How Raise Errors

Try … catch statements are used to control the flow of execution when an error occurs. To start, you code the TRY block around any statements that might cause an error to be raised. The TRY block begins with the BEGIN TRY and ends with END TRY (as shown in the following). Immediately following the TRY block, you must code a single CATCH block. A CATCH block begins with BEGIN CATCH and ends with END CATCH. Within the CATCH block you can include any statements that handle the error that might be raised in the TRY block.
The syntax of try…catch is given in the following:

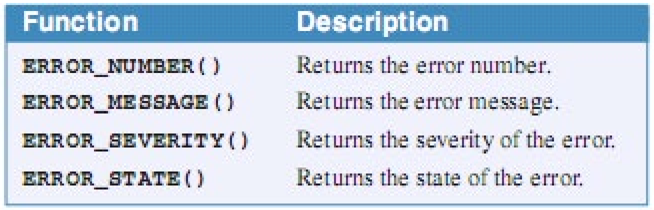
```
BEGIN TRY
     { sql_statement | statement_block }
END TRY
BEGIN CATCH
     [ { sql_statement | statement_block } ]
END CATCH
```

- If there is no error in the code inside the TRY block, then after the last statement is executed, control passes to the statement after the END CATCH. In other words if the code in TRY block doesn't lead to an error the CATCH block is not executed.

In the following example the TRY block is coded around an INSERT and PRINT statements. In the CATCH block a PRINT statement is used to display a message that indicates the INSERT statement in the TRY block did not execute successfully. Then the second PRINT statement uses two functions that are designed to work within a CATCH block to provide more detailed information about the error.

```
BEGIN TRY
     INSERT INTO Invoices
      VALUES (799, 'ZXK-799','2016-05-07',299.95,0,0,1,'2016-06-
06',NULL)
      PRINT 'Success: Record was inserted'
END TRY
BEGIN CATCH
     PRINT 'Failure: Record was not inserted'
      PRINT 'Error' + CONVERT (varchar,ERROR_NUMBER(),1)
+':'+ERROR_MESSAGE()
END CATCH
```

All four of the functions you can use within a CATCH block are shown in the following. You can also use the CATCH block to perform other tasks such as writing info in a log table or rolling back a transaction.



Function	Description
ERROR_NUMBER()	Returns the error number.
ERROR_MESSAGE()	Returns the error message.
ERROR_SEVERITY()	Returns the severity of the error.
ERROR_STATE()	Returns the state of the error.

In addition to TRY … CATCH statements to handle errors after they occur, you can also prevent errors before they occur by checking data before it's used to make sure it's valid. Checking data before it's used is often referred to as data validation and we have been practicing it in our stored procedures. If the data is not valid you can execute code that makes it valid, or you can return an error to the calling program.

To return an error it's often helpful to use the THROW statement. Then if the calling program contains a TRY…CATCH statement, it can catch and handle the error.

The syntax for THROW is shown in the following:

```
THROW [error_number, message, state]
```

- Error number: the value of this parameter must be 50000 or greater. You can use this value to indicate the type of error that occurred.
- Message: the error message you want to display if the error is raised.
- State: the state that you want to associate with the error. The state code is strictly informational and has no system meaning. You can use any value between 0 and 255 to represent the state that the system was in when the error was raised. In most cases, you'll just code 1 for this argument.

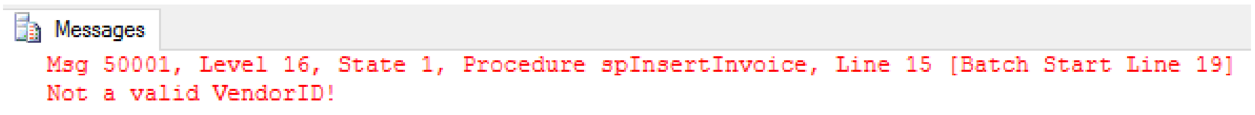
The following is a stored procedure with a THROW statement inside it. The procedure checks the VendorID that's passed from the calling program before it performs the insert operation that's specified by the INSERT statement. That way, the system error that's raised when you try to insert a row with an invalid foreign key will never occur. Instead if the VendorID value is invalid, the THROW statement will raise a custom error that provides a user-friendly message.

```
CREATE PROC spInsertInvoice
      @VendorID int,
      @InvoiceNumber varchar(50),
      @InvoiceDate smalldatetime,
      @InvoiceTotal money,
      @TermsID int,
      @InvoiceDueDate smalldatetime
AS

IF EXISTS (SELECT * FROM Vendors WHERE VendorID=@VendorID)
      INSERT INTO Invoices
      VALUES (@VendorID,@InvoiceNumber,@InvoiceDate,@InvoiceTotal,
                 0,0,@TermsID,@InvoiceDueDate,NULL);
ELSE
      THROW 50001, 'Not a valid VendorID!', 1;
```

The following is the result you get when you run the EXEC script

```
EXEC spInsertInvoice 799,'ZXK-799', '2016-05-01',299.95, 1, '2016-06-01';
```

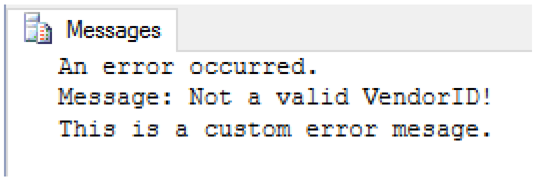


Messages

```
Msg 50001, Level 16, State 1, Procedure spInsertInvoice, Line 15 [Batch Start Line 19]
Not a valid VendorID!
```

The following is the script that calls the procedure inside a TRY block and the error you get.

```
BEGIN TRY
	EXEC spInsertInvoice 799,'ZXK-799', '2016-05-01',299.95, 1,
'2016-06-01';
END TRY
BEGIN CATCH
	PRINT 'An error occurred.'
	PRINT 'Message: '+ ERROR_MESSAGE();
	IF ERROR_NUMBER()>=50000
		PRINT 'This is a custom error message.'
END CATCH
```



Messages

```
An error occurred.
Message: Not a valid VendorID!
This is a custom error mesage.
```

- The THROW statement manually raises an error. Unless this error is caught by a TRY…CATCH statement within the stored procedure, the error will be returned to the caller just like an error that's raised by the database engine.
- A THROW statement that's coded within a block must be preceded by a semicolon. That's true even if the THROW statement is the first or only statement in the block. E.g.

```
BEGIN
	;
	THROW 50001, 'Not a valid VendorID!', 1;
END
```