**Triggers - Lecture Notes**

Explain triggers as a special kind of stored procedure. Demonstrate how to create a trigger.

Be sure the student knows about the **inserted** and **deleted** internal and temporary tables; and that they can (beginning with SQL Server 2005) be accessed outside of a trigger (covered in a later lesson).

|  |  |  |
| --- | --- | --- |
| **Operation** | ***deleted* Table** | ***inserted* Table** |
| INSERT | (not used) | Contains the rows being inserted |
| DELETE | Contains the rows being deleted | (not used) |
| UPDATE | Contains the rows as they were before the UPDATE statement | Contains the rows as they were after the UPDATE statement |

Explain the sequence of events for AFTER and INSTEAD OF triggers, nested triggers, and recursive triggers.

After is the default type of trigger unless you manually specify that’s an Instead of trigger.

Look at AdventureWorks2012.Person.Person table and iuPerson Trigger

Know the difference between DML and DDL Triggers. Know where the different types are stored, know that DML Triggers are stored on the Table Level, while DDL Triggers are stored on the Database Programmability Level.

**Disable Nesting/Recursing Triggers**  
Following script will stop executing all the nested triggers.  
**sp\_CONFIGURE 'nested\_triggers',0  
GO  
RECONFIGURE  
GO**

**ALTER DATABASE databasename  
SET RECURSIVE\_TRIGGERS ON | OFF**

**CREATE TRIGGER trg\_Customers\_Upd**

**ON Customers**

**FOR UPDATE**

**AS**

**begin**

**SELECT \* FROM INSERTED *-- show data in INSERTED logical table***

**SELECT \* FROM DELETED *-- show data in DELETED logical table***

**end**

CREATE TRIGGER dbo.InvUpdate ON dbo.Orders FOR INSERT

AS BEGIN

UPDATE p

SET p.instock = (p.instock - i.qty)

FROM Products p JOIN inserted i ON p.prodid = i.prodid

END

GO

CREATE TRIGGER tr\_OrderItems\_LastUpdateDate ON Orders AFTER UPDATE  
AS  
  
UPDATE Orders  
SET LastUpdateDate = GetDate()  
FROM Orders o  
INNER JOIN Inserted i  
 ON o.OrderId = i.OrderId  
 AND o.ItemId = i.ItemId  
  
GO

CREATE TRIGGER dbo.AZDel ON dbo.Customers FOR DELETE AS BEGIN IF (SELECT

state FROM deleted) = 'AZ' BEGIN PRINT 'Cannot remove customers from AZ'

PRINT 'Transaction has been cancelled' ROLLBACK END END

GO

**Functions - Lecture Notes**

Have both you and your students display Books Online on their monitors. Find the list of functions by finding any one and then pressing the synchronize button. Scroll through the list just to overwhelm everyone with how much there is to learn. Randomly open different functions; scroll to the bottom of the text and, together, examine some examples. Quit when everyone has had enough; when everyone has gone into overload.

Main Concept - encapsulate reusable logic

* The types of user-defined functions (UDFs) that SQL Server supports, both scalar (which return a single value) and table-valued (which return a table), and how to use them.
* Some of the more interesting built-in functions
* How and why functions can get you into trouble, and cause terrible performance, if you're not careful about how you use them.

SQL aggregate functions return a single value, calculated from values in a column.

SQL scalar functions return a single value, based on the input value.

CREATE FUNCTION [schema\_name.]function\_name

    ([@parameter\_name data\_type [= default]] [, ...])

    RETURNS TABLE

    [WITH [ENCRYPTION] [, SCHEMABINDING]]

    [AS]

RETURN [(] select\_statement [)]

DECLARE @A varchar(2)

DECLARE @B varchar(2)

DECLARE @C varchar(2)

set @A=25

set @B=15

set @C=33

Select CAST(@A as int) + CAST(@B as int) +CAST (@C as int) as Result

select convert(varchar(20),GETDATE(),108)

SELECT CONVERT(VARCHAR(15),GETDATE(),6)

go

SELECT CONVERT(VARCHAR(16),GETDATE(),106)

go

SELECT CONVERT(VARCHAR(24),GETDATE(),113)

Select db\_name() as CurrentDB

Select db\_name(1) as dbname

Select database\_id, name from sys.databases

CREATE FUNCTION fn\_Zip

CREATE FUNCTION fn\_FullName

CREATE FUNCTION fn\_TotalValue

**USE AdventureWorks2012  
GO  
SELECT  CustomerID, COUNT(\*) AS OrderCount,  
CHOOSE(COUNT(\*), 'Bronze','Silver','Gold','Platinum') AS MemberLevel,  
AVG(TotalDue) AS AvgOrder,  
IIF(AVG(TotalDue) > 1000, 'High','Low') AS Priority  
FROM Sales.SalesOrderHeader  
WHERE CustomerID IN (11060, 11062, 11065, 11068, 11070, 12165)  
GROUP BY CustomerID;**

**Stored Procedures - Lecture Notes**

Demonstrate by HELP\_TEXT code on the classroom display device. This system stored procedure exemplifies good technique and most available features usable in procedures.

USE [AdventureWorks2012]

GO

-- Method #1

SELECT ROUTINE\_DEFINITION, ROUTINE\_NAME, ROUTINE\_TYPE

FROM INFORMATION\_SCHEMA.ROUTINES

WHERE ROUTINE\_NAME = 'ufnGetContactInformation';

GO

-- Method #2

select c.text, object\_name(c.id), o.type

from sys.syscomments c

join sys.sysobjects o

on o.id = c.id

where c.id = object\_ID('ufnGetContactInformation');

GO

<p>-- Method #3

exec sp\_helptext 'dbo.ufnGetContactInformation';

GO

Before you create a stored procedure you need to know what your end result is, whether you are selecting data, inserting data, etc..

In this simple example we will just select all data from the Person.Address table that is stored in the AdventureWorks database.

So the simple T-SQL code would be as follows which will return all rows from this table.

|  |
| --- |
| SELECT \* FROM AdventureWorks.Person.Address |

To create a stored procedure to do this the code would look like this:

|  |
| --- |
| CREATE PROCEDURE uspGetAddress  AS  SELECT \* FROM AdventureWorks.Person.Address  GO |

To call the procedure to return the contents from the table specified, the code would be:

|  |
| --- |
| EXEC uspGetAddress  --or just simply  uspGetAddress |

When creating a stored procedure you can either use CREATE PROCEDURE or CREATE PROC.  After the stored procedure name you need to use the keyword "AS" and then the rest is just the regular SQL code that you would normally execute.

On thing to note is that you cannot use the keyword "GO" in the stored procedure.  Once the SQL Server compiler sees "GO" it assumes it is the end of the batch.

Just like you have the ability to use parameters with your SQL code you can also setup your stored procedures to accept one or more parameter values.

**One Parameter**

In this example we will query the Person.Address table from the AdventureWorks database, but instead of getting back all records we will limit it to just a particular city.  This example assumes there will be an exact match on the City value that is passed.

|  |
| --- |
| CREATE PROCEDURE uspGetAddress @City nvarchar(30)  AS  SELECT \*  FROM AdventureWorks.Person.Address  WHERE City = @City  GO |

To call this stored procedure we would execute it as follows:

|  |
| --- |
| EXEC uspGetAddress @City = 'New York' |

We can also do the same thing, but allow the users to give us a starting point to search the data.  Here we can change the "=" to a LIKE and use the "%" wildcard.

|  |
| --- |
| CREATE PROCEDURE uspGetAddress @City nvarchar(30)  AS  SELECT \*  FROM AdventureWorks.Person.Address  WHERE City LIKE @City + '%'  GO |