Triggers

Triggers are a type of stored procedure attached to a table or view and are executed only when the contents of a table are changed. They can be used to enforce almost any business rule, and are especially important for dealing with situations that are too complex for a check constraint to handle. Triggers can be used when one of the following needs to be done:

- Perform cross-database referential integrity.
- Check intra-table inter-row constraints.
- Check intertable constraints.
- Introduce desired side effects to your modification queries (`INSERT`, `UPDATE`, `DELETE`).

In a trigger you can run almost every T-SQL command, except for the following:

`ALTER DATABASE`	`CREATE DATABASE`	`DROP DATABASE`
`RESTORE LOG`	`RECONFIGURE`	`RESTORE DATABASE`

There are two different models of triggers that I'll show you how to use:

- `AFTER`: The trigger fires *after* the command has affected the table, though not on views. `AFTER` triggers are used for building enhanced business rule handlers and generally putting any kind of logging mechanisms into them. You may have an unlimited number of `AFTER` triggers that fire on `INSERT`, `UPDATE`, and `DELETE` or any combination of them. Even if you have an `INSTEAD OF` trigger, you may still have as many `AFTER` triggers as you want, because they can all be combined into a single trigger.

- `INSTEAD OF`: The trigger operates *instead of* the command (`INSERT`, `UPDATE`, or `DELETE`) affecting the table or view. In this way, you can do whatever you want with the data, either modifying it as sent, or putting it into another place. You can have only a single `INSERT`, `UPDATE`, and `DELETE` trigger of this type per table. They can be combined into one and have a single trigger that fires for all three operations.

The following section will describe some common techniques for employing triggers in your data protection code.

Coding Triggers

The following contains the basic trigger syntax:

```
CREATE TRIGGER <tableName$triggerType$usageType>
ON <tableName> | <viewName>
AFTER | INSTEAD OF
[DELETE][,][INSERT][,][UPDATE]
AS
<sqlStatements>
```

The following sections look at several parts of the `CREATE TRIGGER` statement that need further explanation:

- `CREATE TRIGGER <tableName$triggerType$usageType>`

  The name of the trigger isn't tremendously important, as it won't be explicitly called from code. I generally like to give them names that resemble all of the other objects you've seen previously, though as with every object you've built, the name is only a personal choice.

- `INSTEAD OF | AFTER`: Tells the trigger when to fire, as outlined previously.

- `[DELETE][,][INSERT][,][UPDATE]`: You can specify one or more of these to tell your trigger when to fire, for example, specifying

```
AFTER DELETE, INSERT
```

  would give you a trigger that fired on `DELETE` and `INSERT`, but not `UPDATE`. Specifying

```
INSTEAD OF INSERT, UPDATE
```

  would give you a trigger that fired instead of `INSERT`s and `UPDATE`s, but `DELETE` would work as normal.

- `<sqlStatements>`: Any SQL Statements, as long as they don't include the statements listed earlier in this appendix.

The rest of this section will be devoted to which statements to put in the trigger to handle certain situations. You'll look at

- Accessing modified row information
- Determining which columns have been modified
- Writing validations for many rows at a time
- Having multiple triggers for the same action (`INSERT`, `UPDATE`, `DELETE`)
- Error handling

Accessing Modified Rows

The best part about coding triggers is that no matter how many rows are being modified, you get access to them all. You access these values via two tables that let you see the state of your data before and after your modification. These tables are called the `DELETED` and `INSERTED` tables, and are tables in memory that only exist for the lifetime of the trigger, being inaccessible outside of its scope.

The `INSERTED` table will be filled in during an `INSERT` or `UPDATE` operation, and the `DELETED` table will be filled in during an `UPDATE` or `DELETE` operation. `INSERT` and `DELETE` operations are obvious, but the SQL Server's handling of an `UPDATE` is less so. All `UPDATE` operations in SQL Server are logically divided into a `DELETE` and an `INSERT` operation, removing the rows that are going to be modified, then inserting them into the new format.

In the following example, you'll see how to build a trigger to show the contents of the `INSERTED` and `DELETED` tables after an `UPDATE`:

```
--build a trigger that returns as a result set
--inserted and deleted tables for demonstration purposes
--only. For real triggers, this is a bad idea
CREATE TRIGGER artist$afterUpdate$showInsertedAndDeleted
ON artist
AFTER UPDATE -- fires after the update has occurred
AS

SELECT 'contents of inserted' -- informational output
SELECT *
FROM INSERTED

SELECT 'contents of deleted' -- informational output
SELECT *
FROM DELETED
GO
```

Then you run an UPDATE on the table that will set all names to uppercase as an example of the differences between the INSERTED and DELETED table contents.

```
UPDATE artist
SET name = UPPER(name)
```

This returns the following four result sets:

```
---------------------------------
contents of inserted

artistId name …
---------------------------------
1 THE BEATLES …
2 THE WHO …
```

```
---------------------------------
contents of deleted

artistId name …
---------------------------------
1 the beatles …
2 the who …
```

Determining Which Columns Have Been Modified

There are two functions that you can use in your triggers to determine which columns have been involved in either an INSERT or UPDATE operation, but note that there is no equivalent for a DELETE operation. You'll use this to avoid checking for data that hasn't been affected by the operation, and hence enhance the performance of your triggers.

Specifically, the functions get information from

- The columns clause of an INSERT statement:

```
INSERT tablename (<column1>,<column2>,…,<columnN>)
```

- The SET clauses of the UPDATE statement:

```
UPDATE tablename
SET <column1> = value1,
<column2> = value2,
...
<columnN> = valueN
```

The first method of seeing that the column has been updated is the UPDATE(<columnName>) function. This will return a Boolean value that tells you whether or not the column has been referenced in the modification statement. For example, the following would check to see if the name or description columns had been modified:

```
IF update(name) OR update(description)
BEGIN
   <SQL statements>
END
```

As an example, you'll create the following table:

```
CREATE TABLE columnUpdatedTest
(
   columnUpdatedTestId int IDENTITY,   --bitmask value 1
   column1 int,                        --bitmask value 2
   column2 int,                        --bitmask value 4
   column3 int                         --bitmask value 8
)
```

You can then define the following trigger and attach it to the table:

```
CREATE TRIGGER testIt
ON columnUpdatedTest
AFTER INSERT,UPDATE
AS
IF update(columnUpdatedTestId)
BEGIN
   SELECT 'columnUpdatedTestId modified'
END
IF update(column1)
BEGIN
   SELECT 'column1 modified'
END
IF update(column2)
BEGIN
   SELECT 'column2 modified'
END
IF update(column3)
BEGIN
   SELECT 'column3 modified'
END
GO
```

Now when you create a new record as follows:

```
INSERT columnUpdatedTest(column1,column2, column3)
VALUES (1,1,1)
```

the following rows are returned:

```
----------------------------------------
columnUpdatedTestId modified

------------------------
column1 modified

------------------------
column2 modified

------------------------
column3 modified
```

If you update a single column like this:

```
UPDATE columnUpdatedTest
SET column2 = 2
WHERE columnUpdatedTestId = 1
```

the same trigger now informs you of the following:

```
------------------------
column2 modified
```

The other method is worth a mention, but isn't really all that practical. It's the `columns_updated()` function. By executing the `columns_updated()` function, you get back a hexadecimal integer with a bitmask of the columns in the table. For instance, in the table that you created, you have the following columns:

```
columnUpdatedTestId     --bitmask value 1
column1                 --bitmask value 2
column2                 --bitmask value 4
column3                 --bitmask value 8
```

The bitmask for every column in the table would be 1 + 2 + 4 + 8 = 15.

So in the preceding trigger, you can use `columns_updated()`in place of `updated()` in order to get a bitmask of the columns that have been updated. For instance, if you insert all columns in an `INSERT` statement as follows:

```
INSERT columnUpdatedTest (column1,column2,column3)
VALUES (1,1,1)
```

and you execute the `columns_updated()` function in an insert trigger, you get the following bitmask back:

```
Ox0F
```

which is equal to 15, or the bitmask for the entire table as you displayed earlier. If you use the following code:

```
UPDATE columnUpdatedTest
SET column2 = 2
WHERE columnUpdatedTestId = 1
```

you then get the bitmask for this single column that you've changed, and not the one in the `WHERE` clause.

```
0x04
```

So you can test to see which columns have been updated by using bitwise operations, by using code such as

```
If columns_updated() & 4 = 4    -- the third column in the table has been
                                -- modified

If columns_updated() & 15 = 15 -- the first four columns have been changed.
```

So what is wrong with using the `columns_updated()` function? The code that is written using this function must change if your column order changes. This makes your code unpredictable, as you have to check the trigger code if you add a column to any position in the table other than the end. The `update()` function is by far the most preferable way of checking modified columns. The `columns_updated()` function is shown here for completeness.

Writing Multirow Validations

Because triggers fire only once for a multirow operation, statements within triggers have to be coded with multiple rows in mind. It's fairly easy to write triggers that only seek to correct a single row; however, every piece of code written will typically need to look for rows that don't meet the criteria, instead of those that do. Unless you want to force users into entering one row at a time, you have to code triggers in a way that recognizes that more than one row in the table is being modified. In the next example, you build a trigger with two very similar validations, and although both work for single rows, only one will catch multirow failures.

```
CREATE TRIGGER artist$afterUpdate$demoMultiRow
ON artist
AFTER INSERT, UPDATE --fires after the insert and update has occurred
AS
IF NOT EXISTS ( SELECT *
   FROM INSERTED
   WHERE name IN ('the who','the beatles','jethro tull')
   )
   BEGIN
      SELECT 'Invalid artist validation 1'
   END

IF EXISTS ( SELECT *
   FROM INSERTED
   WHERE name NOT IN ('the who','the beatles','jethro tull')
   )
   BEGIN
      SELECT 'Invalid artist validation 2'
   END
GO
```

So you've written your trigger, and you want to test it. Because you're going to write a query that will succeed (and you want to be able to repeatedly execute the statement without deleting new data every time), you surround your INSERTs with BEGIN TRANSACTION and ROLLBACK TRANSACTION statements, in order to allow you to test the statement while preventing any of your data modifications from actually being saved to the table. The trigger I've written is for demonstration only, and in the sections written on the AFTER and INSTEAD OF triggers, you'll look at possible ways to actually return errors from triggers.

```
BEGIN TRANSACTION
   INSERT INTO artist (name, defaultFl, catalogNumberMask)
   VALUES ('ROLLING STONES',0, '%')

   ROLLBACK TRANSACTION --undo our test rows
```

Here it returns the following:

```
---------------------------------
Invalid artist validation 1

---------------------------------
Invalid artist validation 2
```

because in this validation:

```
IF NOT EXISTS ( SELECT *
   FROM INSERTED
   WHERE name IN ('the who','the beatles','jethro tull'))
```

and this one:

```
IF EXISTS ( SELECT *
   FROM INSERTED
   WHERE name NOT IN ('the who','the beatles','jethro tull'))
```

'ROLLING STONES' doesn't fall within the boundaries. In the first, the name isn't in the group, so no set of rows exists, which passes the NOT EXISTS test. In the second, the name isn't in the group, so all rows are returned by the query, and a row does exist. Next you try a multirow test.

The best way to do a multirow test is to use a UNION:

```
SELECT 'ROLLING STONES',0,'%'
UNION
SELECT 'JETHRO TULL',0,'%'
```

which returns

```
-------------------------------------------
JETHRO TULL               0 %
ROLLING STONES            0 %
```

By carrying out such unions, in this case one valid and another invalid, you can test the multirow capabilities of your triggers.

```
BEGIN TRANSACTION

INSERT INTO artist (name, defaultFl, catalogNumberMask)
SELECT 'ROLLING STONES',0,'%'
UNION
SELECT 'JETHRO TULL',0,'%'

ROLLBACK TRANSACTION --undo our test rows
```

This time only one of your validations fail.

```
-----------------------------------
Invalid artist – validation 2
```

Let's have to look at why this has happened. In the following validation, the trigger won't send you a message:

```
IF NOT EXISTS ( select *
   FROM INSERTED
   WHERE name IN ('the who','the beatles','jethro tull'))
```

Because you have one row that meets the criteria (the one with the `jethro tull` entry), you have one row where `INSERTED.name` is in your test group. So a single row *is* returned, and a row does exist. This is a pretty common problem when writing triggers. Even after 10 years of trigger writing, I must test my triggers to make sure I haven't fallen into this trap. It's of course best to test each snippet of code like this to make sure that it can handle whatever data is thrown at it.

Note also that you'll drop the `artist$afterUpdate$demoMultiRow` trigger from your database as it's no longer needed.

```
Drop trigger artist$afterUpdate$demoMultiRow
```

Having Multiple Triggers for the Same Action

One cool SQL Server 2000 feature is the ability to have multiple `AFTER` triggers to take care of different data validation operations. This allows you to build triggers that are very specific in nature, as in the following options:

- **Nested trigger option:** A server global setting accessed via

  ```
  sp_configure 'nested triggers', 1 | 0 – where 1 = ON, 0 = OFF
  ```

  If a cascading operation is carried out, then the triggers on all the participating tables are fired. There is a limit of 32 nested trigger operations. Note that you're able to build infinite loop triggers that will cascade the operation to one table, and if any modifies the initial table, then it will start the process over again. Nested triggers are set at the server level.

- **Recursive trigger option:** A database-specific setting accessed via

  ```
  sp_dboption '<dbName>','recursive triggers', 'TRUE' | 'FALSE'
  ```

  This causes triggers to be refired when the trigger executes modifications on the same table as it's attached to. This requires that the nested trigger option is set and is set at the database level.

In this example, you'll learn how the nested and recursive trigger settings work. First, you create two very simple tables into which you can insert a single value as follows:

```
CREATE TABLE tableA
(
   field varchar(40) NOT NULL
)
CREATE TABLE tableB
(
   field varchar(40) not NULL
)
GO
```

and then two very simple triggers each inserting the same values into the other table like this:

```
CREATE TRIGGER tableA$afterUpdate$demoNestRecurse
ON tableA
AFTER INSERT
AS

--tell the user that the trigger has fired
SELECT 'table a insert trigger'

--insert the values into tableB
INSERT INTO tableB (field)
SELECT field
FROM INSERTED
GO

CREATE TRIGGER tableB$afterUpdate$demoNestRecurse
ON tableB
AFTER INSERT
AS

--tell the user that the trigger has fired
SELECT 'table b insert trigger'

--insert the values into tableB
INSERT INTO tableA (field)
SELECT field
FROM INSERTED
GO
```

Now that you have your setup, you'll turn off nested triggers for the server.

```
EXEC sp_configure 'nested triggers', 0 -- (1 = on, 0 = off)
RECONFIGURE      --required for immediate acting server configuration
```

Next you insert a row into `tableA` as follows:

```
INSERT INTO tableA (field)
VALUES ('test value')
```

and the only trigger that fires is the `tableA` trigger.

```
---------------------------
table a insert trigger
```

If you turn on nested triggers, and make certain that recursive triggers is turned off for your test database as follows:

```
EXEC sp_configure 'nested triggers', 1 -- where 1 = ON, 0 = OFF
RECONFIGURE
EXEC sp_dboption 'bookTest','recursive triggers','FALSE'
```

then you can now execute the same code on `tableA`.

```
INSERT INTO tableA (field)
VALUES ('test value')
```

But this time, you get something very different:

```
---------------------------
table a insert trigger

---------------------------
table b insert trigger

<rows removed>
---------------------------
table a insert trigger

---------------------------
table b insert trigger

Server: Msg 217, Level 16, State 1, Procedure tableB$afterUpdate$demoNestRecurse, Line 7
Maximum stored procedure, function or trigger nesting level exceeded (limit 32).
```

This is pretty much what you would probably have expected. A recursive trigger situation only occurs when the trigger acts upon the same table that it's set on. To illustrate this, just change the `tableA` trigger so that it inserts a row into `tableA` and not `tableB`.

```
ALTER TRIGGER tableA$afterUpdate$demoNestRecurse
ON tableA
AFTER INSERT
AS

--tell the user that the trigger has fired
SELECT 'table a insert trigger'

--insert the values into tableB
INSERT INTO tableA (field)
SELECT field
FROM INSERTED
GO
```

When you enter your single row as follows:

```
INSERT INTO tableA (field)
VALUES ('test value')
```

the statement fails and rolls back the transaction. This illustrates that recursive triggers can only be used if you very carefully code the triggers so they can't run more than 32 times.

```
-----------------------------
table a insert trigger

-----------------------------
table a insert trigger

……..……
```

Server: Msg 217, Level 16, State 1, Procedure tableA$afterUpdate$demoNestRecurse, Line 7
Maximum stored procedure, function or trigger nesting level exceeded (limit 32).

Error Handling

There are two ways to implement error handling using a trigger. Either the error handling code is placed in the trigger or it's contained within the client calling code.

Handling Errors Within the Trigger

You saw in the section on constraints that you have to check for the error status after every SQL statement that modifies data. Sadly, handling errors in this manner doesn't help you that much when you get errors from triggers, for two reasons:

- **Executing a rollback transaction in a trigger rolls back the transaction and halts all execution of proceeding statements:** The crux of this problem is that there is absolutely no way to tell all triggers that the current operation failed and that only the single operation should be rolled back. Say you have several triggers working consecutively and you have set nested transactions on. If the final transaction fails, then all the transactions are rolled back regardless of the success of the previous transactions. It isn't possible to just roll back the failed transaction and continue with the successful ones.
- **If a severe error is detected, the entire transaction automatically rolls back:** The definition of a severe error is quite loose. It does, however, include constraint errors, so any data modifications you do in a trigger that violate another table's constraints not only fail, but once the trigger has finished, the transaction is rolled back and no further commands are executed.

There are three generic ways in which to handle errors in your triggers, which can be summarized as follows:

- Checking that the values in the table are proper
- Cascading an operation
- Calling a stored procedure

Checking That the Values in the Table Are Proper

The basic method you'll use to check the validity of a condition is as follows:

```
IF EXISTS (      SELECT *
                 FROM INSERTED
                 WHERE <condition is true for invalid values>)
BEGIN
     --always raise another error to tell the caller where they are
     RAISERROR 50000 'Invalid data exists in inserted table'
     ROLLBACK TRANSACTION
     RETURN  --note the trigger doesn't end until you stop it, as
             --the batch is canceled AFTER the trigger finishes when ROLLBACK
             --is called
END
```

Cascading an Operation

In the next piece of code, I show you the way to cascade an update of one value to another table:

```
DECLARE @errorHold int,     --used to hold the error after modification
 @msg varchar(8000)         --statement for formatting messages

UPDATE anotherTable
SET <column> = <value>
FROM anotherTable
JOIN INSERTED
ON table.tableId = INSERTED.tableId

SET @errorHold = @@error    --get error value after update

IF @errorHold <> 0          --something went awry
BEGIN
SET @msg = 'Error ' + CAST(@errorHold) + 'occurred cascading update to table'
--always raise another error to tell the caller where they are
RAISERROR 50000 @msg
ROLLBACK TRANSACTION
RETURN      --note the trigger doesn't end until you stop it, as
            --the batch is canceled AFTER the trigger finishes
END
```

Executing a Stored Procedure

The next piece of code demonstrates the method of calling a stored procedure from your trigger:

```
DECLARE @errorHold int,     --used to hold the error after modification
                            --statement holds return value from stored
@returnValue int,           --procedure for formatting messages
@msg varchar(8000)

EXEC @retval = <procedureName> <parameters>
SET @errorHold = @@error    --get error value after update

IF @errorHold <> 0 or @retval < 0     --something went awry
BEGIN
SET @msg = CASE WHEN @errorHold <> 0
THEN 'Error ' + CAST(@errorHold) + 'occurred cascading update to table'
ELSE 'Return value ' + CAST(@retval as varchar(10))
                        + ' returned from procedure'
END
--always raise another error to tell the caller where they are
RAISERROR 50000 @msg
ROLLBACK TRANSACTION
RETURN --note the trigger doesn't end until you stop it, as
       --the batch is canceled AFTER the trigger
END
```

As you've seen, the error messages that you get back from constraints are neither very readable nor very meaningful. However, when you build triggers, you write your own error messages that can be made as readable and meaningful as you want. You could go further and put every database predicate into your triggers (including unique constraints), but this would require more coding and wouldn't be that advantageous to you; triggers are inherently slower than constraints and aren't the best method of protecting your data except in circumstances when constraints can't be used.

Handling Errors in the Calling Code

The best part about error handling and triggers is that because the entire sequence of trigger commands is rolled back when an error occurs, there is no need to worry about external error handling other than monitoring the error status when protecting against constraint errors.

AFTER Triggers

As I've previously stated, **AFTER triggers** are so called because they fire after the data is actually in the table. This lets you verify that whatever the statement did to the data in the table doesn't violate your specifications.

> *In previous versions of SQL Server, this was the only kind of trigger available, but instead of the keyword* `AFTER`*, the* `FOR` *keyword was used. In SQL Server 2000, the latter keyword should no longer be used as a trigger type, but it's retained for backward compatibility.*

Let's look at a simple example of an `AFTER` trigger by considering the status of the `INSERTED` table and the contents of the database table while the trigger is running.

```
CREATE TRIGGER artist$trInsertDemo ON artist
AFTER INSERT
AS

SELECT 'contents of inserted'
SELECT artistId, name --just PK and name so it fits on a line in the book
FROM INSERTED

SELECT 'contents of artist'
SELECT artistId, name
FROM artist
GO
```

You then execute an `INSERT`, setting the name value as follows:

```
INSERT INTO artist (name)
VALUES ('jethro tull')
```

which returns the following result sets:

```
-------------------------------
contents of INSERTED

artistId name
-----------------------------------------------------------------------------------------------
27 jethro tull

----------------------
contents of artist

artistId name
-----------------------------------------------------------------------------------------------
27  jethro tull
1   the beatles
2   the who
```

So you see that the row in the `INSERTED` table contains the data for the inserted row (naturally), and you see from the results of the second query that the table already contains the value that you passed in. (If you did this with an `INSTEAD OF` trigger, the second query wouldn't return the new record.)

Implementing Multiple AFTER Triggers

Let's now extend the preceding multiple trigger by altering the coding of the `tableA$afterUpdate$demoNestRecurse` trigger, so as to insert the name of the procedure into `tableB.field`, enabling you to see the different triggers that are firing.

```
ALTER TRIGGER tableA$afterUpdate$demoNestRecurse
ON tableA
AFTER INSERT
AS
SELECT 'table a insert trigger'

INSERT INTO tableB (field)
VALUES (object_name(@@procid))      --@@procid gives the object_id of the
                                    --of the current executing procedure
GO
```

You then create two more triggers that are identical to the first (though with different names).

```
CREATE TRIGGER tableA$afterUpdate$demoNestRecurse2
ON tableA
AFTER INSERT
AS
SELECT 'table a insert trigger2'

INSERT INTO tableB (field)
VALUES (object_name(@@procid))
GO

CREATE TRIGGER tableA$afterUpdate$demoNestRecurse3
ON tableA
AFTER INSERT
AS
SELECT 'table a insert trigger3'

INSERT INTO tableB (field)
VALUES (object_name(@@procid))
GO
```

Next, you clear out the tables as follows:

```
--truncate table deletes the rows in the table very fast with a single log
--entry, plus it resets the identity values
TRUNCATE TABLE tableA
TRUNCATE TABLE tableB
```

and you insert a single row like this:

```
INSERT INTO tableA
VALUES ('Test Value')
```

which returns the following three rows:

```
------------------------------
table a insert trigger

------------------------------
table a insert trigger2

------------------------------
table a insert trigger3
```

To show that the tables are as you expect, you execute `SELECT` statements from the tables.

```
SELECT * FROM tableA
SELECT * FROM tableB
```

You then see that `tableA` has the single row you expected and `tableB` has a row resulting from each of the three `tableA` triggers that were executed.

```
column
------------------------------

Test Value

column
------------------------------------------------------------
tableA$afterUpdate$demoNestRecurse
tableA$afterUpdate$demoNestRecurse2
tableA$afterUpdate$demoNestRecurse3
```

Finally, you want to show how to reorder the execution order of the triggers.

```
EXEC sp_settriggerorder
                @triggerName = 'tableA$afterUpdate$demoNestRecurse3',
                @order = 'first',
                @stmttype = 'insert'
```

Note `@order` may be '`first`', '`last`', or '`none`' to fire in an unspecific location. You can only set the first and last trigger to fire, whereas any other triggers fire in a random (as far as you're concerned) order. The `@stmttype` value may be '`insert`', '`update`', or '`delete`'.

You can now execute the commands to truncate the table, insert a row, and finally select the rows from the table as follows:

```
TRUNCATE TABLE tableA
TRUNCATE TABLE tableB

INSERT INTO tableA
VALUES ('Test Value')

SELECT * FROM tableA
SELECT * FROM tableB
```

and in this case you return a different order of triggers.

```
column
------------------------------

Test Value

column
------------------------------------------------------------
tableA$afterUpdate$demoNestRecurse3
tableA$afterUpdate$demoNestRecurse
tableA$afterUpdate$demoNestRecurse2
```

As a final note in this section, I would urge caution in having multiple triggers for the same action on the same table. It's often simpler to build a single trigger that handles all trigger needs. You'll normally design your systems to use one `INSTEAD OF` trigger and one `AFTER` trigger for each operation when necessary. You'll also seldom use a multipurpose trigger that will fire on insert and update. Basically a good principle is to keep it as simple as possible to avoid any ordering issues.

INSTEAD OF Triggers

`INSTEAD OF` triggers only fire in place of data manipulation operations, and before check constraints have been validated. You may only have a single `INSTEAD OF` trigger for each operation on each table. Note that although this trigger will be typically referred to as a "before trigger," it's more than that. When coding an `INSTEAD OF` trigger, you have to redo the operation in the trigger. For example, in the case of an `INSTEAD OF DELETE` trigger, the trigger contains a delete just like the operation that caused the trigger to fire in the first place.

The following example shows a trigger that will return the state of the `INSERTED` table and then the physical table:

```
CREATE TRIGGER artist$insteadOfInsert ON artist
INSTEAD OF INSERT
AS
--output the contents of the inserted table
SELECT 'contents of inserted'
SELECT * FROM inserted

--output the contents of the physical table
SELECT 'contents of artist'
SELECT * FROM artist
GO
```

Next, you execute an `INSERT` against the table like this:

```
INSERT INTO artist (name)
VALUES ('John Tesh')
```

and you get the following results:

```
------------------------------
contents of inserted

artistId    name            ...
-----------------------------------------------------------------------------------------------------------------------
0           John Tesh       ...

------------------------------
contents of artist

artistId    name            ...
-----------------------------------------------------------------------------------------------------------------------
1           THE BEATLES     ...
2           THE WHO         ...
```

This is exactly as you would expect; the trigger fired before the columns were in the table, which explains why John Tesh's record doesn't appeared in the second result set. However, if you look at the contents of the table now that the trigger has executed:

```
SELECT artistId, name
FROM artist
```

You get back

```
artistId    name ...
-----------------------------------------------------------
1           THE BEATLES ...
2           THE WHO ...
```

This illustrates one difficulty with `INSTEAD OF` triggers, namely that any insertions are added to the `INSERTED` or `DELETED` table and not the database. This means that you have to manually insert the data into the table to make the trigger work properly. Therefore, `INSTEAD OF` triggers force you to completely reinvent the way you code data manipulation operations in order to make the most of them. In the rest of this section, you'll see some other problems with `INSTEAD OF` triggers, followed by a couple of very important uses that make them a very desirable addition to the SQL Server toolkit.

Caveats

The problems associated with `INSTEAD OF` triggers are as follows:

- The operation that the user has performed has to be repeated in the trigger.
- It isn't easy to do range checks on multiple rows.
- Cascading operations ignore `INSTEAD OF` triggers.

I'll discuss each of the difficulties in a bit more depth in the following sections.

The Operation That the User Has Performed Has to Be Repeated

In the following trigger, I show a simple demonstration of this:

```
CREATE TRIGGER artist$insteadOfInsert on artist
INSTEAD OF INSERT
AS

-- must mimic the operation we are doing “instead of”.
-- note that must supply a column list for the insert
-- and we cannot simply do a SELECT * FROM INSERTED because of the
-- identity column, which complains when you try to pass it a value
INSERT INTO artist(name, defaultFl, catalogNumberMask)
SELECT name, defaultFl, catalogNumberMask
FROM INSERTED
GO

CREATE TRIGGER artist$insteadOfDelete on artist
INSTEAD OF DELETE
AS

-- must mimic the operation we are doing “instead of”
DELETE FROM artist
FROM DELETED
   -- we always join the inserted and deleted tables
   -- to the real table via the primary key,
   JOIN ARTIST
   ON DELETED.artistId = artist.artistId

GO
```

Once you’ve created the triggers to operate instead of `INSERT` and `DELETE`, you test them as shown here:

```
INSERT INTO artist(name, defaultFl, catalogNumberMask)
VALUES ('raffi',0,'%')

SELECT * FROM artist WHERE name = 'raffi'
```

which returns

```
artistId    name ...
----------------------------------
14          raffi ...
```

Then you delete the row and execute a select to see if it exists as follows:

```
DELETE FROM artist
WHERE name = 'raffi'

SELECT * FROM artist WHERE name = 'raffi'
```

which produces no output as the delete has done its job.

Not Easy to Do Range Checks on Multiple Rows

In `AFTER` triggers, because the data has already been added to the table, it's easy to check to see if multirow constraints were being met. However, `INSTEAD OF` trigger data hasn't yet made it to the database. In the next two samples of code, you'll look at the two different methods required to validate that a table has only nonnegative balances for an `amount` field, in an `accountTransaction` table, where balances are grouped by `accountId`.

AFTER Trigger for Comparison

```
IF EXISTS ( SELECT accountId
   FROM accountTransaction
   JOIN (SELECT distinct accounted FROM INSERTED) AS accounts
      ON accountTransaction.accountId = accounts.accountId
      GROUP BY accountId
      HAVING sum(amount) > 0)
BEGIN
   RAISERROR 50000 'Negative balances exist as due to operation'
END
```

INSTEAD OF Trigger

```
IF EXISTS( SELECT accountId
   FROM         --union inserted values with the transactions for
                --any accounts that are affected by the operation
      ( SELECT accountTransactionId, accountId, amount
      FROM INSERTED
   UNION
   SELECT accountTransactionId, accountId, amount
   FROM accountTransaction
   JOIN (SELECT distinct accounted FROM INSERTED) AS accounts
      ON accountTransaction.accountId = accounts.accountId
      --remove any accountTransactionRecords that you are
      --updating, by removing the rows that are now "deleted"
      WHERE accountTransactionId NOT IN (
         SELECT accountTransactionId FROM DELETED )
      ) as accountValues
      GROUP BY accountId
      HAVING sum(amount) > 0 )
BEGIN
   RAISERROR 50000 'Negative balances exist as due to operation'
END
```

In the AFTER trigger, all that was required was to check the current state of the table, whereas in the INSTEAD OF trigger, you had to add the values from the INSERTED table to the values in the actual table. One interesting thing to note is that it wasn't just a simple query of the INSERTED table. You always have to consider all of the rows in the current table that relate to the rows in the INSERTED table, which requires a complex JOIN.

Cascading Operations Ignore INSTEAD OF Triggers

One of the more troubling problems with INSTEAD OF triggers is that cascading operations ignore them, and so using INSTEAD OF triggers in enforcing business rules can prove problematic. For example, if in a given table the primary key changes to a value that would fail the checks in the INSTEAD OF trigger, the change won't be prevented, because the trigger won't fire. These limitations mean that INSTEAD OF triggers should not be used to enforce update business rules that involve the primary key or any business rules that involve deletes. This won't prevent you from using them for other purposes, which is what you shall turn to now.

Summary

For the database system where the protection of the data is database centric, triggers are very important tools. You can use them to seamlessly and automatically perform complex validation and cascading actions when a user makes a change to a table, or even a view. Some example operations for which you'll likely use triggers are listed here:

- **Perform cross-database referential integrity:** This isn't possible in constraints.
- **Check intra-table inter-row constraints:** Because CHECK constraints are used to look at the current row, and it isn't straightforward to build functions that look at data in the table being updated. Triggers give you access to the before and after images of the rows that are modified by the statement.
- **Check inter-table constraints:** This can be done using CHECK constraints using user-defined functions, but it can be slower because it works one row at a time, rather than all rows at a time
- **Introduce desired side effects to your modification queries:** Triggers are the only automatic way you can introduce side effects (that is, changing the state of the system beyond the expected INSERT, UPDATE, or DELETE of one or more rows).

Triggers are employed in several situations in Chapter 12.