Microsoft .Net - C# - Customized

Microsoft® Technology Courses

Introduction

Microsoft .NET Framework
Microsoft Visual Studio
C# Programming

Microsoft

Day7 – Session 1

Programming with T-SQL

Programming with T-SQL

- Although T-SQL is not a general-purpose programming language, it does support programming constructs such as variables and conditional processing.
- All of the SQL statements used thus far have been single standalone statements:
- SELECT statements to retrieve data, ALTER TABLE statements to make table changes, and so on.
- But some data-retrieval tasks are more complex, often involving multiple statements, conditional processing, and mid-process data manipulation.

3

Understanding T-SQL Programming

- As such, it is worthwhile to briefly examine these capabilities, specifically the following:
  - Using variables
  - Performing conditional processing
  - Repeating processes (looping)
  - Working with Stored Procedures
  - Using Cursors
  - Using Triggers
  - Managing Transaction Processing
  - Exception Handling Basics

4

Using Variables

- In computer programming, a variable is a named object that stores values.
- Although variable use and capabilities vary from one programming language to the next, the basic ability to define variables and store values in them for subsequent use is pretty universal.
- Variables are declared in the body of a batch or procedure with the DECLARE statement and are assigned values by using either a SET or SELECT statement.

5

Rules and Requirements

- T-SQL variables have specific rules and requirements:
  - All T-SQL variables names must start with @, local variables are prefixed with @
  - Global variables, which are used extensively by SQL Server itself and generally should not be used for your own purposes, are prefixed with @@.
  - T-SQL variables must be declared using the DECLARE statement.
  - When a variable is declared, its datatype must be specified.

6

Presented by
Ranjan Bhatnagar

Few More

- Multiple variables may be defined using individual DECLARE statements, or they may be comma-delimited in a single DECLARE statement.
- There is no way to "undeclare" variables. They remain present until they go out of scope. For local variables, this means they'll exist until the process completes.

7

Declaring Variables

- The following example demonstrates the use of DECLARE:

```
DECLARE @age INT;
DECLARE @firstName CHAR(20), @lastName CHAR(20);
```

- The first DECLARE statement declares a single variable named @age of type INT (an integer, a numeric value), and the second declares two variables of type CHAR(20)
- Notice that the second DECLARE statement requires a comma between the variables being declared.

8

Assigning Values to Variables

- When variables are first declared, they contain no values (actually, they contain NULL).
- To assign values to variables, you use the SET statement, as shown here:

```
DECLARE @age INT;
DECLARE @firstName CHAR(20), @lastName CHAR(20);
SET @lastName='Forta';
SET @firstName='Ben';
SET @age=21;
```

- This example declares the same three variables as seen previously, and then uses three SET statements to assign values to those variables

9

SET or SELECT?

- You can also use SELECT to assign values to variables, as shown here:

```
SELECT @age=21;
```

- There is one important difference between using SET or SELECT to assign variable values.
- SET only sets a single variable, and to assign values to multiple variables you must use multiple SET statements.
- SELECT, on the other hand, can be used to assign values to multiple variables in a single statement.

10

Viewing Variable Contents

- When using variables, it is often necessary to inspect their contents. The simplest way to view variable contents is to output them, and there are two ways to do this. SELECT can be used to retrieve variable values, as shown here:

```
DECLARE @age INT;
DECLARE @firstName CHAR(20), @lastName CHAR(20);
SET @lastName='Forta';
SET @firstName='Ben';
SET @age=21;
SELECT @lastName, @firstName, @age
```

11

Use Aliases

- T-SQL also supports the PRINT statement, which is used to display messages along with any returned results. Here is an example:

```
DECLARE @age INT;
DECLARE @firstName CHAR(20), @lastName CHAR(20);
SET @lastName='Forta';
SET @firstName='Ben';
SET @age=21;
PRINT @lastName + ', ' + @firstName;
PRINT @age;
```

- PRINT simply outputs text. In this example, the first line is a string made up of three strings, a variable, static (fixed) text, and then another variable. The second line prints a numeric variable.

12

Presented by
Ranjan Bhatnagar

Converting Variables to Strings

- What if you wanted to display the age output as Age: 21?
- You'd need to concatenate the string Age: with the variable @age, but the simple + concatenation operation used to join the strings together would fail because @age is a number, not a string.
- To get around this problem, you would need to convert @age to a string so as to be able to concatenate it, like this:

```
PRINT 'Age: ' + Convert(CHAR, @age);
```

13

Using Variables in T-SQL Statements

- Now that you know how to declare, populate, and view the contents of variables, let's look at a practical example of how variables could be used.
- Suppose you need to run two queries, one to return customer information for a specific customer and another to return orders placed by that customer.
- This requires two SELECT statements, as shown here:

```
SELECT cust_name, cust_email FROM customers
WHERE cust_id = 10001;
SELECT order_num, order_date FROM orders
WHERE cust_id = 10001 ORDER BY order_date;
```

- This batch-processing example is pretty self-explanatory; two SELECT statements are used, so two sets of results are returned.

14

Alternative

- Alternative to the preceding example would be to only define the customer ID once, thus only requiring one change to perform a different search. Look at this example:

```
-- Define @cust_id
DECLARE @cust_id INT;
SET @cust_id = 10001;
-- Get customer name and e-mail
SELECT cust_name, cust_email FROM customers
WHERE cust_id = @cust_id;
-- Get customer order history
SELECT order_num, order_date FROM orders
WHERE cust_id = @cust_id ORDER BY order_date;
```

15

Comment Your Code

- You may have noticed that the preceding example contains lines of code beginning with double hyphens (--).
- These are comments (messages included in your SQL code that are ignored by SQL Server), and they help explain what the code is doing.
- As the complexity of SQL statements increases, it is invaluable to be able to read embedded comments the next time someone has to understand what was done and why.

16

Using Conditional Processing

- Conditional processing is a way to make decisions within programming code, performing some action based on the decision made.
- Like most other programming languages, T-SQL allows developers to write code where decisions are made at runtime, and what makes this work is the IF statement.
- Let's start with a basic example. Imagine you are writing a SQL statement that has to process open orders (perhaps updating values, or copying rows to another table).
- This SQL code would need to run regularly, but what it does might differ based on whether today is a weekday (and thus a day when you are open for business and processing orders) or part of the weekend (when your are not processing orders).

17

Simple Example

- The following is a simple IF statement that sets a variable to either 0 or 1, based on whether or not today is Sunday:

```
-- Define variables
DECLARE @open BIT
-- Open for business today?
IF DatePart(dw, GetDate()) = 1
SET @open = 0
ELSE
SET @open = 1
-- Output
SELECT @open AS OpenForBusiness
```

18

Presented by
Ranjan Bhatnagar

Some Error

- There is a flaw in this code because it only checks to see if today is Sunday. As such, if today were Saturday, @open would mistakenly be set to 1. To fix this we need to add an OR to the IF statement:

```
-- Define variables
DECLARE @dow INT
DECLARE @open BIT
-- Get the day of week
SET @dow = DatePart(dw, GetDate());
-- Open for business today?
IF @dow = 1 OR @dow = 7
    SET @open = 0
ELSE
    SET @open = 1
-- Output
SELECT @open AS OpenForBusiness
```

19

Grouping Statements

- In the previous examples, a single statement was processed if the IF or ELSE conditions were met. But what would happen if you had to execute multiple statements? Look at this example:

```
-- Define variables
DECLARE @dow INT
DECLARE @open BIT, @process
BIT
-- Get the day of week
SET @dow = DatePart(dw,
GetDate());
-- Open for business today?
IF @dow = 1 OR @dow = 7
BEGIN
SET @open = 0
SET @process = 0
END
ELSE
BEGIN
SET @open = 1
SET @process = 1
END
```

20

Case Statement

- In SQL Server (Transact-SQL), the CASE statement has the functionality of an IF-THEN-ELSE statement. You can use the CASE statement within a SQL statement.

```
DECLARE @t INT=1    -- Try assigning 2, -5, 45
SELECT CASE
        WHEN @t>0 THEN
          CASE
            WHEN @t=1 THEN 'One'
            WHEN @t=2 THEN 'Two'
            ELSE 'not a Number required'
          END
        ELSE 'less than one'
      END
```

21

Using Looping

- SQL statements are processed sequentially, one at a time, and each being processed once.
- Like other programming languages, T-SQL supports looping, the ability to repeat a block of code as needed.
- In T-SQL, looping is accomplished using the WHILE statement.

```
DECLARE @counter INT
SET @counter=1
WHILE @counter <= 10
BEGIN
PRINT @counter
SET @counter=@counter+1
END
```

22

WHILE and BEGIN/END

- Just like IF, WHILE repeats only the single statement that follows it.
- To repeat multiple lines of code, use BEGIN and END to delimit that code block, as shown in the previous example.
- Two other statements are frequently used in conjunction with WHILE:
  - BREAK immediately exits the current WHILE loop (or IF).
  - CONTINUE restarts processing at the top of the loop.

23

Understanding Stored Procedures

- Not all operations are that simple; often, multiple statements will be needed to perform a complete operation. For example, consider the following scenario:
  - To process an order, checks must be made to ensure that items are in stock.
  - If items are in stock, they need to be reserved so they are not sold to anyone else, and the available quantity must be reduced to reflect the correct amount in stock.
  - Any items not in stock need to be ordered; this requires some interaction with the vendor.
  - The customer needs to be notified as to which items are in stock (and can be shipped immediately) and which are backordered.

24

Presented by
Ranjan Bhatnagar

Why Use Stored Procedures?

- To simplify complex operations by encapsulating processes into a single easy-to-use unit.
- To ensure data integrity by not requiring that a series of steps be created over and over.
- To simplify change management. If tables, column names, or business logic (or just about anything) changes, only the stored procedure code needs to be updated, and no one else will need even to be aware that changes were made.
- Stored procedures typically execute quicker than individual SQL statements.
- Stored procedures can use these to create code that is more powerful and flexible.

25

Downside

- Before you run off to turn all your SQL code into stored procedures, here's the downside:
  - Stored procedures tend to be more complex to write than basic SQL statements, and writing them requires a greater degree of skill and experience.
  - You might not have the security access needed to create stored procedures. Many database administrators restrict stored procedure– creation rights, allowing users to execute them but not necessarily create them.
- Nonetheless, stored procedures are very useful and should be used whenever possible.

26

Creating Stored Procedures

- As already explained, writing a stored procedure is not trivial. To give you a taste for what is involved, let's look at a simple example, a stored procedure that returns the average product price. Here is the code:

```
CREATE PROCEDURE productpricing AS
BEGIN
SELECT Avg(prod_price) AS priceaverage
FROM products;
END;
```

- When SQL Server processes this code, it creates a new stored procedure named productpricing.
- No data is returned because the code does not call the stored procedure; it simply creates the code for future use.

27

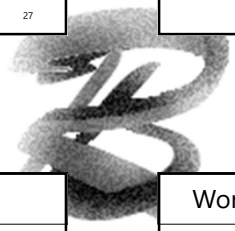
Executing Stored Procedures

- SQL Server procedures are executed using the EXECUTE statement.
- EXECUTE takes the name of the stored procedure and any parameters (if required) that need to be passed to it.

```
EXECUTE productpricing;
```

- Here, a stored procedure named productpricing is executed and displays the returned result.;
- EXECUTE may be shortened to EXEC, but both EXECUTE and EXEC do the exact same thing.

28



Dropping Stored Procedures

- After Stored Procedures are created, they remain on the server, ready for use, until dropped.
- The DROP command removes the stored procedure from the server.
- To remove the stored procedure we just created, use the following statement:

```
DROP PROCEDURE productpricing;
```

- This drops (deletes) the just-created stored procedure.

29

Working with Parameters

- The productpricing is a really simple stored procedure; it simply displays the results of a SELECT statement.
- Typically stored procedures do not display results; rather, they return them to variables that you specify.
- Here is an updated version of productpricing (you'll not be able to create the stored procedure again if you did not previously drop it).

30

Presented by
Ranjan Bhatnagar

Updated Procedure

```
CREATE PROCEDURE productpricing
@price_min MONEY OUTPUT,
@price_max MONEY OUTPUT,
@price_avg MONEY OUTPUT
AS
BEGIN
SELECT @price_min = Min(prod_price)
FROM products;
SELECT @price_max = Max(prod_price)
FROM products;
SELECT @price_avg = Avg(prod_price)
FROM products;
END;
```

31

Executing Procedure

- To call this updated stored procedure, we must specify three variable names, as shown here:

```
DECLARE @cheap MONEY
DECLARE @expensive MONEY
DECLARE @average MONEY
EXECUTE productpricing @cheap OUTPUT,
@expensive OUTPUT,
@average OUTPUT
SELECT @cheap, @expensive, @average;
```

32

Another Example

- Here is another example. This time the example passes parameters to the stored procedure as well as returns OUTPUT parameters.
- Ordertotal accepts an order number and returns the total for that order:

```
CREATE PROCEDURE ordertotal
@order_num INT, @order_total MONEY OUTPUT
AS
BEGIN
SELECT @order_total = Sum(item_price*quantity)
FROM orderitems
WHERE order_num = @order_num;
END;
```

33

Invoke Procedure

- To invoke this new stored procedure, you can use the following:

```
DECLARE @order_total MONEY
EXECUTE ordertotal 20005, @order_total OUTPUT
SELECT @order_total
```

- Two parameters must be passed to ordertotal; the first is the order number and the second is the name of the variable that will contain the calculated total.
- Here, one of those parameters (the order number) is a static value (not a variable).

34

More Intelligent Stored Procedures

- The real power of stored procedures is realized when business rules and intelligent processing are included within them.
- Consider this scenario: You need to obtain order totals as before, but also need to add sales tax to the total, but only for some customers (perhaps the ones in your own state).
- Now you need to do several things:
  - Obtain the total (as before).
  - Conditionally add tax to the total.
  - Return the total (with or without tax).
- That's a perfect job for a stored procedure:

35

Code

```
-- Name: ordertotal
-- Parameters: @order_num = order number
-- @taxable = 0 if not taxable, 1 if taxable
-- @order_total = order total variable
CREATE PROCEDURE ordertotal
@order_num INT,
@taxable BIT,
@order_total MONEY OUTPUT
AS
BEGIN
-- Declare variable for total
DECLARE @total MONEY;
-- Declare tax percentage
DECLARE @taxrate INT;
-- Set tax rate (adjust as needed)
SET @taxrate = 6;
-- Get the order total
SELECT @total =
Sum(item_price*quantity)
FROM orderitems
WHERE order_num = @order_num
-- Is this taxable?
IF @taxable = 1
-- Yes, so add taxrate to the total
SET
@total=@total+(@total/100*@taxrate
);
-- And finally, save to output variable
SELECT @order_total = @total;
END;
```

36

Presented by
Ranjan Bhatnagar

Try Executing

- This is obviously a more sophisticated and powerful stored procedure.
- To try it out, use the following two statements:
- First Execution

```
DECLARE @order_total MONEY
EXECUTE ordertotal 20005, 0, @order_total OUTPUT
SELECT @order_total
```

- Second Execution

```
DECLARE @order_total MONEY
EXECUTE ordertotal 20005, 1, @order_total OUTPUT
SELECT @order_total
```

37

Using Cursors

- As you have seen in previous examples, T-SQL retrieval operations work with sets of rows known as result sets.
- The rows returned are all the rows that match a SQL statement, zero or more of them.
- Using simple SELECT statements, there is no way to get the first row, the next row, or the previous 10 rows, for example.
- Nor is there an easy way to process all rows, one at a time (as opposed to all of them in a batch).

38

That's Why Cursors!

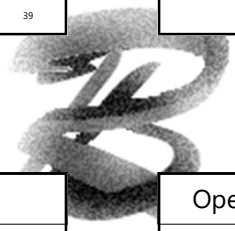
- Sometimes there is a need to step through rows forward or backward, and one or more at a time.
- This is what cursors are used for.
- A cursor is a database query stored in SQL Server, not a SELECT statement, but the result set retrieved by that statement.
- Once the cursor is stored, applications can scroll or browse up and down through the data as needed.
- Cursors are used primarily by interactive applications in which users need to scroll up and down through screens of data, browsing or making changes.

39

Working with Cursors

- Using cursors involves several distinct steps:
  - Before a cursor can be used, it must be declared (defined). This process does not actually retrieve any data; it merely defines the SELECT statement to be used.
  - After it is declared, the cursor must be opened for use. This process actually retrieves the data using the previously defined SELECT statement.
  - With the cursor populated with data, individual rows can be fetched (retrieved) as needed.
  - Once the desired data has been fetched, the cursor must be closed.
  - Finally, the cursor must be removed.

40



Example - Cursor

- For example, this statement defines a cursor named orders_cursor using a SELECT statement that retrieves all order numbers:

```
DECLARE orders_cursor CURSOR
FOR
SELECT order_num FROM orders ORDER BY order_num;
DEALLOCATE orders_cursor;
```

- A DECLARE statement is used to define and name the cursor, in this case, orders_cursor.
- Nothing is done with the cursor, and it is immediately removed using DEALLOCATE.

41

Opening and Closing Cursors

OPEN orders_cursor;

- When the OPEN statement is processed, the query is executed, and the retrieved data is stored for subsequent browsing and scrolling.
- After cursor processing is complete, the cursor should be closed using the CLOSE statement, as follows:

CLOSE orders_cursor;

- CLOSE frees up internal memory and resources used by the cursor, so every cursor should be closed when it is no longer needed.

42

Presented by
Ranjan Bhatnagar

Better Example

- After a cursor is closed, it cannot be reused without being opened again.
- However, a cursor does not need to be declared again to be used; an OPEN statement is sufficient (so long as the cursor has not been removed with DEALLOCATE).
- Here is an updated version of the previous example:

```
-- Define the cursor
DECLARE orders_cursor CURSOR
FOR
SELECT order_num FROM orders ORDER BY order_num;
-- Open cursor (retrieve data)
OPEN orders_cursor;
-- Close cursor
CLOSE orders_cursor
-- And finally, remove it
DEALLOCATE orders_cursor;
```

43

Using Cursor Data

- After a cursor is opened, you can access each row individually using a FETCH statement.
- FETCH specifies the cursor to be used and where retrieved data should be stored. It also advances the internal row pointer within the cursor so the next FETCH statement will retrieve the next row (and not the same one over and over).
- The first example given in next slide retrieves a single row from the cursor (the first row):

44

Example Code

```
-- Local variables
DECLARE @order_num INT;
-- Define the cursor
DECLARE orders_cursor CURSOR
FOR
SELECT order_num FROM orders ORDER BY order_num;
-- Open cursor (retrieve data)
OPEN orders_cursor;
-- Perform the first fetch (get first row)
FETCH NEXT FROM orders_cursor INTO @order_num;
-- Close cursor
CLOSE orders_cursor
-- And finally, remove it
DEALLOCATE orders_cursor;
```

45

What to Fetch?

- The FETCH statements in this example use FETCH NEXT to fetch the next row.
- This is the most frequently used FETCH, but other FETCH options are available.
- These include FETCH PRIOR to retrieve the previous row, FETCH FIRST and FETCH LAST to retrieve the first and last rows, respectively, FETCH ABSOLUTE to fetch a specific row number starting from the top, and FETCH RELATIVE to fetch a specific row number starting from the current row.

46

Fetch with Loop

- In the example, the retrieved data is looped through from the first row to the last:

```
-- Local variables
DECLARE @order_num INT;
-- Define the cursor
DECLARE orders_cursor CURSOR
FOR
SELECT order_num FROM orders
ORDER BY order_num;
-- Open cursor (retrieve data)
OPEN orders_cursor;
-- Perform the first fetch (get first
row)
FETCH NEXT FROM orders_cursor
INTO @order_num;
-- Check @@FETCH_STATUS to see if
there are any more rows
-- to fetch.
WHILE @@FETCH_STATUS = 0
BEGIN
-- This is executed as long as the
previous fetch succeeds.
FETCH NEXT FROM orders_cursor
INTO @order_num;
END
-- Close cursor
CLOSE orders_cursor
-- And finally, remove it
DEALLOCATE orders_cursor;
```

47

Analysis

- Like the previous example, this code uses FETCH to retrieve the current order_num and place it into a declared variable named @order_num.
- Unlike the previous example, the FETCH here is followed by a WHILE loop, so it is repeated over and over. When does the looping terminate?
- Each time FETCH is used, an internal function named @@FETCH_STATUS obtains a status code.
- @@FETCH_STATUS will return 0 if the FETCH succeeded, and a negative value otherwise. So the WHILE loop simply continues WHILE @@FETCH_STATUS = 0.

48

Presented by
Ranjan Bhatnagar

Finally

```
-- Local variables
DECLARE @order_num INT;
DECLARE @order_total MONEY;
DECLARE @total MONEY;
-- Initialize @total
SET @total=0;
-- Define the cursor
DECLARE orders_cursor CURSOR
FOR
SELECT order_num FROM orders
ORDER BY order_num;
-- Open cursor (retrieve data)
OPEN orders_cursor;
-- Perform the first fetch (get first
row)
FETCH NEXT FROM orders_cursor
INTO @order_num;
-- Check @@FETCH_STATUS to see if
there are any more rows
-- to fetch.
WHILE @@FETCH_STATUS = 0
BEGIN
-- Get this order total (including tax)
EXECUTE ordertotal @order_num, 1,
@order_total OUTPUT
-- Add this order to the total
SET @total = @total + @order_total
-- Get next row
FETCH NEXT FROM orders_cursor
INTO @order_num;
END
-- Close cursor
CLOSE orders_cursor
-- And finally, remove it
DEALLOCATE orders_cursor;
-- And finally display calculated total
SELECT @total AS GrantTotal;
```

49

Analysis

- In this example, we've declared a variable named @order_total (to store the total for each order) and another named @total (to store the running total of all orders).
- FETCH fetches each @order_num as it did before, and then EXECUTE is used to execute a stored procedure to calculate the total with tax for each order (the result of which is stored in @order_total).
- Each time an @order_total is retrieved, it is added to @total using a SET statement.
- And finally, the grant total is returned using a SELECT.
- And there you have it, a complete working example of cursors, row-byrow processing, and even stored procedures execution.

50

Using Triggers

- T-SQL statements are executed when needed, as are stored procedures. But what if you want a statement (or statements) to be executed automatically when events occur?
- Here are some examples:
  - Every time a customer is added to a database table, check that the phone number is formatted correctly and that the state abbreviation is in uppercase.
  - Every time a product is ordered, subtract the ordered quantity from the number in stock.
  - Whenever a row is deleted, save a copy in an archive table.

51

What is a Trigger?

- What all these examples have in common is that they need to be processed automatically whenever a table change occurs.
- And that is exactly what triggers are. A trigger is a T-SQL statement (or a group of statements enclosed within BEGIN and END statements) that is automatically executed by SQL Server in response to any of these statements:
  - DELETE
  - INSERT
  - UPDATE
- No other T-SQL statements support triggers.

52

Creating Triggers

- When creating a trigger, you need to specify three pieces of information:
  - The unique trigger name
  - The table to which the trigger is to be associated
  - The action that the trigger should respond to (DELETE, INSERT, or UPDATE)
- Triggers are created using the CREATE TRIGGER statement. Here is a really simple example:

```
CREATE TRIGGER newproduct_trigger ON products
AFTER INSERT
AS
SELECT 'Product added';
```

53

What we did?

- CREATE TRIGGER is used to create the new trigger named newproduct_trigger.
- This trigger is defined as AFTER INSERT, so the trigger will execute after a successful INSERT statement has been executed, and the text Product added will be displayed once for each row inserted.
- To test this trigger, use the INSERT statement to add one or more rows to products; you'll see the Product added message displayed for each successful insertion.
- Triggers are defined per event per table, and only one trigger per event per table is allowed. As such, up to three triggers are supported per table (one for each of INSERT, UPDATE, and DELETE).

54

Presented by
Ranjan Bhatnagar

Dropping Triggers

- By now the syntax for dropping a trigger should be self-apparent.
- To drop a trigger, use the DROP TRIGGER statement, as shown here:

```
DROP TRIGGER newproduct_trigger;
```

- If required, triggers can be updated using ALTER TRIGGER, or they can be dropped and re-created.
- Rather than dropping the triggers and then having to re-create them, SQL Server allows you to disable triggers and then enable them as needed.

```
DISABLE TRIGGER newproduct_trigger ON products;
ENABLE TRIGGER newproduct_trigger ON products;
```

55

Determining Trigger Assignments

- Triggers are very useful and very powerful. But they can also change the way SQL Server behaves, executing code that you may be unaware of.
- And because most triggers execute silently (they provide no feedback), it can be difficult to determine whether triggers are running at any given time.
- To solve this problem, you can use the built-in stored procedure SP_ HELPTRIGGER:

```
SP_HELPTRIGGER products;
```

56

INSERT Triggers

- INSERT triggers are executed after an INSERT statement is executed.
- Within INSERT trigger code, you can refer to a virtual table named INSERTED to access the rows being inserted.

```
CREATE TRIGGER neworder_trigger ON orders
AFTER INSERT
AS
SELECT @@IDENTITY AS order_num;
```

- The code creates a trigger named neworder_trigger that is executed AFTER INSERT on the table orders.
- When a new order is saved in orders, SQL Server generates a new order number and saves it in order_num.

57

Insert Trigger Cont…

- This trigger simply obtains this value from @@IDENTITY and returns it.
- Using this trigger for every insertion into orders will always return the new order number.
- To test this trigger, try inserting a new order, like this:

```
INSERT INTO orders(order_date, cust_id)
VALUES(GetDate(), 10001);
```

- orders contains three columns. order_date and cust_id must be specified, order_num is automatically generated by SQL Server, and order_num is now returned automatically.

58

DELETE Triggers

- DELETE triggers are executed after a DELETE statement is executed.
- Within DELETE trigger code, you can refer to a virtual table named DELETED to access the rows being deleted.
- The following example demonstrates the use of DELETED to save deleted rows into an archive table:

```
CREATE TRIGGER deleteorder_trigger ON orders
AFTER DELETE
AS
BEGIN
INSERT INTO orders_archive(order_num, order_date, cust_id)
SELECT order_num, order_date, cust_id FROM DELETED;
END;
```

59

UPDATE Triggers

- UPDATE triggers are executed after an UPDATE statement is executed.
- Within UPDATE trigger code, you can refer to a virtual table named DELETED to access the previous (pre-UPDATE statement) values and INSERTED to access the new updated values.
- The following example ensures that state abbreviations are always in uppercase (regardless of how they were actually specified in the UPDATE statement):

```
CREATE TRIGGER vendor_trigger ON vendors
AFTER INSERT, UPDATE
AS
BEGIN
UPDATE vendors SET vend_state=Upper(vend_state)
WHERE vend_id IN (SELECT vend_id FROM INSERTED);
END;
```

60

Presented by
Ranjan Bhatnagar

More on Triggers

- Here are some important points to keep in mind when using triggers:
  - Creating triggers might require special security access. However, trigger execution is automatic. If an INSERT, UPDATE, or DELETE statement may be executed, any associated triggers will be executed, too.
  - Triggers should be used to ensure data consistency (case, formatting, and so on). The advantage of performing this type of processing in a trigger is that it always happens, and happens transparently, regardless of client application.
  - One very interesting use for triggers is in creating an audit trail. Using triggers, it would be very easy to log changes (even before and after states if needed) to another table.
  - Triggers can call stored procedures as well as most T-SQL statements.

61

Managing Transaction Processing

- Transaction processing is used to maintain database integrity by ensuring that batches of T-SQL operations execute completely or not at all.
- The orders tables you've been using in prior lessons are a good example of this. Orders are stored in two tables: orders stores actual orders, and orderitems stores the individual items ordered.
- These two tables are related to each other using unique IDs called primary keys
- These tables, in turn, are related to other tables containing customer and product information.

62

Requirement

- Consider the process of adding an order to the system is as follows:
  - Check whether the customer is already in the database (present in the customers table). If not, add him or her.
  - Retrieve the customer's ID.
  - Add a row to the orders table associating it with the customer ID.
  - Retrieve the new order ID assigned in the orders table.
  - Add one row to the orderitems table for each item ordered, associating it with the orders table by the retrieved ID (and with the products table by product ID).

63

What if Failure?

- Now imagine that some database failure (for example, out of disk space, security restrictions, table locks) prevents this entire sequence from completing.
- What would happen to your data?
- Well, if the failure occurred after the customer was added and before the orders table was added, there is no real problem.
- It is perfectly valid to have customers without orders. When you run the sequence again, the inserted customer record will be retrieved and used. You can effectively pick up where you left off.

64

More Problems

- But what if the failure occurred after the orders row was added, but before the orderitems rows were added? Now you'd have an empty order sitting in your database.
- Worse, what if the system failed during adding the orderitems rows?
- Now you'd end up with a partial order in your database, but you wouldn't know it.

65

How do you solve this problem?

- That's where transaction processing comes in. Transaction processing is a mechanism used to manage sets of T-SQL operations that must be executed in batches to ensure that databases never contain the results of partial operations.
- With transaction processing, you can ensure that sets of operations are not aborted mid-processing; they either execute in their entirety or not at all (unless explicitly instructed otherwise).
- If no error occurs, the entire set of statements is committed (written) to the database tables. If an error does occur, a rollback (undo) can occur to restore the database to a known and safe state.

66

This is how the process would work

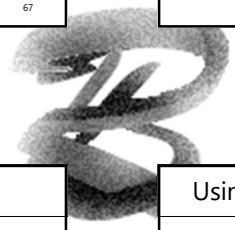
- Check whether the customer is already in the database; if not, add him or her.
- Commit the customer information.
- Retrieve the customer's ID.
- Add a row to the orders table.
- If a failure occurs while adding the row to orders, roll back.
- Retrieve the new order ID assigned in the orders table.
- Add one row to the orderitems table for each item ordered.
- If a failure occurs while adding rows to orderitems, roll back all the orderitems rows added and the orders row.
- Commit the order information.

67

Keywords used for Transactions

- When working with transactions and transaction processing, you'll notice a few keywords that keep reappearing. Here are the terms you need to know:
  - **Transaction:** A block of SQL statements
  - **Rollback:** The process of undoing specified SQL statements
  - **Commit:** Writing unsaved SQL statements to the database tables
  - **Savepoint:** A temporary placeholder in a transaction set to which you can issue a rollback (as opposed to rolling back an entire transaction)

68



Controlling Transactions

- Now that you know what transaction processing is, let's look at what is involved in managing transactions.
- The key to managing transactions involves breaking your SQL statements into logical chunks and explicitly stating when data should be rolled back and when it should not.
- The T-SQL statement used to mark the start of a transaction is

  BEGIN TRANSACTION;
- Transactions may optionally be named. This is useful when you are working with multiple transactions so as to be able to explicitly define the transaction to be committed if rolled back.

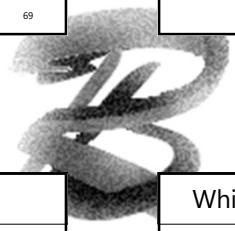
69

Using ROLLBACK

- The T-SQL ROLLBACK command is used to roll back (undo) T-SQL statements, as shown in this next statement

```
-- What is in orderitems?
SELECT * FROM orderitems;
-- Start the transaction
BEGIN TRANSACTION;
-- Delete all rows from orderitems
DELETE FROM orderitems;
-- Verify that they are gone
SELECT * FROM orderitems;
-- Now rollback the transaction
ROLLBACK;
-- And the deleted rows should all be back
SELECT * FROM orderitems;
```

70



Understand Execution

- This example starts by displaying the contents of the orderitems table.
- First, a SELECT is performed to show that the table is not empty.
- Then a transaction is started, and all of the rows in orderitems are deleted with a DELETE statement.
- Another SELECT verifies that, indeed, orderitems is empty.
- Then a ROLLBACK statement is used to roll back all statements until the BEGIN TRANSACTION, and the final SELECT shows that the table is no longer empty.
- Obviously, ROLLBACK can only be used within a transaction (after a BEGIN TRANSACTION command has been issued).

71

Which Statements Can You Roll Back?

- Transaction processing is used to manage INSERT, UPDATE, and DELETE statements.
- You cannot roll back SELECT statements.
- There would not be much point in doing so anyway.
- You cannot roll back CREATE and DROP operations.
- These statements may be used in a transaction block, but if you perform a rollback, they will not be undone.

72

Using COMMIT

- T-SQL statements are usually executed and written directly to the database tables.
- This is known as an autocommit; the commit (write or save) operation happens automatically.
- Within a transaction block, however, commits do not occur implicitly. To force an explicit commit, you use the COMMIT statement, as shown here:

```
BEGIN TRANSACTION;
DELETE FROM orderitems WHERE order_num = 20010;
DELETE FROM orders WHERE order_num = 20010;
COMMIT;
```

73

Using Savepoints

- Simple ROLLBACK and COMMIT statements enable you to write or undo an entire transaction.
- Although this works for simple transactions, more complex transactions might require partial commits or rollbacks.
- For example, the process of adding an order described previously is a single transaction.
- If an error occurs, you only want to roll back to the point before the orders row was added. You do not want to roll back the addition to the customers table (if there was one).

74

Save Transaction

- To support the rollback of partial transactions, you must be able to put placeholders at strategic locations in the transaction block.
- Then, if a rollback is required, you can roll back to one of the placeholders.
- These placeholders are called savepoints, and to create one use the SAVE TRANSACTION statement, as follows:

```
SAVE TRANSACTION delete1;
```

- To roll back to this savepoint, do the following:

```
ROLLBACK TRANSACTION delete1;
```

75

Rollback from Beginning

- To roll back to the very beginning of the transaction, do the following:

```
ROLLBACK TRANSACTION;
```

- Implicit Transaction Closes
- After a COMMIT or ROLLBACK statement has been executed, the transaction is automatically closed (and future changes will implicitly commit).

76

TRY...CATCH

- Implements error handling for Transact-SQL that is similar to the exception handling in the Microsoft Visual C# languages. A group of Transact-SQL statements can be enclosed in a TRY block. If an error occurs in the TRY block, control is passed to another group of statements that is enclosed in a CATCH block.
- Syntax for SQL Server

```
BEGIN TRY
    { sql_statement | statement_block }
END TRY
BEGIN CATCH
    [ { sql_statement | statement_block } ]
END CATCH
[ ; ]
```

77

Using TRY...CATCH

- The following example shows a SELECT statement that will generate a divide-by-zero error. The error causes execution to jump to the associated CATCH block.

```
BEGIN TRY
   -- Generate a divide-by-zero error.
   SELECT 1/0;
END TRY
BEGIN CATCH
   SELECT   ERROR_NUMBER() AS ErrorNumber
      ,ERROR_SEVERITY() AS ErrorSeverity
      ,ERROR_STATE() AS ErrorState
      ,ERROR_PROCEDURE() AS ErrorProcedure
      ,ERROR_LINE() AS ErrorLine
      ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

78

Using TRY...CATCH in a transaction

```
BEGIN TRANSACTION;
BEGIN TRY
   -- Generate a constraint violation error.
   DELETE FROM products WHERE prod_id = 'FB';
END TRY
BEGIN CATCH
   SELECT ERROR_NUMBER() AS ErrorNumber
      ,ERROR_SEVERITY() AS ErrorSeverity
      ,ERROR_STATE() AS ErrorState
      ,ERROR_PROCEDURE() AS ErrorProcedure
      ,ERROR_LINE() AS ErrorLine
      ,ERROR_MESSAGE() AS ErrorMessage;
IF @@TRANCOUNT > 0
      ROLLBACK TRANSACTION;
END CATCH;
IF @@TRANCOUNT > 0
   COMMIT TRANSACTION;
```

79

Presented by
Ranjan Bhatnagar