Stored Procedure in SQL Server

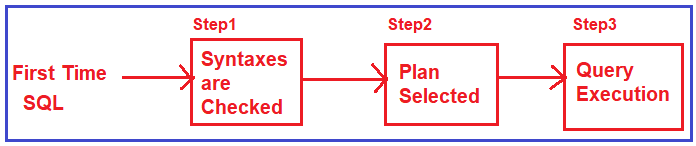
**Stored Procedure in SQL Server with Examples**

discuss **Stored Procedures in SQL Server** with Examples. As a SQL Server developer, it is the most important concept for you to understand. As part of this article, we are going to discuss the following pointers related to the Stored Procedure.

1. **Why do we need a Procedure in SQL Server?**
2. **What is a Stored Procedure in SQL Server?**
3. **How can we create a Stored Procedure?**
4. **How to call a Stored Procedure in SQL Server?**
5. **Multiple Examples to understand SQL Server Stored Procedure.**
6. **How to view the text of a Procedure in SQL Server?**
7. **How to Drop and Alter a Procedure in SQL Server?**
8. **Different Types of Parameters in SQL Server Stored Procedure**
9. **Understanding the Input Parameters in SQL Server Procedure**
10. **Understanding the** **SQL Server Stored Procedure Output Parameters**
11. **What are the advantages of using a Stored Procedure?**
12. **What is an execution plan?**

**Why do we need a Stored Procedure in SQL Server?**

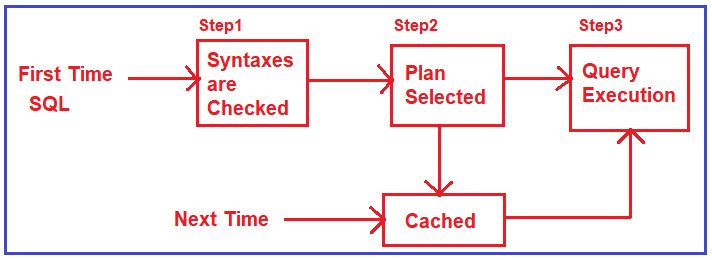
Before going to understand why we need a Stored Procedure, let us first understand what happens when we execute a simple SQL statement in SQL Server. When any SQL Statements are fired on SQL Server, then three steps are happening in order which is shown in the below image.



1. **Syntax Checked:** This step ensures that the syntaxes are correct and there is no error and the query is ready for executing on SQL Server.
2. **Plan Selected:** Once the syntaxes are checked, the second step is to select a plan. The SQL Query must be using some tables. It will go and check what types of indexes are exists on these tables, it will also check can use these indexes or a table scan is fine. So, the second step is to select a proper execution plan to execute the query.
3. **Query Execution:** Once the plan is selected, the final step is to execute the query and the output is seen by the end-user.

So, any SQL Statement fire on SQL Server should go through these three steps.

Now somehow, if we ensure that the first two steps (i.e. Syntax Checked and Plan Selected) are executed only once, would not it be great. In other words, the first time the SQL is executed, the syntaxes are checked, the execution plan is selected and the execution plan is cached in memory. So, if the same SQL statements are fired again, then these two steps are not going to be executed, rather the execution plan is taken from the cache and executed and that will definitely increase the performance of the application which is shown in the below image.



This is what exactly the stored procedure does in SQL Server. When we create a stored procedure, the syntaxes are checked while creating the procedure or we can say at the design. When we execute the procedure for the first time, the best execution plan is selected and is cached in memory. And after that whenever we call the stored procedure, the query execution plan is taken from the cache rather than creating again and again and executed.

There are also other advantages of using stored procedures which we will discuss in our upcoming articles. With this keep in mind, let us proceed and understand the SQL Server Stored Procedure in detail.

**What is a Stored Procedure in SQL Server?**

A SQL Server Stored Procedure is a database object which contains pre-compiled queries (a group of T-SQL Statements). In other words, we can say that the Stored Procedures are a block of code designed to perform a task whenever we called.

There are 2 types of stored procedures

1 system stored Proceduers

2.UserDefined stored procedure

These are some of the commonly used system procedures.

|  |  |
| --- | --- |
| PROCEDURE | DESCRIPTION |
| sp\_monitor | Provides runtime statistics about the SQL instance. |
| sp\_help | Provides information about database objects |
| sp\_helptext | Returns the definition (DDL) of a database object |
| sp\_who2 | Provides information about current users, sessions, and processes. |
| sp\_kill | Kills a session. Useful in deadlock situations. |
| sp\_tables | Returns information about tables and views |
| sp\_depends | Returns the dependencies of a database object |
| sp\_executesql | Used in T-SQL to execute dynamic SQL. Be mindful of SQL injection. |
| sp\_getapplock | Identifies transaction and object locks. |

## List All Stored Procedures

## [#](https://www.dofactory.com/sql/stored-procedures#list)

This is one way to list all user-generated stored procedures:

**SELECT \***

**FROM sys.procedures**

And another way.

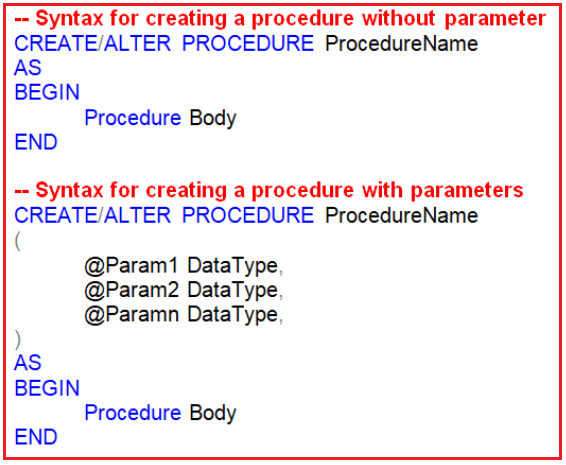
**SELECT \***

**FROM INFORMATION\_SCHEMA.ROUTINES**

**WHERE ROUTINE\_TYPE = 'PROCEDURE'**

**How can we create a Stored Procedure in SQL Server?**

In SQL Server, you can create a stored procedure by using the **CREATE PROCEDURE** or **CREATE PROC** statement. Again, you can create a procedure with or without parameters. Please have a look at the below image for the **Syntax of Stored Procedure**.



Syntax to run a stored procedure.

1. **EXECUTE procedure\_name**

Or, use the shorthand.

1. **EXEC procedure\_name**

Syntax to change a stored procedure.

1. **ALTER PROCEDURE procedure\_name**
2. **AS**
3. **BEGIN**
4. **sql\_statement**
5. **END**

This effectively overwrites the prior version of the stored procedure.

Syntax to remove a stored procedure.

1. **DROP PROCEDURE procedure\_name**

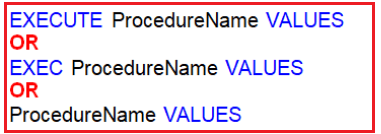
A stored procedure is very much similar to a function in C, C++ languages or a method in Java or C# languages. The procedure definitions contain two parts in it

1. **Procedure header**
2. **Procedure body**

The content above “**AS**” is known as the procedure header and the content below the “**AS”** is known as the procedure body. If required we can pass the parameter to a procedure to make the procedure more dynamic.

**How to call a Stored Procedure in SQL Server?**

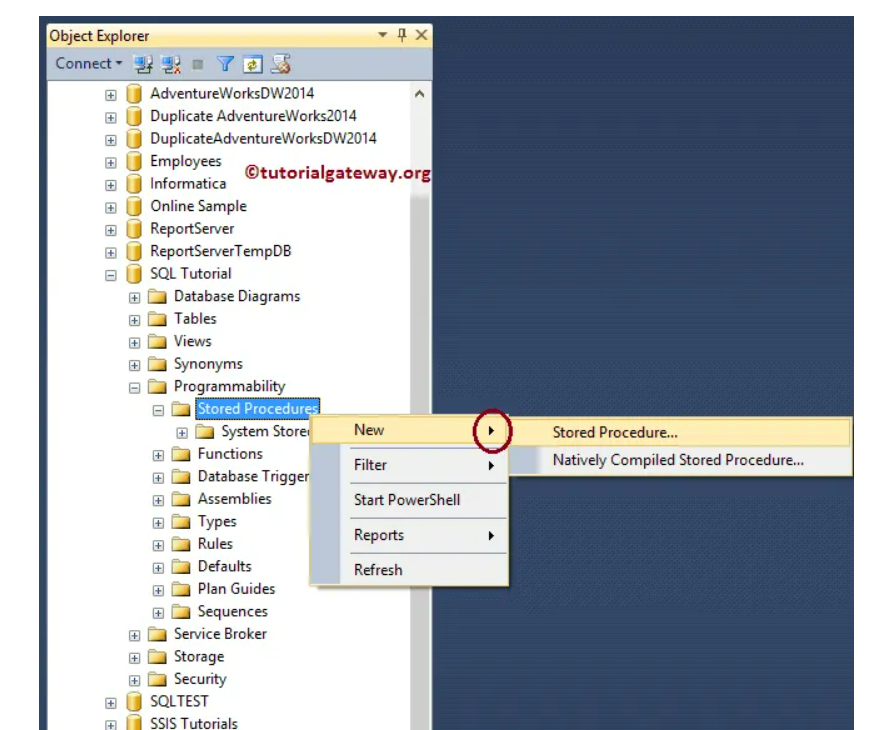
Once we create the stored procedure, then it is physically stored on the server as a “**database object**” which can be called from anywhere connecting to the server. We can call the procedure from anywhere that is from a new query window or from any application that is developed using java or .net language also in three different ways as shown in the below image.



**Note:** Another way to execute a stored procedure is to right-click on the procedure name in object explorer and select “**Execute Stored Procedure**”.

**Stored Procedure in SQL Server Without Parameter**

The following stored procedure simply print a welcome message on the screen



**CREATE** **PROCEDURE** spDisplayWelcome

**AS**

**BEGIN**

**PRINT** 'WELCOME TO PROCEDURE in SQL Server'

**END**

**Calling a Stored Procedure:**

**EXECUTE** spDisplayWelcome

Or

**EXEC** spDisplayWelcome

Or

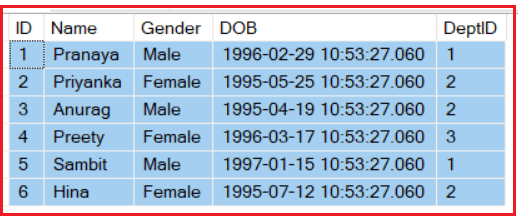
spDisplayWelcome

**Output**: **WELCOME TO PROCEDURE in SQL Server**

**Note:** While naming the user-defined stored procedures we should not have to use **“sp\_”** as a prefix as it is recommended by Microsoft. The reason is all the system-defined procedures in SQL Server are prefixed with **“sp\_”**. So to avoid the ambiguity between the user-defined stored procedure and the system stored procedures and for any conflicts with some future coming system procedure we should not use sp\_ as a prefix to our user-defined stored procedure.

**Let’s see another example where we will fetch the data from a database table.**

We are going to use the following Employee table.



**Please use the following SQL Script to create and populate the Employee table with the required sample data.**

-- Create Employee Table

**CREATE** **TABLE** Employee

(

**ID** **INT** **PRIMARY KEY**,

Name **VARCHAR**(50),

Gender **VARCHAR**(50),

**DOB** **DATETIME**,

DeptID **INT**

)

**GO**

-- Populate the Employee Table with test data

**INSERT** **INTO** Employee **VALUES**(1, 'Pranaya', 'Male','1996-02-29 10:53:27.060', 1)

**INSERT** **INTO** Employee **VALUES**(2, 'Priyanka', 'Female','1995-05-25 10:53:27.060', 2)

**INSERT** **INTO** Employee **VALUES**(3, 'Anurag', 'Male','1995-04-19 10:53:27.060', 2)

**INSERT** **INTO** Employee **VALUES**(4, 'Preety', 'Female','1996-03-17 10:53:27.060', 3)

**INSERT** **INTO** Employee **VALUES**(5, 'Sambit', 'Male','1997-01-15 10:53:27.060', 1)

**INSERT** **INTO** Employee **VALUES**(6, 'Hina', 'Female','1995-07-12 10:53:27.060', 2)

**GO**

**Create a stored procedure to get the names, gender, and the dob of all employees from the table Employee table.**

**CREATE** **PROCEDURE** spGetEmployee

**AS**

**BEGIN**

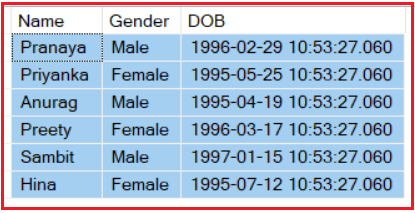
**Select** Name, Gender, **DOB** **from** Employee

**END**

-- To Execute the Procedure

**EXEC** spGetEmployee

**When we execute the above statement it will give us the below output.**



**How to View the text of a Stored Procedure in SQL Server?**

Once you created the stored procedure and later if you want to view the text of the stored procedure then you need to use the **sp\_helptext**system-defined stored procedure by supplying the procedure name as a parameter as shown below

**Example:** **sp\_helptext spGetEmployee**

Else just right click on the stored procedure in object explorer Script procedure as Create To new query editor window

**How to change the name and body of a stored procedure in SQL Server?**

**CREATE** **PROCEDURE** spGetEmployee

**As**

**BEGIN**

**SELECT** Name,Gender, **DOB** **FROM** Employee

**END**

-- How to change the body of a stored procedure

-- User Alter procedure to change the body

**ALTER** **PROCEDURE** spGetEmployee

**AS**

**BEGIN**

**SELECT** Name, Gender, **DOB**

**FROM** Employee

**ORDER BY** Name

**END**

-- To change the procedure name from spGetEmployee to spGetEmployee1

-- Use sp\_rename system defined stored procedure

**EXEC** sp\_rename 'spGetEmployee', 'spGetEmployee1'

**How to Drop a Stored Procedure?**

In order to drop a stored procedure, all you need to use the following syntax  
**DROP PROCEDURE ProcedureName**  
**Example:** **Drop proc spGetEmployee1** or **Drop Procedure spGetEmployee1**

**Different Types of Parameters in SQL Server Stored Procedure.**

The parameters of a Stored Procedure in SQL Server can be of two types

1. **Input parameters**
2. **Output parameters**

The Input Parameters in SQL Server Stored Procedure are used for bringing the values into the procedure for execution. On the other hand, the Output Parameters are used to carrying a value out of the procedure after execution.

When a parameter is declared with the output keyword, then we only require assigning a value to the parameter inside the procedure so that the procedure will send that value out at the end of procedure execution.

**Understanding the Input Parameters in SQL Server Procedure:**

Let us understand the Input Parameters in SQL Server with an example. Let’s create a procedure that will take two input integer parameters and then perform the sum operation and finally print the result.

**Example: Stored Procedure for adding two variables value**

-- Create a Procedure

**ALTER** **PROCEDURE** spAddTwoNumbers(@no1 **INT**, @no2 **INT**)

**AS**

**BEGIN**

**DECLARE** @Result **INT**

**SET** @Result = @no1 + @no2

**PRINT** 'RESULT IS: '+ **CAST**(@Result **AS** **VARCHAR**)

**END**

**GO**

-- Calling the procedure:

**EXECUTE** spAddTwoNumbers 10, 20

-- OR

**EXECUTE** spAddTwoNumbers @no1=10, @no2=20

-- OR calling the procedure by declaring two variables as shown below

**DECLARE** @no1 **INT**, @no2 INt

**SET** @no1 = 10

**SET** @no2 = 20

**EXECUTE** spAddTwoNumbers @no1, @no2

**Note:** The Parameters and variables that we created must have an **@** prefix in their name.

**Example: Create a Procedure to get the employee information bypassing the employee gender and department id from the Employee table**

**CREATE** **PROCEDURE** spGetEmployeesByGenderAndDepartment

@Gender **VARCHAR**(20),

@DeptID **INT**

**AS**

**BEGIN**

**SELECT** Name, Gender, **DOB**, DeptID

**FROM** Employee

**WHERE** Gender = @Gender AND DeptID = @DeptID

**END**

**GO**

In order to invoke the above SQL Server Stored Procedure, we need to pass the value for **@Gender** and **@DeptID** input parameters. If we don’t specify the name of the parameters we have to first pass the value for the **@Gender** parameter and then for the **@DeptID** parameter as shown below.

**EXECUTE spGetEmployeesByGenderAndDepartment ‘Male’, 1**

On the other hand, if we change the order, then we will get an error stating “**Error converting data type varchar to int.**” This is because the value of **“Male”** is passed into the **@DeptID** parameter. Since **@DeptID** is an integer, we get the type conversion error.

**EXEC spGetEmployeesByGenderAndDepartment 1, ‘Male’**

When we specify the names of the parameters when executing the stored procedure the order doesn’t matter. The example is given below.

**EXEC spGetEmployeesByGenderAndDepartment @DeptID=1, @Gender=’Male’**

**Note:**While we are calling the stored procedure passing the values in order is very important. In the order they are declared in the procedure we need to pass the values in the same order. You can also pass the value in any order, but at that time you have to specify the variable name before the value to which you are passing the values.

**Example: Create a procedure to update the Employee details in the Employee table based on the Employee id.**

-- Create a Procedure

**CREATE** **PROCEDURE** spUpdateEmployeeByID

(

@**ID** **INT**,

@Name **VARCHAR**(50),

@Gender **VARCHAR**(50), @**DOB** **DATETIME**,

@DeptID **INT**

)

**AS**

**BEGIN**

**UPDATE** Employee **SET**

Name = @Name,

Gender = @Gender,

**DOB** = @**DOB**,

DeptID = @DeptID

**WHERE** **ID** = @**ID**

**END**

**GO**

-- Executing the Procedure

-- If you are not specifying the Parameter Names then the order is important

**EXECUTE** spUpdateEmployeeByID 3, 'Palak', 'Female', '1994-06-17 10:53:27.060', 3

-- If you are specifying the Parameter Names then order is not mandatory

**EXECUTE** spUpdateEmployeeByID @**ID** =3, @Gender = 'Female', @**DOB** = '1994-06-17 10:53:27.060', @DeptID = 3, @Name = 'Palak'

**SQL Server Stored Procedure Output Parameters:**

The Input parameters of SQL Server Stored Procedure are used for bringing the values into the procedure for execution. On the other hand, the SQL Server Stored Procedure output parameters are used for carrying a value out of the procedure after its execution. We only require assigning a value to the output parameter inside the procedure so that procedure will send that value out at the end of the procedure execution. The Output parameter in SQL Server can be declared either by using the **OUT** or **OUTPUT** keyword

**Example: Stored Procedure with Output Parameter**

**Let’s create a simple stored procedure to understand the SQL Server Stored Procedure Output Parameters**

**CREATE** **PROCEDURE** spGetResult

@No1 **INT**,

@No2 **INT**,

@Result **INT** **OUTPUT**

**AS**

**BEGIN**

**SET** @Result = @No1 + @No2

**END**

The above SQL Server Stored Procedure takes 3 parameters. The **@No1** and **@No2** are input parameters by default whereas the **@Result** is the output parameter. The Parameter which is created using the **OUT** or **OUTPUT** keyword is called the output parameter in SQL Server.

To execute a procedure with output parameter, First, we need to declare a variable, then we need to pass that variable while calling the procedure by specifying the type as output as shown below.

-- To Execute Procedure

DECLARE @Result INT

EXECUTE spGetResult 10, 20, @Result OUT

PRINT @Result

When we execute the above code it will print 30.

**Let’s see more examples for a better understanding of SQL Server stored procedure output parameters.**

Create a stored procedure to get the total number of employees in the Employee table by Gender. As we already discussed to create a Stored Procedure in SQL Server with an output parameter, we need to use the keyword **OUT or OUTPUT**. In the following Stored Procedure, the **@EmployeeCount** is an output parameter as we specified the parameter with the OUTPUT keyword.

CREATE PROCEDURE spGetEmployeeCountByGender

@Gender VARCHAR**(**30**)**,

@EmployeeCount INT OUTPUT

AS

BEGIN

SELECT @EmployeeCount = COUNT**(**ID**)**

FROM Employee

WHER Gender = @Gender

END

Let’s see the different ways to execute the above SQL Server Stored Procedure with the output parameter.

1. **Step1**: First declare a variable of the same data type as that of the output parameter. Here we have declared the **@EmployeeTotal** integer variable.
2. **Step2**: Then we need to pass the **@EmployeeTotal** variable to the stored procedure. We have to specify the variable with the **OUTPUT**keyword. If we don’t specify the **OUTPUT** keyword, the variable will be **NULL**.
3. **Step3.** Execute

**Way1: Allowed**

DECLARE @EmployeeTotal INT

EXECUTE spGetEmployeeCountByGender 'Male', @EmployeeTotal OUTPUT

PRINT @EmployeeTotal

**Note**: If we don’t specify the output keyword when executing the stored procedure then the **@EmployeeTotal** value will be null. For example, see the following

DECLARE @EmployeeTotal INT

EXECUTE spGetEmployeeCountByGender 'Male', @EmployeeTotal

PRINT @EmployeeTotal

**Whether it will print null or not check the following:**

DECLARE @EmployeeTotal INT

EXECUTE spGetEmployeeCountByGender'Male', @EmployeeTotal

IF **(**@EmployeeTotal IS **NULL)**

PRINT '@EmployeeTotal IS **NULL**'

ELSE

PRINT '@EmployeeTotal IS NOT **NULL**'

**Way2: Not Allowed**

DECLARE @EmployeeTotal INT

EXECUTE spGetEmployeeCountByGender @EmployeeTotal OUTPUT, 'Male'

PRINT @EmployeeTotal

**Way3: Allowed**

We can pass the parameters in any order when we use the parameter names. Here, we are first passing the output parameter and then the input **@Gender** parameter.

DECLARE @EmployeeTotal INT

EXECUTE spGetEmployeeCountByGender @EmployeeCount = @EmployeeTotal OUTPUT, @Gender ='Male'

PRINT @EmployeeTotal

**Example-3**

**To search an employee using ouput parameters which takes input as**

**An employeeid**

create procedure SerchEmployee1(@empid int,@empname varchar(50) out,@age int out,

@salary decimal out,

@designation varchar(50) out,@deptid int out)

as

begin

select \* from Employee\_tbl where empid=@empid

end

declare @empname1 varchar(50),@age1 int,@salary1 decimal ,@designation1 varchar(50),@deptid1 int

exec SerchEmployee1 1,@empname1 out,@age1 out,@salary1 out,@designation1 out,@deptid1 out

7)create a procedure to insert records in two tables.

create procedure spinserttwotables @eid int,@ename varchar(50),@salary money,@Address char(40),@Deptno int,@Dname char(30),@Loc char(20)

as

begin

insert into Employee values(@eid,@ename,@salary,@Address,@Deptno)

insert into Dept values(@Deptno,@Dname,@Loc)

end

Output: exec spinsert 7,'mohan',62000,'mumbai',10,'dotnet','hyd'

9) A Procedure which takes the Empno and prints the Net Salary of the Employee

. CREATE PROCEDURE Net\_Sal(@Empno int)

As

Begin

Declare @VSal money, @NSal money, @VPF money, @VPT money EXEC Deductions @Empno, @VPF OUTPUT, @VPT OUTPUT SELECT @Sal=Sal FROM Emp WHERE Empno=@Empno SET @NSal = @VSal - @VPF - @VPT

Print „Net Salary of the Employee is: „ + Cast(@NSal as Varchar) End -Executing the above Procedure: EXEC Net\_Sal 1005

**Stored Procedure with Default Values:**

Let’s see an example of how to use the stored procedure with default values.

**CREATE** **PROCEDURE** spAddNumber(@No1 **INT**= 100, @No2 **INT**)

**AS**

**BEGIN**

**DECLARE** @Result **INT**

**SET** @Result = @No1 + @No2

**PRINT** 'The SUM of the 2 Numbers is: '+ **CAST**(@Result **AS** **VARCHAR**)

**END**

-- Executing the above procedure:

1. **EXEC** spAddNumber 3200, 25

2. **EXEC** spAddNumber @No1=200, @No2=25

3. **EXEC** spAddNumber @No1=**DEFAULT**, @No2=25

4. **EXEC** spAddNumber @No2=25

In the 3rd and 4th cases, it uses the default value of 100 to the variable @No1 which has been given while creating the procedure.

**What are the advantages of using a Stored Procedure in an SQL Server?**

This is one of the most frequently asked interview questions in SQL Server. Let discuss this question in detail.

**Execution Plan Retention which Improves the Application Performance**

As there is no unnecessary compilation of queries this will reduce the burden on the database (when we send a query to a SQL Server three things happen in order, 1st it checks the syntax of that query, 2ndit compiles that query, 3rd it generates an execution plan) as response user will get a quick response. Let’s get into more details.

The Stored Procedures are pre-compiled and their execution plan is cached and used again when the same stored procedure is executed again. Although ad-hoc queries also create and reuse plans, the plan is reused only when the query is the textual match and the datatypes are matching with the previous call. Any changes in the datatype or you have an extra space in the query then, a new plan is created.

**Reduces the Network Traffic**

The Stored Procedure reduces network traffic. When we execute a stored procedure we need to send the procedure name and parameters so only these things are passed on the network but if we are not using the stored procedure then we need to write the ad-hoc queries and we need to execute them which may contain many numbers of lines. So the stored procedure reduces the network traffic as a result performance of the application increase.

**Code Re-usability and Better Maintainability**

Multiple applications can use the same stored procedure. The different applications which want similar kind of data then they can use the same stored procedure. The advantage is that if we want to change the stored procedure then we need to change it in one place that will affect to all the application that uses it whereas if it is inline SQL query and if we have to use it in multiple applications, then we end up with multiple copies of the same inline SQL query, and if the logic has to change, then we have to change the logic at all the places, which makes it harder maintaining inline SQL. So, the stored procedure provides code reusability and maintainability.

**Better Security**

By granting permission to the underlying database the user can do everything. He can view all the records as well as modify the records. But if we want to restrict the user only to view the records then we need to grant only for that stored procedure which will display the records. In that way, we achieve better security with a stored procedure. Using a stored procedure we can also avoid the SQL Injection attack.

**What is an execution plan?**

An execution plan is nothing but for the query to retrieve the data what is the best possible way available. This depends on the indexes that available on the SQL Server to help that query. Based on those it generates the execution plan and then it executes the query.

## SQL Stored Procedures Best Practices

The following is a list of suggestions that might help you to improve the SQL Server stored procedures performance.

* Try to use the Schema Names while you were creating or referencing any database object. It will decrease the database engine processing time.
* Always specify the required column names within the SELECT Statement. And avoid the SELECT \* Statement
* While Creating or altering a table using ([CREATE TABLE](https://www.tutorialgateway.org/sql-create-table/) or [ALTER TABLE](https://www.tutorialgateway.org/sql-alter-table/)), use the DEFAULT keyword to assign the default values to the Columns. It will prevent the NULL values, and assign those default values to the column data.
* When you are creating [Temporary Tables](https://www.tutorialgateway.org/temp-table-in-sql-server/) inside the sp, You have to specify whether the Column accepts NULLS, or NOT explicitly. It can be done by using NULL, or NOT NULL
* Rather than Extracting, or Inserting a large amount of data, try to work with less and essential data. It reduces the query processing load, and increase the query performance.
* Use the SET NOCOUNT ON statement within the SQL server stored procedure. It will turn off the messages that are sent by the SQL Server to the Client. It includes the number of rows updated, deleted, etc.
* Try to replace the UNION Operator, or OR Operator with the UNION ALL Operator, unless you are looking for distinct values.
* If possible, avoid using the SCALAR Functions in the SELECT statement that returns a large amount of data. It is because the scalar function applied on each row (row basic), so it will affect the query performance.
* To handle the errors, SQL allows us to use the [TRY CATCH](https://www.tutorialgateway.org/sql-try-catch/) feature inside the Stored Procedures. So, try to use the TRY CATCH feature
* Always use the BEGIN..COMMIT [TRANSACTION](https://www.tutorialgateway.org/sql-transaction/) within the SP. Remember that the transaction should be as short as possible. Otherwise, there is a danger in either deadlock or longer locking.

### Best Approach to create Stored Procedures

It will be an ideal way to create in real-time

IF OBJECT\_ID ( 'SelectingEmployeeRecords', 'P' ) IS NOT NULL

DROP PROCEDURE SelectingEmployeeRecords;

GO

CREATE PROCEDURE SelectingEmployeeRecords

AS

BEGIN

SET NOCOUNT ON;

SELECT [FirstName]

,[LastName]

,[Education]

,[Occupation]

,[YearlyIncome]

,[Sales]

,[HireDate]

FROM [Employee]

END

GO

### Encrypted Stored Procedure

By adding the WITH ENCRYPTION keyword, we can create the encrypted Stored Procedure. For example, if we add the below line in the above query

## ****SQL Server Stored Procedure Return Value With Examples****

In this article, I am going to discuss the **SQL Server Stored Procedure Return Value** with examples. Please read our previous article before proceeding to this article where we discussed the [**Output Parameters in SQL Server Stored Procedure**](https://dotnettutorials.net/lesson/stored-procedure-output-parameters-sql-server/) with examples. As part of this article, we are going to discuss the following pointers.

1. **What is Return Value in Stored Procedure?**
2. **Multiple Examples to understand the Return Status Variable in SQL Server Stored Procedure.**
3. **Disadvantages of Return Status Value in SQL Server?**
4. **What are the differences between Return Status Value and Output Parameters in SQL Server?**

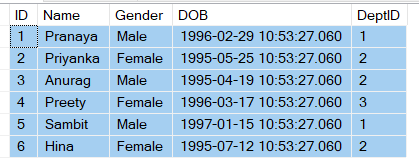
##### ****What is Return Value in SQL Server Stored Procedure?****

Whenever we execute a stored procedure in SQL Server, it always returns an integer status variable indicating the status, usually, zero indicates success, and non-zero indicates the failure. To see this yourself, execute any stored procedure from the object explorer, in SQL server management studio.

1. Right Click and select Execute Stored Procedure
2. If the procedure, expects parameters, provide the values and click OK
3. Along with the result that you expect, the stored procedure also returns a Return Value = 0

So, from this point, we understood that, when a stored procedure is executed, it returns an integer status variable. With this in mind, let’s understand the difference between the output parameters and SQL Server Stored Procedure Return Values.

We are going to use the following Employee table to understand the Stored Procedure Output Parameters and Return values in SQL Server.



**Please use the below SQL Script to create and populate the Employee table with some test data.**

-- Create Employee Table

**CREATE** **TABLE** Employee

(

**ID** **INT** **PRIMARY KEY**,

Name **VARCHAR**(50),

Gender **VARCHAR**(50),

**DOB** **DATETIME**,

DeptID **INT**

)

**GO**

-- Populate the Employee Table with test data

**INSERT** **INTO** Employee **VALUES**(1, 'Pranaya', 'Male','1996-02-29 10:53:27.060', 1)

**INSERT** **INTO** Employee **VALUES**(2, 'Priyanka', 'Female','1995-05-25 10:53:27.060', 2)

**INSERT** **INTO** Employee **VALUES**(3, 'Anurag', 'Male','1995-04-19 10:53:27.060', 2)

**INSERT** **INTO** Employee **VALUES**(4, 'Preety', 'Female','1996-03-17 10:53:27.060', 3)

**INSERT** **INTO** Employee **VALUES**(5, 'Sambit', 'Male','1997-01-15 10:53:27.060', 1)

**INSERT** **INTO** Employee **VALUES**(6, 'Hina', 'Female','1995-07-12 10:53:27.060', 2)

**GO**

###### ****Example1:****

**Create a procedure that will count the total number of employees in the Employee table using the output parameter.**

CREATE PROCEDURE spGetTotalCountOfEmployee1

@TotalCount INT OUTPUT

AS

BEGIN

SELECT @TotalCount =COUNT**(**ID**)**FROM Employee

END

-- For calling the procedure:

DECLARE @EmployeeTotal INT

EXECUTE spGetTotalCountOfEmployee1 @EmployeeTotal OUTPUT

PRINT @EmployeeTotal

When we execute the above SP it returns 6.

**Note**: While calling a procedure with output parameters we need to declare variables first and substitute them in the place of the parameter list so that the results come and sit in those variables but here also we need to use **OUT/OUTPUT** keywords.

###### ****Example2:****

**Create a procedure that will count the total number of employees in the Employee table using return status.**

CREATE PROCEDURE spGetTotalCountOfEmployee2

AS

BEGIN

RETURN **(**SELECT COUNT**(**ID**)** FROM Employee**)**

END

-- For calling the procedure:

DECLARE @EmployeeTotal INT

EXECUTE @EmployeeTotal = spGetTotalCountOfEmployee2

PRINT @EmployeeTotal

When we execute the above SP, it also returns 6.

So we are able to achieve what we want using both output parameters as well as return values. Now let’s look at an example where the return status variable cannot be used but the output parameter can be used.

##### ****Example3:****

**Create a procedure that will take the id value of an employee and return the name of that employee using the output parameter.**

CREATE PROCEDURE spGetEmplloyeeNameById1

@ID INT,

@Name VARCHAR**(**30**)** OUTPUT

AS

BEGIN

SELECT @Name = Name FROM Employee WHERE ID = @ID

END

GO

-- For calling the procedure:

DECLARE @EmployeeName VARCHAR**(**30**)**

EXECUTE spGetEmplloyeeNameById1 3, @EmployeeName OUTPUT

PRINT @EmployeeName

When we execute the above query, it will print the name of the employee as expected.

###### ****Now let’s achieve the same thing using return status value.****

CREATE PROCEDURE spGetEmplloyeeNameById2

@ID INT

AS

BEGIN

RETURN **(**SELECT Name FROM Employee WHERE ID = @ID**)**

END

GO

-- For calling the procedure:

DECLARE @EmployeeName VARCHAR**(**30**)**

EXECUTE @EmployeeName = spGetEmplloyeeNameById2 3

PRINT @EmployeeName

When we execute the spGetEmplloyeeNameById2 Stored Procedure it returns an error stating ‘**Conversion failed when converting the nvarchar value Anurag to data type int.**‘. The return status variable is an integer, and hence when we select the Name of an employee and try to return that we get a conversion error.

So, in SQL Server by using the return values, we can return only one integer. It is not possible, to return more than one value using return values, whereas in output parameters, we can return any data type and a stored procedure can have more than one output parameter.

In general, Return value is used to indicate the success or failure of the stored procedure, especially when we are dealing with nested stored procedures. A return value of 0, indicates success, and any nonzero value indicates failure.

###### ****What are the Disadvantages of Return Status Value in SQL Server?****

Following are the things that we can’t achieve using the Return Value in SQL Server.

1. We cannot return more than one value.
2. We cannot return values other than an integer.

But these two can possible with output parameters.

###### ****What are the differences between Return Status Value and Output Parameters in SQL Server Stored Procedure?****

|  |  |
| --- | --- |
| **Return Status Variable** | **Output parameters** |
| Only integer data type it can return | Any data type value it can return |
| Only one value | More than one value |
| Use to indicate success or failure | Use to return values like name, age, salary, count, etc. |

# SQL Server Stored Procedure with Encryption and Recompile Attribute

## ****SQL Server Stored Procedure with Encryption and Recompile Attribute****

In this article, I am going to discuss the **SQL Server Stored Procedure with Encryption and Recompile Attribute** with examples. Please read our previous article where we discussed the [**Temporary Stored Procedure in SQL Server**](https://dotnettutorials.net/lesson/temporary-stored-procedure-sql-server/) with examples. As part of this article, we are going to discuss the following important concepts.

1. **Learning Some useful system-defined stored Procedure in SQL Server.**
2. **How to view the text of a stored procedure?**
3. **How to encrypt the text of a stored procedure in SQL Server?**
4. **Why do we need to encrypt the text of a stored procedure?**
5. **Understanding the With Encryption Attribute in SQL Server Stored Procedure.**
6. **Understanding the With Recompile Attribute in SQL Server Stored Procedure.**

##### ****Learning Some useful system-defined stored Procedure in SQL Server.****

Let’s have a look at some of the useful system-defined procedures that are very important while working with SQL Server.

###### ****sp\_help procedure name:****

This **sp\_help** system-defined stored procedure is used to view the information of a stored procedure like parameter names, their data type, etc. The **Sp\_help** stored procedure not only used to give information about a stored procedure but also give information about other database objects like tables, views, triggers, etc. You can also press the **ALT+F1** key to get the information when the name of the object is highlighted.

**Example:**  
**sp\_help spGetEmployeeCountByGender**  
**sp\_help Employee**

###### ****sp\_helptext procedure name:****

This **sp\_helptext** system procedure is used when you want to view the text of a subprogram such as function and procedures. The most important point to keep in mind is that whenever we created a stored procedure or function then the content or text of the function or procedure is going to be saved under the **syscomments** system table

###### ****Example****:

**Select \* from syscomments**

In the above **syscomments** table, there is a column called **“text”** which actually stores the complete create procedure or function statement. The **sp\_helptext** system procedure will retrieve the information from the **text** column of the **syscomments** table and then displays it.

###### ****Sp\_depends procedure name:****

The **sp\_depends**system-defined stored procedure is used when we want to see the dependency object of a stored procedure. This procedure is very useful when we want to check if there are any stored procedures that are referencing a table that we are going to drop. The Sp\_depends system-defined stored procedure can also be used with other database objects like table, view, etc.

**sp\_depends Employee**It gives information about the functions, procedures, etc which depends on this table

**sp\_depends spGetEmployee :**it gives information on which fields of which table it depends

##### ****How to view the text of the stored procedure?****

To view the text of a stored procedure you need to use the system stored procedure sp\_helptext <Procedure Name>.

**Example: sp\_helptext spGetEmployeeByGender**

Alternatively, right-click on the stored procedure name in object explorer then select Script procedure as Create To new query editor window.

##### ****SQL Server Stored Procedure Encryption and Recompile Attribute:****

There are two types of attributes that can be used while creating a stored procedure in SQL Server, they are as follows

1. **With Encryption Attribute**
2. **With Recompile Attribute**

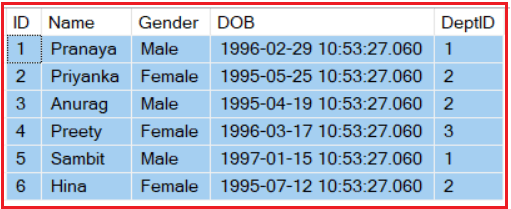
##### ****Understanding the With Encryption Attribute in SQL Server Stored Procedure:****

If you want to encrypt the text of a stored procedure in SQL Server then you need to use the With Encryption Option while creating the Stored Procedure. Once you create the stored procedure using the **“With Encryption”** option then you cannot view the text or content of the stored procedure by using the **sp\_helptext** system-defined stored procedure. If you try to view the text using the **sp\_helptext** system stored procedure, then we will get a message stating ‘**the text for the object is encrypted**’.

That means if this **With Encryption** attribute is used while creating the stored procedure, then the text or content of the stored procedure is encrypted and will not be stored in the **text** column of the **syscomments** system table. As a result, we cannot view the text of the stored procedure.

###### ****Let us understand the use of the SQL Server Stored Procedure Encryption option with one example.****

We are going to use the following Employee table



**Please use the below SQL Script to create and populate the Employee table**

-- Create Employee Table

CREATE TABLE Employee

**(**

ID INT PRIMARY KEY,

Name VARCHAR**(**50**)**,

Gender VARCHAR**(**50**)**,

DOB DATETIME,

DeptID INT

**)**

GO

-- Populate the Employee Table with test data

INSERT INTO Employee VALUES**(**1, 'Pranaya', 'Male','1996-02-29 10:53:27.060', 1**)**

INSERT INTO Employee VALUES**(**2, 'Priyanka', 'Female','1995-05-25 10:53:27.060', 2**)**

INSERT INTO Employee VALUES**(**3, 'Anurag', 'Male','1995-04-19 10:53:27.060', 2**)**

INSERT INTO Employee VALUES**(**4, 'Preety', 'Female','1996-03-17 10:53:27.060', 3**)**

INSERT INTO Employee VALUES**(**5, 'Sambit', 'Male','1997-01-15 10:53:27.060', 1**)**

INSERT INTO Employee VALUES**(**6, 'Hina', 'Female','1995-07-12 10:53:27.060', 2**)**

GO

##### ****Now let’s create one Stored Procedure without using the With Encryption attribute****

-- Stored Procedure without with encryption option

CREATE Procedure sp\_GetEmployeeDetailsById

**(**

@ID INT

**)**

AS

BEGIN

SELECT Name, Gender, CAST**(**DOB AS DATE**)** AS DOB

FROM Employee

WHERE ID = @ID

END

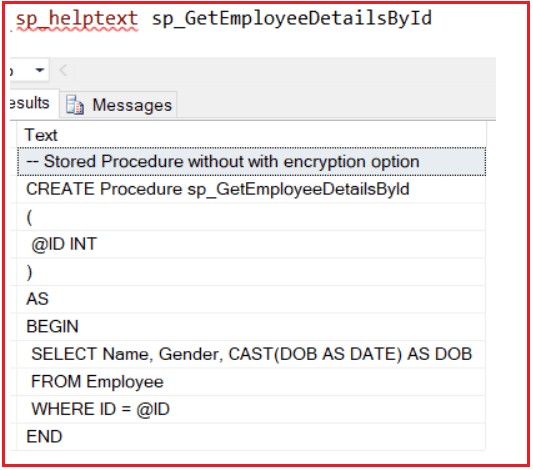
Now let’s view the text using the SYSCOMMENTS table as shown below. Have a look at the text column of this table which actually stores the text of our stored procedure.

**SELECT \* FROM SYSCOMMENTS WHERE ID = OBJECT\_ID(‘sp\_GetEmployeeDetailsById’)**

**Now let’s view the text using the sp\_helptext system stored procedure as shown below.**

**sp\_helptext sp\_GetEmployeeDetailsById**

The **sp\_helptext**system stored procedure actually retrieves the data from the text column of the **syscomments** table. When we execute the above statement it will give us the below output.



###### ****Now let’s alter the above SQL Server Stored Procedure to use the With Encryption Attribute.****

-- Stored Procedure with encryption option

ALTER Procedure sp\_GetEmployeeDetailsById

**(**

@ID INT

**)**

WITH ENCRYPTION

AS

BEGIN

SELECT Name, Gender, CAST**(**DOB AS DATE**)** AS DOB

FROM Employee

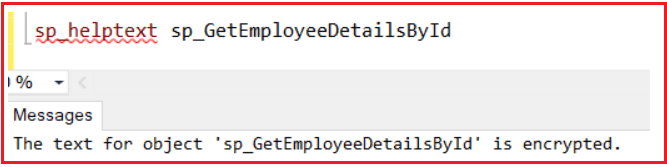
WHERE ID = @ID

END

Now if we check the text column of the **syscomments system**table, then the value of the text column will be null. Please execute the below code to check.

**SELECT \* FROM SYSCOMMENTS WHERE ID = OBJECT\_ID(‘sp\_GetEmployeeDetailsById’)**

If we try to retrieve the text of the above SP using the **sp\_helptext sp\_GetEmployeeDetailsById**. You will get a message stating ‘**The text for object ‘sp\_GetEmployeeDetailsById’ is encrypted.**‘ as shown in the below image.



**Note:** When an application is developed for a client at the time of installing this application on the client system we will be using the encryption option on all the views, procedures, functions, triggers, etc, and install on the client machine. So that they will not have the chance of viewing the source code or altering the source code.

##### ****With Recompiled Attribute in SQL Server Stored Procedure:****

Whenever a procedure is compiled for the first time it prepares the best execution plan according to the current state of the database. Then it executes that query plan when the procedure is called.

The compilation of the procedure and preparing the execution plan is prepared not only at the time of procedure creation but each and every time the server is restarted (Implicitly occurs).

If the procedure is created by using the Recompile Attribute. Then it is forced to be compiled each time it is executed and whenever it compiles it prepares the execution plan.

Forcing a procedure for recompilation and prepared the execution plan is required when the database undergoes significant changes to its data or structure.

Another reason to force a procedure to recompile is if at all the tables are added with new indexes from which the procedure might be benefited forcing for recompilation is very important because we cannot wait until the server is restarted for preparing a new query plan.

**Note:** Even if the “Recompile“ Attribute is available it is not suggested to be used if at all there are no significant changes in the structure of the databases.

##### ****Example:****

Alter the stored procedure to use the Recompile option.

ALTER Procedure sp\_GetEmployeeDetailsById

**(**

@ID INT

**)**

WITH RECOMPILE

AS

BEGIN

SELECT Name, Gender, CAST**(**DOB AS DATE**)** AS DOB

FROM Employee

WHERE ID = @ID

END

**stored Procedures vs Functions**

[**#**](https://www.dofactory.com/sql/stored-procedures#functions)

A list of differences between stored procedures and functions.

|  |  |
| --- | --- |
| STORED PROCEDURE | FUNCTION |
| Return value is optional | Must return a value |
| Supports input and output parameters | Only supports input parameters |
| Cannot be called from a function | Can be called from a procedure |
| Allows SELECT, INSERT, UPDATE, and DELETE queries. | Only allows SELECT statements |
| Cannot be utilized in a SELECT statement | Can be embedded in a SELECT statement |
| Supports Try-Catch exceptions | Does not support error exception |
| Supports Transactions | Does not support Transactions |

# Exception Handling in SQL Server

Back to: [SQL Server Tutorial For Beginners and Professionals](https://dotnettutorials.net/course/ms-sql-server/)

## ****Exception Handling in SQL Server with Examples****

In this article, I am going to discuss **Exception Handling in SQL Server**with Examples. As a developer, it is very important for you to understand Exception Handling in SQL Server. As part of this article, we are going to discuss the following pointers related to SQL Server Exception Handling.

1. **Why do we need Exception Handling in SQL Server?**
2. **What Happens in SQL Server when an Exception Occurred?**
3. **What is Exception handling in SQL Server?**
4. **Exception Handling Using RAISERROR System Function.**
5. **Understanding the RaiseError System Function in SQL Server.**
6. **Using @@Error System Function in SQL Server**
7. **Understanding Error Number, Error Message, Severity Level, and Error State**

##### ****Why do we need Exception Handling in SQL Server?****

Let us understand the need for exception handling in SQL Server with an example. So, create a SQL Server Stored Procedure for dividing two numbers by executing the below query.’

**CREATE** **PROCEDURE** spDivideTwoNumber(

@Number1 **INT**,

@Number2 **INT**

)

**AS**

**BEGIN**

**DECLARE** @Result **INT**

**SET** @Result = 0

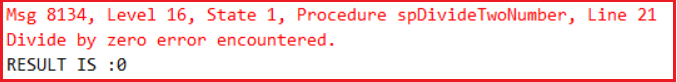
**SET** @Result = @Number1 / @Number2

**PRINT** 'RESULT IS :'+**CAST**(@Result **AS** **VARCHAR**)

**END**

Execution: Now call the stored procedure by giving the numbers as 100 and 5 as shown below  
**EXEC spDivideTwoNumber 100, 5**  
When you execute the above, you will get the following output  
**RESULT IS: 20**

Now execute the stored procedure by giving the second value as 0 as shown below.  
**EXEC spDivideTwoNumber 100, 0**  
When we execute the procedure with 100 and 0, we get the following error. This is because the divisor value is 0 as we know a number cannot be divided by zero as per the rule of mathematics.



As you can see in the above image, whenever an error occurs at a particular line, in SQL Server, it will first print that error message and then it will continue with its execution. So in our example, the statement **SET @Result = @Number1 / @Number2**is the error statement in the procedure and once this statement executes with the second number as 0, it will print Divide by zero error message and then continues its execution with from next statement onwards and this is the reason why it still prints the message **“RESULT IS :0”**.

So, the problem with the above execution is that even if an error occurred in the program, it still showing the result so there are chances of users being confused.

##### ****What Happens in SQL Server when an Exception Occurred?****

In SQL Server, whenever an exception occurred, it displays the exception message and then continues the program execution. But in programming languages such as C#, Java, C++, etc. whenever an exception occurred then the program execution terminates abnormally on the line where the exception occurred.

In the above case both the behaviors are wrong because when errors occur in a programming language it will skip the execution of all the statements after the error even if those statements are not related to the error whereas in SQL Server the execution will not stop when the error occurred, statements related with the error also executed but it should not be happening. For example in the above-stored procedure when the exception occurred it is still displaying the “**RESULT IS: 0**” which should not happen.

##### ****What is Exception handling in SQL Server?****

We handle errors both in programming languages as well as in databases. Handling errors in a programming language means stopping the abnormal termination of the program and allowing the statements which are not related to the error to execute but handling an error in SQL Server means stopping the execution of the statements which are related to the error.

With the introduction of **Try/Catch** blocks in SQL Server 2005, the error handling in the SQL server is now very much similar to programming languages like C#, and Java. But, before understanding the error handling using the **try/catch** block, let’s step back and understand how error handling was done in SQL Server prior to 2005, using the system function **RAISERROR** and **@@Error**.

##### ****Exception Handling Using RAISERROR System Function in SQL Server:****

Let’s alter the same stored procedure that we created in the previous example as shown below to use the **Raiseerror System Function** to handling the exception in SQL Server.

**ALTER** **PROCEDURE** spDivideTwoNumber

@Number1 **INT**,

@Number2 **INT**

**AS**

**BEGIN**

**DECLARE** @Result **INT**

**SET** @Result = 0

**IF**(@Number2 = 0)

**BEGIN**

**RAISERROR**('Second Number Cannot be zero', 16, 1)

**END**

**ELSE**

**BEGIN**

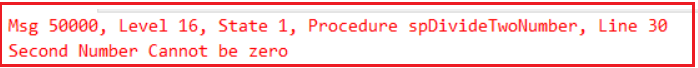
**SET** @Result = @Number1 / @Number2

**PRINT** 'RESULT IS : '+ **CAST**(@Result **AS** **VARCHAR**)

**END**

**END**

Let’s execute the procedure by giving the second value as zero as shown below.  
**EXEC spDivideTwoNumber 100, 0**  
When we execute the above procedure with 100 and 0, it gives us the below error message.



In the above procedure, we are using the system-defined **Raiserror**() function to return an error message back to the calling application, if the second number is zero.

##### ****What is RaiseError System Function in SQL Server?****

The RaiseError System defined Function in SQL Server takes 3 parameters as shown below.   
**RAISERROR(‘Error Message’, ErrorSeverity, ErrorState)**

1. **Error Message:** The custom error message that you want to display whenever the exception is raised.
2. **Error Severity:** When we are returning any custom errors in SQL Server, we need to set the ErrorSeverity level as 16, which indicates this is a general error and this error can be corrected by the user. In our example, the error can be corrected by the user by giving a nonzero value for the second parameter.
3. **Error State:** The ErrorState is also an integer value between 1 and 255. The RAISERROR() function can only generate custom errors if you set the Error State value between 1 to 127.

##### ****@@Error System Function in SQL Server:****

In SQL Server 2000, in order to detect errors, we use the **@@Error** system function. The **@@Error** system function returns a **NON-ZERO** value if there is an error, otherwise, **ZERO** indicates that the previous SQL statement was executed without any error. Let’s modify the stored procedure to make use of the **@@ERROR** system function as shown below.

**ALTER** **PROCEDURE** spDivideTwoNumber

@Number1 **INT**,

@Number2 **INT**

**AS**

**BEGIN**

**DECLARE** @Result **INT**

**SET** @Result = 0

**IF**(@Number2 = 0)

**BEGIN**

**RAISERROR**('Second Number Cannot be zero',16,1)

**END**

**ELSE**

**BEGIN**

**SET** @Result = @Number1 / @Number2

**END**

**IF**(@@**ERROR** <> 0)

**BEGIN**

**PRINT** 'Error Occurred'

**END**

**ELSE**

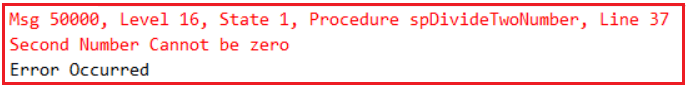
**BEGIN**

**PRINT** 'RESULT IS :'+**CAST**(@Result **AS** **VARCHAR**)

**END**

**END**

Let’s execute the procedure as EXEC spDivideTwoNumber 100, 0  
When we execute the above procedure passing 100 and 0, it gives us the below output.



##### ****Predefined Error Terms in SQL Server:****

Whenever an error occurs under a program like dividing a number by zero, violation of primary key, violation of Check constraint, etc. the system displays an error message telling us the problem encountered in the code. Every error that occurs in the program is associated with four attributes.

1. **Error Number**
2. **Error Message**
3. **Severity Level**
4. **Error State**

##### ****Example:****

**Message 8134 (Error Number), Level 16(SEVERITY Level), State 1 (state), Divide by Zero error encountered (Error Message)**

**Error Number:**The Error number is a unique identifier given for each and every error that occurs in SQL Server. This value will be below 50,000 for predefined errors and must be above or equals to 50,000 for errors defined by the user. While raising custom errors, if we don’t specify the error number, then by default it will set the Error Number as 50000.

**Error Message:** It is a piece of brief information describing the error that occurred which should be maxing from 2047 characters.

**Severity Level:**This tells about the importance of the error which can be ranging between 0 to 24. In which

1. **0 to 9:** are not serves which can be considered as information or status messages.
2. **11 to 16:**  Indicates these errors can be created by the user.
3. **17 to 19:** Indicates these are software errors that cannot be corrected by the user must be reported to the system administrator.
4. **20 to 24:** Indicates fatal errors and if these errors occur they can damage the system or database. So here the connection immediately terminates with the database.

**State:**It is an arbitrary value that is not that important can be ranging between 0 to 127. We use this whenever the same error has to occur in multiple places

# RaiseError and @@ERROR Function in SQL Server

Back to: [SQL Server Tutorial For Beginners and Professionals](https://dotnettutorials.net/course/ms-sql-server/)

## ****RaiseError and @@ERROR function in SQL Server with Example****

In this article, I am going to discuss the **RaiseError and @@ERROR Function in SQL Server** with Example. Please read our previous article where we discussed the [**Basics Concepts of Exception Handling in SQL Server**](https://dotnettutorials.net/lesson/exception-handling-in-sql-server/)with examples.

##### RaiseError Function in SQL Server

The system-defined **Raiserror**() function returns an error message back to the calling application. The RaiseError System defined Function in SQL Server takes 3 parameters as shown below.   
**RAISERROR(‘Error Message’, ErrorSeverity, ErrorState)**

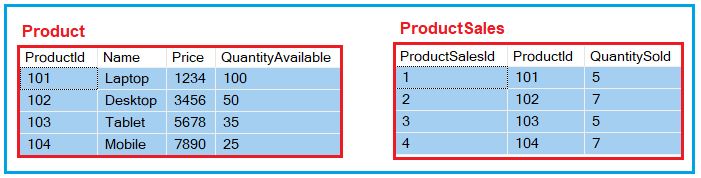
1. **Error Message:** The custom error message that you want to display whenever the exception is raised.
2. **Error Severity:** When we are returning any custom errors in SQL Server, we need to set the ErrorSeverity level as 16, which indicates this is a general error and this error can be corrected by the user. In our example, the error can be corrected by the user by giving a nonzero value for the second parameter.
3. **Error State:** The ErrorState is also an integer value between 1 and 255. The RAISERROR() function can only generate custom errors if you set the Error State value between 1 to 127.

##### ****@@Error System Function in SQL Server:****

In SQL Server 2000, in order to detect errors, we use the @**@Error** system function. The @**@Error** system function returns a **NON-ZERO** value if there is an error, otherwise, **ZERO** indicates that the previous SQL statement was executed without any error.

##### Example: RaiseError and @@ERROR Function in SQL Server

We are going to use the following Product and ProductSales table to understand how to handle errors in SQL Server using RaiseError and @ERROR System-Defined Functions.



Please use the below SQL Script to create and populate the Product and ProductSales table with sample data.

-- Create Product table

**CREATE** **TABLE** Product

(

ProductId **INT** **PRIMARY KEY**,

Name **VARCHAR**(50),

Price **INT**,

QuantityAvailable **INT**

)

**GO**

-- Populate the Product Table with some test data

**INSERT** **INTO** Product **VALUES**(101, 'Laptop', 1234, 100)

**INSERT** **INTO** Product **VALUES**(102, 'Desktop', 3456, 50)

**INSERT** **INTO** Product **VALUES**(103, 'Tablet', 5678, 35)

**INSERT** **INTO** Product **VALUES**(104, 'Mobile', 7890, 25)

**GO**

-- Create ProductSales table

**CREATE** **TABLE** ProductSales

(

ProductSalesId **INT** **PRIMARY KEY**,

ProductId **INT**,

QuantitySold **INT**

)

**GO**

-- Populate the ProductSales Table with some test data

**INSERT** **INTO** ProductSales **VALUES**(1, 101, 5)

**INSERT** **INTO** ProductSales **VALUES**(2, 102, 7)

**INSERT** **INTO** ProductSales **VALUES**(3, 103, 5)

**INSERT** **INTO** ProductSales **VALUES**(4, 104, 7)

Go

###### ****Create the following stored procedure for product sales.****

The following stored procedure accepts 2 parameters – ProductID and QuantityToSell. The ProductID parameter specifies the product that we want to sell and the QuantityToSell parameter specifies the quantity that we want to sell. In the below procedure, if enough stock is not available then we are raising a custom exception by using the Raiserror statement.

**CREATE** **PROCEDURE** spSellProduct

@ProductID **INT**,

@QuantityToSell **INT**

**AS**

**BEGIN**

-- First we need to Check the stock available for the product we want to sell

**DECLARE** @StockAvailable **INT**

**SELECT** @StockAvailable = QuantityAvailable **FROM** Product **WHERE** ProductId = @ProductId

-- We need to throw an error to the calling application

-- if the stock is less than the quantity we want to sell

**IF**(@StockAvailable < @QuantityToSell)

**BEGIN**

Raiserror('Enough Stock is not available', 16, 1)

**END**

-- If enough stock is available

**ELSE**

**BEGIN**

-- We need to start the transaction

**BEGIN** **TRANSACTION**

-- First we need to reduce the quantity available

**UPDATE** Product **SET** QuantityAvailable = (QuantityAvailable - @QuantityToSell)

**WHERE** ProductID = @ProductID

-- Then Calculate MAX ProductSalesId

**DECLARE** @MaxProductSalesId **INT**

**SELECT** @MaxProductSalesId = **CASE**

**WHEN** **MAX**(ProductSalesId) IS NULL **THEN** 0

**ELSE** **MAX**(ProductSalesId)

**END**

**FROM** ProductSales

-- Increment @MaxProductSalesId by 1, so we don't get a primary key violation

Set @MaxProductSalesId = @MaxProductSalesId + 1

-- We need to insert the quantity sold into the ProductSales table

**INSERT** **INTO** ProductSales(ProductSalesId, ProductId, QuantitySold)

**VALUES**(@MaxProductSalesId, @ProductId, @QuantityToSell)

**COMMIT** **TRANSACTION**

End

**END**

The problem with the above-stored procedure is that the transaction is always going to be committed even though there is an error somewhere between updating the Product table and inserting data into the ProductSales table.

The main purpose of wrapping these 2 statements (Update Product Statement and Insert into ProductSales statement) in a transaction is to ensure that, both of these statements are treated as a single unit. For example, if we have an error when executing the second statement, then the first statement should be rolled back.

Let us modify the stored procedure to use the @@ERROR function to check if there any error occurred. If no error occurred then we are committing the transaction else we roll backing the transaction.

**ALTER** **PROCEDURE** spSellProduct

@ProductID **INT**,

@QuantityToSell **INT**

**AS**

**BEGIN**

-- First we need to Check the stock available for the product we want to sell

**DECLARE** @StockAvailable **INT**

**SELECT** @StockAvailable = QuantityAvailable **FROM** Product **WHERE** ProductId = @ProductId

-- We need to throw an error to the calling application

-- if the stock is less than the quantity we want to sell

**IF**(@StockAvailable< @QuantityToSell)

**BEGIN**

Raiserror('Enough Stock is not available',16,1)

**END**

-- If enough stock is available

**ELSE**

**BEGIN**

-- We need to start the transaction

**BEGIN** **TRANSACTION**

-- First we need to reduce the quantity available

**UPDATE** Product **SET** QuantityAvailable = (QuantityAvailable - @QuantityToSell)

**WHERE** ProductID = @ProductID

-- Calculate MAX ProductSalesId

**DECLARE** @MaxProductSalesId **INT**

**SELECT** @MaxProductSalesId = **CASE**

**WHEN** **MAX**(ProductSalesId) IS NULL **THEN** 0

**ELSE** **MAX**(ProductSalesId)

**END**

**FROM** ProductSales

-- Increment @MaxProductSalesId by 1, so we don't get a primary key violation

Set @MaxProductSalesId = @MaxProductSalesId + 1

-- We need to insert the quantity sold into the ProductSales table

**INSERT** **INTO** ProductSales(ProductSalesId, ProductId, QuantitySold)

**VALUES**(@MaxProductSalesId, @ProductId, @QuantityToSell)

-- The @@Error returns a NON-ZERO value if there is an error, otherwise it will return ZERO,

-- indicating that the previous SQL statement encountered no errors

**IF**(@@**ERROR** <> 0)

**BEGIN**

**ROLLBACK** **TRANSACTION**

**PRINT** 'Rolled Back the Transaction'

**END**

**ELSE**

**BEGIN**

**COMMIT** **TRANSACTION**

**PRINT** 'Committed the Transaction'

**END**

End

**END**

In the above procedure, if you comment the line (**Set @MaxProductSalesId = @MaxProductSalesId + 1**), and then execute the stored procedure there will be a primary key violation error when trying to insert into the ProductSales table as a result of which the entire transaction will be rolled back.

**Note**: The @@ERROR is cleared and reset on each statement execution. Check it immediately following the statement being verified, or save it to a local variable that can be checked later.

In the **Product** table, we already have a record with **ProductID = 4**. So the insert statement causes a primary key violation error. The @@ERROR retains the error number, as we are checking for it immediately after the statement that causes the error.

**INSERT** **INTO** Product values(4, 'Mobile Phone', 1500, 100)

**IF**(@@**ERROR** <> 0)

**PRINT** 'Error Occurred'

**ELSE**

**PRINT** 'No Errors'

On the other hand, when you execute the code below, you will get the message**‘No Errors’**. This is because the @@ERROR is cleared and reset on each statement execution.

**INSERT** **INTO** Product values(4, 'Mobile Phone', 1500, 100)

-- At this point the @@ERROR will have a NON ZERO value

**SELECT** \* **FROM** Product

-- At this point the @@ERROR reset to ZERO, because the

-- select statement successfully executed

**IF**(@@**ERROR** <> 0)

**PRINT** 'Error Occurred'

**ELSE**

**PRINT** 'No Errors'

In the below example, we are storing the value of the **@@Error** function to a local variable, which is used later.

**DECLARE** @Error **INT**

**INSERT** **INTO** Product **VALUES**(4, 'Mobile Phone', 1500, 100)

Set @Error = @@**ERROR**

**SELECT** \* **FROM** Product

**IF**(@Error <> 0)

**PRINT** 'Error Occurred'

**ELSE**

**PRINT** 'No Errors'

# How to Raise Errors Explicitly in SQL Server

Back to: [SQL Server Tutorial For Beginners and Professionals](https://dotnettutorials.net/course/ms-sql-server/)

## ****How To Raise Errors Explicitly in SQL Server****

In this article, I am going to discuss **how to raise errors explicitly in SQL Server** with examples along with we will also discuss the different options that we can use with Raiserror in SQL Server. Please read our previous article where we discussed the [**real-time example of the Raise Error**](https://dotnettutorials.net/lesson/raiseerror-in-sql-server/) system function. As part of this article, we are going to discuss the following pointers.

1. **How to Raise Errors Explicitly in SQL Server?**
2. **Raise error using RAISERROR statement in SQL Server**
3. **Raise Error using throw statement in SQL Server**
4. **What is the difference between the RAISERROR function and the throw statement?**
5. **Understanding the RaiseError statement with the Log option.**
6. **How to Raise Errors By storing the Error message in the SysMessage table.**

##### ****How to Raise Errors Explicitly in SQL Server?****

Generally, errors are raised in a program for predefined reasons like dividing a number by zero, violation of primary key, violation of check, violation of referential integrity, etc. But if you want then you can also raise an error in your programs in two different ways. They are as follows.

1. **Raiserror Statement**
2. **throw Statement (new feature of SQL Server 2012)**

**Raiserror Syntax: Raiserror (errorid/errormsg, SEVERITY, state)[with log]**

**throw Syntax: Throw errorid, errormsg, state**

##### ****Example: Raise Error using RAISERROR statement in SQL Server.****

In the following stored Procedure, we raise an error when the division is 1 by using the **RAISERROR** statement.

**CREATE** **PROCEDURE** spDivideBy1(@No1 **INT**, @No2 **INT**)

**AS**

**BEGIN**

**DECLARE** @Result **INT**

**SET** @Result = 0

**BEGIN** **TRY**

**IF** @No2 = 1

**RAISERROR** ('DIVISOR CANNOT BE ONE', 16, 1)

**SET** @Result = @No1 / @No2

**PRINT** 'THE RESULT IS: '+**CAST**(@Result **AS** **VARCHAR**)

**END** **TRY**

**BEGIN** **CATCH**

**PRINT** ERROR\_NUMBER()

**PRINT** ERROR\_MESSAGE()

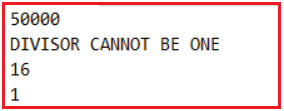
**PRINT** ERROR\_SEVERITY()

**PRINT** ERROR\_STATE()

**END** **CATCH**

**END**

**Example of execution: EXEC spDivideBy1 10, 1**



##### ****Example: Raise Error using throw statement in SQL Server.****

The above procedure can also be rewritten with the help of a throw statement in place of Raiserror as following.

**ALTER** **PROCEDURE** spDivideBy2(@No1 **INT**, @No2 **INT**)

**AS**

**BEGIN**

**DECLARE** @Result **INT**

**SET** @Result = 0

**BEGIN** **TRY**

**IF** @No2 = 1

**THROW** 50001,'DIVISOR CANNOT BE ONE', 1

**SET** @Result = @No1 / @No2

**PRINT** 'THE RESULT IS: '+**CAST**(@Result **AS** **VARCHAR**)

**END** **TRY**

**BEGIN** **CATCH**

**PRINT** ERROR\_NUMBER()

**PRINT** ERROR\_MESSAGE()

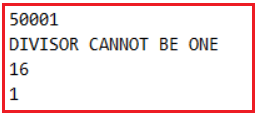
**PRINT** ERROR\_SEVERITY()

**PRINT** ERROR\_STATE()

**END** **CATCH**

**END**

**EXECUTION: EXEC spDivideBy2 10, 1**



##### ****What is the difference between the RAISERROR function and the throw statement in SQL Server?****

If we use any of the two statements in a program for raising a custom error without try and catch blocks, the RAISERROR statement after raising the error will still continue the execution of the program whereas the throw statement will terminate the program abnormally on that line. But if they are used under try block both will behave in the same way that it will jump directly to catch block from where the error got raised.

The RAISERROR statement will give an option of specifying the ERROR SEVERITY Level of the error message whereas we don’t have this option in the case of the throw statement where all error messages will have a default  ERROR SEVERITY level as 16.

In the case of RAISERROR, there is a chance of recording the error message into the server log file by using the with log option whereas we cannot do this in case of a throw statement.

In the case of throw, we need to specify both the errorid and error message to raise the error whereas in the case of RAISERROR we can specify either id or message. If the id is not specified default error id is 50000 but if we want to specify only the error id first we need to add the error message in the sysmessage table by specifying a unique id to the table.

##### ****OPTIONS WITH RAISERROR STATEMENT:****

**With Log:**By using this option in the RAISERROR statement we can record the error message in the SQL Server log file so that if the errors are fatal database administrator can take care of fixing those errors. If the severity of the error is greater than 20 specifying the With Log option is mandatory. To test this ALTER the procedure spDivideBy1 by changing the raiserror statement as following

**RAISERROR (‘DIVISOR CANNOT BE ONE’, 16, 1) WITH LOG**

###### ****Below is the complete procedure****

**ALTER** **PROCEDURE** spDivideBy1(@No1 **INT**, @No2 **INT**)

**AS**

**BEGIN**

**DECLARE** @Result **INT**

**SET** @Result = 0

**BEGIN** **TRY**

**IF** @No2 = 1

**RAISERROR** ('DIVISOR CANNOT BE ONE', 16, 1) **WITH** **LOG**

**SET** @Result = @No1 / @No2

**PRINT** 'THE RESULT IS: '+**CAST**(@Result **AS** **VARCHAR**)

**END** **TRY**

**BEGIN** **CATCH**

**PRINT** ERROR\_NUMBER()

**PRINT** ERROR\_MESSAGE()

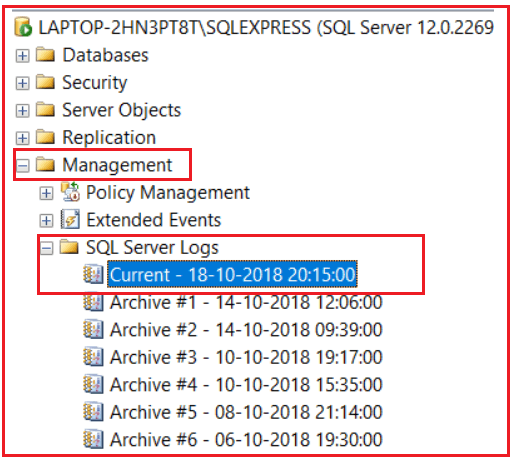
**PRINT** ERROR\_SEVERITY()

**PRINT** ERROR\_STATE()

**END** **CATCH**

**END**

Now execute the procedure and whenever the given error raises we can watch the error messages recorded under the SQL Server log file. To view the log file. In object explorer, go to the management node, then open SQL Server logs node and open the current log file by double-clicking on it as shown below.



##### ****Using substitutional parameters in the error message of RAISERROR:****

Just like C language, we can also substitute values into the error message to make the error message as dynamic as following

**RAISERROR (‘THE NUMBER %d CANNOT BE DIVIDED BY %d’,16, 1, @No1, @No2)WITH LOG**

##### ****Raising Errors By storing the Error message in the SysMessage table:****

We can raise an error without giving the error message in the RAISERROR statement but in place of the error message we need the specify the error id and to specify the error id we need to record that error id with the error message in the SysMessage table by using the system defined procedure “**SP\_ADDMESSAGE**”.

**Syntax: SP\_ADDMESSAGE <error id>, <severity>, <error message>**

**To test this first add a record to sysmessage table as following**

**EXEC sp\_Addmessage 51000, 16, ‘DIVIDE BY ONE ERROR ENCOUNTERED’**

**Now alter the procedure by changing the RAISERROR statement as following**

**RAISERROR (51000,16, 1)WITH LOG**

##### ****Deleting the error messages from sysmessages table:****

**Syntax: SP\_DROPMESSAGE <error id>**

**Example: EXEC sp\_dropMessage 51000**

# xception Handling Using Try Catch in SQL Server

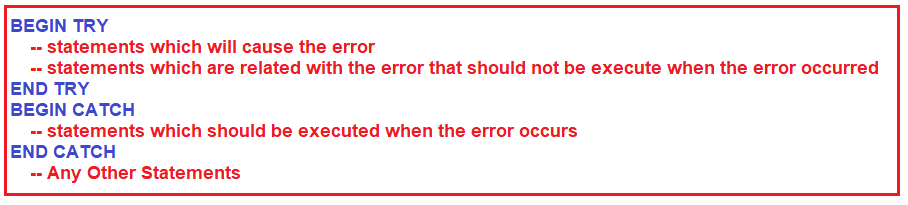
## ****Exception Handling Using Try Catch in SQL Server****

In this article, I am going to discuss **Exception Handling Using Try Catch in SQL Server** with examples. Please read our previous article where we discussed [**how to raise errors explicitly in SQL Server**](https://dotnettutorials.net/lesson/how-to-raise-errors-explicitly-in-sql-server/) using RaiseError and Throw statement. As part of this article, we are going to discuss the following pointers.

1. **How to use Try Catch in SQL Server to Handle Error?**
2. **Example To understand the Try-Catch implementation in SQL Server.**
3. **Try-catch Implementation in SQL Server with system-defined error statements.**
4. **What is ERROR\_MESSAGE() in SQL Server?**
5. **A real-time example of Exception handling using the try-catch in SQL Server.**

##### ****How to use Try Catch in SQL Server to Handle Error?****

From SQL Server 2005 we are provided with a structure error handling mechanism with the help of **TRY and CATCH** blocks. The syntax of using **TRY- CATCH** is shown in the below image.



The **SQL statements** which can have the possibility to throw an exception need to be placed in between the **BEGIN TRY** and **END TRY** blocks. If there is an exception that occurred in the TRY block, then the control immediately moves to the Corresponding **CATCH** block. If there is no exception occurred in the TRY block, then the **CATCH** block simply skipped, and the statements which are present after the **CATCH** block are going to be executed.

**Note**: The Errors trapped by a CATCH block are not going to be returned to the calling application. If you want to return the error information back to the calling application then you need to use the **RAISERROR()** function explicitly with the catch block. In our previous article, we discussed [**how to raise errors explicitly using the RAISERROR()**](https://dotnettutorials.net/lesson/how-to-raise-errors-explicitly-in-sql-server/) function.

##### ****Example: To understand the Try-Catch implementation in SQL Server.****

In the following example, we create a stored procedure for dividing 2 variables values by using the SQL Server **TRY CATCH**implementation with user-defined error statements.

**CREATE** **PROCEDURE** spDivideTwoNumbers

@Number1 **INT**,

@Number2 **INT**

**AS**

**BEGIN**

**DECLARE** @Result **INT**

**SET** @Result = 0

**BEGIN** **TRY**

**SET** @Result = @Number1 / @Number2

**PRINT** 'RESULT IS : '+**CAST**(@Result **AS** **VARCHAR**)

**END** **TRY**

**BEGIN** **CATCH**

**PRINT** 'SECOND NUMBER SHOULD NOT BE ZERO'

**END** **CATCH**

**END**

**EXEC spDivideTwoNumbers 10, 2**  
**OUTPUT: RESULT IS : 5**

**EXEC spDivideTwoNumbers 10, 0**  
**OUTPUT: SECOND NUMBER SHOULD NOT BE ZERO**

When we execute the above-stored procedure with correct values, then the error will not occur in the program. That means after executing all the statements in the try block the control directly jumps to the statements present after the catch block without executing the catch block.

If any error occurs in the execution process i.e. in the try block, then in such case from the line where the error got occurred, the control directly jumps to the catch block. So rest of the statements in the try block will not execute whereas the catch block will execute.

**Note:** In the above program when the error got occurred, we are displaying an error message “**SECOND NUMBER SHOULD NOT BE ZERO**”. In place of that error message we can also display the original error message by calling a function “**Error\_Message**”. To test this rewriting the code inside the catch block as following

**Print Error\_Message()**

##### ****Example: try-catch with system defined error statements in SQL Server:****

In the following example, we create a stored procedure for dividing two variables values by using try-catch implementation with system-defined error statements in SQL Server.

**ALTER** **PROCEDURE** spDivideTwoNumbers

@Number1 **INT**,

@Number2 **INT**

**AS**

**BEGIN**

**DECLARE** @Result **INT**

**SET** @Result = 0

**BEGIN** **TRY**

**SET** @Result = @Number1 / @Number2

**PRINT** 'RESULT IS : '+**CAST**(@Result **AS** **VARCHAR**)

**END** **TRY**

**BEGIN** **CATCH**

**PRINT** ERROR\_MESSAGE()

**END** **CATCH**

**END**

**Execution: EXEC spDivideTwoNumbers 10, 0**

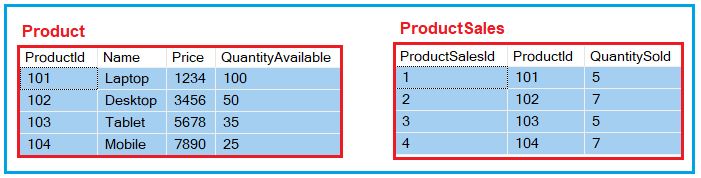
**OUTPUT:** **Divide by zero error encountered.**

##### ****What is ERROR\_MESSAGE() in SQL Server?****

This method is used to display what type of error has occurred in the try block.

##### ****A real-time example of using the try-catch implementation in SQL Server.****

We are going to use the following Product and ProductSales table to understand how to handle errors in SQL Server using RaiseError and @ERROR System-Defined Functions.



Please use the below SQL Script to create and populate the Product and ProductSales table with sample data.

-- Create Product table

**CREATE** **TABLE** Product

(

ProductId **INT** **PRIMARY KEY**,

Name **VARCHAR**(50),

Price **INT**,

QuantityAvailable **INT**

)

**GO**

-- Populate the Product Table with some test data

**INSERT** **INTO** Product **VALUES**(101, 'Laptop', 1234, 100)

**INSERT** **INTO** Product **VALUES**(102, 'Desktop', 3456, 50)

**INSERT** **INTO** Product **VALUES**(103, 'Tablet', 5678, 35)

**INSERT** **INTO** Product **VALUES**(104, 'Mobile', 7890, 25)

**GO**

-- Create ProductSales table

**CREATE** **TABLE** ProductSales

(

ProductSalesId **INT** **PRIMARY KEY**,

ProductId **INT**,

QuantitySold **INT**

)

**GO**

-- Populate the ProductSales Table with some test data

**INSERT** **INTO** ProductSales **VALUES**(1, 101, 5)

**INSERT** **INTO** ProductSales **VALUES**(2, 102, 7)

**INSERT** **INTO** ProductSales **VALUES**(3, 103, 5)

**INSERT** **INTO** ProductSales **VALUES**(4, 104, 7)

Go

##### ****Stored procedure for product sales using TRY Catch Implementation in SQL Server****

The following stored procedure accepts 2 parameters i.e. ProductID and QuantityToSell. The ProductID parameter specifies the product that we want to sell and the QuantityToSell parameter specifies the quantity that we want to sell. In the below procedure, if enough stock is not available then we are raising a custom exception by using the Raiserror statement. If enough stock is available, then we are performing the required operation as part of a transaction and moreover, the transaction is within the Begin TRY and End TRY block.

**CREATE** **PROCEDURE** spSellProduct

@ProductID **INT**,

@QuantityToSell **INT**

**AS**

**BEGIN**

-- First we need to Check the stock available for the product we want to sell

**DECLARE** @StockAvailable **INT**

**SELECT** @StockAvailable = QuantityAvailable

**FROM** Product

**WHERE** ProductId = @ProductId

-- We need to throw an error to the calling application

-- if the stock is less than the quantity we want to sell

**IF**(@StockAvailable< @QuantityToSell)

**BEGIN**

Raiserror('Enough Stock is not available',16,1)

**END**

-- If enough stock is available

**ELSE**

**BEGIN**

**BEGIN** **TRY**

-- We need to start the transaction

**BEGIN** **TRANSACTION**

-- First we need to reduce the quantity available

**UPDATE** Product **SET**

QuantityAvailable = (QuantityAvailable - @QuantityToSell)

**WHERE** ProductID = @ProductID

-- Calculate MAX ProductSalesId

**DECLARE** @MaxProductSalesId **INT**

**SELECT** @MaxProductSalesId = **CASE**

**WHEN** **MAX**(ProductSalesId) IS NULL **THEN** 0

**ELSE** **MAX**(ProductSalesId)

**END**

**FROM** ProductSales

-- Increment @MaxProductSalesId by 1, so we don't get a primary key violation

Set @MaxProductSalesId = @MaxProductSalesId + 1

-- We need to insert the quantity sold into the ProductSales table

**INSERT** **INTO** ProductSales(ProductSalesId, ProductId, QuantitySold)

**VALUES**(@MaxProductSalesId, @ProductId, @QuantityToSell)

**COMMIT** **TRANSACTION**

**END** **TRY**

**BEGIN** **CATCH**

**ROLLBACK** **TRANSACTION**

**SELECT** ERROR\_NUMBER() **as** ErrorNumber,

ERROR\_MESSAGE() **as** ErrorMessage,

ERROR\_PROCEDURE() **as** ErrorProcedure,

ERROR\_STATE() **as** ErrorState,

ERROR\_SEVERITY() **as** ErrorSeverity,

ERROR\_LINE() **as** ErrorLine

**END** **CATCH**

End

**END**

In the procedure spSellProduct the Begin Transaction and Commit Transaction statements are wrapped between the Begin Try and End Try block. If there is no error occured in the code that is enclosed within the BEGIN TRY and END TRY block, then the COMMIT TRANSACTION statement gets executed and the changes are made permanent to the database.

On the other hand, if there is an error occurred within the try block then the control immediately jumps to the CATCH block, and in the CATCH block, we are rolling back the transaction. So it’s much easier to handle errors with the Try/Catch construct than with the @@Error system function in SQL Server.

SQL Server also provides some built-in functions that we can use in the scope of a CATCH block which is used to retrieve more information about the error that occurred and these functions will return NULL if they are executed outside the scope of the CATCH block.

**Note:** We cannot use the **TRY/CATCH** implementation within a user-defined function.