Module 17: Implementing Error Handling

Contents:

Module Overview

When creating applications for Microsoft® SQL Server® using the T-SQL language, appropriate handling of errors is critically important. A large number of myths surround how error handling works in T-SQL. In this module, you will explore T-SQL error handling, look at how it has traditionally been implemented, and how structured exception handling can be used.

Objectives

After completing this module, you will be able to:

- Implement T-SQL error handling.
- Implement structured exception handling.

Lesson 1: Implementing T-SQL Error Handling

You should consider how errors can be handled or reported in T-SQL. The T-SQL language offers a variety of error handling capabilities. This lesson covers basic T-SQL error handling, including how you can raise errors intentionally and set up alerts to fire when errors occur. In the next lesson, you will see how to implement a more advanced form of error handling known as structured exception handling.

Lesson Objectives

After completing this lesson, you will be able to:

- Raise errors using the RAISERROR statement.
- Raise errors using the THROW statement.
- Use the @@ERROR system variable.
- Create custom errors.
- Create alerts that fire when errors occur.

Errors and Error Messages

Elements of Database Engine Errors	
Error number	Unique number identifying the specific error
Error message	Text describing the error
Severity	Numeric indication of seriousness from 1 to 25
State	Internal state code for the database engine condition
Procedure	The name of the stored procedure or trigger in which the error occurred
Line number	Which statement in the batch or procedure generated the error

- System error messages are in **sys.messages**
- Add custom application errors using **sp_add_message**

An error indicates a problem or notable issue that arises during a database operation. Each error includes the following elements:

- **Error number**. Unique number identifying the specific error.
- **Error message**. Text describing the error.
- **Severity**. Numeric indication of seriousness from 1 to 25.
- **State**. Internal state code for the database engine condition.
- **Procedure**. The name of the stored procedure or trigger in which the error occurred.
- **Line number**. Which statement in the batch or procedure generated the error.

Errors can be generated by the SQL Server Database Engine in response to an event or failure at the system level; or you can generate application errors in your Transact-SQL code.

System Errors

System errors are predefined, and you can view them in the sys.messages system view. When a system error occurs, SQL Server may take automatic remedial action, depending on the severity of the error. For example, when a high-severity error occurs, SQL Server may take a database offline or even stop the database engine service.

Custom Errors

You can generate errors in Transact-SQL code to respond to application-specific conditions or to customize information sent to client applications in response to system errors. These application errors can be defined inline where they are generated, or you can predefine them in the sys.messages table alongside the system-supplied errors. The error numbers used for custom errors must be 50001 or above.

To add a custom error message to **sys.messages**, use **sp_addmessage**. The user for the message must be a member of the sysadmin or serveradmin fixed server roles.

sp_addmessage Syntax

```
sp_addmessage [ @msgnum= ] msg_id , [ @severity= ] severity , [
@msgtext= ] 'msg'
     [ , [ @lang= ] 'language' ]
     [ , [ @with_log= ] { 'TRUE' | 'FALSE' } ]
     [ , [ @replace= ] 'replace' ]
```

sp_addmessage Example

```
sp_addmessage 50001, 10, N’Unexpected value entered’;
```

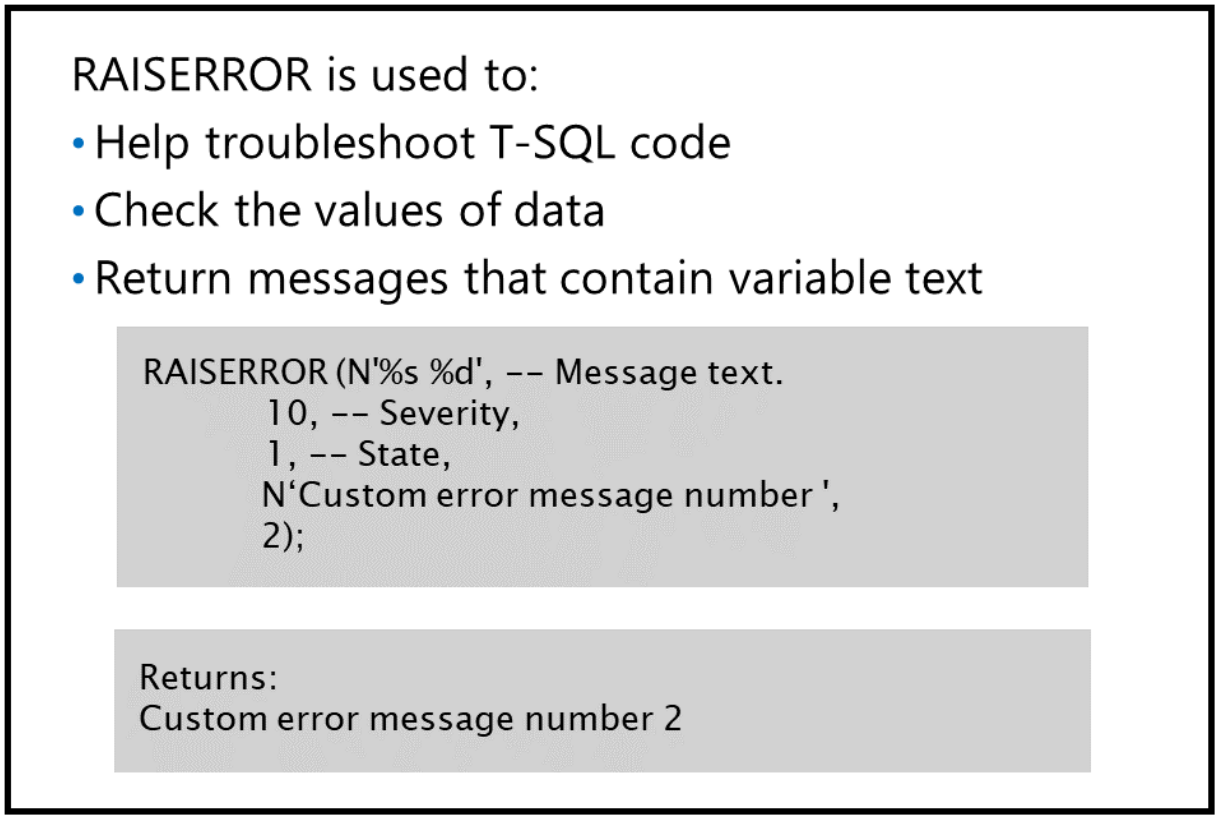
In addition to being able to define custom error messages, members of the sysadmin server role can also use an additional parameter, @with_log. When set to TRUE, the error will also be recorded in the Windows Application log. Any message written to the Windows Application log is also written to the SQL Server error log. Be judicious with the use of the @with_log option because network and system administrators tend to dislike applications that are “chatty” in the system logs. However, if the error needs to be trapped by an alert, the error must first be written to the Windows Application log.

Note that raising system errors is not supported.

Messages can be replaced without deleting them first by using the @replace = ‘replace’ option.

The messages are customizable and different ones can be added for the same error number for multiple languages, based on a language_id value. (Note: English messages are language_id 1033.)

Raising Errors Using RAISERROR



Both PRINT and RAISERROR can be used to return information or warning messages to applications. RAISERROR allows applications to raise an error that could then be caught by the calling process.

RAISERROR

The ability to raise errors in T-SQL makes error handling in the application easier, because it is sent like any other system error. RAISERROR is used to:

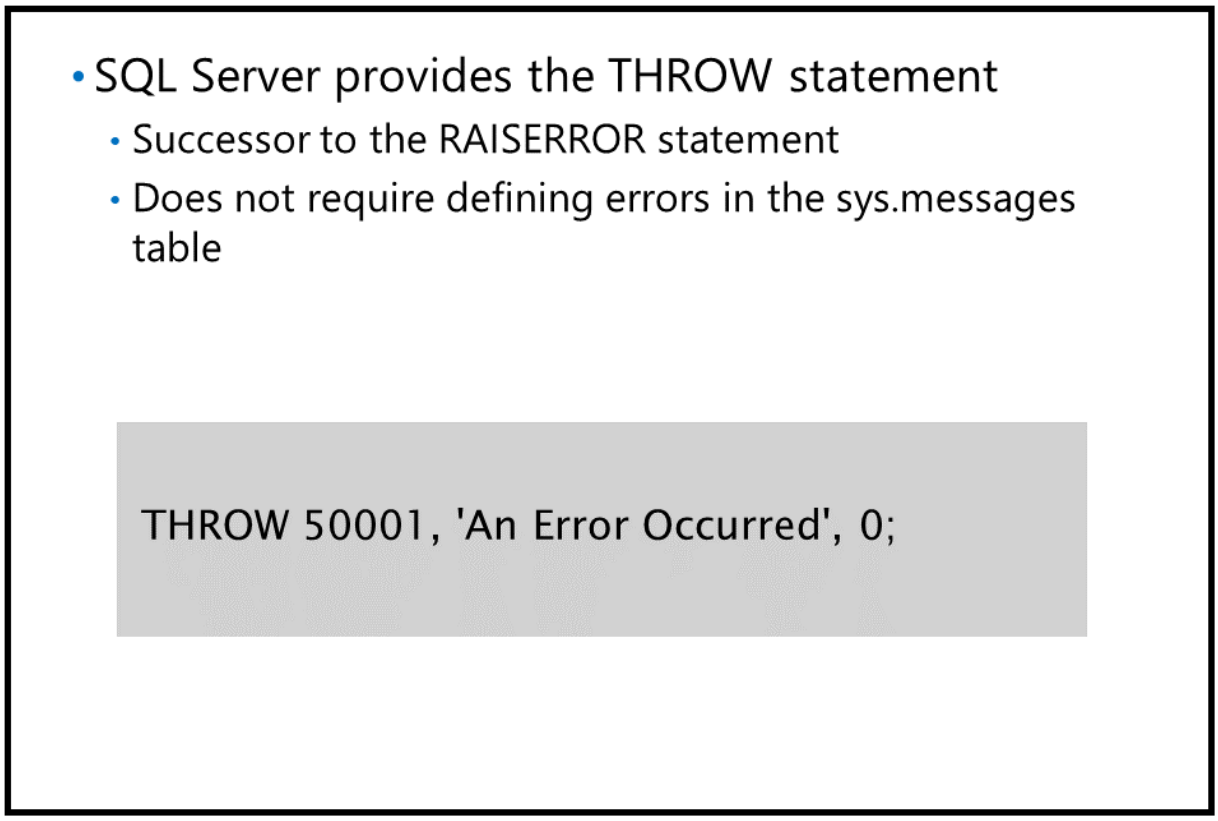
- Help troubleshoot T-SQL code.
- Check the values of data.
- Return messages that contain variable text.

Note that using a PRINT statement is similar to raising an error of severity 10, as shown in the sample on the slide.

Substitution Placeholders and Message Number

Note that, in the message shown in the example on the slide, %d is a placeholder for a number and %s is a placeholder for a string. Note also that a message number was not mentioned. When errors with message strings are raised using this syntax, they always have error number 50000.

Raising Errors Using THROW



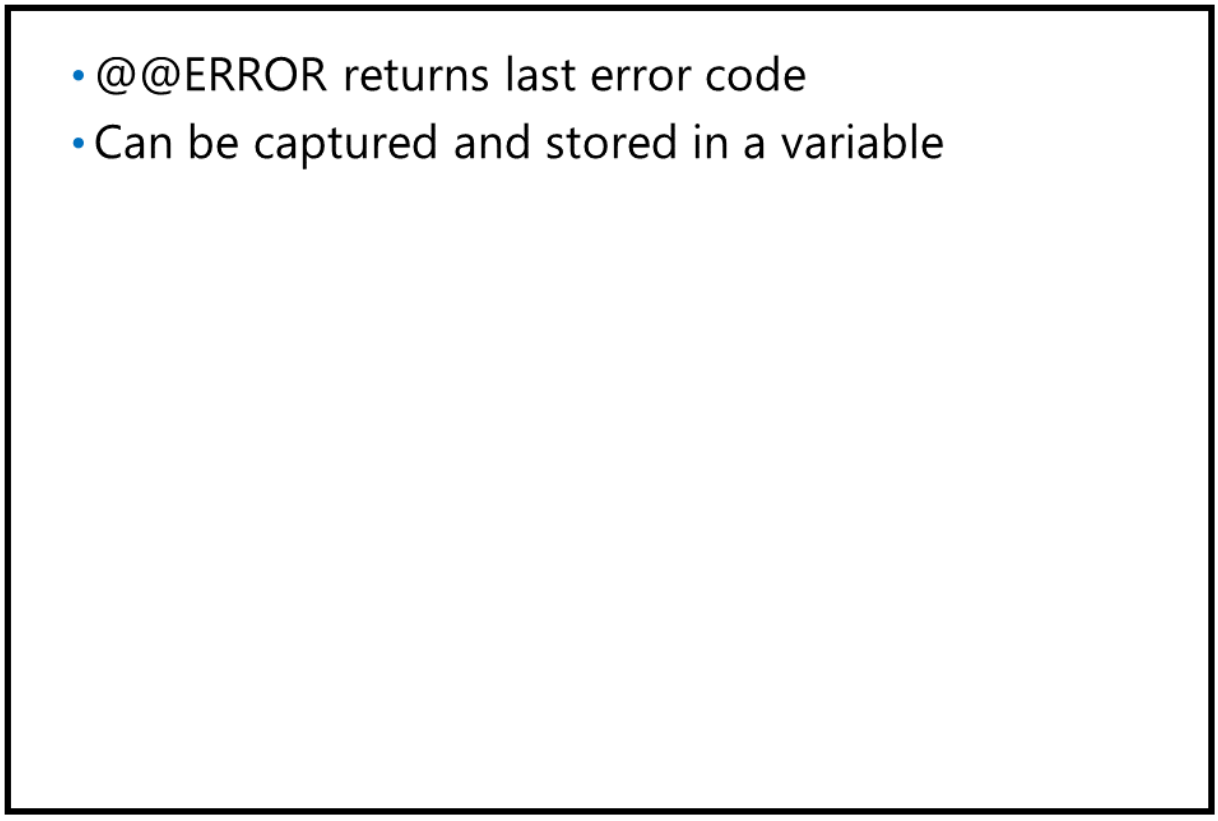
The THROW statement offers a simpler method of raising errors in code. Errors must have an error number of at least 50000.

THROW

THROW differs from RAISERROR in several ways:

- Errors raised by THROW are always severity 16.
- The messages returned by THROW are not related to any entries in sys.sysmessages.
- Errors raised by THROW only cause transaction abort when used in conjunction with SET XACT_ABORT ON and the session is terminated.

Using @@Error



Most traditional error handling code in SQL Server applications has been created using @@ERROR. Note that structured exception handling was introduced in SQL Server 2005 and provides a strong alternative to using @@ERROR. It will be discussed in the next lesson. A large amount of existing SQL Server error handling code is based on @@ERROR, so it is important to understand how to work with it.

@@ERROR

@@ERROR is a system variable that holds the error number of the last error that has occurred. One significant challenge with @@ERROR is that the value it holds is quickly reset as each additional statement is executed.

@@ERROR Example


```
RAISERROR(N'Message', 16, 1);
IF @@ERROR <> 0
PRINT 'Error=' + CAST(@@ERROR AS VARCHAR(8));
GO
```

You might expect that, when the code is executed, it would return

```
the error number in a printed string. However, when the code is
executed, it returns:
Msg 50000, Level 16, State 1, Line 1
Message
Error=0
```

Note that the error was raised but the message printed was “Error=0”. In the first line of the output, you can see that the error, as expected, was actually 50000, with a message passed to RAISERROR. This is because the IF statement that follows the RAISERROR statement was executed successfully and caused the @@ERROR value to be reset.

For this reason, when working with @@ERROR, it is important to capture the error number into a variable as soon as it is raised, and then continue processing with the variable.

Capturing @@ERROR Into a Variable

```
DECLARE @ErrorValue int;
RAISERROR(N'Message', 16, 1);
SET @ErrorValue = @@ERROR;
IF @ErrorValue <> 0
PRINT 'Error=' + CAST(@ErrorValue AS VARCHAR(8));
```

When this code is executed, it returns the following output:

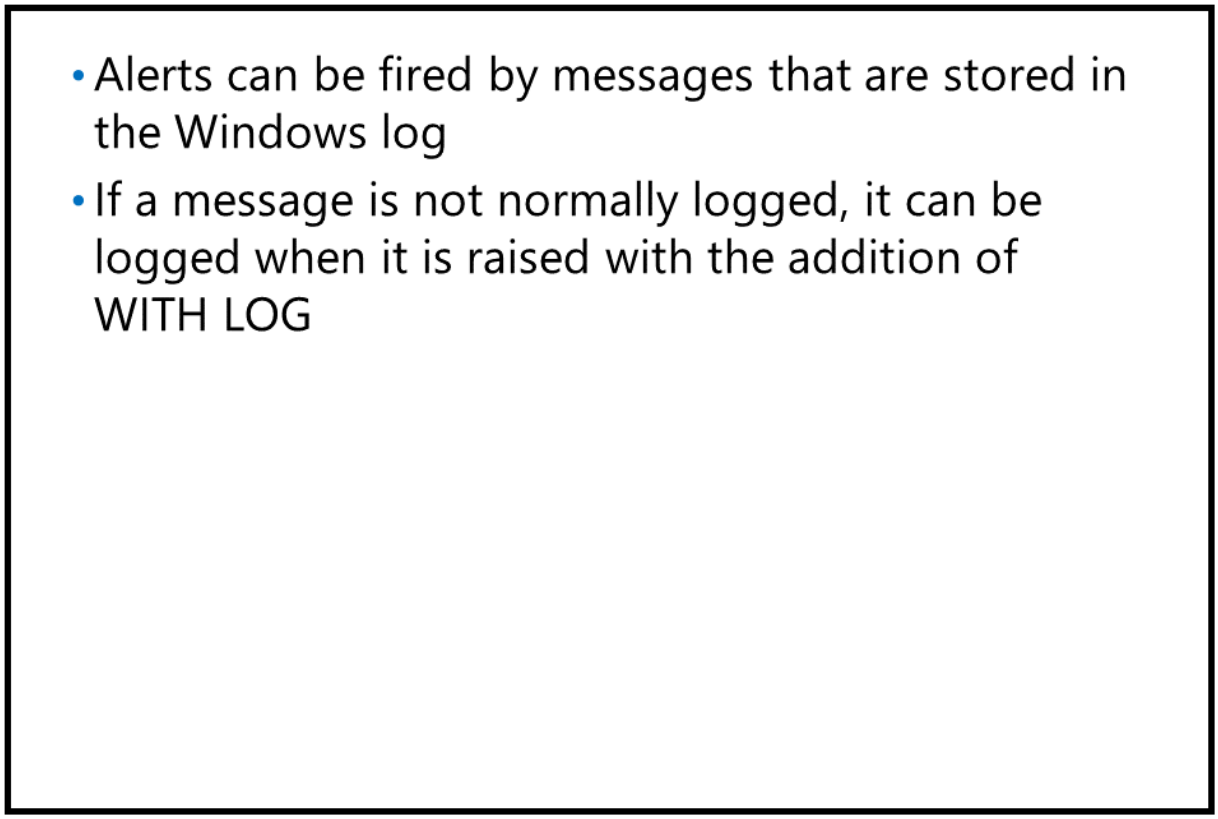
```
Msg 50000, Level 16, State 1, Line 2
Message
Error=50000
```

Note that the error number is correctly reported now.

Centralizing Error Handling

One other significant issue with using @@ERROR for error handling is that it is difficult to centralize within your T-SQL code. Error handling tends to end up scattered throughout the code. It would be possible to centralize error handling using @@ERROR to some extent, by using labels and GOTO statements. However, this would be frowned upon by most developers today as a poor coding practice.

Creating Alerts When Errors Occur



For certain categories of errors, administrators might create SQL Server alerts, because they wish to be notified as soon as these occur. This can even apply to user-defined error messages. For example, you might want to raise an alert whenever a transaction log fills. Alerting is commonly used to bring high severity errors (such as severity 19 or above) to the attention of administrators.

Raising Alerts

Alerts can be created for specific error messages. The alerting service works by registering itself as a callback service with the event logging service. This means that alerts only work on logged errors.

There are two ways to make an error raise an alert—you can use the WITH LOG option when raising the error or the message can be altered to make it logged by executing sp_altermessage. The WITH LOG option affects only the current statement. Using sp_altermessage changes the error behavior for all future use. Modifying system errors via sp_altermessage is only possible from SQL Server 2005 SP3 or SQL Server 2008 SP1 onwards.

Demonstration: Handling Errors Using T-SQL

In this demonstration, you will see how to handle errors.

Demonstration Steps

Handle Errors

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod17\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. At the command prompt, type **y**, and then press Enter.
5. When the script completes, press any key to continue.
6. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
7. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod17\Demo** folder.
8. Open the **11 - Demonstration A.sql** script file.
9. Select the code under the comment **Step 1**, and then click **Execute**.
10. Select the code under the comment **Step 2**, and then click **Execute**.
11. Select the code under the comment **--Capture @@ERROR into a variable**, and then click **Execute**.
12. Select the code under the comment **--Create a custom error message**, and

then click **Execute**.

13. Select the code under the comment **--Use a custom error message**, and then click **Execute**.

14. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Discovery

You are writing some error handling in a T-SQL script. If a problem arises, you want to raise an error with a severity of 20. Should you use RAISERROR or THROW for this error handling?

Show solution Reset

Lesson 2: Implementing Structured Exception Handling

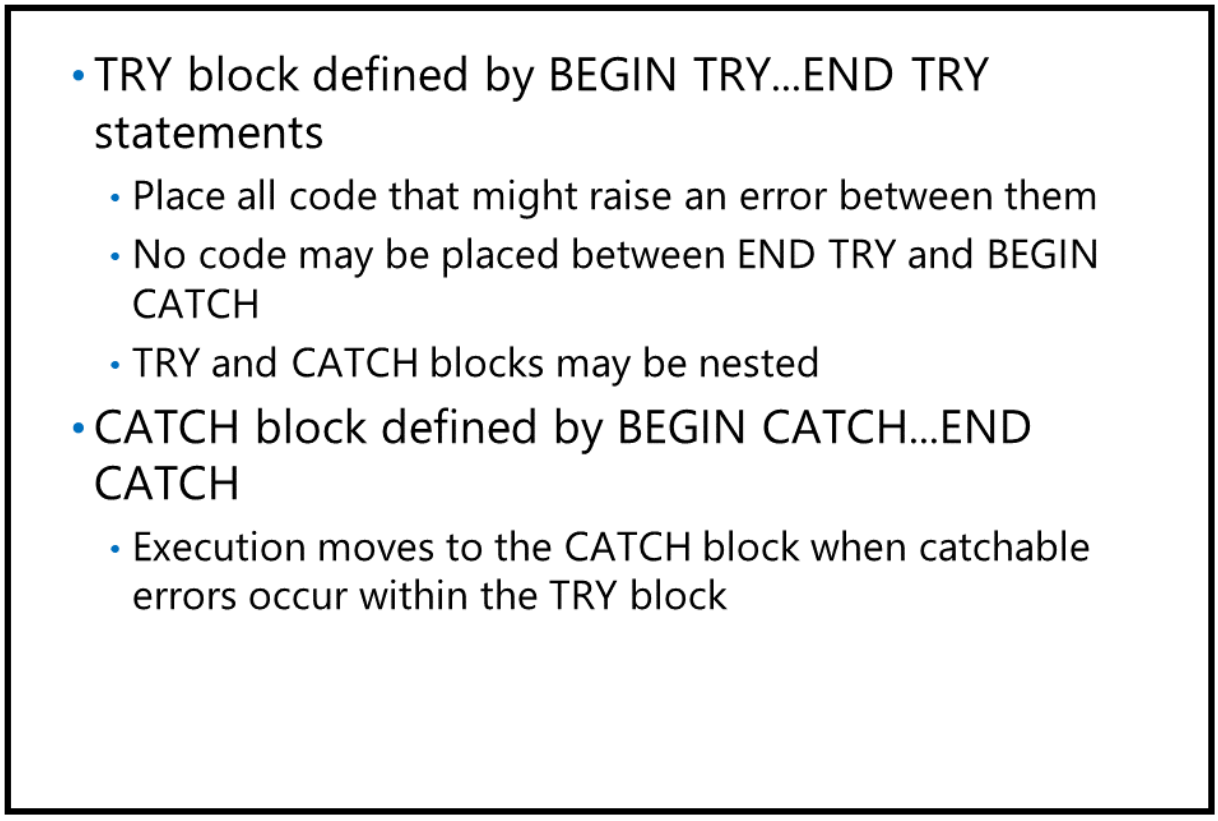
Now you have an understanding of the nature of errors and basic error handling in T-SQL, it is time to look at a more advanced form of error handling. Structured exception handling was introduced in SQL Server 2005. You will see how to use it and evaluate its benefits and limitations.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain TRY CATCH block programming.
- Describe the role of error handling functions.
- Describe catchable versus noncatchable errors.
- Explain how TRY CATCH relates to transactions.
- Explain how errors in managed code are surfaced.

TRY/CATCH Block Programming



Structured exception handling has been part of high level languages for some time, after being introduced to the T-SQL language by SQL Server 2005.

TRY/CATCH Block Programming

Structured exception handling is more powerful than error handling based on the @@ERROR system variable. It allows you to prevent code from being littered with error handling code and to centralize that error handling code.

Centralization of error handling code also means you can focus more on the purpose of the code rather than the error handling it contains.

TRY Block and CATCH Block

When using structured exception handling, code that might raise an error is placed within a TRY block. TRY blocks are enclosed by BEGIN TRY and END TRY statements.

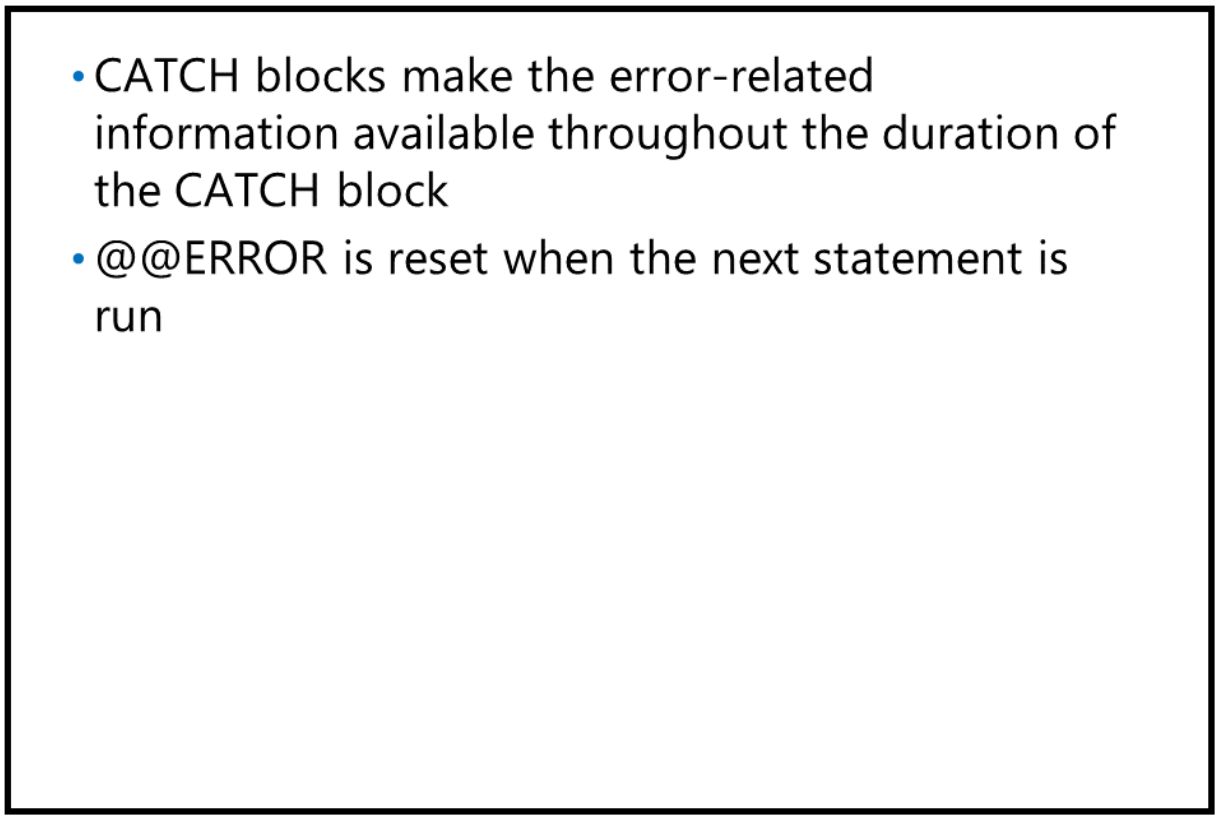
Should a catchable error occur (most errors can be caught), execution control moves to the CATCH block. The CATCH block is a series of T-SQL statements enclosed by BEGIN CATCH and END CATCH statements.

Note that, while BEGIN CATCH and END TRY are separate statements, the BEGIN CATCH must immediately follow the END TRY.

Current Limitations

High level languages often offer a try/catch/finally construct, and are often used to release resources implicitly. There is no equivalent FINALLY block in T-SQL.

Error Handling Functions



CATCH blocks make the error-related information available throughout the duration of the CATCH block. This includes subscopes, such as stored procedures, run from within the CATCH block.

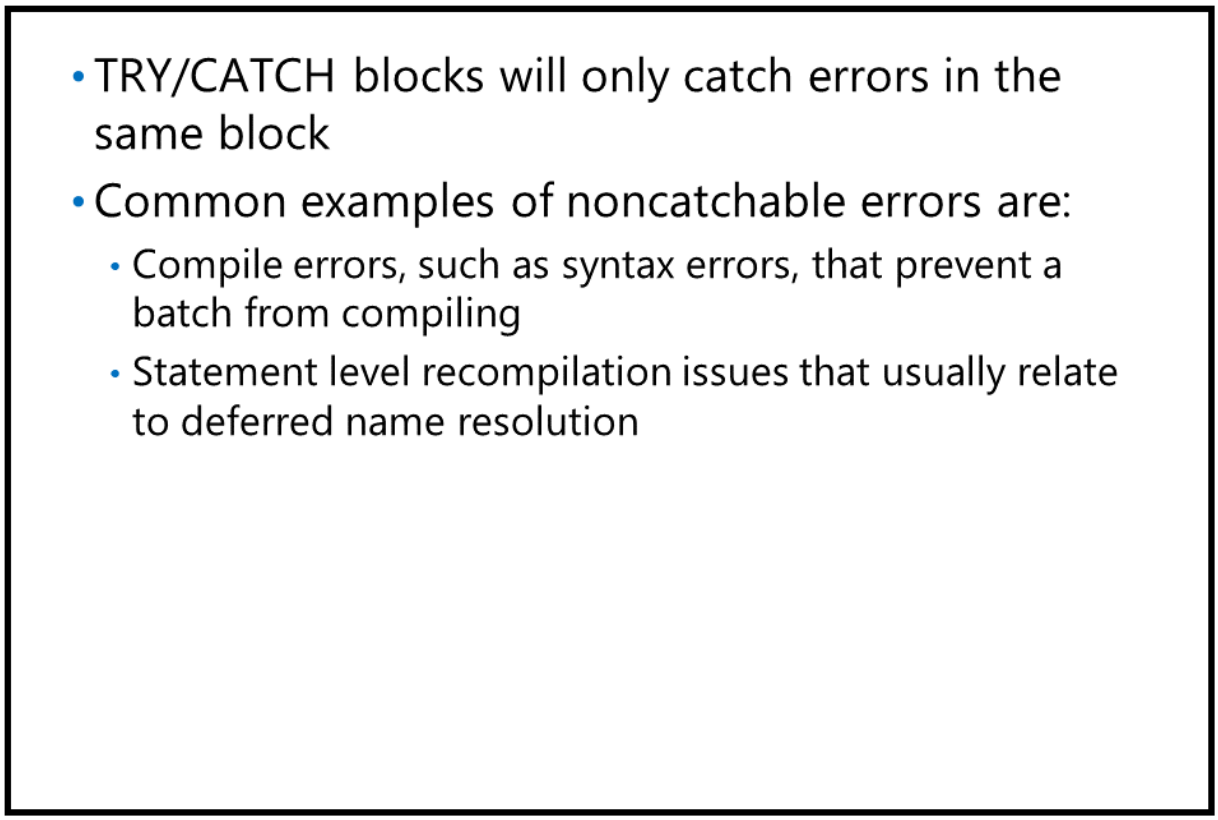
Error Handling Functions

You should recall that, when programming with @@ERROR, the value held by the @@ERROR system variable was reset as soon as the next statement was executed.

Another key advantage of structured exception handling in T-SQL is that a series of error handling functions has been provided and these retain their values throughout the CATCH block. Separate functions provide each property of an error that has been raised.

This means you can write generic error handling stored procedures that can still access the error-related information.

Catchable vs. Noncatchable Errors



It is important to realize that, while TRY/CATCH blocks allow you to catch a much wider range of errors than you could with @@ERROR, you cannot catch every type.

Catchable vs. Noncatchable Errors

Not all errors can be caught by TRY/CATCH blocks within the same scope where the TRY/CATCH block exists. Often, errors that cannot be caught in the same scope can be caught in a surrounding scope. For example, you might not be able to catch an

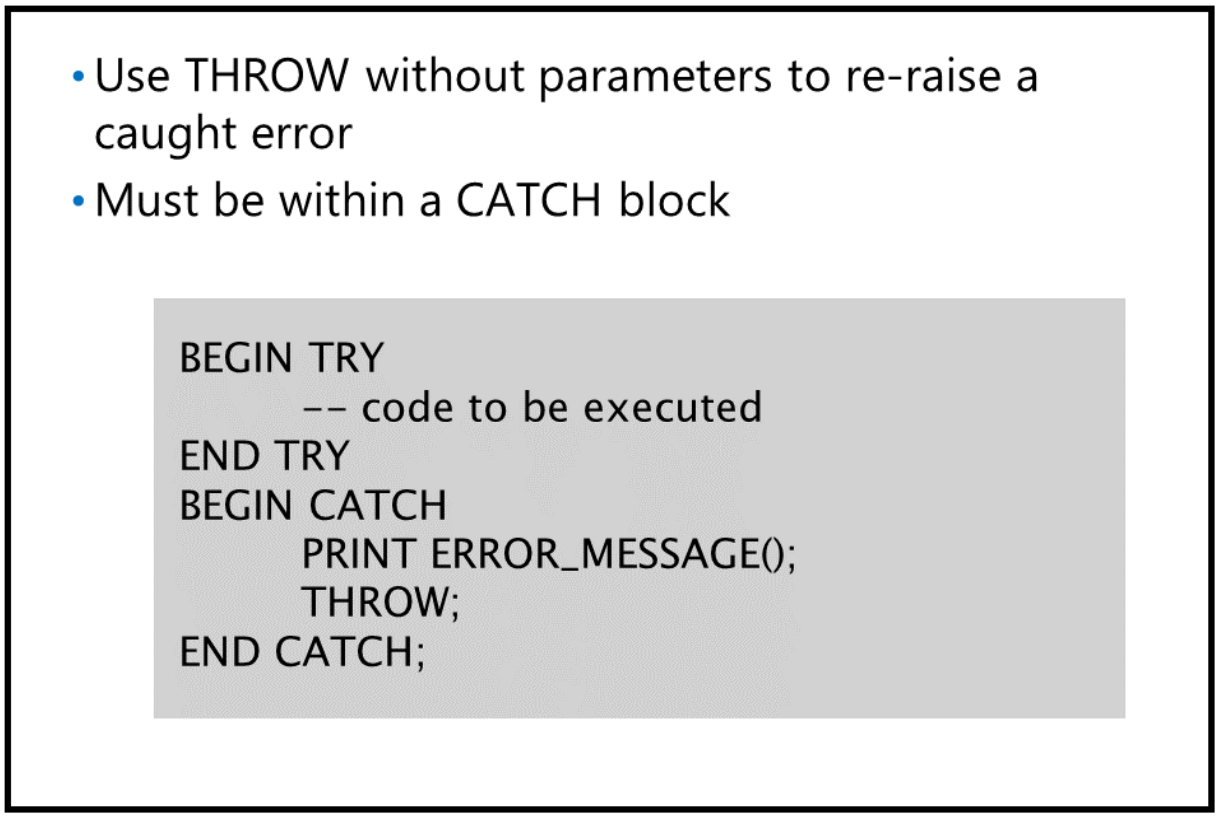
error within the stored procedure that contains the TRY/CATCH block. However, you are likely to catch that error in a TRY/CATCH block in the code that called the stored procedure where the error occurred.

Common Noncatchable Errors

Common examples of noncatchable errors are:

- Compile errors, such as syntax errors, that prevent a batch from compiling.
- Statement level recompilation issues that usually relate to deferred name resolution. For example, you could create a stored procedure that refers to an unknown table. An error is only thrown when the procedure tries to resolve the name of the table to an objectid.

Rethrowing Errors Using THROW

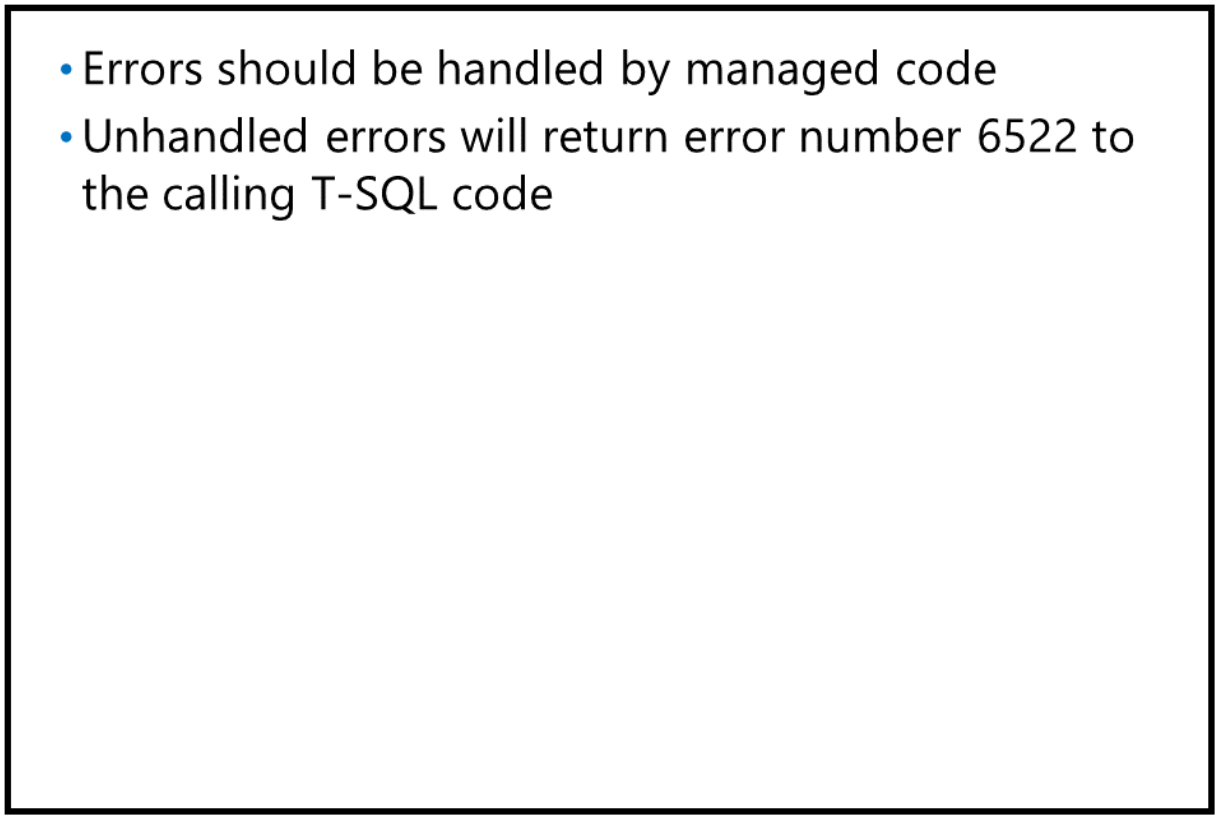


If the THROW statement is used in a CATCH block without any parameters, it will rethrow the error that caused the code to enter the CATCH block. You can use this technique to implement error logging in the database by catching errors and logging

their details, and then throwing the original error to the client application, so that it can be handled there.

In some earlier versions of SQL Server, there was no method to throw a system error. While THROW cannot specify a system error to raise, when THROW is used without parameters in a CATCH block, it will re-raise both system and user errors.

Errors in Managed Code



SQL CLR integration allows for the execution of managed code within SQL Server. High level .NET languages, such as C# and VB, have detailed exception handling available to them. Errors can be caught using standard .NET try/catch/finally blocks.

Errors in Managed Code

In general, you might wish to catch errors within managed code as much as possible.

It is important to realize, though, that any errors not handled in the managed code are passed back to the calling T-SQL code. Whenever any error that occurs in managed code is returned to SQL Server, it will appear to be a 6522 error. Errors can be nested and that particular error will be wrapping the real cause of the error.

Another rare but possible cause of errors in managed code would be that the code could execute a RAISERROR T-SQL statement via a SqlCommand object.

Demonstration: Using a TRY/CATCH Block

In this demonstration, you will see how to use a TRY/CATCH block.

Demonstration Steps

Use a TRY/CATCH Block

1. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Close SQL Server Management Studio without saving any files.

Check Your Knowledge

Discovery

You have the following T-SQL script:

```
BEGIN TRY
  INSERT INTO HumanResources.PossibleSkills(SkillName, Category)
  VALUES ('Database Administration', 'IT Professional');
END TRY
DECLARE @prefix AS NVARCHAR(50) = 'There has been an error: ';
BEGIN CATCH
  PRINT @prefix + ERROR_MESSAGE();
  THROW;
END CATCH;
GO
```

The code will not compile and execute. What should you do to troubleshoot this code?

Show solution Reset

Lab: Implementing Error Handling

Scenario

As a junior database developer for Adventure Works, you will be creating stored procedures using corporate databases stored in SQL Server 2012. To create more robust procedures, you will be implementing error handling in your code.

Objectives

After completing this lab, you will be able to:

- Redirect errors with TRY/CATCH.
- Use THROW to pass an error message to a client.

Lab Setup

Estimated Time: 30 minutes

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Redirecting Errors with TRY/CATCH

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a Basic TRY/CATCH Construct
3. Display an Error Number and an Error Message
4. Add Conditional Logic to a CATCH Block
5. Execute a Stored Procedure in the CATCH Block

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Result: After this exercise, you should be able to capture and handle errors using a TRY/CATCH construct.

Exercise 2: Using THROW to Pass an Error Message Back to a Client

---

Scenario

You will practice how to pass an error message using the THROW statement, and how to send custom error messages.

The main tasks for this exercise are as follows:

1. Rethrow the Existing Error Back to a Client
2. Add an Error Handling Routine
3. Add a Different Error Handling Routine
4. Remove the Stored Procedure

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Detailed Steps ▼

Result: After this exercise, you should know how to throw an error to pass messages back to a client.

Module Review and Takeaways

In this module, you have learned how to:

- Implement T-SQL error handling.
- Implement structured exception handling.

Review Question(s)

Check Your Knowledge

Discovery

Which error types cannot by caught by structured exception handling?

Show solution Reset

Check Your Knowledge

Discovery

Can TRY/CATCH blocks be nested?

Show solution Reset

Check Your Knowledge

Discovery

How can you use THROW outside of a CATCH block?

Show solution Reset