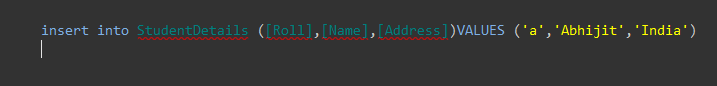
**Error Handling**

* SQL Server also has an exception model to handle exceptions and errors that occurs in T-SQL statements.
* An exception is a problem (usually an error) that prevents the continuation of a program.
* Can consider @@ERROR as one of the basic error handling mechanisms in SQL Server.
* Automatically populates the error message when a certain error occurred in any statement. But we have to trace it within just after the next line where the actual error occurred, otherwise, it will reset to 0.

**@@ERROR Syntax**

@@ERROR -- It returns the error number as int

A table named StudentDetails with columns, Roll (int), Name (varchar) and Address (varchar). Intentionally trying to insert a char in Roll field:



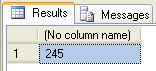
This will throw the following Error:

Msg 245, Level 16, State 1, Line 1

Conversion failed when converting the varchar value 'a' to data type int.



The output is:



So, @@Error returns the same error as return by insert command.

**When We Should Use @@Error**

There are some scenarios where we should use @@ERROR:

* With Insert, Delete, Update, Select Into Statement
* While using Cursor in SQL Server (Open, Fetch Cursor)
* While executing any Stored Procedure

**TRY…CATCH**

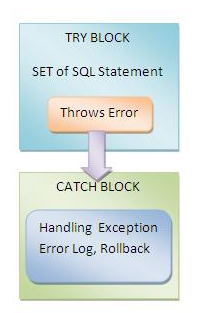
* To handle exception in SQL Server we have TRY...CATCH blocks.
* Put T-SQL statements in TRY block and to handle exception we write code in CATCH block. T
* The role of the TRY statement is to capture the exception.
* If there is an error in code within TRY block then the control will automatically jump to the corresponding CATCH blocks.
* In SQL Server, against a Try block we can have only one CATCH block.
* The **TRY…CATCH** block makes it easy to return or audit error-related data
* **CATCH** includes a **RAISERROR** statement in order to re-throw error-related data to the calling application.
* However, with the release of SQL Server 2012, a replacement for **RAISERROR,** the **THROW** statement, makes it easier than ever to capture the error-related data.

Exception handling with the TRY and CATCH blocks gives a programmer a lot of benefits, such as:

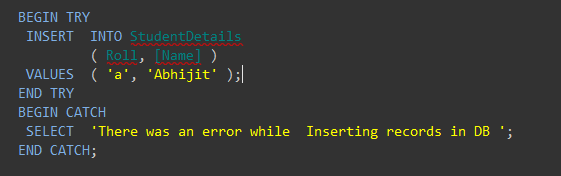
* Provide a clean way to check for errors without cluttering code
* Provide a mechanism to signal errors directly rather than using some side effects
* Can be seen by the programmer and checked during the compilation process

**TRY…CATCH Syntax**

1. **BEGIN** TRY
2. // SQL Statements
3. **END** TRY
4. **BEGIN** CATCH
5. //Handle the exception details
6. //Control **is** pulled **to** CATCH block **only** if there are any exceptions in the TRY
7. block
8. **END** CATCH



As I have already discussed about the studentDetails table, I am now going to insert one record in the table with Roll='a'.



As Roll is an int type but I am trying to insert a char type data which will violate the type conversion rule, an error will be thrown. So the execution pointer will jump to Catch block. And below is the output:

There was an error while inserting records in DB

To get the details of the error SQL Server provides the following System function that we can use inside our Catch-block for retrieving the details of the error.

| **Function Name** | **Description** |
| --- | --- |
| 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 |

**ERROR NUMBER**

* Any error number <= 50000 is a System Defined Message and the ones that are > 50000 are User Defined Messages.
* SYS.Messages catalog view can be used to retrieve both System and User Defined Messages.
* Can add a user defined message using sp\_addmessage
* Can remove it using the system stored procedure sp\_dropmessage.

**ERROR SEVERITY**  
Error Severity can be between 0-25.

* 0-10: Informational or a warning.
* 11-16: Programming Errors.
* 17-25: Resource / Hardware / OS/ SQL Server Internal Errors.
* 20-25: Terminates the Connection.
* 19-25: Only User with SysAdmin rights can raise errors with this severity.

**ERROR STATE**

* The same error can be raised for several different conditions in the code.
* Each specific condition that raises the error assigns a unique state code.
* Also the SQL Support team uses it to find the location in the source code where that error is being raised.

**ERROR PROCEDURE**

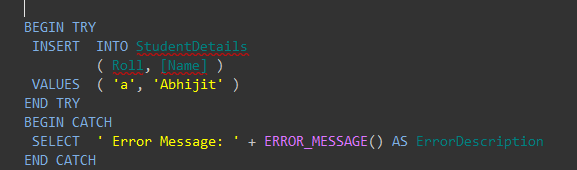
* The name of the Stored Procedure or the function in which the error occurred.
* It will be blank if it is a normal batch of statements.

**ERROR LINE**

* Line number of the statement within a SP, UDF or Batch that triggered the error.
* Will be 0 if a SP or UDF invoke causes the error.

**ERROR MESSAGE**

* Error description detailing the reason for the error.

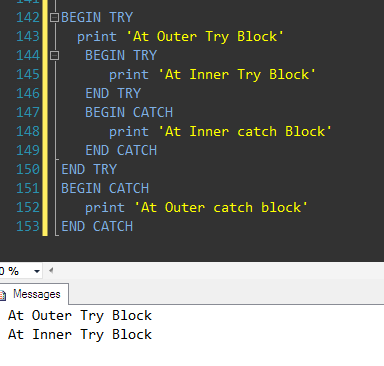


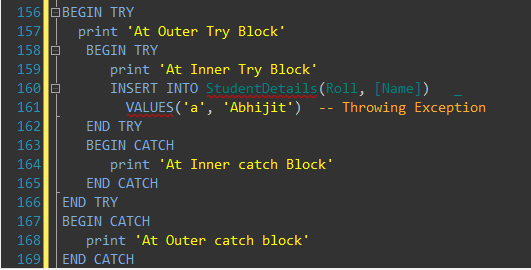
Executed the same code block here but rather than showing custom message, I am showing the internal Error message by Calling ERROR\_MESSAGE () System function. Below is the output:

SystemFunction.JPG

### 

### Nested TRY-CATCH Block



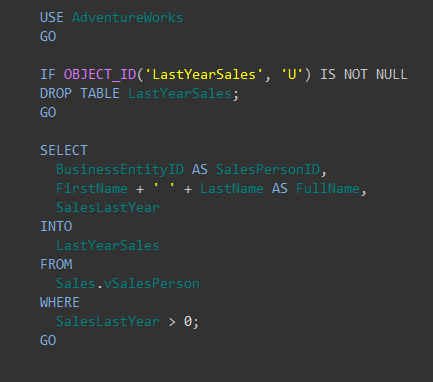


**At Outer Try Block**

**At Inner Try Block**

**At Inner Catch Block**

The examples are based on a table I created in the **AdventureWorks2012** sample database, on a local instance of SQL Server 2012. This is the T-SQL script used to create the **LastYearSales** table.

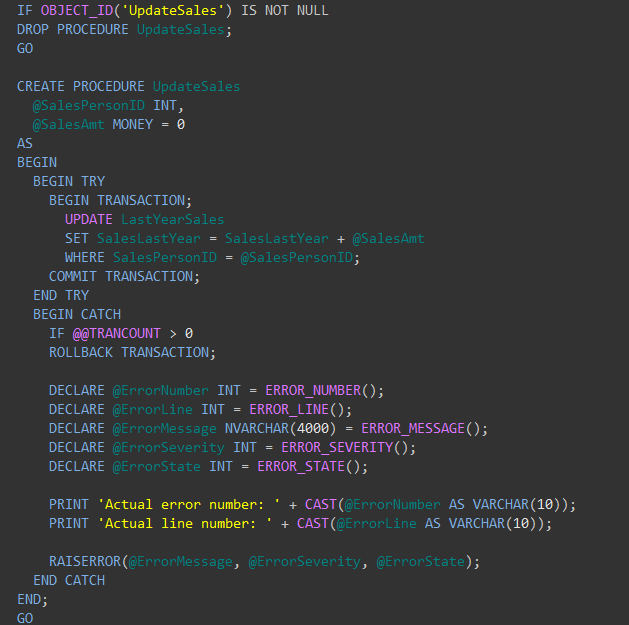


|  |  |
| --- | --- |
|  |  |

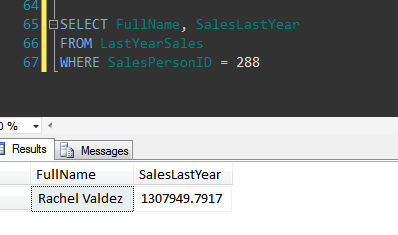
* Used a **SELECT…INTO** statement to retrieve data from the **Sales.vSalesPerson** view and insert it into the newly created table.
* To demonstrate how to handle errors, we need to add one more element to our table: a check constraint that ensures the **SalesLastYear** value is never less than zero.

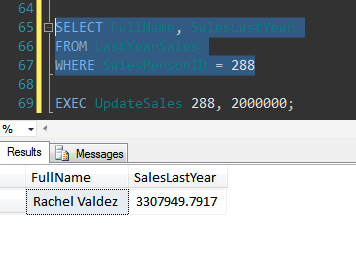
|  |
| --- |
|  |

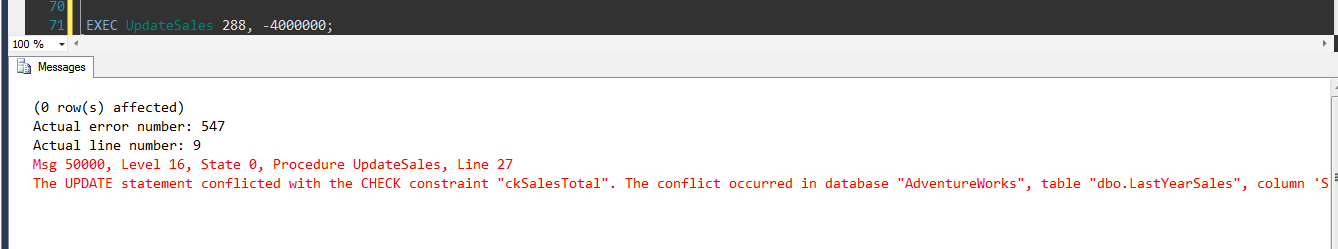
* The constraint makes it easy to generate an error when updating the table.
* Try to add a negative amount to the **SalesLastYear** column, an amount large enough to cause SQL Server to throw an error.



* The procedure, **UpdateSales**, modifies the value in the **SalesLastYear** column in the **LastYearSales** table for a specified salesperson.
* Works by adding or subtracting an amount from the current value in that column. Listing 3 shows the script I used to create the procedure.
* Included two input parameters-**@SalesPersonID** and **@SalesAmt**-which coincide with the table’s **SalesPersonID** and **SalesLastYear** columns.
* The main body of the procedure definition, enclosed in the **BEGIN…END** block, contains the **TRY…CATCH** block, which itself is divided into the **TRY** block and the **CATCH** block.
* The **TRY** block starts with **BEGINTRY** and ends with **ENDTRY** and encloses the T-SQL necessary to carry out the procedure’s actions.
* Included an **UPDATE** statement that adds the **@SalesAmount** value to the SalesLastYear column.
* Statement enclosed in **BEGINTRANSACTION** and **COMMITTRANSACTION** statements to explicitly start and commit the transaction.
* If the **UPDATE** statement runs successfully, the **SalesLastYear** value is updated and the operation is completed, in which case, the code in the **CATCH** block is never executed.
* If the **UPDATE** statement fails and SQL Server generates an error, the transaction is terminated and the database engine jumps to the **CATCH** block.
* The **CATCH** block starts with **BEGINCATCH** and ends with **ENDCATCH** and encloses the statements necessary to handle the error.
* Use the **@@TRANCOUNT** function to determine whether any transactions are still open. **@@TRANCOUNT** is a built-in SQL Server function that returns the number of running transactions in the current session.
* In this case, there should be only one (if an error occurs), so roll back that transaction.
* Declare a set of variables based on system functions that SQL Server makes available within the scope of the **CATCH** block.
* The functions return error-related information that you can reference in your T-SQL statements.
* Include two **PRINT** statements that display the values of **@ErrorNumber and @ErrorLine variables.**
* The **RAISERROR** statement comes after the **PRINT** statements. The statement returns error information to the calling application.
* Generally, when using **RAISERROR**, include an error message, error severity level, and error state.
* **RAISERROR** will show the line number where the RAISERROR statement was executed, but not the actual exception position



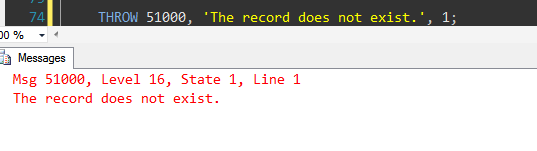




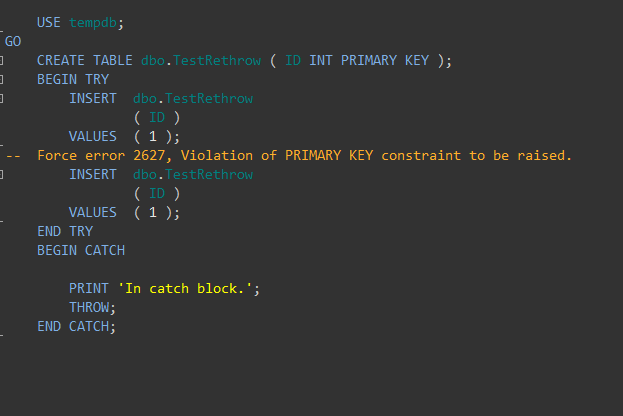
The actual error number (**547**) is different from the **RAISERROR** message number (**50000**) and that the actual line number (**9**) is different from the **RAISERROR** line number (**27**).

* To simplify returning errors in a **CATCH** block, SQL Server 2012 introduced the **THROW** statement.
* With the **THROW** statement, you don’t have to specify any parameters and the results are more accurate.

The following example shows how to use the THROW statement to raise an exception.



The following example shows how use the THROW statement to raise the last thrown exception again.



**(1 row(s) affected)**

**(1 row(s) affected)**

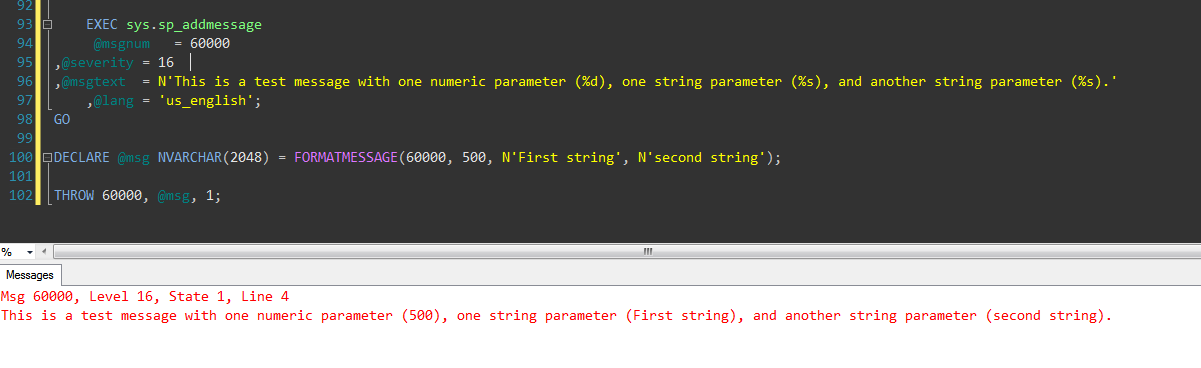
**(0 row(s) affected)**

**In catch block.**

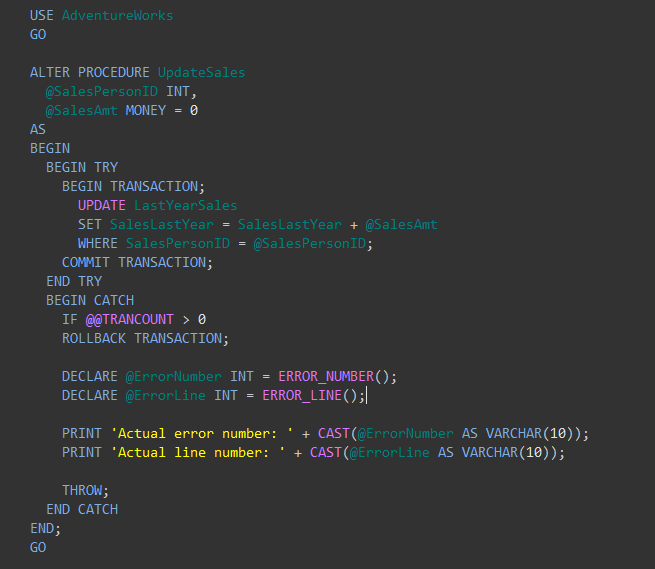
**Msg 2627, Level 14, State 1, Line 7**

**Violation of PRIMARY KEY constraint 'PK\_\_TestReth\_\_3214EC27F7476E1C'. Cannot insert duplicate key in object 'dbo.TestRethrow'. The duplicate key value is (1).**

The following example shows how to use the FORMATMESSAGE function with THROW to throw a customized error message. The example first creates a user-defined error message by using sp\_addmessage. Because the THROW statement does not allow for substitution parameters in the message parameter in the way that RAISERROR does, the FORMATMESSAGE function is used to pass the three parameter values expected by error message 60000.



To demonstrate the **THROW** statement, **ALTER PROCEDURE** statement that modifies the **UpdateSales** procedure, specifically the **CATCH** block.



* Retained the **@ErrorNumber** and **@ErrorLine** variable declarations and their related **PRINT**statements to demonstrate the **THROW** statement’s accuracy.
* In actually, need only to roll back the transaction and specify the **THROW** statement, without any parameters.



**(0 row(s) affected)**

**Actual error number: 547**

**Actual line number: 9**

**Msg 547, Level 16, State 0, Procedure UpdateSales, Line 9**

**The UPDATE statement conflicted with the CHECK constraint "ckSalesTotal". The conflict occurred in database "AdventureWorks", table "dbo.LastYearSales", column 'SalesLastYear'.**

## Raiserror Vs Throw

Below table lists-out 10 major differences between RAISERROR and THROW with examples:

|  |  |
| --- | --- |
| **RAISERROR** | **THROW** |
| **Version of the Sql Server in which it is introduced?** | |
| Introduced in SQL SERVER 7.0. And as per BOL, [Microsoft](http://msdn.microsoft.com/en-us/library/ms178592%28SQL.110%29.aspx) is suggesting to start using THROW statement instead of RAISERROR in New Applications.  RAISERROR can’t be used in the Sql Server 2014’s Natively compiled Stored Procedures. | Introduced in SQL SERVER 2012. THROW statement seems to be simple and easy to use than RAISERROR.  THROW statement can be used in the Sql Server 2014’s Natively Compiled Stored Procedure. |
| **SYNTAX** | |
| |  | | --- | | RAISERROR   ( { error\_number | message          | @local\_variable }      { ,severity ,state }      [ ,argument [ ,...n ] ] )      [ WITH option [ ,...n ] ] | | |  | | --- | | THROW   [ { error\_number       | @local\_variable },     { message | @local\_variable },     { state | @local\_variable } ]  [ ; ] | |
| **Can re-throw the original exception that invoked the CATCH block?** | |
| **NO.** It always generates new exception and results in the loss of the original exception details. Below example demonstrates this:   |  | | --- | | BEGIN TRY    DECLARE @result INT  --Generate divide-by-zero error    SET @result = 55/0  END TRY  BEGIN CATCH  --Get the details of the error  --that invoked the CATCH block   DECLARE     @ErMessage NVARCHAR(2048),     @ErSeverity INT,     @ErState INT     SELECT     @ErMessage = ERROR\_MESSAGE(),     @ErSeverity = ERROR\_SEVERITY(),     @ErState = ERROR\_STATE()     RAISERROR (@ErMessage,               @ErSeverity,               @ErState )  END CATCH |   **RESULT:** Msg **50000**, Level 16, State 1, **Line 19** Divide by zero error encountered.  **NOTE:*The actual line number of the code which generated Divided By Zero error here is 4, but the exception message returned by RAISERROR is showiung it as 19. Also the error number corresponding to divide by zero error is 8134 in the SYS.Messages table, but the one returned by RAISERROR is 50000.*** | **YES.** To Re-THROW the original exception caught in the TRY Block, we can just specify the THROW statement without any parameters in the CATCH block. Below example demonstrates this:   |  | | --- | | BEGIN TRY    DECLARE @result INT  --Generate divide-by-zero error    SET @result = 55/0  END TRY  BEGIN CATCH      THROW  END CATCH |   **RESULT:** Msg 8134, Level 16, State 1, Line 4 Divide by zero error encountered.  With above example it is clear that THROW statement is very simple for RE-THROWING the exception. And also it returns correct error number and line number. |
| **Causes the statement batch to be ended?** | |
| **Example 1:** In the below Batch of statements the PRINT statement after RAISERROR statement will be executed.   |  | | --- | | BEGIN   PRINT 'BEFORE RAISERROR'   RAISERROR('RAISERROR TEST',16,1)   PRINT 'AFTER RAISERROR'  END |   **RESULT:**  BEFORE RAISERROR Msg 50000, Level 16, State 1, Line 3 RAISERROR TEST AFTER RAISERROR  **Example 2:** In the below example all the statement’s after RAISERROR statement are executed.   |  | | --- | | BEGIN TRY   DECLARE @RESULT INT = 55/0  END TRY  BEGIN CATCH   PRINT 'BEFORE RAISERROR';    --Get the details of the error  --that invoked the CATCH block   DECLARE    @ErMessage NVARCHAR(2048),    @ErSeverity INT,    @ErState INT     SELECT    @ErMessage = ERROR\_MESSAGE(),    @ErSeverity = ERROR\_SEVERITY(),    @ErState = ERROR\_STATE()     RAISERROR (@ErMessage,               @ErSeverity,               @ErState )     PRINT 'AFTER RAISERROR'  END CATCH   PRINT 'AFTER CATCH' |   **RESULT:** BEFORE RAISERROR Msg 50000, Level 16, State 1, Line 19 Divide by zero error encountered. AFTER RAISERROR AFTER CATCH | **Example 1:** In the below Batch of statements the PRINT statement after THROW statement will not executed.   |  | | --- | | BEGIN      PRINT 'BEFORE THROW';      THROW 50000,'THROW TEST',1      PRINT 'AFTER THROW'  END |   **RESULT:**  BEFORE THROW Msg 50000, Level 16, State 1, Line 3 THROW TEST  **Example 2:** In the below example no PRINT statement’s after THROW statement are executed.   |  | | --- | | BEGIN TRY    DECLARE @RESULT INT = 55/0  END TRY  BEGIN CATCH    PRINT 'BEFORE THROW';    THROW;    PRINT 'AFTER THROW'  END CATCH    PRINT 'AFTER CATCH' |   **RESULT:** BEFORE THROW Msg 8134, Level 16, State 1, Line 2 Divide by zero error encountered. |
| **CAN SET SEVERITY LEVEL?** | |
| **YES.** The severity parameter specifies the severity of the exception. | **NO.** There is no severity parameter. The exception severity is always set to 16. (unless re-throwing in a CATCH block) |
| **Requires preceding statement to end with semicolon (;) statement terminator?** | |
| **NO.** | **YES.** The statement before the THROW statement must be followed by the semicolon (;) statement terminator. |
| **CAN RAISE SYSTEM ERROR MESSAGE?** The SYS.MESSAGES Table will have both system-defined and user-defined messages. Message IDs less than 50000 are system messages. | |
| **YES.** With RAISERROR we can raise the System Exception. Example: RAISERROR (40655,16,1)**RESULT:** Msg 40655, Level 16, State 1, Line 1 Database ‘master’ cannot be restored. | **NO.**With THROW we can’t raise the System Exception. But when it used in CATCH BLOCK it can Re-THROW the system exception.Example: Trying to raise system exception (i.e. exception with ErrorNumber less than 50000).  THROW 40655, ‘Database master cannot be restored.’, 1  **RESULT:** Msg 35100, Level 16, State 10, Line 1 Error number 40655 in the THROW statement is outside the valid range. Specify an error number in the valid range of 50000 to 2147483647 |
| **CAN RAISE user-defined message with message\_id greater than 50000 which is not defined in SYS.MESSAGES table?** | |
| **NO.**If a msg\_id is passed to RAISERROR, the ID must be defined in sys.messages.Example:  RAISERROR (60000, 16, 1)  **RESULT:** Msg 18054, Level 16, State 1, Line 1 Error 60000, severity 16, state 1 was raised, but no message with that error number was found in sys.messages. If error is larger than 50000, make sure the user-defined message is added using sp\_addmessage.  Now add the Message to SYS.MESSAGES Table by using the below statement:  EXEC sys.sp\_addmessage 60000, 16, ‘Test User Defined Message’  Now try to Raise the Error: RAISERROR (60000, 16, 1)  **RESULT:** Msg 60000, Level 16, State 1, Line 1 Test User Defined Message | **YES.**The error\_number parameter does not have to be defined in sys.messages.Example: THROW 60000, ‘Test User Defined Message’, 1**RESULT:** Msg 60000, Level 16, State 1, Line 1 Test User Defined Message |
| **Allows substitution parameters in the message parameter?** By using the below statement add a sample test message with parameteres to the SYS.Messages Table: **EXEC sp\_addmessage 70000,16,‘Message with Parameter 1: %d and Parameter 2:%s’** | |
| **YES.**The msg\_str parameter can contain **printf**formatting styles.Example 1:  RAISERROR (70000, 16, 1, 505,‘Basavaraj’ )  RESULT: Msg 70000, Level 16, State 1, Line 1 Message with Parameter 1: 505 and Parameter 2:Basavaraj | **NO.**The message parameter does not accept **printf** style formatting.Example 1:  THROW 70000, ‘Message with Parameter 1: %d and Parameter 2:%s’, 1, 505,’Basavaraj’  RESULT: Msg 102, Level 15, State 1, Line 1 Incorrect syntax near ‘,’.  **Alternative Way of doing this is:**  DECLARE @ErrorMsg NVARCHAR(2048) = FORMATMESSAGE(70000, 505, ‘Basavaraj’ ); THROW 70000, @ErrorMsg, 1  Example 2: Message manipulation is not allowed in the THROW statement  Below statement will fail  THROW 58000,‘String1’ + ‘ String2’,1  RESULT: Msg 102, Level 15, State 1, Line 1 Incorrect syntax near ‘+’.  We can solve such problems, we can prepare the message prior to the THROW statement and then pass it to throw statement as a variable. Below example illustrates this.  DECLARE @message NVARCHAR(2048) SET @message = ‘String1’ + ‘ String2’**;** THROW 58000, @message, 1  RESULT: Msg 58000, Level 16, State 1, Line 3 String1 String2 |