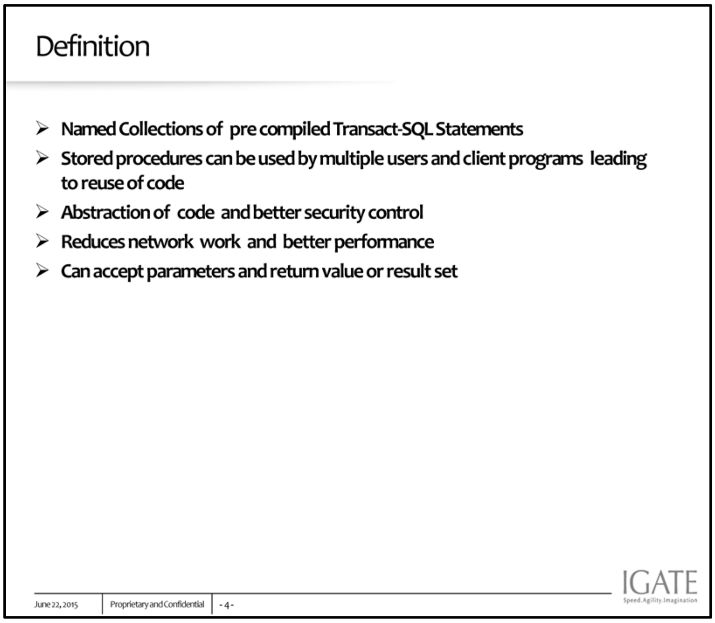
IGATE
Speed.Agility.Imagination



Overview

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

June 22, 2015 | Proprietary and Confidential | - 3 -



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

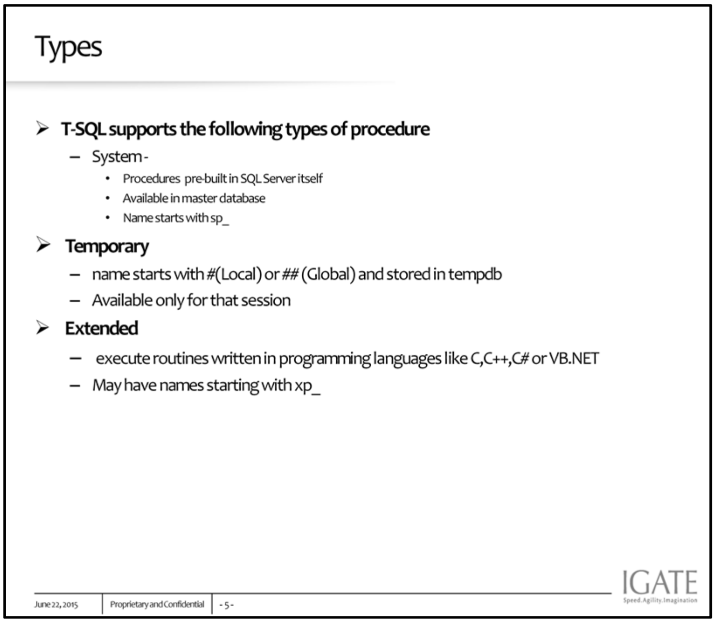
June 22, 2015 | Proprietary and Confidential | - 4 -

IGATE
Speed.Agility.Imagination

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.



Types

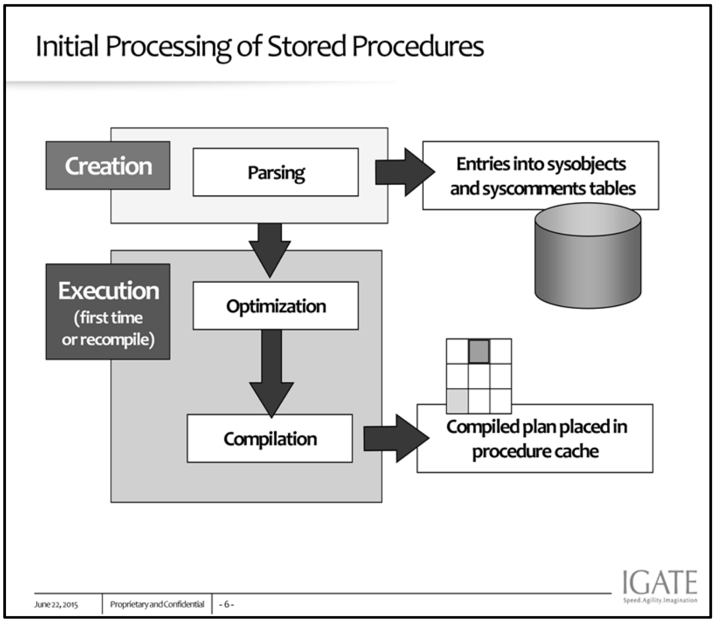
- T-SQL supports the following types of procedure
  - System -
    - Procedures pre-built in SQL Server itself
    - Available in master database
    - Name starts with sp_
- **Temporary**
  - name starts with #(Local) or ## (Global) and stored in tempdb
  - Available only for that session
- **Extended**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

June 22, 2015 | Proprietary and Confidential | - 5 -

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.



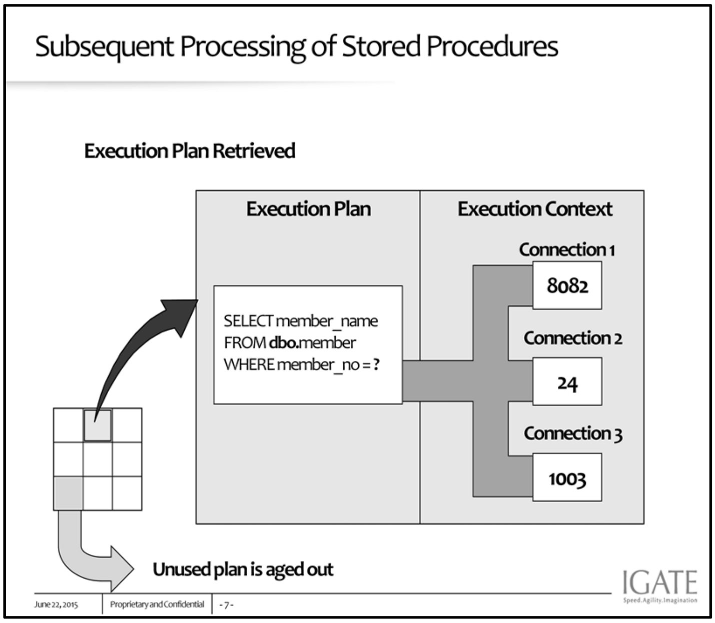
Initial Processing of Stored Procedures

Creation
Parsing
Entries into sysobjects and syscomments tables
Execution (first time or recompile)
Optimization
Compilation
Compiled plan placed in procedure cache

June 22, 2015 | Proprietary and Confidential | - 6 -

When you create a stored procedure
SQL server parses your code and make entries in appropriate system table.
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.
SQL server generates multiple permutation of execution plan and choose the one with lowest cost

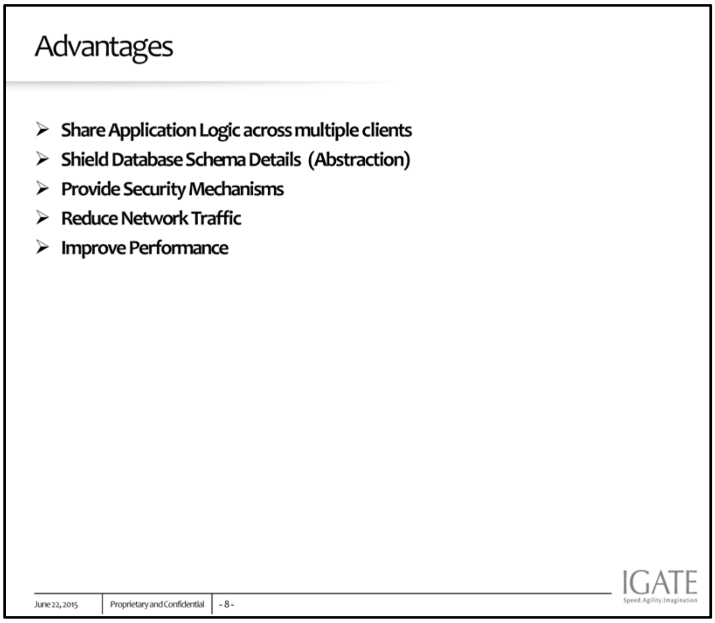
After execution of query or procedure we can see the execution plan in result window by clicking on display estimated execution plan button in the tool bar.



Subsequent Processing of Stored Procedures

```
SELECT member_name
FROM dbo.member
WHERE member_no = ?
```

3. Stored procedures can reuse a previously cached execution plan, which saves the resources involved in generating a new execution plan
4. 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.

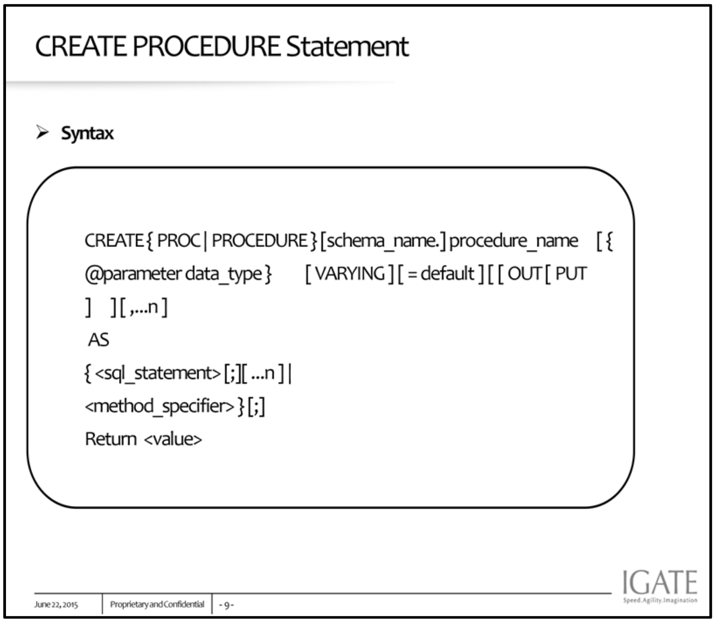


Advantages

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

June 22, 2015 | Proprietary and Confidential | - 8 -

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.



CREATE PROCEDURE Statement

➢ **Syntax**

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

June 22, 2015 | Proprietary and Confidential | - 9 -

IGATE
Speed.Agility.Imagination

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name
   [ { @parameter [ type_schema_name. ] data_type }       [ [ 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.

@ *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.

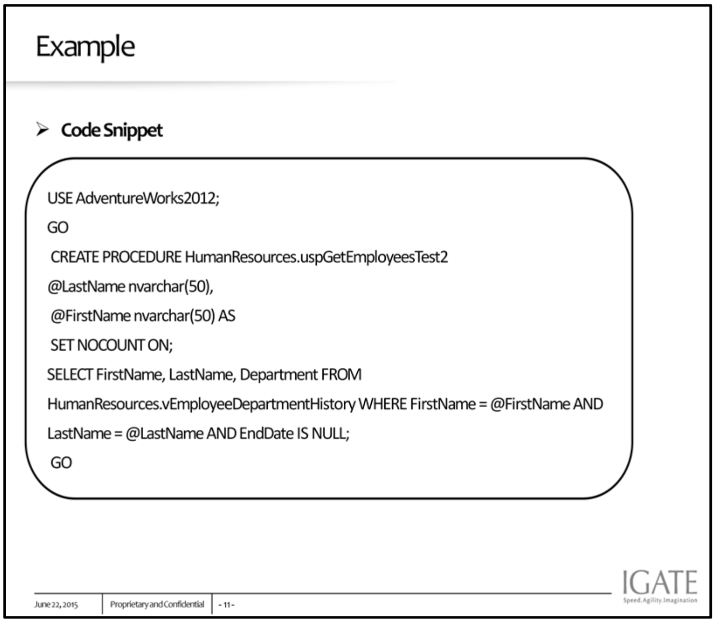
Specify a parameter name by using an at sign (**@**) as the first character. The parameter name must comply with the rules for identifiers. Parameters are local to the procedure; the same parameter names can be used in other procedures. By default, parameters can take the place only of constant expressions; 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.



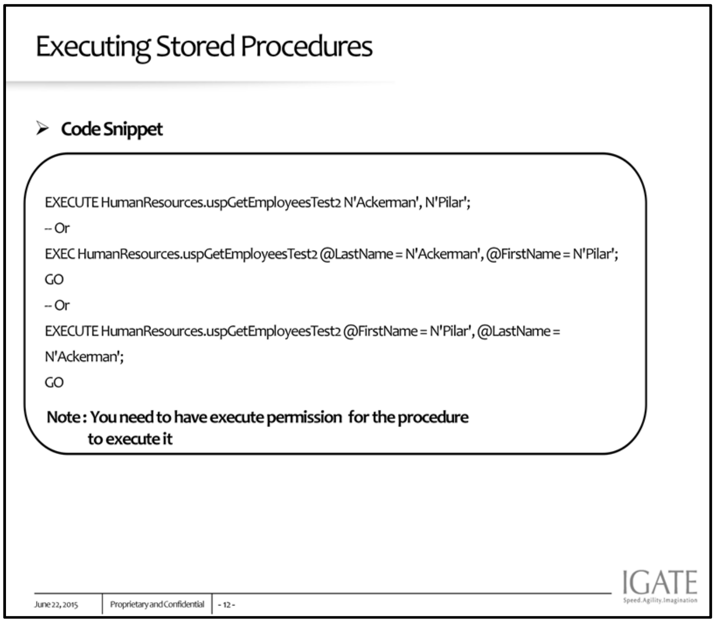
Example

➢ **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
```

June 22, 2015 | Proprietary and Confidential | - 11 -

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



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

June 22, 2015 | Proprietary and Confidential | - 12 -

IGATE
Speed.Agility.Imagination

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, scalar-valued user-defined function, 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
```

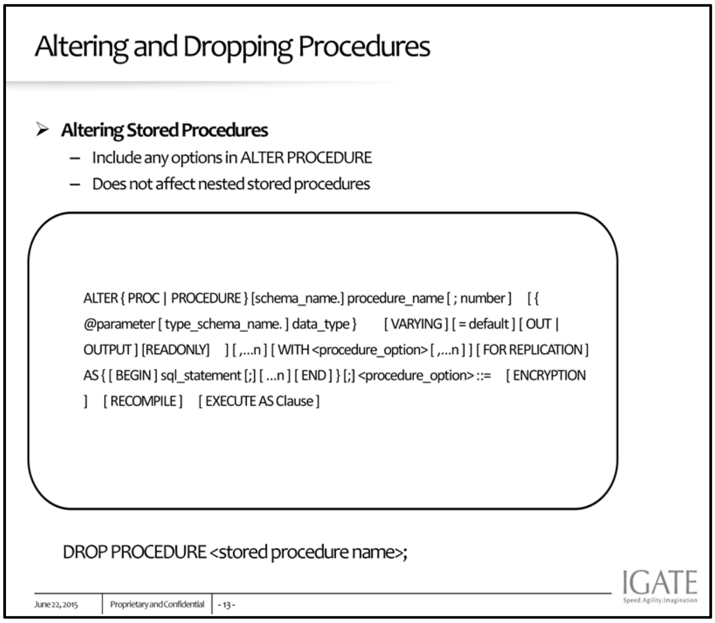
Executing a Stored Procedure Within an INSERT Statement

```
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
```



Altering and Dropping Procedures

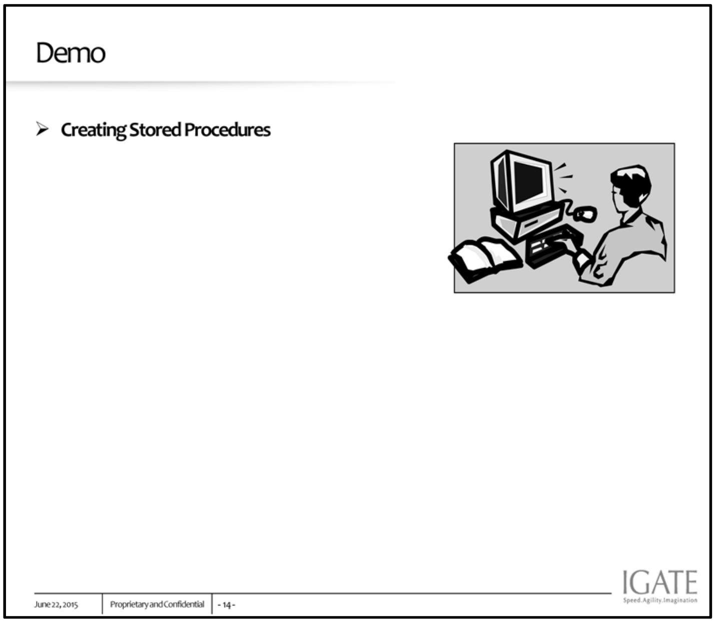
- **Altering Stored Procedures**
  - Include any options in ALTER PROCEDURE
  - Does not affect nested stored procedures

```
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>;

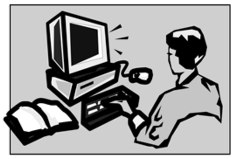
June 22, 2015 | Proprietary and Confidential | - 13 -

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.



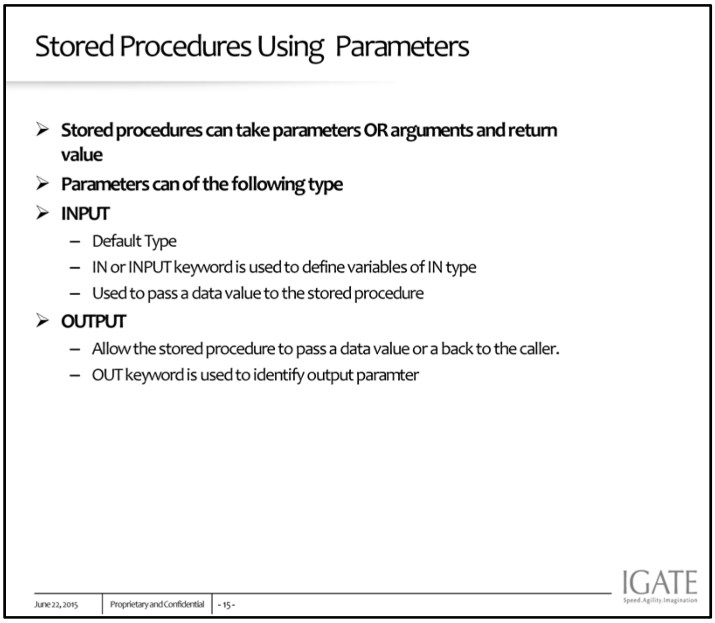
Demo

- Creating Stored Procedures



June 22, 2015 | Proprietary and Confidential | - 14 -

IGATE
Speed.Agility.Imagination



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 paramter

June 22, 2015 | Proprietary and Confidential | - 15 -

IGATE
Speed.Agility.Imagination

Parameters are used to exchange data between stored procedures and functions that called the stored procedure or function:

Input parameters allow the caller to pass a data value to the stored procedure or function.

Output parameter

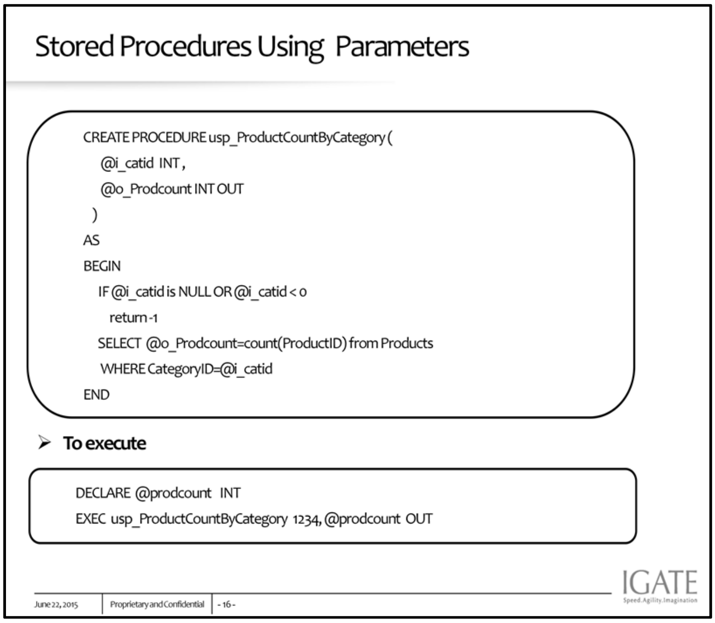
Output parameters allow the stored procedure to pass a data value or a cursor variable back to the caller.

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'
```



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
```

June 22, 2015 | Proprietary and Confidential | - 16 -

IGATE
Speed.Agility.Imagination

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 valkidated 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

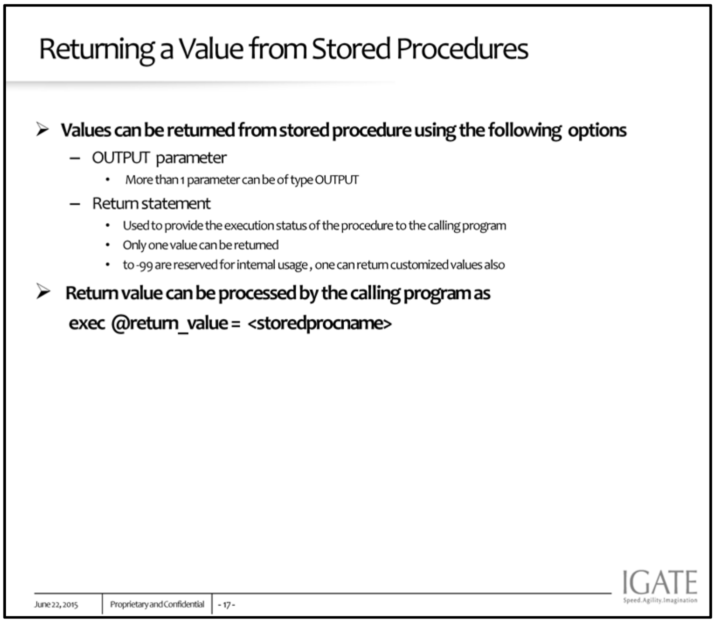
```
Exec   usp_ProductCountByCategor DEFAULT, @pcount OUTPUT
```

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
```



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>

June 22, 2015 | Proprietary and Confidential | - 17 -

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.

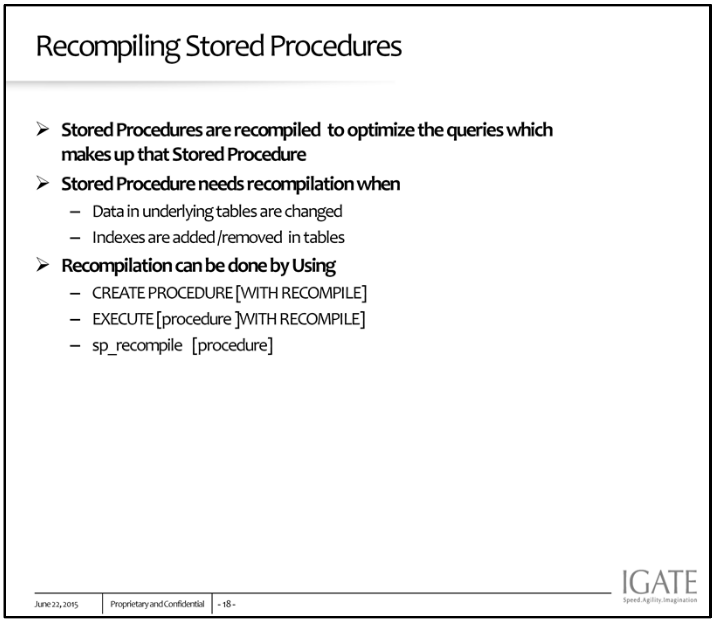
Example 1:

```
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
```



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]

June 22, 2015 | Proprietary and Confidential | - 18 -

IGATE Speed.Agility.Imagination

Example of Procedure with Recompile

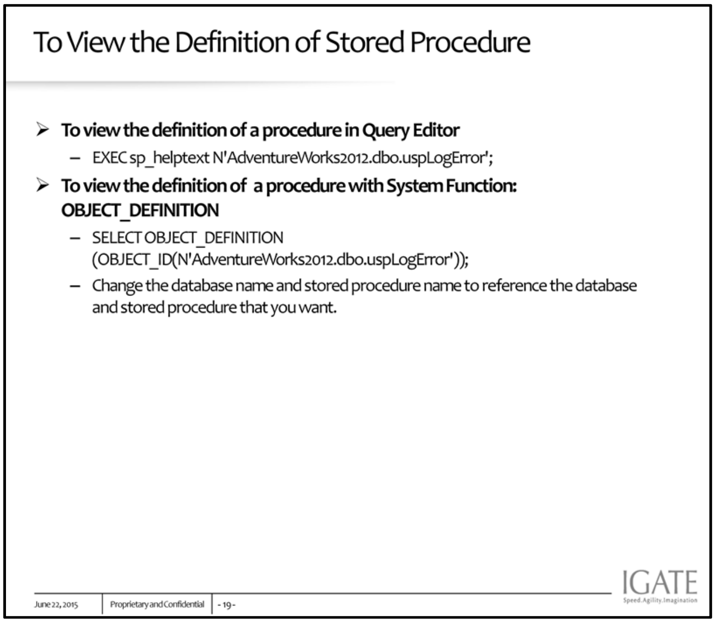
```
USE AdventureWorks2012;
CREATE PROCEDURE dbo.uspProductByVendor @Name varchar(30) = '%'
WITH RECOMPILE
 AS
SET NOCOUNT ON;
SELECT v.Name AS 'Vendor name', p.Name AS 'Product name' FROM
Purchasing.Vendor AS v JOIN Purchasing.ProductVendor AS pv ON
v.BusinessEntityID = pv.BusinessEntityID JOIN Production.Product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

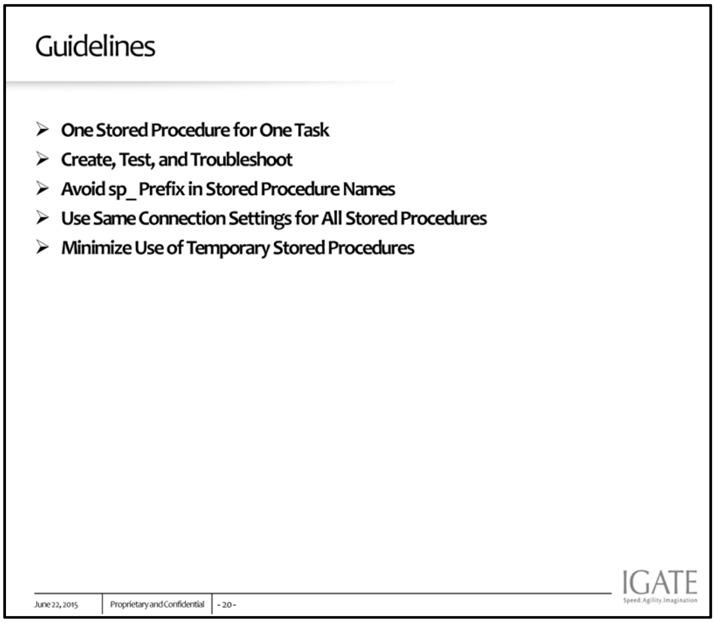


To View the Definition of Stored Procedure

- **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.

June 22, 2015 | Proprietary and Confidential | - 19 -

IGATE
Speed.Agility.Imagination

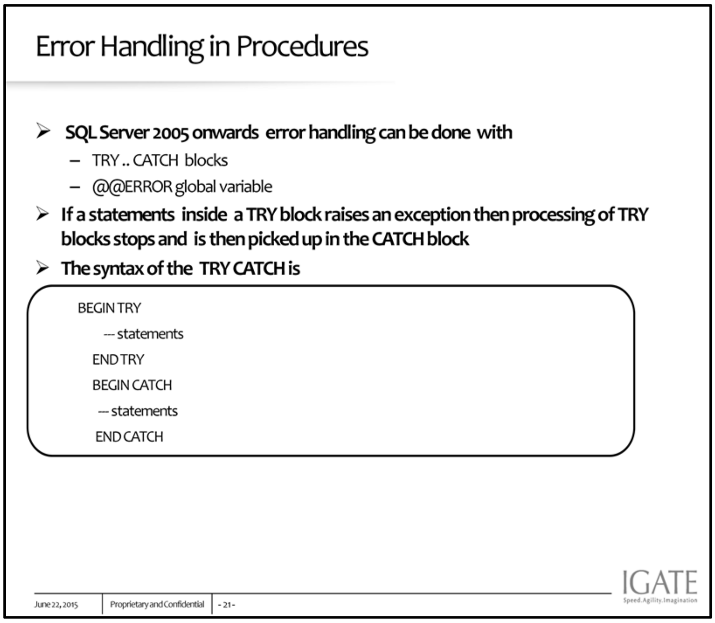


Guidelines

- 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

June 22, 2015 | Proprietary and Confidential | - 20 -

IGATE
Speed.Agility.Imagination



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
```

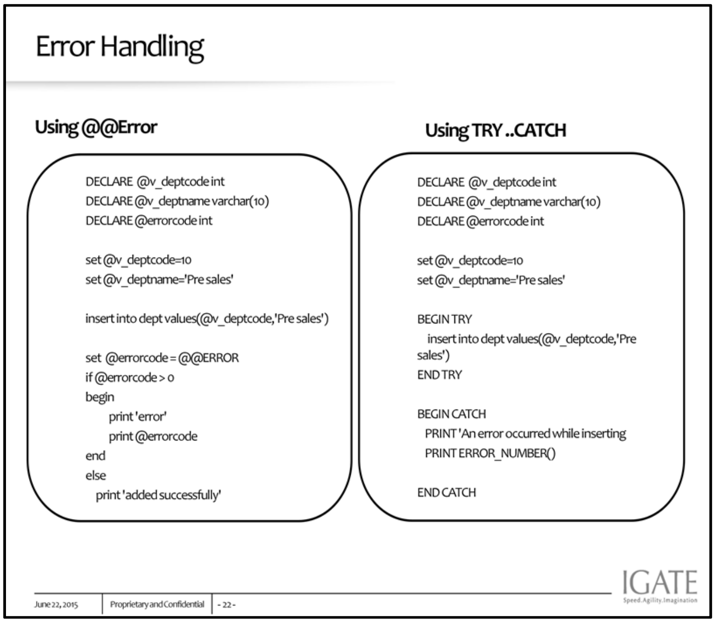
June 22, 2015 | Proprietary and Confidential | - 21 -

IGATE
Speed.Agility.Imagination

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



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
```

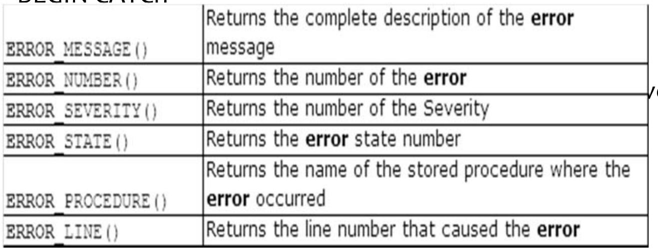
June 22, 2015 | Proprietary and Confidential | - 22 -

IGATE
Speed.Agility.Imagination

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

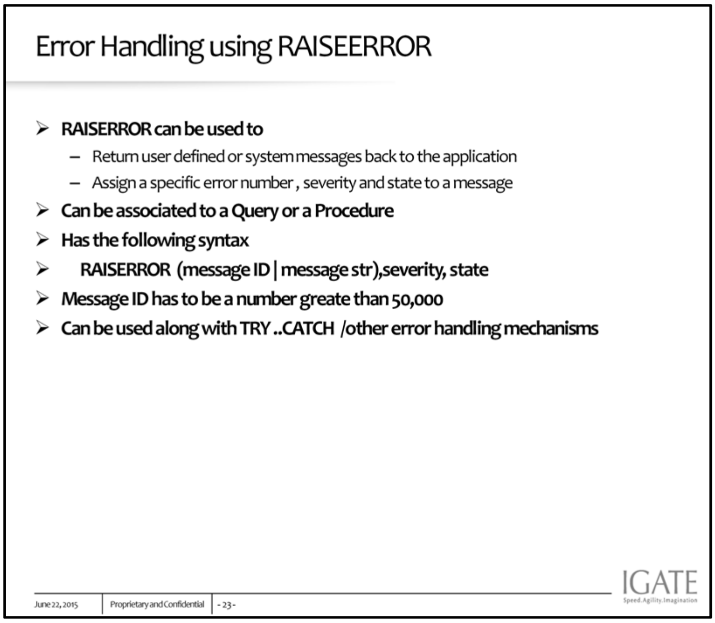
The general syntax of the TRY..CATCH is as follows

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



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**

ver



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 str),severity, state
- Message ID has to be a number greate than 50,000
- Can be used along with TRY ..CATCH /other error handling mechanisms

June 22, 2015 | Proprietary and Confidential | - 23 -

IGATE Speed.Agility.Imagination

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.

@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.

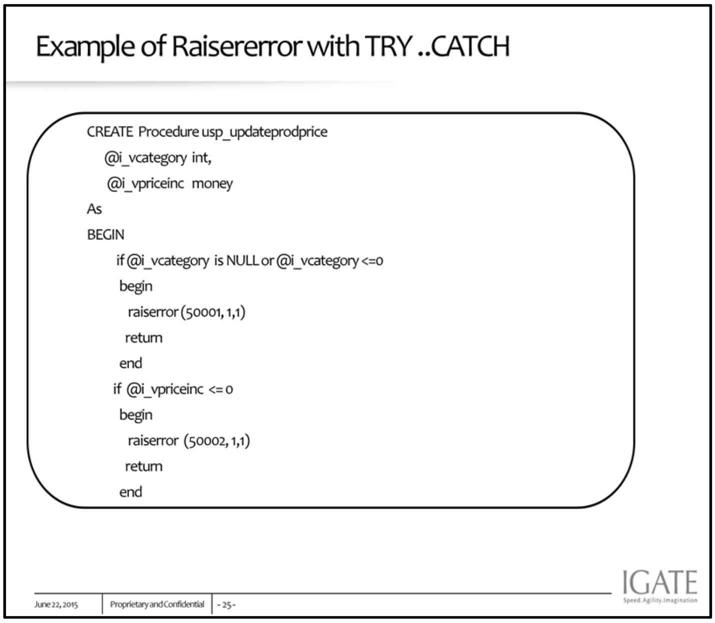
RAISERROR can return either:

- 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



Example of Raisererror 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
```

June 22, 2015 | Proprietary and Confidential | - 25 -

The structure of the new table is as follows

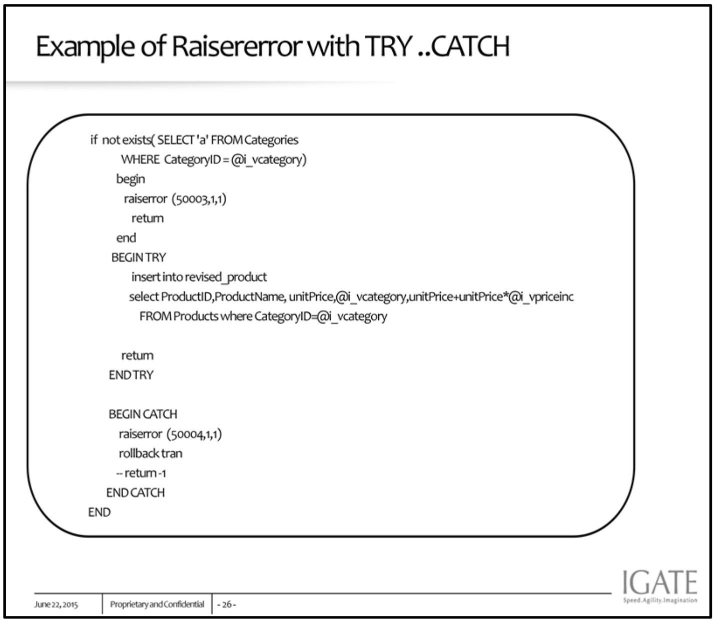
```
create table revised_product
   (ProductID  int not null,
    ProductName  nvarchar(80),
    unitPrice money,
    CategoryID  int,
    revisedprice  money
)
```

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
```



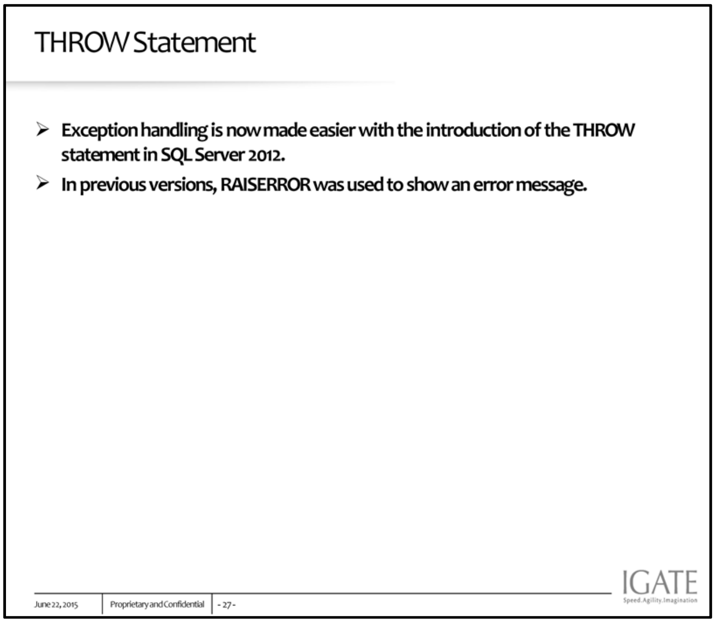
Example of Raisererror with TRY ..CATCH

```
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
```

June 22, 2015 | Proprietary and Confidential | - 26 -



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.

June 22, 2015 | Proprietary and Confidential | - 27 -

IGATE
Speed.Agility.Imagination

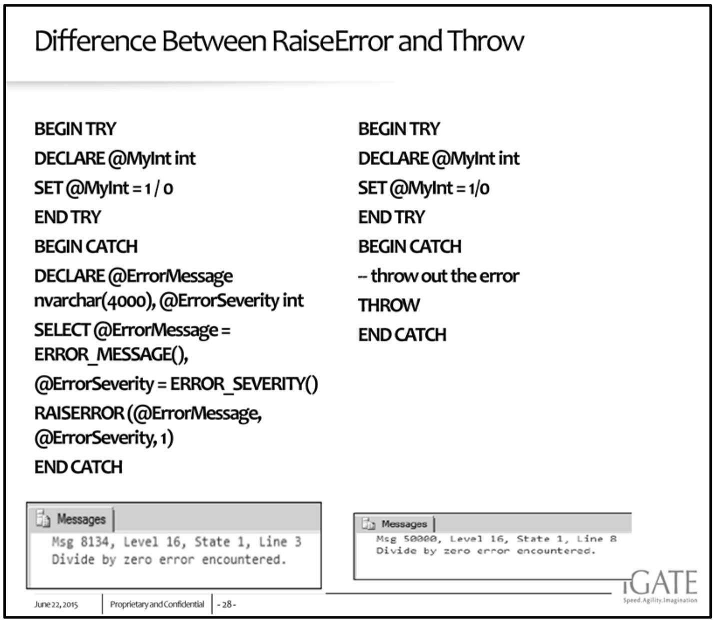
Drawbacks of RAISERROR:
It requires a proper message number to be shown when raising any error.
The message number should exist in sys.messages.
RAISERROR cannot be used to re-throw an exception raised in a TRY..CATCH block.
RAISERROR has been considered as deprecated features

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
```



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
```

```
Msg 8134, Level 16, State 1, Line 3
Divide by zero error encountered.
```

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

```
Msg 50000, Level 16, State 1, Line 8
Divide by zero error encountered.
```

June 22, 2015 | Proprietary and Confidential | - 28 -

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

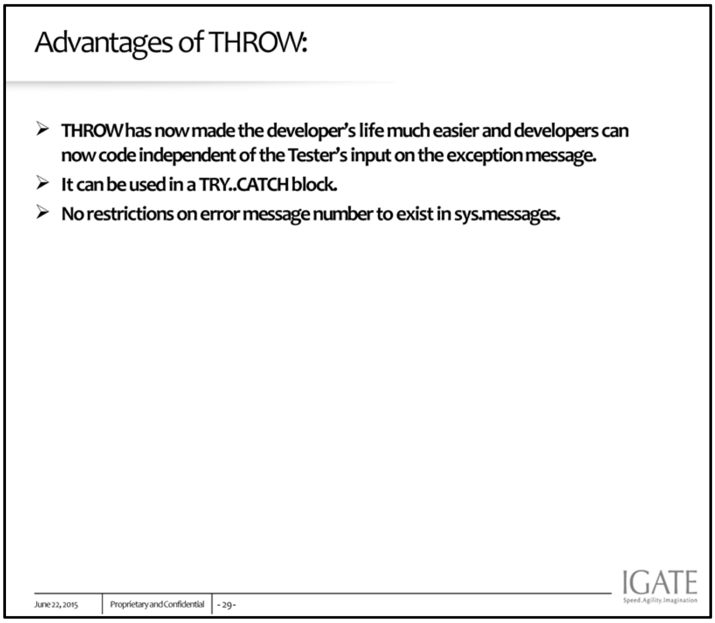
RAISERROR statement THROW statement
If a msg_id is passed to RAISERROR, the ID must be defined in sys.messages.
The error_number parameter does not have to be defined in sys.messages.

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
```

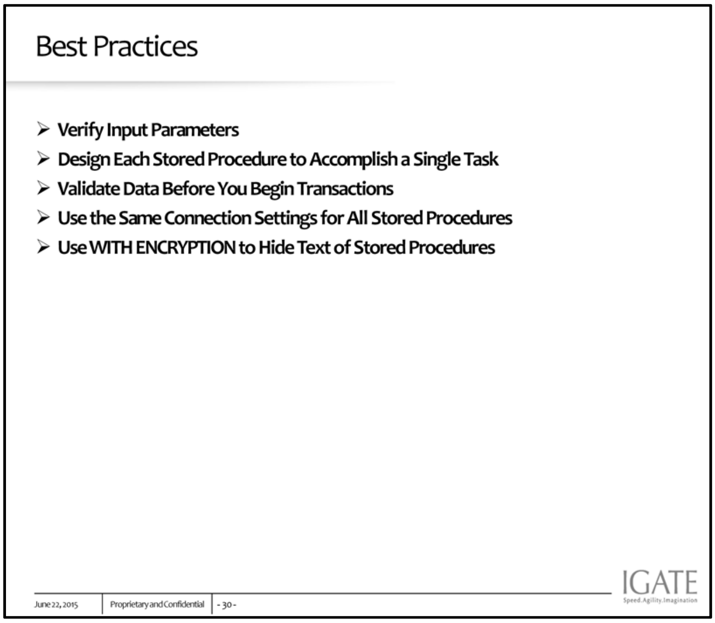
```
USE tempdb;
GO
CREATE TABLE dbo.TestRethrow (  ID INT PRIMARY KEY );
BEGIN TRY
INSERT dbo.TestRethrow(ID) VALUES(1);
-- Force error 2627, Violation of PRIMARY KEY constraint to be raised.
INSERT dbo.TestRethrow(ID) VALUES(1);
END TRY
BEGIN CATCH
PRINT 'In catch block.';
THROW;
END CATCH;
```



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.

June 22, 2015 | Proprietary and Confidential | - 29 -



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

June 22, 2015 | Proprietary and Confidential | - 30 -

IGATE
Speed.Agility.Imagination

Example of WITH ENCRYPTION

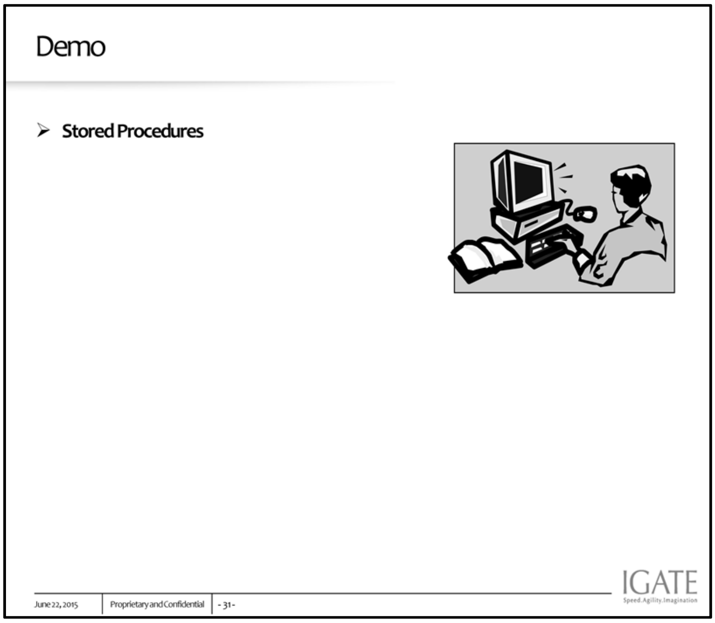
```
Use AdventureWorks2012
CREATE PROCEDURE HumanResources.uspEncrypthis
 WITH ENCRYPTION
AS
SET NOCOUNT ON;
 SELECT BusinessEntityID, JobTitle, NationalIDNumber,
VacationHours, SickLeaveHours FROM
HumanResources.Employee;
GO
```

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.
```

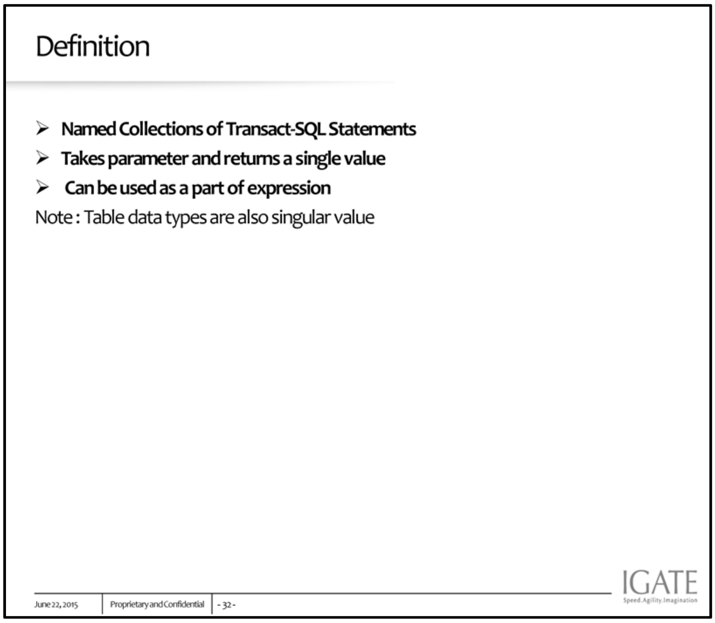


Demo

- Stored Procedures

June 22, 2015 | Proprietary and Confidential | - 31 -

IGATE Speed.Agility.Imagination



Definition

- 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

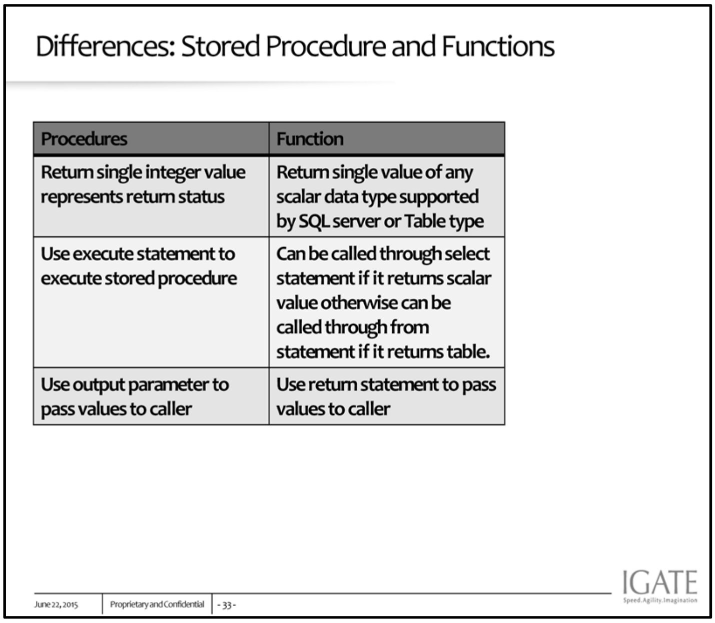
June 22, 2015 | Proprietary and Confidential | - 32 -

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
```

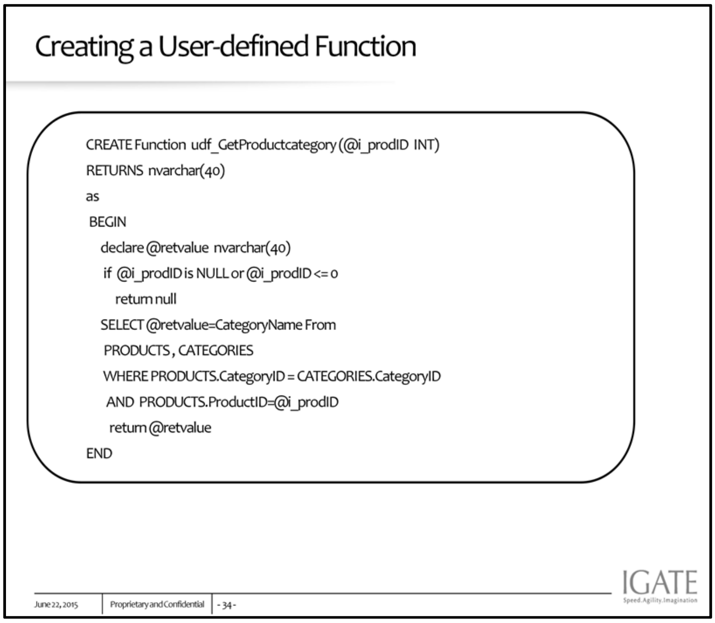
The definition of a table variable looks almost like the definition of a normal table, except that you use the word DECLARE instead of CREATE, and the name of the table variable comes before the word TABLE.
The definition of a table variable can include:
A column list defining the datatypes for each 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



Differences: Stored Procedure and Functions

Procedures	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

June 22, 2015 | Proprietary and Confidential | - 33 -



Creating a 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
```

June 22, 2015 | Proprietary and Confidential | - 34 -

IGATE Speed.Agility.Imagination

```
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.

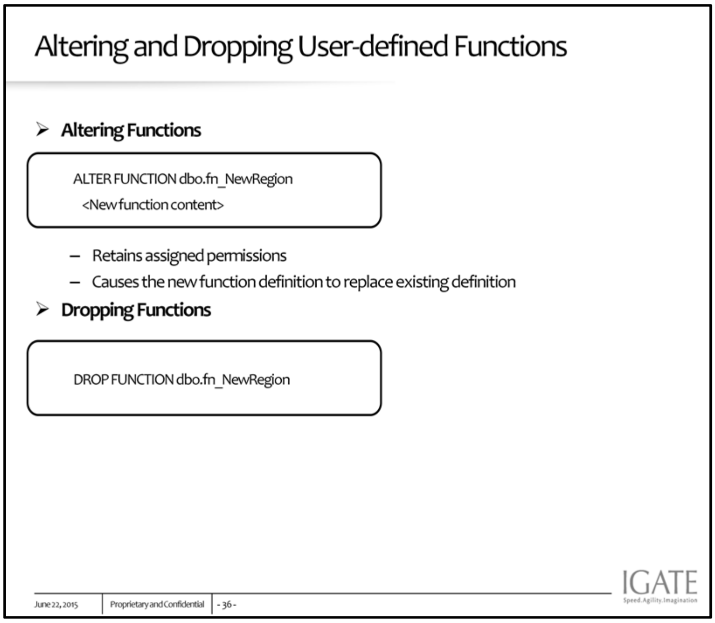
@parameter_name
: Is a parameter in the user-defined function. One or more parameters can be declared.
A function can have a maximum of 1,024 parameters. The value of each declared parameter must be supplied by the user when the function is executed, unless a default for the parameter is defined.
Specify 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.

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.



Altering and Dropping User-defined Functions

- **Altering Functions**

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

  – Retains assigned permissions
  – Causes the new function definition to replace existing definition

- **Dropping Functions**

```
DROP FUNCTION dbo.fn_NewRegion
```

June 22, 2015 | Proprietary and Confidential | - 36 -

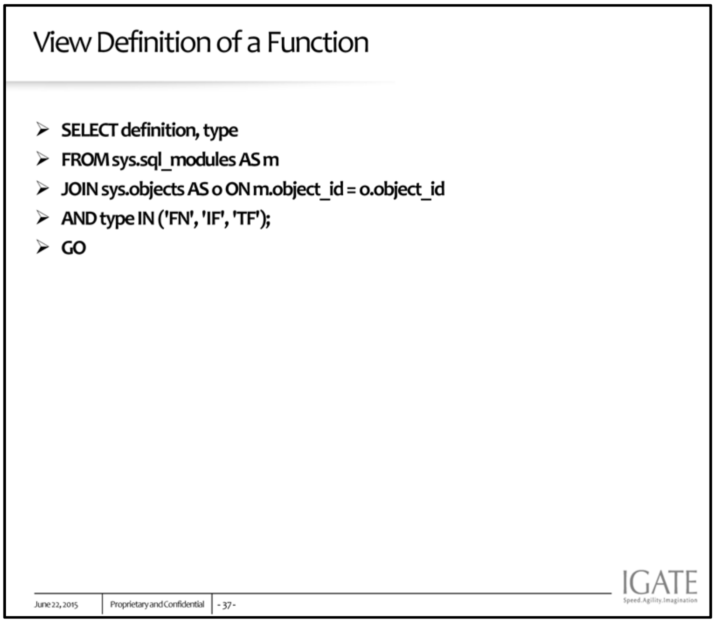
IGATE Speed.Agility.Imagination

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

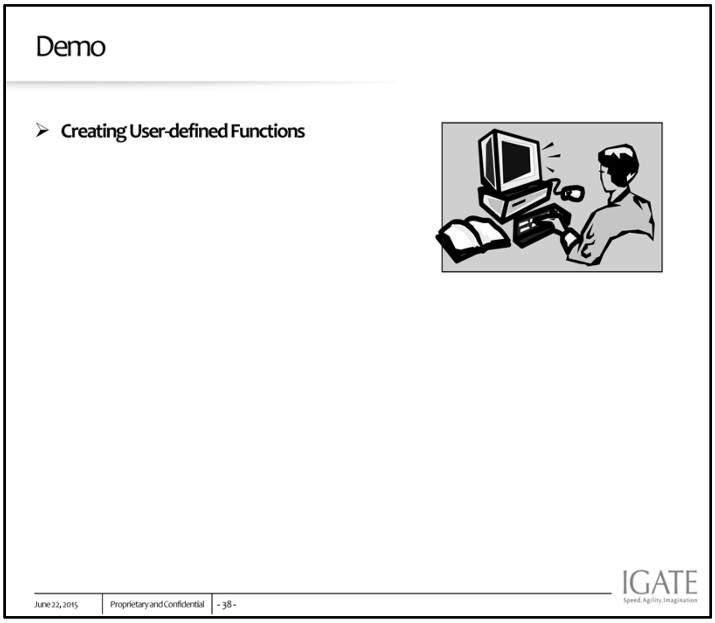
DROP FUNCTION
Removes one or more user-defined functions from the current database. User-defined functions are created by using create function and modified by using alter function.



View Definition of a Function

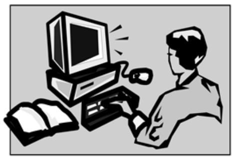
- SELECT definition, type
- FROM sys.sql_modules AS m
- JOIN sys.objects AS o ON m.object_id = o.object_id
- AND type IN ('FN', 'IF', 'TF');
- GO

June 22, 2015 | Proprietary and Confidential | - 37 -

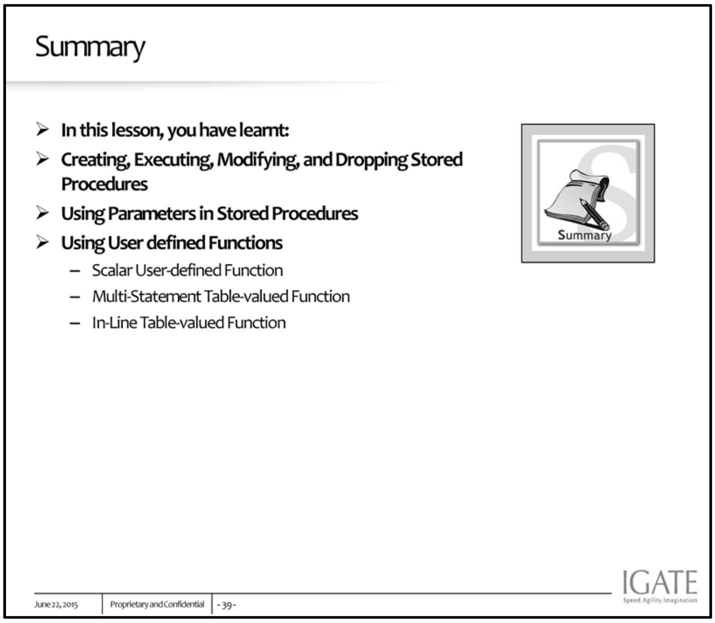


Demo

- Creating User-defined Functions



IGATE
Speed.Agility.Imagination

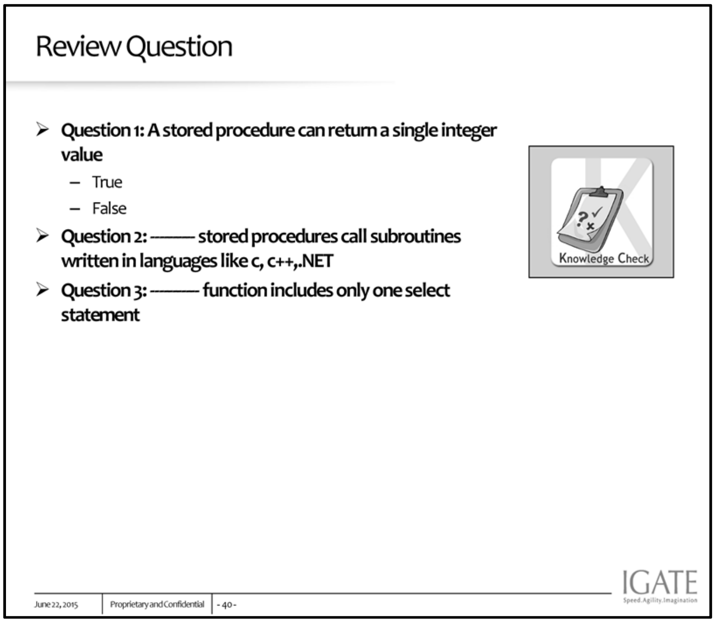


Summary

- 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

June 22, 2015 | Proprietary and Confidential | - 39 -

IGATE Speed.Agility.Imagination



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**

June 22, 2015 | Proprietary and Confidential | - 40 -