Intermediate Stored Procedures 1

# Calling Procedures

We have already learned how to call stored procedures using the EXECUTE statement and passing parameter values one by one in order. We have also learned that default values exist for parameters, but not how to use them.

Today we’re going to learn how to use optional parameters and a new way to call procedures. Let’s start with a new procedure to use the Orders and OrderLines tables we created in TestDB. NOTE: If you are missing these tables, the script is available in “Intro to Stored Procedures”.

USE TestDB;

GO

CREATE PROCEDURE dbo.CreateOrder (

@CustomerName NVARCHAR(50),

@OrderID INT OUTPUT,

@CustomerAddr NVARCHAR(100) = 'Same as mailing address',

@OrderDate DATE = NULL

)

AS

BEGIN;

SET NOCOUNT ON;

INSERT INTO dbo.Orders ( CustomerName, CustomerAddr, OrderDate )

VALUES ( @CustomerName, @CustomerAddr, ISNULL(@OrderDate, GETDATE()) );

SET @OrderID = SCOPE\_IDENTITY();

END;

GO

This procedure will create a new entry in the orders table. We have two optional parameters, @CustomerAddr and @OrderDate, which we’ve created defaults for. We’re also catching the identity that is created for this record and returning it to the caller.

There are two important things to note in this procedure.

First, we are using NULL instead of GETDATE() as a default for @OrderDate. This is because parameter defaults must be a constant or NULL. Unlike default constraints, they cannot be any expression.

Second, we are using a new system function called SCOPE\_IDENTITY(). This returns the last identity value that was created in the current scope. When we insert into the Orders table, a new identity is created for OrderID. If we called another procedure that also created a new identity, it would not affect SCOPE\_IDENTITY() in this procedure, because it is in a different scope.

**Parameter Defaults**

Default values used for parameters must be constants. This may include NULL.

**SCOPE\_IDENTITY()**

Returns the last identity value created in the current scope.

## Optional Parameters

If we want to use the defaults for the optional parameters, we can do this by just omitting them in the EXECUTE statement. While we are at it, let’s capture the OrderID that is being created and double-check our new row.

DECLARE @OrderID INT;

EXECUTE dbo.CreateOrder 'Ralph', @OrderID OUTPUT;

SELECT \* FROM dbo.Orders WHERE OrderID = @OrderID;

Does that match what you expected?

If we want to include a value for @CustomerAddr, that’s easy enough, because it’s the next value after @OrderID. And if we want to include all values, that’s easy as well, we just keep adding values in order:

DECLARE @OrderID INT;

-- Include address, but not date

EXECUTE dbo.CreateOrder 'Kim Ball', @OrderID OUTPUT, '123 Dimension Drive';

SELECT \* FROM dbo.Orders WHERE OrderID = @OrderID;

-- Include all parameters

EXECUTE dbo.CreateOrder 'Fully Loaded', @OrderID OUTPUT, '999 Max Ave', '2018-01-01'

SELECT \* FROM dbo.Orders WHERE OrderID = @OrderID;

What if we want to include the order date, but use the default for the address?

What happens if we try to omit just address?

DECLARE @OrderID INT;

-- Include date, but not address

EXECUTE dbo.CreateOrder 'Kim Ball', @OrderID OUTPUT, , '2018-01-01';

SELECT \* FROM dbo.Orders WHERE OrderID = @OrderID;

If we are passing values to parameters by order, we can only exclude columns from the end. When passing by order, we cannot use a default for a parameter if we are supplying values for parameters that follow it. So how do we do this?

## Parameters By Name

We also have the option of explicitly stating which parameters we are supplying values for. In our example above, where we want to omit the address but specify the date, we could do this using the following:

DECLARE @OrderID INT;

-- Include date, but not address

EXECUTE dbo.CreateOrder

@CustomerName = 'Kim Ball',

@OrderID = @OrderID OUTPUT,

@OrderDate = '2018-01-01';

SELECT \* FROM dbo.Orders WHERE OrderID = @OrderID;

Notice that it is able to differentiate between the parameter @OrderID and the variable @OrderID. This is because the parameter is always to the left of the “=” and the value is always to the right.

There are many reasons you might choose to call a procedure with named parameters:

* The procedure has many parameters, so keeping them straight may be difficult, particularly if you have many repeating values.
* You are using complex expressions as values, which may be difficult to read.
* You want to use a default for one parameter, but specify values for parameters that follow it (like we did above).
* You want to change the order of the parameters. When using named parameters, you don’t have to specify them in order.

NOTE: It is possible to use a combination of providing value in order and by name, but combining approaches is confusing and is generally a bad practice.

**Best Practice**

When creating procedures with optional parameters, the optional parameters should be at the end of the parameter list. This makes it easier to call the procedure by specifying values by order if desired.

# Controlling Flow

## IF

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/if-else-transact-sql>

IF works very similar in T-SQL as it does in other languages, but there is one very important difference. An IF statement only considers the next line of code to be part of the if statement. This means that if we want to include multiple lines of code, we need to use the BEGIN; statement. To see this in action, try the following:

IF 1 = 0

PRINT 'I should not print';

PRINT 'Neither should I';

What returns? Is that what you expected? Let’s fix this and see the syntax for addition conditions while we’re at it:

IF 1 = 0

BEGIN;

PRINT 'I should not print';

PRINT 'Neither should I';

END;

ELSE IF 1 = 1

BEGIN;

PRINT 'I should print.';

END;

ELSE

BEGIN;

PRINT 'Else';

END;

What prints this time? Is this what you expected?

**Best Practice**

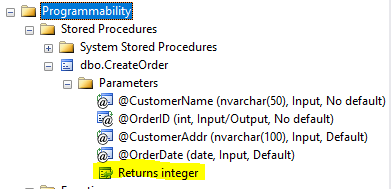
Always use BEGIN and END after an IF or ELSE statement, even if you are only including a single line of code afterwards. This helps make your code more consistent and easier to read. It also helps guard against future syntax errors.

## A Word on Loops and Cursors

It is possible to create loops in stored procedures. In practice it is somewhat rare because it results in RBAR (row by agonizing row) processing. As we learned earlier in the course, it is far better to work with data sets than individual rows.

If you find you’re working on a problem where a loop is absolutely necessary, you can review Microsoft’s documentation on while loops here: <https://docs.microsoft.com/en-us/sql/t-sql/language-elements/while-transact-sql>

## RETURN

Looking at our CreateOrder procedure in the object explorer, we may see something that looks a little confusing:  


Where did this come from? Aren’t return values only for procedures?

Yes and no. All procedures can return an integer. This is used to return the return code. This value is optional and defaults to 0 (i.e. completed normally). In practice is not frequently used. However, this does mean that just like in functions, we can use RETURN statements to short-circuit a stored procedure.

In our system, perhaps we are able to silently ignore orders from the past; they’re not considered errors, but they can be omitted from the database. We could do this by adding a guard clause to short circuit our stored procedure:

ALTER PROCEDURE dbo.CreateOrder (

@CustomerName NVARCHAR(50),

@OrderID INT OUTPUT,

@CustomerAddr NVARCHAR(100) = 'Same as mailing address',

@OrderDate DATE = NULL

)

AS

BEGIN;

SET NOCOUNT ON;

IF @OrderDate < GETDATE()

BEGIN;

RETURN;

END;

INSERT INTO dbo.Orders ( CustomerName, CustomerAddr, OrderDate )

VALUES ( @CustomerName, @CustomerAddr, ISNULL(@OrderDate, GETDATE()) );

SET @OrderID = SCOPE\_IDENTITY();

END;

GO

Test this out by trying to insert an old order. Check the maximum OrderID in Orders to be sure nothing is being added:

DECLARE @OrderID INT;

SELECT MAX(OrderID) FROM dbo.Orders;

EXECUTE dbo.CreateOrder 'Old Order', @OrderID OUTPUT, '999 Old Order Lane', '2000-01-01'

SELECT MAX(OrderID) FROM dbo.Orders;

If you’re unclear about what is happening here, or if you want to explore it more in depth, add some print statements before, inside, and after the IF statement to see what is returned.

# Handling Errors

## THROW

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/throw-transact-sql>

More commonly, when we short circuit a stored procedure using guard clauses, it is to identify errors and raise them to the caller. We can do that by using THROW.

Let’s try the example above, but this time let’s treat old order as an error instead of something that we can safely ignore.

ALTER PROCEDURE dbo.CreateOrder (

@CustomerName NVARCHAR(50),

@OrderID INT OUTPUT,

@CustomerAddr NVARCHAR(100) = 'Same as mailing address',

@OrderDate DATE = NULL

)

AS

BEGIN;

SET NOCOUNT ON;

IF @OrderDate < GETDATE()

BEGIN;

THROW 50001, '@OrderDate is in the past', 1;

END;

INSERT INTO dbo.Orders ( CustomerName, CustomerAddr, OrderDate )

VALUES ( @CustomerName, @CustomerAddr, ISNULL(@OrderDate, GETDATE()) );

SET @OrderID = SCOPE\_IDENTITY();

END;

GO

In addition to the error message, we specify an error number (50001 in our example) and state (1 in our example). User defined error numbers in SQL Server must be equal to or larger than 50000, but are otherwise up to the user to define. State can be any number between 0 and 255.

We can see this error in action by calling the procedure again:

DECLARE @OrderID INT;

SELECT MAX(OrderID) FROM dbo.Orders;

EXECUTE dbo.CreateOrder 'Old Order', @OrderID OUTPUT, '999 Old Order Lane', '2000-01-01'

SELECT MAX(OrderID) FROM dbo.Orders;

## XACT ABORT and TRY CATCH

<https://docs.microsoft.com/en-us/sql/t-sql/statements/set-xact-abort-transact-sql>

In stored procedures, it is possible to catch errors using TRY CATCH ([link](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/try-catch-transact-sql)). However, it is more typical to raise an error to the caller and allow them to handle it appropriately. In the database, when an error is encountered our primary concern is data integrity.

Let’s create some simple procedures to illustrate this issue.

### Attempt 1 – Encountering an Error

First, let’s create a procedure that will insert into Orders, intentionally try a bad insert, then try another insert. What do you think might happen? How will our proc handle this error?

CREATE PROCEDURE dbo.ErrorProblems

AS

BEGIN;

SET NOCOUNT ON;

INSERT INTO dbo.Orders ( CustomerName ) VALUES ('Before Bad Insert NO Err handle');

-- FK ERROR

INSERT INTO dbo.OrderLines ( OrderID ) VALUES ( -1 );

INSERT INTO dbo.Orders ( CustomerName ) VALUES ('After Bad Insert No Err Handle');

END;

GO

-- Clear out dbo.Orders, then execute proc – OR keep all orders-see which kept

-- DELETE FROM dbo.Orders -- Note IF there are Orderlines records already

-- Added, then MUST delete those FIRST…

EXECUTE dbo.ErrorProblems;

SELECT \* FROM dbo.Orders;

Did that produce what you expected?

### Attempt 2 – Adding a Transaction

That didn’t seem right, but maybe the problem is that were not using a transaction, so all three statements were executed independently. Let’s create a transaction to help alleviate this.

ALTER PROCEDURE dbo.ErrorProblems

AS

BEGIN;

SET NOCOUNT ON;

BEGIN TRANSACTION;

INSERT INTO dbo.Orders ( CustomerName ) VALUES ('Before Bad Insert Trans Committ');

-- FK ERROR

INSERT INTO dbo.OrderLines ( OrderID ) VALUES ( -1 );

INSERT INTO dbo.Orders ( CustomerName ) VALUES ('After Bad Insert- Trans Committ');

COMMIT TRANSACTION;

END;

GO

Attempt to execute this. Is it better, worse, or the same? Is it what you expected?

### Attempt 3 – TRY CATCH

We can catch this in a TRY CATCH. That way we can roll back on error if we need to.

ALTER PROCEDURE dbo.ErrorProblems

AS

BEGIN;

SET NOCOUNT ON;

BEGIN TRANSACTION;

BEGIN TRY;

INSERT INTO dbo.Orders ( CustomerName ) VALUES ('Before Bad Insert- TryCatch');

-- FK ERROR

INSERT INTO dbo.OrderLines ( OrderID ) VALUES ( -1 );

INSERT INTO dbo.Orders ( CustomerName ) VALUES ('After Bad Insert - TryCatch');

COMMIT TRANSACTION;

END TRY

BEGIN CATCH;

ROLLBACK TRANSACTION;

END CATCH;

END;

GO

Does that produce the rows you were expecting?  
This does handle the error and roll back the transaction as we might expect, but what message is displayed in the messages? The issue now is that our caller has no idea anything bad happened. A hidden error is worse than an unhandled error because we still have the error, but now we don’t know about it and can’t do anything to handle it.

### Attempt 4 – Re-raising the Error

To get the error to be raised to the caller, we need to re-raise it in the catch block. The simplest way to do that is to use THROW with no parameters. This can only be done in a catch block and it simply raises the error that was caught in the TRY block.

ALTER PROCEDURE dbo.ErrorProblems

AS

BEGIN;

SET NOCOUNT ON;

BEGIN TRANSACTION;

BEGIN TRY;

INSERT INTO dbo.Orders ( CustomerName ) VALUES ('Before Bad Insert- TryCatchthrow attempt');

-- FK ERROR

INSERT INTO dbo.OrderLines ( OrderID ) VALUES ( -1 );

INSERT INTO dbo.Orders ( CustomerName ) VALUES ('After Bad Insert- TryCatchthrow attempt');

COMMIT TRANSACTION;

END TRY

BEGIN CATCH;

ROLLBACK TRANSACTION;

THROW;

END CATCH;

END;

GO

Try to execute this. Does it behave the way you would expect? Does it properly raise the error?

Even though this behaves the way we would like, it seems more complex than it needs to be, especially since we aren’t logging any messages about the error and we’re just re-raising it. **Also/most importantly, there are some errors (such as timeouts) that aren’t handled by TRY CATCH, how would we deal with those consistently?**

### Attempt 5 – XACT\_ABORT

SQL Server has a setting that tells the server to automatically end processing and rollback any open transactions when an error is encountered. It is the simplest and most consistent way to raise errors to the caller and leave your database in a consistent state.

Let’s try creating the proc with XACT\_ABORT on.

ALTER PROCEDURE dbo.ErrorProblems

AS

BEGIN;

SET NOCOUNT ON;

SET XACT\_ABORT ON;

BEGIN TRANSACTION;

INSERT INTO dbo.Orders ( CustomerName ) VALUES ('Before Bad Insert – XACT\_ABORT!');

-- FK ERROR

INSERT INTO dbo.OrderLines ( OrderID ) VALUES ( -1 );

INSERT INTO dbo.Orders ( CustomerName ) VALUES ('After Bad Insert – XACT\_ABORT !');

COMMIT TRANSACTION;

END;

GO

When we run this, we see that we’re getting the results we expect (no records updated – test it out by doing Select \*’s … ) and we’re being notified of the error.

**Best Practice**

**Enable XACT\_ABORT**

XACT\_ABORT should be enabled on all of your procedures.

# Homework

## Understand

1. Create a stored procedure called “CreateOrderLine” to write to our OrderLines table. It should require an OrderID and UnitPrice. It should also accept optional parameters for ItemDesc and Quantity. The default for ItemDesc should be ‘Unknown Item’ and the Quantity should be 1.
2. Create a procedure called “ProcessOrder” that will insert a new order with a new order line. It should do this by calling the CreateOrder proc we created in this lesson and the CreateOrderLine proc you created above. It should have parameters for CustomerName, CustomerAddr, OrderDate, ItemDesc, Quantity, and UnitPrice. The OrderID that is required in CreateOrderLine should come from CreateOrder (hint: you will need to use a variable to store that value).
3. Alter “ProcessOrder” to assign parameter values by name when calling CreateOrder and CreateOrderLine (if you haven’t already).
4. Alter “ProcessOrder” to throw an error if the quantity is less than 1.