MICROSOFT
TRAINING
AND CERTIFICATION

Microsoft Official
Curriculum

Module 11: Implementing Triggers

Contents

Microsoft

Project Lead: Rich Rose
Instructional Designers: Rich Rose, Cheryl Hoople, Marilyn McGill
Instructional Software Design Engineers: Karl Dehmer, Carl Raebler, Rick Byham
Technical Lead: Karl Dehmer
Subject Matter Experts: Karl Dehmer, Carl Raebler, Rick Byham
Graphic Artist: Kirsten Larson (Independent Contractor)
Editing Manager: Lynette Skinner
Editor: Wendy Cleary
Copy Editor: Edward McKillop (S&T Consulting)
Production Manager: Miracle Davis
Production Coordinator: Jenny Boe
Production Support: Lori Walker (S&T Consulting)
Test Manager: Sid Benavente
Courseware Testing: TestingTesting123
Classroom Automation: Lorrin Smith-Bates
Creative Director, Media/Sim Services: David Mahlmann
Web Development Lead: Lisa Pease
CD Build Specialist: Julie Challenger
Online Support: David Myka (S&T Consulting)
Localization Manager: Rick Terek
Operations Coordinator: John Williams
Manufacturing Support: Laura King; Kathy Hershey
Lead Product Manager, Release Management: Bo Galford
Lead Product Manager, Data Base: Margo Crandall
Group Manager, Courseware Infrastructure: David Bramble
Group Product Manager, Content Development: Dean Murray
General Manager: Robert Stewart

Instructor Notes

Presentation:
45 Minutes

Lab:
30 Minutes

This module provides the student with a definition of what triggers are and how to create them. Triggers are a useful tool for database implementers who want certain actions to be performed whenever data in a specific table is inserted, updated, or deleted. They are an especially useful method for enforcing business rules and ensuring data integrity.

The following parts of a trigger are discussed:

- The statement that activates it (INSERT, UPDATE, or DELETE)
- The table that the trigger protects
- The action that the trigger takes when it is invoked

This section also discusses the uses of triggers and issues that you must consider when determining whether a trigger is the appropriate tool to accomplish a task.

The next section discusses the process of creating and dropping triggers and provides detailed information about the operation of each of the four types of triggers: INSERT, UPDATE, DELETE, and INSTEAD OF. This section also includes a discussion of how to alter triggers.

A discussion on working with triggers follows, including examples of nesting triggers and how triggers can be useful for enforcing data integrity, complex referential integrity, and business rules.

The module concludes with a list of recommended practices and performance considerations that implementers should consider when they create and work with triggers.

In the lab, students create triggers and test the effectiveness of these triggers.

After completing this module, students will be able to:

- Create a trigger.
- Drop a trigger.
- Alter a trigger.
- Evaluate the performance considerations that affect using triggers.

Materials and Preparation

This section provides the materials and preparation tasks that you need to teach this module.

Required Materials

To teach this module, you need the following materials:

- Microsoft® PowerPoint® file 2073A_11.ppt
- The C:\Moc\2073A\Demo\D11_Ex.sql example file, which contains all of the example scripts from the module, unless otherwise noted in the module.

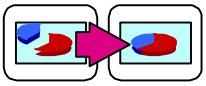
Preparation Tasks

To prepare for this module, you should:

- Read all of the materials for this module.
- Complete the lab.

Other Activities

This section provides procedures for implementing interactive activities to present or review information, such as games or role playing exercises.



Displaying the Animated PowerPoint Slides

All animated slides are identified with an icon of links on the lower left corner of the slide.

► To display the How an INSERT Trigger Works slide

This slide shows how an INSERT trigger inserts a row and how the **inserted** table is used.

1. Display the slide, which shows the **Order Details** table before any modification is done.
2. Display the next image, which shows that a row has been inserted into the table. The slide automatically continues to show that the inserted row is also written to the log, and is accessible to the trigger and users as the **inserted** table.
3. Display the next image, which shows the trigger definition script, and discuss the actions that the statements perform.
4. Display the last image, which summarizes how the trigger works.

► To display the How a DELETE Trigger Works slide

This slide shows how a DELETE trigger deletes a row and how the **deleted** table is used.

1. Display the slide, which shows the **Categories** table before any modification is done.
2. Display the next image, which shows that a row has been deleted from the table and is available as the **deleted** table.
3. Display the next image, which shows how the trigger definition script and the trigger act on the **Products** table.
4. Display the last image, which summarizes how the trigger works.

► To display the How an UPDATE Trigger Works slide

This slide shows how an UPDATE trigger modifies a row by using the **inserted** and **deleted** tables.

1. The slide displays an UPDATE statement to be executed on the **Employees** table before any modification is done. The slide also displays the rows that will be stored in the **deleted** and **inserted** tables.
2. Display the next image, which shows the trigger definition script and discusses the actions that result from the trigger.
3. Display the last image, which summarizes the order of events when a trigger is executed.

► To display the How an INSTEAD OF Trigger Works slide

This slide shows how an INSTEAD OF trigger can be used to redirect and update to a view.

1. The slide displays a view that is based on two tables. Discuss how these tables partition the customers by country.
2. Display the next image that updates the view. As the presentation proceeds, discuss how the update is redirected and the original insert is not attempted against the Customers view.
3. Display the last image, which summarizes the key points regarding INSTEAD OF triggers.

► To display the Enforcing Business Rules slide

This slide shows how a trigger enforces the "products with outstanding orders cannot be deleted" business rule.

1. The slide initially displays the DELETE trigger that enforces the "products with outstanding orders cannot be deleted" business rule. It also displays the **Products** table, from which **ProductID 2** is being removed.
2. Display the next image, in which the product with the **ProductID of 2** is removed from the **Products** table.
3. Display the next image, which shows the **Order Details** table with the **ProductID 2** row selected, indicating that there are orders for that product.
4. Display the next image, in which an error message is printed.
5. Display the final image, in which the **ProductID 2** row is returned to the **Products** table.

Module Strategy

Use the following strategy to present this module:

- Introduction to Triggers

  Define triggers for students. Tell them that a trigger is a special type of stored procedure that is assigned to a specific table. Then discuss the three key points: that triggers are invoked automatically; that they cannot be called by anything other than a trigger action to the trigger table; and that they are transactions.

  Briefly discuss some of the primary uses of triggers. Triggers can cascade changes throughout a database. They can enforce more complex business rules than can be defined with CHECK constraints or rules. Triggers also provide a way to compare the before and after states (images) of data that are modified by an INSERT, UPDATE, or DELETE statement. Mention that cascading referential integrity actions can replace the use of triggers.

  Discuss some of the facts and guidelines that implementers must consider when determining whether to use triggers instead of other tools.

- Defining Triggers

  Discuss the CREATE TRIGGER statement and the various options that users have when they create a trigger. Emphasize that students have full control over the statements that a trigger executes and that users cannot circumvent the trigger from firing.

  Discuss permissions, the Microsoft SQL Server™ 2000 statements that cannot be used in trigger definitions, and that users can use the **sp_depends** system stored procedure on a table to review trigger information that is associated with the table.

  Discuss the procedure for altering a trigger and give (or ask students for) examples where they might want to alter trigger definitions. Also discuss the DROP TRIGGER syntax.

- How Triggers Work

  Discuss how the INSERT, DELETE, UPDATE, and INSTEAD OF statements work, displaying the animated slides.

  Describe the use of nested triggers and mention that users can enable or disable nesting. Discuss situations in which students would want to use nesting and the implications of doing so, such as lengthy transactions, locks, and ROLLBACK TRANSACTION statements.

  Finally, discuss how recursive triggers work. Inform students that a trigger can call itself but that students should include a recursive termination check statement in the trigger definition so that the trigger does not end up in an endless loop.

- Examples of Triggers

  Discuss and describe each example of effective triggers.

- Performance Considerations

  Discuss some of the performance considerations for using triggers.

Customization Information

This section identifies the lab setup requirements for a module and the configuration changes that occur on student computers during the labs. This information is provided to assist you in replicating or customizing Microsoft Official Curriculum (MOC) courseware.

Important The lab in this module is dependent on the classroom configuration that is specified in the Customization Information section at the end of the *Classroom Setup Guide* for course 2073A, *Programming a Microsoft SQL Server 2000 Database*.

Lab Setup

The following section describes the setup requirement for the lab in this module.

Setup Requirement

The lab in this module requires the **ClassNorthwind** database to be in a state required for this lab. To prepare student computers to meet this requirement, perform one of the following actions:

- Complete the prior lab
- Execute the C:\Moc\2073A\Batches\Restore11.cmd batch file.

Warning If this course has been customized, students must execute the C:\Moc\2073A\Batches\Restore11.cmd batch file to ensure that the lab will function properly.

Lab Results

There are no configuration changes on student computers that affect replication or customization.

Overview

Topic Objective
To provide an overview of the module topics and objectives.

Lead-in
In this module, you will learn about creating triggers.

- **Introduction to Triggers**
- **Defining Triggers**
- **How Triggers Work**
- **Examples of Triggers**
- **Performance Considerations**

A *trigger* is a stored procedure that executes when data in a specified table is modified. You often create triggers to enforce referential integrity or consistency among logically related data in different tables. Because users cannot circumvent triggers, you can use triggers to enforce complex business rules that maintain data integrity.

After completing this module, you will be able to:

- Create a trigger.
- Drop a trigger.
- Alter a trigger.
- Describe how various triggers work.
- Evaluate the performance considerations that affect using triggers.

Introduction to Triggers

Topic Objective
To introduce the concept of trigger objects.

Lead-in
In this section, you will learn when and how to use triggers.

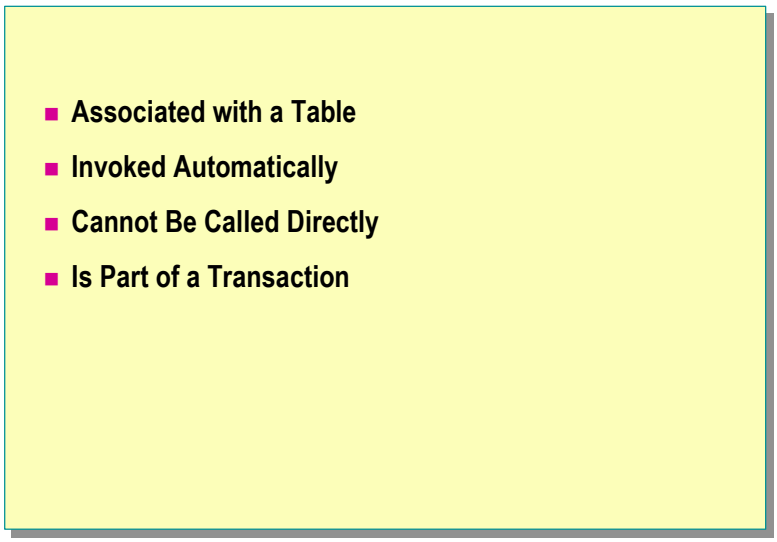
- What Is a Trigger?
- Uses of Triggers
- Considerations for Using Triggers

This section introduces triggers and describes when and how to use them.

What Is a Trigger?

Topic Objective
To introduce the concept of a trigger and to state the advantages of using one.

Lead-in
A trigger is a special kind of stored procedure that executes automatically whenever an attempt is made to modify data that the trigger protects. Triggers are tied to specific tables.



Key Points
A trigger is a special type of stored procedure. You can include any Transact-SQL statements in a trigger. Triggers use the procedure cache to store the execution plan.

A trigger is a special kind of stored procedure that executes whenever an attempt is made to modify data in a table that the trigger protects. Triggers are tied to specific tables.

Associated with a Table

Triggers are defined on a specific table, which is referred to as the trigger table.

Invoked Automatically

When an attempt is made to insert, update, or delete data in a table, and a trigger for that particular action has been defined on the table, the trigger executes automatically. It cannot be circumvented.

Cannot Be Called Directly

Unlike standard system-stored procedures, triggers cannot be called directly and do not pass or accept parameters.

Is Part of a Transaction

Delivery Tip
Do *not* discuss transactions in detail. Transactions are described in other modules in this course.

The trigger and the statement that fires it are treated as a single transaction that can be rolled back from anywhere within the trigger. When using triggers, consider these facts and guidelines:

- Trigger definitions can include a ROLLBACK TRANSACTION statement even if an explicit BEGIN TRANSACTION statement does not exist.
- If a ROLLBACK TRANSACTION statement is encountered, the entire transaction rolls back. If a statement in the trigger script follows the ROLLBACK TRANSACTION statement, the statement is executed. It may be necessary to use a RETURN clause in an IF statement to prevent the processing of other statements.

- If a trigger that includes a ROLLBACK TRANSACTION statement is fired from within a user-defined transaction, the ROLLBACK TRANSACTION rolls back the entire transaction. A trigger that is executed from within a batch that executes a ROLLBACK TRANSACTION statement cancels the batch; subsequent statements in the batch are not executed.
- You should minimize or avoid the use of ROLLBACK TRANSACTION in your trigger code. Rolling back a transaction creates additional work because all of the work that was done to that point in the transaction has to be undone. This has a negative impact on performance. It is recommended that information be checked and validated outside the transaction. Start the transaction after everything is checked and verified.
- The user that invokes the trigger must also have permission to perform all statements on all tables.

Uses of Triggers

Topic Objective
To introduce the advantages of using triggers.

Lead-in
There are several advantages to using triggers.

- **Cascade Changes Through Related Tables in a Database**
- **Enforce More Complex Data Integrity Than a CHECK Constraint**
- **Define Custom Error Messages**
- **Maintain Denormalized Data**
- **Compare Before and After States of Data Under Modification**

Delivery Tip
Point out that you can use triggers to cascade updates and deletes through related tables in a database.

Note that many databases created with previous versions of SQL Server can contain this type of trigger.

Triggers are best used to maintain low-level data integrity, *not* to return query results. The primary benefit of triggers is that they can contain complex processing logic. Triggers can cascade changes through related tables in a database, enforce more complex data integrity than a CHECK constraint, define custom error messages, maintain *denormalized* data, and compare before and after states of data under modification.

Cascade Changes Through Related Tables in a Database

You can use a trigger to cascade updates and deletes through related tables in a database. For example, a delete trigger on the **Products** table in the **Northwind** database can delete matching rows in other tables that have rows that match the deleted **ProductID** values. A trigger does this by using the **ProductID** foreign key column as a way of locating rows in the **Order Details** table.

Enforce More Complex Data Integrity Than a CHECK Constraint

Unlike CHECK constraints, triggers can reference columns in other tables. For example, you could place an insert trigger on the **Order Details** table that checks the **UnitsInStock** column for that item in the **Products** table. The trigger could determine that when the **UnitsInStock** value is less than 10, that the maximum order amount is three items. This type of check references columns in other tables. Referencing columns in other tables is not permitted with a CHECK constraint.

You can use triggers to enforce complex referential integrity by:

- Taking action or cascading updates or deletes.

  Referential integrity can be defined by using FOREIGN KEY and REFERENCE constraints with the CREATE TABLE statement. Triggers are useful for ensuring appropriate actions when cascading deletions or updates must occur. If constraints exist on the trigger table, they are checked prior to the trigger execution. If constraints are violated, the trigger is not executed.

- Creating multi-row triggers

  When more than one row is inserted, updated, or deleted, you must write a trigger to handle multiple rows.

- Enforcing referential integrity between databases.

Define Custom Error Messages

Occasionally, your implementation may benefit from custom error messages that indicate the status of an action. By using triggers, you can invoke predefined or dynamic custom error messages when certain conditions occur as a trigger executes.

Maintain Denormalized Data

Triggers can be used to maintain low-level data integrity in denormalized database environments. Maintaining denormalized data is different from cascading in that cascading typically refers to maintaining relationships between primary and foreign key values. Denormalized data is typically contrived, derived, or redundant data values. You must use a trigger if:

- Referential integrity requires something that is not an exact match, such as maintaining derived data (year-to-date sales) or flagging columns (Y or N to indicate whether a product is available).
- You require customized messages and complex error messaging.

Note Redundant data and derived data typically require the use of triggers.

Compare Before and After States of Data Under Modification

Most triggers provide the ability to reference the changes that are made to the data by the INSERT, UPDATE, or DELETE statement. This allows you to reference the rows that are being affected by the modification statements inside the trigger.

Note Constraints, rules, and defaults can communicate errors only through standardized system-error messages. If your application requires (or can benefit from) customized messages and more complex error handling, you must use a trigger.

Considerations for Using Triggers

Topic Objective
To discuss various issues that students must consider when they use triggers.

Lead-in
Consider the following facts and guidelines when you work with triggers.

- **Triggers Are Reactive; Constraints Are Proactive**
- **Constraints Are Checked First**
- **Tables Can Have Multiple Triggers for Any Action**
- **Table Owners Can Designate the First and Last Trigger to Fire**
- **You Must Have Permission to Perform All Statements That Define Triggers**
- **Table Owners Cannot Create AFTER Triggers on Views or Temporary Tables**

Delivery Tip
Be sure to cover *all* of the bulleted items, even though they do not all appear on the slide.

Consider the following facts and guidelines when you work with triggers:

- Most triggers are reactive; constraints and the INSTEAD OF trigger are proactive.

  Triggers are executed after an INSERT, UPDATE, or DELETE statement is executed on the table in which the trigger is defined. For example, an UPDATE statement updates a row in a table, and then the trigger on that table executes automatically. Constraints are checked before an INSERT, UPDATE, or DELETE statement executes.

- Constraints are checked first.

  If constraints exist on the trigger table, they are checked prior to the trigger execution. If constraints are violated, the trigger does not execute.

- Tables can have multiple triggers for any action.

  SQL Server 2000 allows nesting of several triggers on a single table. A table can have multiple triggers defined for it. Each trigger can be defined for a single action or multiple actions.

- Table owners can designate the first and last trigger to fire.

  When multiple triggers are placed on a table, the table owner can use the **sp_settriggerorder** system stored procedure to specify the first and last triggers to fire. The firing order of the remaining triggers cannot be set.

- You must have permission to perform all trigger-defined statements.

  Only the table owner, members of the **sysadmin** fixed-server role, and members of the **db_owner** and **db_ddladmin** fixed-database roles can create and drop triggers for that table. These permissions cannot be transferred.

  In addition, the trigger creator also must have permission to perform all of the statements on all of the affected tables. If permissions are denied to any portion of the Transact-SQL statements inside the trigger, the entire transaction is rolled back.

- Table owners cannot create AFTER triggers on views or temporary tables. Triggers can, however, reference views and temporary tables.

- Table owners can create INSTEAD OF triggers on views and tables, in which case INSTEAD OF triggers greatly extend the types of updates that a view can support.

Delivery Tip
Remind students that triggers do not return result sets or pass parameters.

- Triggers should not return result sets.

  Triggers contain Transact-SQL statements, in the same way that stored procedures do. Like stored procedures, triggers can contain statements that return a result set. However, including statements that return values in triggers is not recommended because users or developers do not expect to see any result sets when an UPDATE, INSERT, or DELETE statement executes.

- Triggers can handle multi-row actions.

  An INSERT, UPDATE, or DELETE action that invokes a trigger can affect multiple rows. You can choose to:

  - Process all of the rows together, in which case all affected rows must meet the trigger criteria for any action to occur.
  - Allow conditional actions.

    For example, if you want to delete three customers from the **Customers** table, you can define a trigger to ensure that there are no active orders or outstanding invoices for each deleted customer. If one of the three customers has an outstanding invoice, that customer will not be deleted, but the qualifying customers will be deleted.

  To determine whether there are multiple affected rows, use the @@ROWCOUNT system function.

Defining Triggers

Topic Objective
To introduce the topics on creating, altering, and dropping triggers that this section covers.

Lead-in
Now that you know what triggers are, let's see how to create, alter, and drop them.

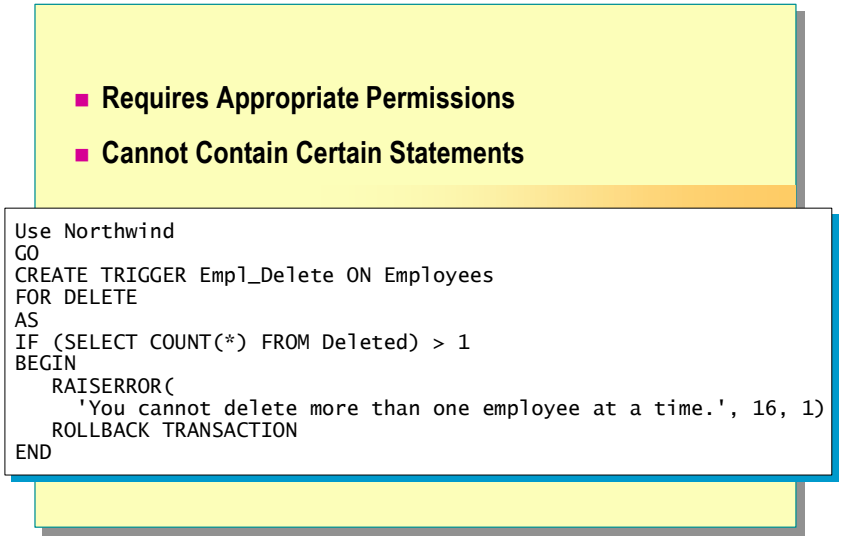
- Creating Triggers
- Altering and Dropping Triggers

This section covers creating, altering, and dropping triggers. It also discusses required permissions and guidelines to follow when defining triggers.

Creating Triggers

Topic Objective
To introduce the CREATE TRIGGER syntax.

Lead-in
Consider these facts and guidelines when you create triggers.



Create triggers by using the CREATE TRIGGER statement. The statement specifies the table on which a trigger is defined, the events for which the trigger executes, and the particular instructions for the trigger.

Syntax

```
CREATE TRIGGER [owner.] trigger_name
  ON [owner.] table_name
  [WITH ENCRYPTION]
  {FOR | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE}
  AS
  [IF UPDATE (column_name)...]
    [{AND | OR} UPDATE (column_name)...]
    sql_statements}
```

When a FOR UPDATE action is specified, the IF UPDATE (*column_name*) clause can be used to focus action on a specific column that is updated.

Both FOR and AFTER are equivalent syntax creating the same type of trigger, which fires after the initiating (INSERT, UPDATE, or DELETE) action.

INSTEAD OF triggers cancel the triggering action and perform a new function instead.

When you create a trigger, information about the trigger is inserted into the **sysobjects** and **syscomments** system tables. If a trigger is created with the same name as an existing trigger, the new trigger will overwrite the original trigger.

Note SQL Server does not support the addition of user-defined triggers on system tables; therefore, you cannot create triggers on system tables.

Requires Appropriate Permissions

Table owners, and members of the database owner (**db_owner**) and the system administrators (**sysadmin**) roles, have permission to create a trigger.

To avoid situations in which the owner of a view and the owner of the underlying tables differ, it is recommended that the **dbo** user own all objects in a database. Because a user can be a member of multiple roles, always specify the **dbo** user as the owner name when you create the object. Otherwise, the object will be created with your user name as the owner.

Cannot Contain Certain Statements

SQL Server does not allow the following statements to be used in a trigger definition:

- ALTER DATABASE
- CREATE DATABASE
- DISK INIT
- DISK RESIZE
- DROP DATABASE
- LOAD DATABASE
- LOAD LOG
- RECONFIGURE
- RESTORE DATABASE
- RESTORE LOG

To determine the tables with triggers, execute the **sp_depends <***tablename***>** system stored procedure. To view a trigger definition, execute the **sp_helptext** <*triggername*> system stored procedure. To determine the triggers that exist on a specific table and their actions, execute the **sp_helptrigger** <*tablename*> system stored procedure.

Example

The following example creates a trigger on the **Employees** table that prevents users from deleting more than one employee at a time. The trigger fires every time a record or group of records are deleted from the table. The trigger checks the number of records being deleted by querying the **Deleted** table. If more than one record is being deleted, the trigger returns a custom error message and rolls back the transaction.

```
Use Northwind
GO

CREATE TRIGGER Empl_Delete ON NewEmployees
FOR DELETE
AS
IF (SELECT COUNT(*) FROM Deleted) > 1
BEGIN
   RAISERROR(
      'You cannot delete more than one employee at a time.',
       16, 1)
   ROLLBACK TRANSACTION
END
```

The following DELETE statement fires the trigger and prevents the transaction.

```
DELETE FROM Employees WHERE EmployeeID > 6
```

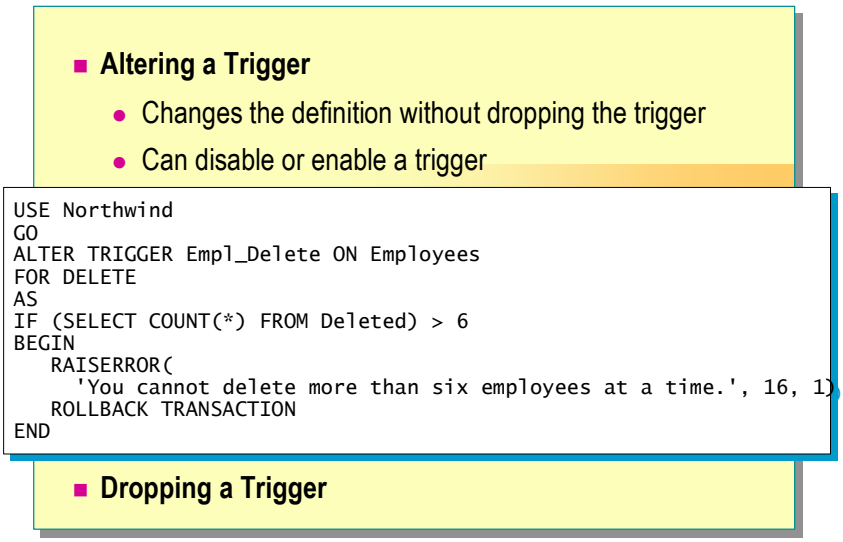
The following DELETE statement fires the trigger and allows the transaction.

```
DELETE FROM Employees WHERE EmployeeID = 6
```

Altering and Dropping Triggers

Topic Objective
To introduce the concept of altering a trigger.

Lead-in
If you must change the definition of an existing trigger, you can alter it without having to drop it.



You can alter or drop a trigger.

Altering a Trigger

If you must change the definition of an existing trigger, you can alter it without having to drop it.

Changes the Definition Without Dropping the Trigger

The altered definition replaces the definition of the existing trigger with the new definition. Trigger action also can be altered. For example, if you create a trigger for INSERT and then change the action to UPDATE, the altered trigger executes whenever the table is updated.

With delayed name resolution, your trigger can reference tables and views that do not yet exist. If the object does not exist when a trigger is created, you receive a warning message and SQL Server updates the trigger definition immediately.

Syntax

```
ALTER TRIGGER trigger_name
  ON table
  [WITH ENCRYPTION]
  {{FOR {[,] [DELETE] [,] [UPDATE] [,][INSERT]}
  [NOT FOR REPLICATION]
  AS
  sql_statement [...n] }
  |
  {FOR {[,] [INSERT] [,] [UPDATE]}
  [NOT FOR REPLICATION]
  AS
  IF UPDATE (column)
  [{AND | OR} UPDATE (column) [,...n]]
  sql_statement [...n] }
  }
```

Example

This example alters the delete trigger created in the previous example. New trigger content is provided, which changes the delete limit from one record to six records.

```
Use Northwind
GO
CREATE TRIGGER Empl_Delete ON Employees
FOR DELETE
AS
IF (SELECT COUNT(*) FROM Deleted) > 6
BEGIN
   RAISERROR(
     'You cannot delete more than six employees at a time.',
      16, 1)
   ROLLBACK TRANSACTION
END
```

Disabling or Enabling a Trigger

You can disable or enable a specific trigger, or all triggers on a table. When a trigger is disabled, it is still defined for the table; however, when an INSERT, UPDATE, or DELETE statement is executed against the table, the actions in the trigger are not performed until the trigger is re-enabled.

You can enable or disable triggers in the ALTER TABLE statement.

Partial Syntax

ALTER TABLE *table*
{ENABLE | DISABLE} TRIGGER
{ALL | *trigger_name*[**,***…n*]}

Dropping a Trigger

You can remove a trigger by dropping it. Triggers are dropped automatically whenever their associated tables are dropped.

Permission to drop a trigger defaults to the table owner and is non-transferable. However, members of the system administrators (**sysadmin**) and database owner (**db_owner**) roles can drop any object by specifying the owner in the DROP TRIGGER statement.

Syntax

DROP TRIGGER *trigger_name*

How Triggers Work

Topic Objective
To introduce the section on how triggers work.

Lead-in
Let's examine how different types of triggers work.

- How an INSERT Trigger Works
- How a DELETE Trigger Works
- How an UPDATE Trigger Works
- How an INSTEAD OF Trigger Works
- How Nested Triggers Work
- Recursive Triggers

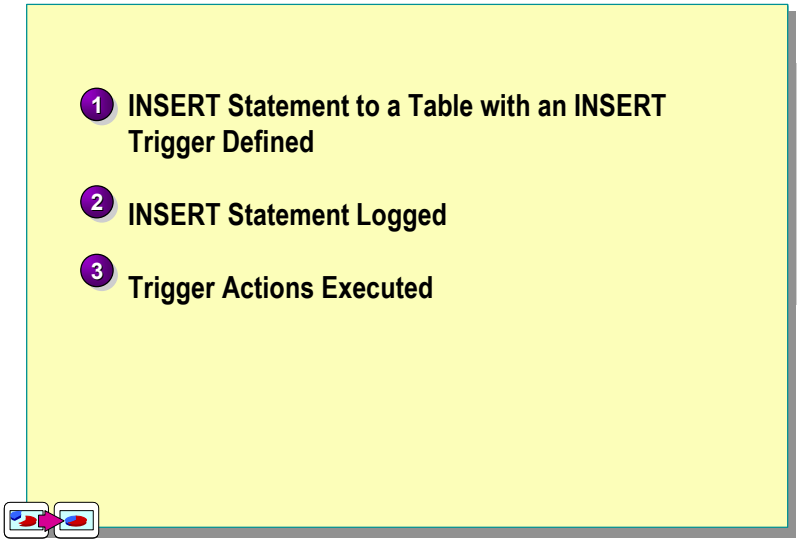
When you design triggers, it is important to understand how they work. This section discusses INSERT, DELETE, UPDATE, INSTEAD OF, nested, and recursive triggers.

How an INSERT Trigger Works

Topic Objective
To show an example of an INSERT trigger.

Lead-in
An INSERT trigger is invoked when an attempt is made to insert a row into a table that the trigger protects.

All inserts are recorded in a special **inserted** table, as illustrated in the slide.



Delivery Tip
Point out how to read CREATE TRIGGER statements quickly to find the trigger name, table, and action The trigger does everything below the AS statement when it executes.

You can define a trigger to execute whenever an INSERT statement inserts data into a table.

When an INSERT trigger is fired, new rows are added to both the trigger table and the **inserted** table. The **inserted** table is a logical table that holds a copy of the rows that have been inserted. The **inserted** table contains the logged insert activity from the INSERT statement. The **inserted** table allows you to reference logged data from the initiating INSERT statement. The trigger can examine the **inserted** table to determine whether, or how, the trigger actions should be carried out. The rows in the **inserted** table are always duplicates of one or more rows in the trigger table.

All data modification activity (INSERT, UPDATE, and DELETE statements) is logged, but the information in the transaction log is unreadable. However, the **inserted** table allows you to reference the logged changes that the INSERT statement caused. Then you can compare the changes to the inserted data in order to verify them or take further action. You also can reference inserted data without having to store the information in variables.

Example

The trigger in this example was created to update a column (**UnitsInStock**) in the **Products** table whenever a product is ordered (whenever a record is inserted into the **Order Details** table). The new value is set to the previous value minus the ordered amount.

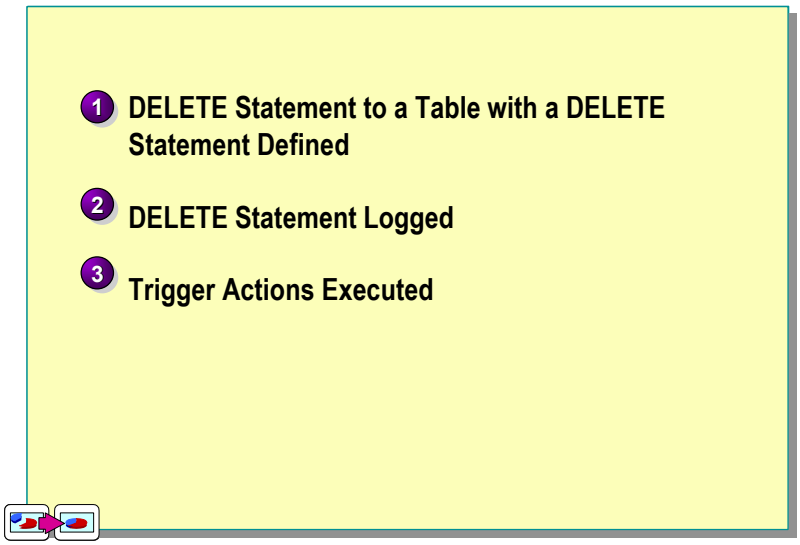
```
USE Northwind
CREATE TRIGGER OrdDet_Insert
ON [Order Details]
FOR INSERT
AS
UPDATE P SET
UnitsInStock = (P.UnitsInStock - I.Quantity)
FROM Products AS P INNER JOIN Inserted AS I
ON P.ProductID = I.ProductID
```

How a DELETE Trigger Works

Topic Objective
To show an example of a DELETE trigger.

Lead-in
A DELETE trigger is invoked whenever an attempt is made to delete information from the table on which the trigger is defined.

When rows are deleted from a table, they are placed in a special **deleted** table, as illustrated in the slide.



When a DELETE trigger is fired, deleted rows from the affected table are placed in a special **deleted** table. The **deleted** table is a logical table that holds a copy of the rows that have been deleted. The **deleted** table allows you to reference logged data from the initiating DELETE statement.

Consider the following facts when you use the DELETE trigger:

- When a row is appended to the **deleted** table, it no longer exists in the database table; therefore, the **deleted** table and the database tables have no rows in common.
- Space is allocated from memory to create the **deleted** table. The **deleted** table is always in the cache.
- A trigger that is defined for a DELETE action does not execute for the TRUNCATE TABLE statement because TRUNCATE TABLE is not logged.

Example

The trigger in this example was created to update the **Discontinued** column in the **Products** table whenever a category is deleted (whenever a record is deleted from the **Categories** table). All affected products are marked as 1, indicating they are discontinued.

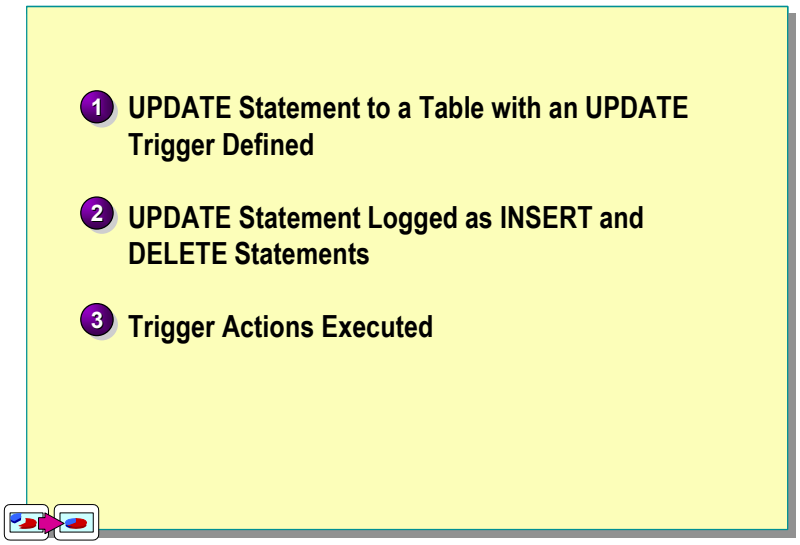
```
USE Northwind
CREATE TRIGGER Category_Delete
  ON Categories
  FOR DELETE
AS
  UPDATE P SET Discontinued = 1
      FROM Products AS P INNER JOIN deleted AS d
      ON P.CategoryID = d.CategoryID
```

How an UPDATE Trigger Works

Topic Objective
To provide an example of an UPDATE trigger.

Lead-in
A trigger that is defined for an UPDATE statement is invoked whenever an attempt is made to update data in a table on which the trigger is defined.

An UPDATE statement moves the original row into the **deleted** table and inserts the updated row into the **inserted** table, as illustrated by this graphic.



An UPDATE statement can be thought of as two steps: the DELETE step that captures the *before image* of the data, and the INSERT step that captures the *after image* of the data. When an UPDATE statement is executed on a table that has a trigger defined on it, the original rows (before image) are moved into the **deleted** table, and the updated rows (after image) are inserted into the **inserted** table.

The trigger can examine the **deleted** and **inserted** tables, as well as the updated table, to determine whether multiple rows have been updated and how the trigger actions should be carried out.

You can define a trigger to monitor data updates on a specific column by using the IF UPDATE statement. This allows the trigger to isolate activity easily for a specific column. When it detects that the specific column has been updated, it can take proper action, such as raising an error message that says that the column cannot be updated, or by processing a series of statements based on the newly updated column value.

Syntax

IF UPDATE (<*column_name*>)

Example 1

This example prevents a user from modifying the **EmployeeID** column in the **Employees** table.

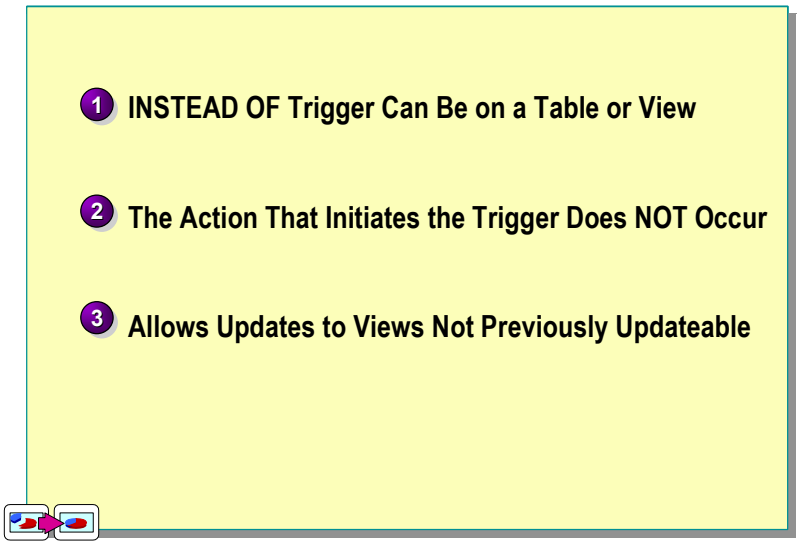
Delivery Tip
The backslash (\) character in the RAISERROR statement is a continuation character that allows the entire error message text to display on one line.

```
USE Northwind
GO
CREATE TRIGGER Employee_Update
  ON Employees
  FOR UPDATE
AS
IF UPDATE (EmployeeID)
BEGIN TRANSACTION
  RAISERROR ('Transaction cannot be processed.\
  ***** Employee ID number cannot be modified.', 10, 1)
  ROLLBACK TRANSACTION
END
```

How an INSTEAD OF Trigger Works

Topic Objective
To show an example of an INSTEAD OF Trigger.

Lead-in
An INSTEAD OF trigger cancels the original triggering action and performs its own function instead.



Key Points
Contrast an INSTEAD OF trigger with an AFTER trigger.

Point out that when you use an INSTEAD OF trigger, the original triggering action (the INSERT, UPDATE, or DELETE) does *not* occur. Note that you can place an INSTEAD OF trigger on tables and views.

You can specify an INSTEAD OF trigger on both tables and views. This trigger executes instead of the original triggering action. INSTEAD OF triggers increase the variety of types of updates that you can perform against a view. Each table or view is limited to one INSTEAD OF trigger for each triggering action (INSERT, UPDATE, or DELETE).

You cannot create an INSTEAD OF trigger on views that have the WITH CHECK OPTION defined.

Example

This example creates a table with customers in Germany and a table with customers in Mexico. An INSTEAD OF trigger placed on the view redirects updates to the appropriate underlying table. The insert to the **CustomersGer** table occurs *instead of* the insert to the view.

Delivery Tip
This example creates the **CustomersGer** and **CustomersMex** tables and then a view called **CustomersView**.

Show how the update fails against the view. Then create the trigger and show how the trigger redirects the update.

Create two tables with customer data

```
SELECT * INTO CustomersGer FROM Customers WHERE
Customers.Country = 'Germany'
SELECT * INTO CustomersMex FROM Customers WHERE
Customers.Country = 'Mexico'
GO
```

Create a view on that data

```
CREATE VIEW CustomersView AS
SELECT * FROM CustomersGer
UNION
SELECT * FROM CustomersMex
GO
```

Create an INSTEAD OF trigger on the view

```
CREATE TRIGGER Customers_Update2
ON CustomersView
INSTEAD OF UPDATE AS
DECLARE @Country nvarchar(15)
SET @Country = (SELECT Country FROM Inserted)
IF @Country = 'Germany'
  BEGIN
   UPDATE CustomersGer
    SET CustomersGer.Phone = Inserted.Phone
    FROM CustomersGer JOIN Inserted
    ON CustomersGer.CustomerID = Inserted.CustomerID
  END
ELSE
  IF @Country = 'Mexico'
  BEGIN
   UPDATE CustomersMex
    SET CustomersMex.Phone = Inserted.Phone
    FROM CustomersMex JOIN Inserted
    ON CustomersMex.CustomerID = Inserted.CustomerID
  END
```

If students ask why this script apparently updated **CustomersView**. Point out that the view gets its information from the **CustomerGer** table.

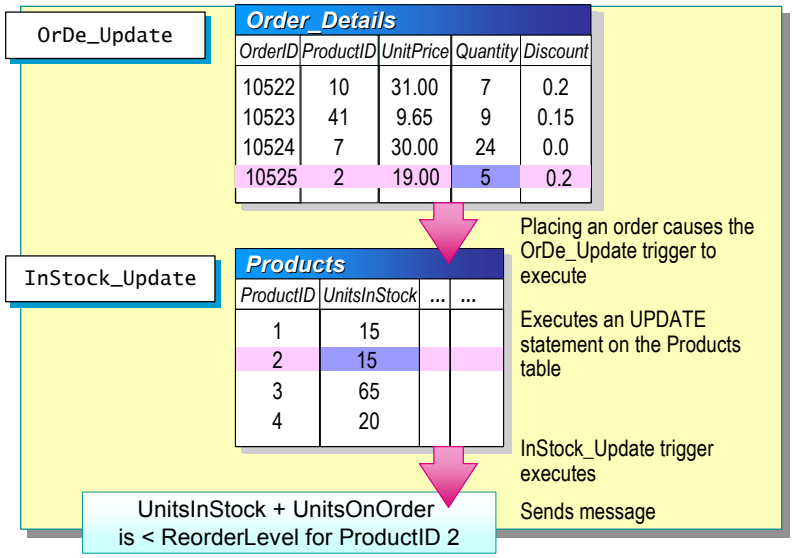
Test the trigger by updating the view

```
UPDATE CustomersView SET Phone = ' 030-007xxxx'
  WHERE CustomerID = 'ALFKI'
SELECT CustomerID, Phone FROM CustomersView
  WHERE CustomerID = 'ALFKI'
SELECT CustomerID, Phone FROM CustomersGer
  WHERE CustomerID = 'ALFKI'
```

How Nested Triggers Work

Topic Objective
To discuss the use of nested triggers.

Lead-in
As mentioned previously, triggers can be nested up to 32 levels deep. If nested triggers are enabled, a trigger that changes a table can activate a second trigger, which in turn can activate a third trigger, and so on.



Any trigger can contain an UPDATE, INSERT, or DELETE statement that affects another table. With nesting enabled, a trigger that changes a table can activate a second trigger, which in turn can activate a third trigger, and so on. Nesting is enabled at installation, but you can disable and re-enable it by using the **sp_configure** system stored procedure.

Triggers can be nested up to 32 levels deep. If any trigger in a nested chain sets off an infinite loop, the nesting level is exceeded. The trigger then terminates and rolls back the transaction. You can use nested triggers to perform functions, such as the storage of a backup copy of rows that were affected by a previous trigger. Consider the following facts when you use nested triggers:

- By default, the nested triggers configuration option is on.
- A nested trigger will not fire twice in the same trigger transaction; a trigger does not call itself in response to a second update to the same table within the trigger. For example, if a trigger modifies a table that, in turn, modifies the original trigger table, the trigger does not fire again.
- Because a trigger is a transaction, a failure at any level of a set of nested triggers cancels the entire transaction, and all data modifications are rolled back. Therefore, you should include PRINT statements when you test triggers so that you can determine where the failure occurred.

Delivery Tip
The @@NESTLEVEL function is useful when testing and troubleshooting triggers but would not typically be included in a production environment.

Checking the Nesting Level

Each time that a nested trigger fires, the nesting level increments. SQL Server supports up to 32 levels of nesting, but you may want to limit the levels of nesting to avoid exceeding the maximum nesting level. You can use the @@NESTLEVEL function to see the current levels of nesting.

Determining Whether to Use Nesting

Nesting is a powerful feature that you can use to maintain data integrity throughout a database. Occasionally, however, you may want to disable nesting. If nesting is disabled, a trigger that modifies another table does not invoke any of the triggers on the second table.

Use the following statement to disable nesting:

Syntax

sp_configure 'nested triggers', 0

You may decide to disable nesting because:

- Nested triggers require a complex and well-planned design. Cascading changes can modify data that you did not intend to affect.
- A data modification at any point in a series of nested triggers sets off the trigger series. Although this offers powerful protection for your data, it can be a problem if your tables must be updated in a specific order.

You can create the same functionality with or without the nesting feature; however, your trigger design will differ substantially. In designing nested triggers, each trigger should initiate only the next data modification—the design should be modular. In designing non-nested triggers, each trigger should initiate all data modifications that you want it to make.

Example

This example shows how placing an order causes the **OrDe_Update** trigger to execute. This trigger executes an UPDATE statement on the **UnitsInStock** column of the **Products** table. When the update occurs, it fires the **Products_Update** trigger and compares the new value of the stock in inventory, plus the stock on order, to the reorder level. If the stock in inventory plus the stock on order falls below the reorder level, a message is sent alerting the buyer to purchase more stock.

```
USE Northwind
GO
CREATE TRIGGER Products_Update
  ON Products
  FOR UPDATE
AS
IF UPDATE (UnitsInStock)
  IF (Products.UnitsInStock + Products.UnitsOnOrder) <
Products.ReorderLevel
BEGIN
  --Send message to the purchasing department
END
```

Recursive Triggers

Topic Objective
To discuss the use of recursive triggers.

Lead-in
With the recursive trigger option enabled, a trigger that changes data in a table can activate a second trigger, which in turn can activate the original calling trigger by modifying data in the original table.

- **Activating a Trigger Recursively**
- **Types of Recursive Triggers**
  - *Direct recursion* occurs when a trigger fires and performs an action that causes the same trigger to fire again
  - *Indirect recursion* occurs when a trigger fires and performs an action that causes a trigger on another table to fire
- **Determining Whether to Use Recursive Triggers**

Any trigger can contain an UPDATE, INSERT, or DELETE statement that affects the same table or another table. With the recursive trigger option enabled, a trigger that changes data in a table can activate itself again, in a recursive execution. The recursive trigger option is disabled by default when a database is created, but you can enable it by using the ALTER DATABASE statement.

Activating a Trigger Recursively

Use the following statement to enable recursive triggers:

Syntax

```
ALTER DATABASE ClassNorthwind SET RECURSIVE_TRIGGERS ON

sp_dboption databasename, 'recursive triggers', True
```

Note Use the **sp_settriggerorder** system stored procedure to specify a trigger that fires as the first AFTER trigger, or the last AFTER trigger. There is no fixed order in which other triggers, that are defined for a given event, are executed. Each trigger should be self-contained.

If the nested trigger option is off, the recursive trigger option is also disabled, regardless of the recursive trigger setting of the database.

The **inserted** and **deleted** tables for a given trigger contain rows that correspond only to the UPDATE, INSERT, or DELETE statement that last invoked the trigger.

Trigger recursion can occur up to 32 levels deep. If any trigger in a recursive loop sets off an infinite loop, the nesting level is exceeded, and the trigger terminates and rolls back the transaction.

Types of Recursive Triggers

There are two different types of recursion:

- Direct recursion, which occurs when a trigger fires and performs an action that causes the same trigger to fire again.

  For example, an application updates table **T1**, which causes trigger **Trig1** to fire. **Trig1** updates table **T1** again, which causes trigger **Trig1** to fire again.

- Indirect recursion, which occurs when a trigger fires and performs an action that causes a trigger on another table to fire, subsequently causes an update to occur on the original table. This then causes the original trigger to fire again.

  For example, an application updates table **T2**, which causes trigger **Trig2** to fire. **Trig2** updates table **T3**, which causes trigger **Trig3** to fire. **Trig3** in turn updates table **T2**, which causes **Trig2** to fire again.

Determining Whether to Use Recursive Triggers

Recursive triggers are a complex feature that you can use to solve complex relationships, such as self-referencing relationships (also known as *transitive closures*). In these special situations, you may want to enable recursive triggers.

Delivery Tip
Point out that the first example does not refer or apply to the **Employees** table in the **Northwind** database.

Recursive triggers may be useful when you must maintain:

- The number of reports columns in the **employee** table where the table contains an **employee ID** column and a **manager ID** column.

  For example, assume that two update triggers, **tr_update_employee** and **tr_update_manager**, are defined on the **employee** table. The **tr_update_employee** trigger updates the **employee** table.

  An UPDATE statement fires both **tr_update_employee** and **tr_update_manager** triggers once. In addition, the execution of **tr_update_employee** triggers the execution of **tr_update_employee** again (recursively) and **tr_update_manager**.

- A chart for production scheduling data in which an implied scheduling hierarchy exists.
- An assembly tracking system in which subparts are tracked to parent parts.

Consider the following guidelines before you use recursive triggers:

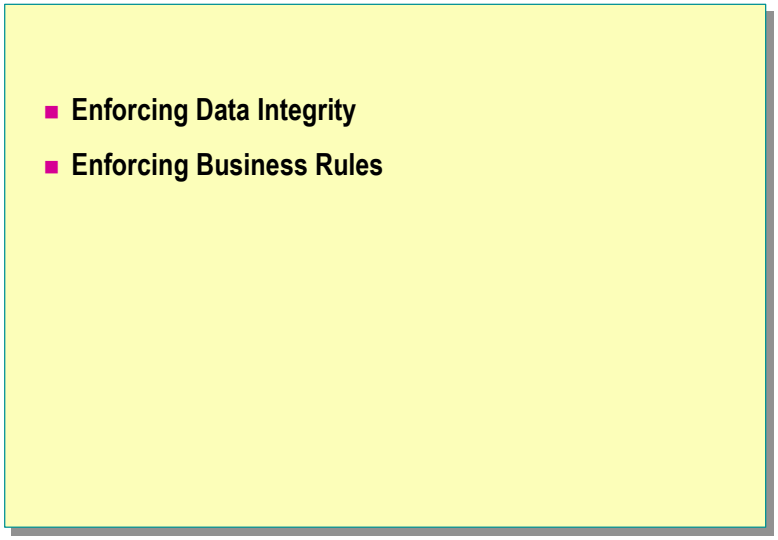
- Recursive triggers are complex and must be well designed and thoroughly tested. Recursive triggers require controlled looping logic code (termination check). Otherwise, you will exceed the 32-level nesting limit.
- A data modification at any point can set off the trigger series. Although this provides the ability to process complex relationships, it can be a problem if your tables must be updated in a specific order.

You can create similar functionality without the recursive trigger feature; however, your trigger design will differ substantially. In designing recursive triggers, each trigger must contain a conditional check in order to stop recursive processing when the condition becomes false. In designing non-recursive triggers, each trigger must contain the full programming looping structures and checks.

Examples of Triggers

Topic Objective
To explain why triggers are necessary in SQL Server.

Lead-in
Triggers enforce data integrity and business rules.

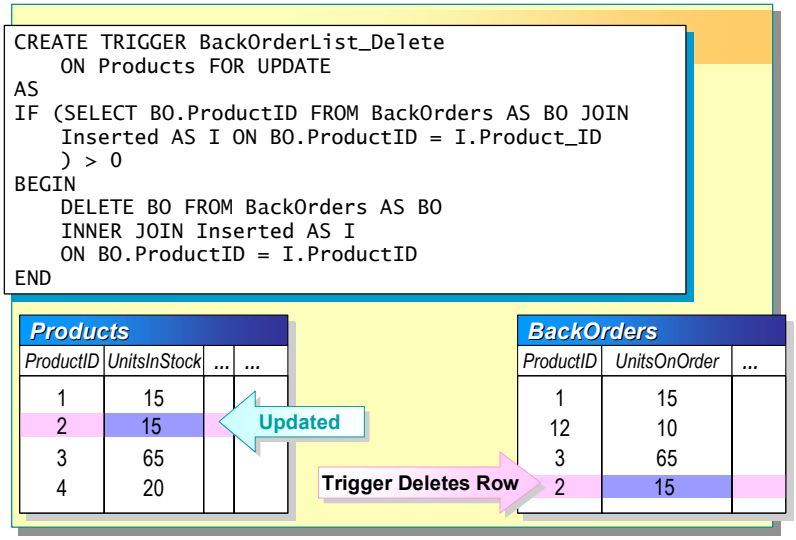


Triggers enforce data integrity and business rules. You can accomplish some of the actions that triggers perform through the use of constraints, and for certain actions, you should first consider constraints. However, triggers are needed to enforce various degrees of denormalization and to enforce complex business rules.

Enforcing Data Integrity

Topic Objective
To show an example of how triggers enforce data integrity.

Lead-in
You can use triggers to maintain data integrity by cascading changes to related tables throughout the database.



You can use triggers to maintain data integrity by cascading changes to related tables throughout the database.

Example

The following example shows how a trigger maintains data integrity on a **BackOrders** table. The **BackOrderList_delete** trigger maintains the list of products in the **BackOrders** table. When products are received, the UPDATE trigger on the **Products** table deletes records from a **BackOrders** table.

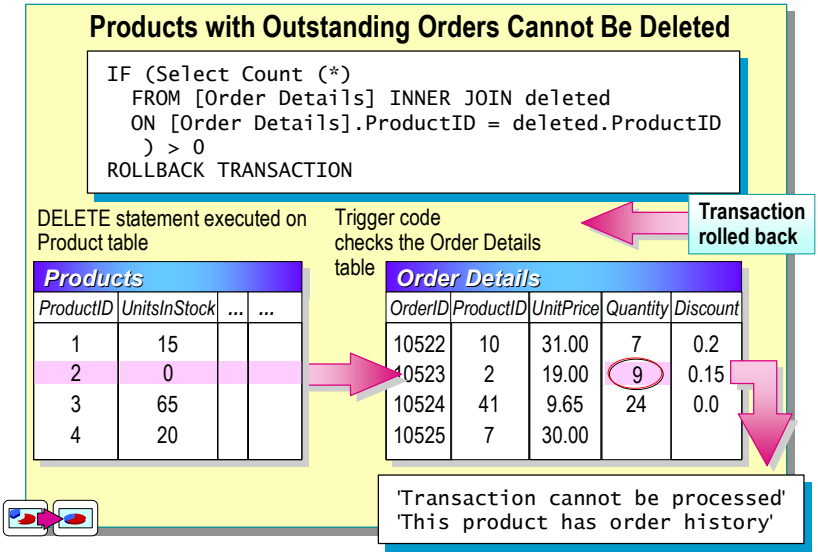
For Your Information
This example is hypothetical. There is no **BackOrders** table in the **Northwind** database.

```
CREATE TRIGGER BackOrderList_Delete
  ON Products FOR UPDATE
AS
IF (SELECT BO.ProductID FROM BackOrders AS BO JOIN
  Inserted AS I ON BO.ProductID = I.Product_ID
  ) > 0
BEGIN
  DELETE BO FROM BackOrders AS BO
  INNER JOIN Inserted AS I
  ON BO.ProductID = I.ProductID
END
```

Enforcing Business Rules

Topic Objective
To show an example of how to enforce business rules.

Lead-in
Triggers also can be used to enforce particular business rules, such as "Don't delete products that have order history."



You can use triggers to enforce business rules that are too complex for the CHECK constraint. This includes checking the status of rows in other tables.

For example, you may want to ensure that members' outstanding fines are paid before they are allowed to discontinue membership.

Example

This example creates a trigger that determines whether a product has order history. If it does, the DELETE is rolled back and the trigger returns a custom error message.

```
Use Northwind
GO
CREATE TRIGGER Product_Delete
  ON Products FOR DELETE
AS
IF (Select Count (*)
    FROM [Order Details] INNER JOIN deleted
    ON [Order Details].ProductID = Deleted.ProductID
    ) > 0
BEGIN
   RAISERROR('Transaction cannot be processed.\
              This product has order history.', 16, 1)
   ROLLBACK TRANSACTION
END
```

Performance Considerations

Topic Objective
To introduce performance considerations for using triggers.

Lead-in
You should consider these performance issues when using triggers.

- **Triggers Work Quickly Because the Inserted and Deleted Tables Are in Cache**
- **Execution Time Is Determined by:**
  - Number of tables that are referenced
  - Number of rows that are affected
- **Actions Contained in Triggers Implicitly Are Part of a Transaction**

You should consider the following performance issues when using triggers:

- Triggers work quickly because the **Inserted** and **Deleted** tables are in cache.

  The **Inserted** and **Deleted** tables are always in memory rather than on a disk, because they are logical tables and are usually very small.
- The number of tables referenced and the number of rows affected determines execution time.

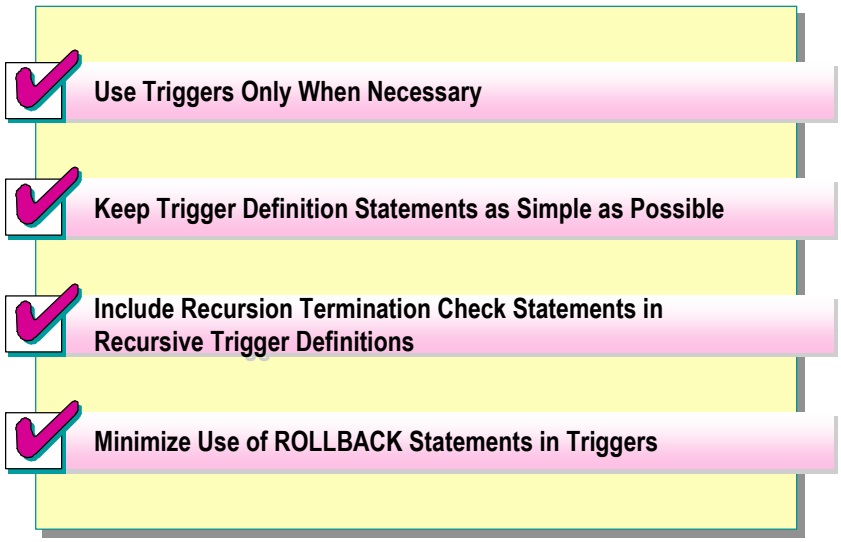
  Time that is spent invoking a trigger is minimal. The largest portion of execution time occurs as a result of referencing other tables (which may be either in memory or on a disk) and modifying data, if the trigger definition calls for it.
- Actions contained in triggers are implicitly part of a transaction.

  After a trigger is defined, the user action (INSERT, UPDATE, or DELETE statement) on the table that executes the trigger is always implicitly part of a transaction, along with the trigger itself. If a ROLLBACK TRANSACTION statement is encountered, the whole transaction rolls back. If any statements exist in the trigger script after the ROLLBACK TRANSACTION statement, those statements are executed. Therefore, it may be necessary to use a RETURN clause in an IF statement to prevent the processing of other statements.

Recommended Practices

Topic Objective
To present recommended practices for using triggers.

Lead-in
The following suggestions are recommended practices for using triggers.



The following recommended practices should help you manage your databases:

- Use triggers only when necessary. Consider a constraint before using a trigger.
- Keep trigger definition statements as simple as possible. Most of the time that is required to process a trigger is spent referencing tables and modifying data. Because triggers are an inherent transaction, locks are maintained until the transaction completes.
- Include recursion-termination check statements in recursive trigger definitions. This prevents the trigger from being stuck in an endless loop.
- Minimize the use of ROLLBACK statements in triggers. When you roll back a trigger, SQL Server must undo all of the actions that it performed up to that point.

Additional information on the following topics is available in SQL Server Books Online.

Topic	Search on
CREATE TRIGGER	“create trigger”
ALTER TRIGGER	“alter trigger”
DROP TRIGGER	“drop trigger”
Creating a trigger	“creating a trigger”
Programming triggers	“programming triggers”

Lab A: Creating Triggers

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will create triggers and test them.

**********ILLEGAL FOR NON-TRAINER USE**********

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Create triggers to maintain data integrity.
- Create triggers to enforce complex business rules.

Prerequisites

Before working on this lab, you must have:

- Script files for this lab, which are located in C:\Moc\2073A\Labfiles\L11.
- Answer files for this lab, which are located in C:\Moc\2073A\Labfiles\L11\Answers.

Lab Setup

To complete this lab, you must have either:

- Completed the prior lab, or
- Executed the C:\Moc\2073A\Batches\Restore11.cmd batch file.

  This command file restores the **ClassNorthwind** database to a state required for this lab.

For More Information

If you require help in executing files, search SQL Query Analyzer Help for "Execute a query".

Other resources that you can use include:

- The **Northwind** database schema.
- Microsoft SQL Server Books Online.

Scenario

The organization of the classroom is meant to simulate that of a worldwide trading firm named Northwind Traders. Its fictitious domain name is nwtraders.msft. The primary DNS server for nwtraders.msft is the instructor computer, which has an Internet Protocol (IP) address of 192.168.*x*.200 (where *x* is the assigned classroom number). The name of the instructor computer is London.

The following table provides the user name, computer name, and IP address for each student computer in the fictitious **nwtraders.msft** domain. Find the user name for your computer, and make a note of it.

User name	Computer name	IP address
SQLAdmin1	Vancouver	192.168.*x*.1
SQLAdmin2	Denver	192.168.*x*.2
SQLAdmin3	Perth	192.168.*x*.3
SQLAdmin4	Brisbane	192.168.*x*.4
SQLAdmin5	Lisbon	192.168.*x*.5
SQLAdmin6	Bonn	192.168.*x*.6
SQLAdmin7	Lima	192.168.*x*.7
SQLAdmin8	Santiago	192.168.*x*.8
SQLAdmin9	Bangalore	192.168.*x*.9
SQLAdmin10	Singapore	192.168.*x*.10
SQLAdmin11	Casablanca	192.168.*x*.11
SQLAdmin12	Tunis	192.168.*x*.12
SQLAdmin13	Acapulco	192.168.*x*.13
SQLAdmin14	Miami	192.168.*x*.14
SQLAdmin15	Auckland	192.168.*x*.15
SQLAdmin16	Suva	192.168.*x*.16
SQLAdmin17	Stockholm	192.168.*x*.17
SQLAdmin18	Moscow	192.168.*x*.18
SQLAdmin19	Caracas	192.168.*x*.19
SQLAdmin20	Montevideo	192.168.*x*.20
SQLAdmin21	Manila	192.168.*x*.21
SQLAdmin22	Tokyo	192.168.*x*.22
SQLAdmin23	Khartoum	192.168.*x*.23
SQLAdmin24	Nairobi	192.168.*x*.24

Estimated time to complete this lab: 30 minutes

Exercise 1
Creating Triggers

In this exercise, you will execute a script that creates a trigger. After the trigger is created, you will test it to verify that it works.

► To create a trigger

In this procedure, you will create a trigger by executing a script file.

1. Log on to the **NWTraders** classroom domain by using the information in the following table.

Option	Value
User name	**SQLAdmin*x*** (where *x* corresponds to your computer name as designated in the **nwtraders.msft** classroom domain)
Password	**password**

2. Open SQL Query Analyzer and, if requested, log in to the (local) server with Microsoft Windows® Authentication.

   You have permission to log in to and administer SQL Server because you are logged as **SQLAdmin*x***, which is a member of the Microsoft Windows 2000 local group, Administrators. All members of this group are automatically mapped to the SQL Server **sysadmin** role.

3. In the **DB** list, click **ClassNorthwind**.
4. Open C:\Moc\2073A\Labfiles\L11\OrdDetInsert.sql and review it.

   This script creates a trigger on the **Order Details** table. This trigger updates the **UnitsInStock** column of the **Products** table whenever a row is inserted into the **Order Details** table (whenever an order is received).

5. Execute C:\Moc\2073A\Labfiles\L11\OrdDetInsert.sql.
6. Execute the **sp_helptrigger** system stored procedure on the **Order Details** table in the **ClassNorthwind** database to determine whether the trigger was created.

► To test the trigger

In this procedure, you will test the trigger that you just created by inserting a row into the **Order Details** table.

1. Execute the following SELECT statement to select a row from the **Products** table to determine the units of a product that are in stock:

   ```
   SELECT * FROM Products WHERE ProductID = 22
   ```

   The **UnitsInStock** column should contain the value of 104. If the **UnitsInStock** column displays a different number, make a note of it.

2. Insert a row into the **Order Details** table ordering 50 units of product 22. Your INSERT statement will be similar to the following:

   ```
   INSERT [Order Details]
   (OrderID, ProductID, UnitPrice, Quantity, Discount)
   VALUES (11077, 22, 21.00, 50, 0.0)
   GO
   ```

3. Query the **Products** table to verify that the **UnitsInStock** column value for the specific **ProductID** has changed to 54.

Exercise 2
Creating a Trigger for Updating Derived Data

In this exercise, you will create two new tables (without PRIMARY and FOREIGN KEY constraints) and then a trigger on the **NewCategories** table to enforce integrity in the **NewProducts.Discontinued** column. C:\Moc\2073A\Labfiles\L11\Answers\CategoryDelete.sql is a completed script for this exercise.

► **To create a trigger that updates derived data**

In this procedure, you will create two new tables called **NewCategories** and **NewProducts.** Neither have the PRIMARY and FOREIGN KEY constraints of the **Categories** and **Products** tables. You will create a trigger on the **NewCategories** table. This trigger updates the **Discontinued** column in the **NewProducts** table whenever a category is deleted (whenever a record is deleted from the **NewCategories** table). All affected products are marked as 1, indicating that they are discontinued. Use C:\Moc\2073A\Labfiles\L11\CategoryDelete.sql and make the appropriate changes.

1. Type and execute the following query to create two new tables called **NewCategories** and **NewProducts**.

```
USE ClassNorthwind
GO
--This creates a NewCategories table
SELECT * INTO NewCategories FROM Categories
--This creates a NewProducts table
SELECT * INTO NewProducts FROM Products
GO
```

2. Type and execute the following query to create a trigger on the **NewCategories** table. This trigger updates the **Discontinued** column of the **NewProducts** table to 1 when a product's parent category is deleted (whenever a row is deleted from the **NewCategories** table).

```
CREATE TRIGGER Category_Delete
   ON NewCategories
   FOR DELETE
AS
   UPDATE P SET Discontinued = 1
   FROM NewProducts AS P INNER JOIN Deleted AS d
   ON P.CategoryID = D.CategoryID
```

3. Type and execute the following SELECT statement that queries the **NewProducts** table to determine the discontinued value of the products in **CategoryID 7**.

```
SELECT ProductID, CategoryID, Discontinued
FROM NewProducts WHERE CategoryID = 7
```

4. Write a DELETE statement that removes a row from the **NewCategories** table, and then verify that the trigger executes correctly.

   Are these triggers necessary to maintain data integrity in the **ClassNorthwind** database? Why or why not?

   Yes. Cascading referential integrity could remove the related products from the NewProducts table, but a trigger is the best way to implement a more complex action, such as leaving the records in NewProducts while updating the Discontinued column.

_____

_____

Exercise 3
Creating a Trigger That Maintains a Complex Business Rule

In this exercise, you will create a DELETE trigger on the **NewProducts** table. This trigger determines whether an order history exists in the **Order Details** table before the trigger permits a deletion from the **NewProducts** table.

► To create a trigger for the loan table

In this procedure, you will use the **NewProducts** table created in the previous exercise. You will create a trigger that determines whether an order history exists for a product that is being deleted. If the product has never been ordered, then the product can be deleted. If the product has a history of orders, then the delete from the product table is rolled back, and the trigger returns a custom error message. C:\Moc\2073A\Labfiles\L11\Answers\BusinessRule.sql is a completed script for this exercise.

1. Create a DELETE trigger on the **NewProducts** table that determines whether an order history exists for a product that is possibly being deleted. If records exist in the **Order Details** table for that product, then display a message and roll back the trigger.
2. Delete product 6 from the **NewProducts** table to test the trigger. Answers\BusinessRule.sql is a completed script for this step.

   Did the trigger fire? Why or why not?

   Yes. The product that was deleted from the NewProducts table contained orders in the Order Details table. The trigger prevented the delete and returned an error message.

   _____

   _____

Exercise 4
Testing the Firing Order of Constraints and Triggers

In this exercise, you will modify the statement from the previous exercise to test the firing order of constraints and triggers.
C:\Moc\2073A\Labfiles\L11\Answers\BusinessRule2.sql is a completed script for this exercise.

► **To modify the trigger from the previous exercise**

1. Create a trigger similar to that used in the previous exercise called **Product_Delete2** on the **Products** table.

   Remember that the previous exercise created a DELETE trigger called **Product_Delete** on the **NewProducts** table.

```
CREATE TRIGGER Product_Delete2
  ON Products FOR DELETE
AS
IF (Select Count (*)
    FROM [Order Details] INNER JOIN deleted
    ON [Order Details].ProductID = Deleted.ProductID
    ) > 0
BEGIN
   RAISERROR('Transaction cannot be processed. This Product
still has a history of orders.', 16, 1)
   ROLLBACK TRANSACTION
END
```

2. Test the trigger.

   Did the trigger fire? Why or why not?

   No. The PRIMARY KEY constraint prevented this trigger from firing.

   _____

   _____

Review

Topic Objective
To reinforce module objectives by reviewing key points.

Lead-in
The review questions cover the key concepts taught in the module.

- Introduction to Triggers
- Defining Triggers
- How Triggers Work
- Examples of Triggers
- Performance Considerations

**********ILLEGAL FOR NON-TRAINER USE**********

Use these questions to review module topics.

Ask students whether they have any questions before continuing.

1. If the inventory manager does not provide the **Products.ProductID** column value in the INSERT statement, what characteristics must exist in the column definition?

   The column must allow NULLs, or if it does not allow NULLs, it must contain a DEFAULT constraint.

2. If the **Products** table contains a PRIMARY KEY constraint on the **ProductID** column, would a trigger work? Why or why not?

   No. Constraints are processed before data is modified (inserted). A PRIMARY KEY constraint does not allow NULLs, so the INSERT statement would fail.

3. What must you do to make a trigger work?

   You could use a default that would place a temporary ProductID number as a placeholder, and then let the trigger assign the correct value. The temporary ProductID number would need to be a value outside the possible range of ProductID numbers. For example, 9999999 would be a good copy number because it is unlikely that this inventory would have more than 9,999,999 products.

   Another option would be to drop the PRIMARY KEY constraint. You would have to replace it by creating an additional trigger to maintain and check referential integrity for the ProductID column. You would possibly have to create a unique index on the ProductID column, as well.

   _____

   _____