FUNCTIONS , VARIABLES AND CONTROL OF FLOW

Lab 4

Variables

- Local Variables
- Global Variables
- Dynamic SQL
- What Are T-SQL Expressions?
- Control-of-flow Statements

What Are T-SQL Variables?

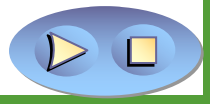
A Transact-SQL local variable is an object that can hold a single data value of a specific type

Variables in batches and scripts are typically used to:

- Count the number of times a loop is performed
- Hold data to be tested by a control-of-flow statement
- Save data values to be returned by a function return value

```
DECLARE @local_variable as <data_type>
```

```
declare @food varchar(20)
set @food = 'ice cream'
…
WHERE Description = @food
```



Local Variable

- A local variable is a named location in memory defined by a user that stores a value
- Null is the Initial Value for local variable
- Typical uses for a local variable
  - To facilitate repeated use of constant values
  - To perform conditional branching in Transact-SQL code
  - To return custom messages to the client that contain variable information
  - To pass information to and from a stored procedure
  - To avoid using a subquery

Declaring Local Variables

- Syntax:

  declare *variable_name* *datatype*
  [, *variable_name* *datatype* ...]

- Example:

```
declare  @myqty        int,
         @myid     char(4)
```

- Variables must be declared before they can be used @
- When declared, the value of a local variable is set to NULL

Viewing Local Variable Values

- Simplified syntax:
  select *variable_name*

- Example:

```
declare @mynumber  int
select @mynumber

-
NULL
```

- This is the same **select** that is used to query data from tables

Assigning Values to Local Variables

- Three methods
  - During Declaration
  - Assignment **select** using an expression
  - Assignment **select** using a table value
  - Assignment **update**

During Declaration

- Example1:

```
declare @x int=1

declare @x int=(Select avg(salary) from Instructor)
```

- Example2:

```
declare @x int
Set @x=1
```

Assignment select and Expressions

- Simplified syntax:
  select *variable_name* = *expression*
  [, *variable_name* = *expression* ...]
- Examples:

```
declare  @number int,
         @copy int,
         @sum int

select   @number = 10

select   @copy = @number,
         @sum = @number + 100
```

- This **select** is an "assignment **select**"
  - No information is returned to the user

Assignment select and Table Values

- Simplified syntax:
  select *variable_name* = *column_name*
  from *table_name*
  [where *condition*]
- Examples:

```
declare @AD_id char(11)
select @AD_id = au_id
    from authors
    where au_fname = "Ann" and au_lname = "Dull"
```

- This **select** is an assignment **select**
  - No information is returned to the user
- If the **select** returns multiple values, only the last value remains in the variable

Assignment update

- Simplified syntax:
  update *table_name*
  set {*column_name* | *variable_name* } = expression
  [, {*column_name* | *variable_name* } = expression …]
  [where *condition*]
- Examples:

```
declare @pub_name varchar(40)
update publishers
    set   city = "Escanaba",
          state = "MI",
          @pub_name = pub_name
    where pub_id = "0736"
```

- No information is returned to the user
- If the **update** modifies multiple rows, only the last assigned value remains in the variable

Global Variable

- A global variable is a named location in memory, defined and maintained by Adaptive Server
- 0 is the initial value for @@error variable
- Rules for global variables
  - Names start with "@@"
  - Cannot be created by users
  - Cannot be assigned values by users
  - Value assignment can be local to the server or to the connection

Common Global Variables

- *@@rowcount*
  - Returns the number of rows affected by the last statement
- *@@error*
  - Returns the error number generated by the last statement
- *@@identity*
  - Returns the value last inserted into an IDENTITY column
- *@@version*
  - Returns the version number of the server

Control-of-flow Statements

These are the control-of-flow keywords:

- BEGIN…END
- BREAK
- GOTO
- CONTINUE
- IF…ELSE
- WHILE
- RETURN
- WAITFOR

Case Expression

Evaluates a list of conditions and returns one of multiple possible result expressions.

The CASE expression has two formats:

- The simple CASE expression compares an expression to a set of simple expressions to determine the result.
- The searched CASE expression evaluates a set of Boolean expressions to determine the result.

Both formats support an optional ELSE argument.

Case Expression Cont.

Syntax

Simple CASE expression:

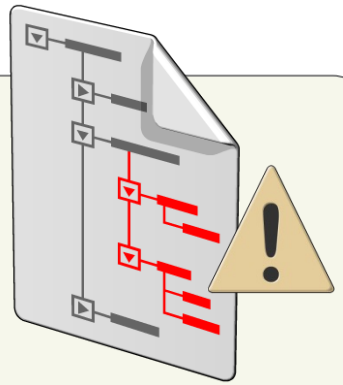
```
CASE input_expression
        WHEN when_exp THEN result_exp[ ..n ]   [ ELSE
        else_result_exp]
END
```

Searched CASE expression:

```
 CASE
WHEN Boolean_exp THEN result_exp[ ...n ]    [ ELSE
else_result_exp]
END
```

Structured Exception Handling

TRY/CATCH



```
BEGIN TRY
   -- Generate divide-by-zero error.
   SELECT 1/0;
END TRY
BEGIN CATCH
   -- Execute error retrieval routine.
   EXECUTE usp_GetErrorInfo;
END CATCH;
```

RAISERROR

```
RAISERROR (N'This is message %s %d.', -- Message text.
        10, -- Severity,
        1, -- State,
        N'number', -- First argument.
        5); -- Second argument.
-- The message text returned is: This is message number 5.
GO
```

Overview of Transactions

- Transaction Fundamentals
- Transactions and the Database Engine
- Basic Transaction Statement Definitions
- What are Transaction Isolation Levels?
- Using Nested Transactions

Transaction Fundamentals

A Transaction:

- Is a sequence of operations performed as a single logical unit of work

- Exhibits the four ACID Properties
  - Atomicity – must be an atomic unit of work
  - Consistency - must leave all data in a consistent state
  - Isolation - must be isolated from the modifications made by any other concurrent transactions
  - Durability – persists even after system failure

Role of Transactions

- Protect data from software, hardware, and power failures
- Allow for data isolation so that multiple users can access data simultaneously without interfering with one another

Transactions and the Database Engine

The Database Engine provides:

- Locking facilities that preserve transaction isolation
  - Transaction Isolation Levels control when locks are taken and how long they are held
- Logging facilities that ensure transaction durability
  - Write-ahead log (WAL) guarantees no data modifications are written before they are logged
  - Checkpoints write records to a data file and contain lists of all active transactions
- Transaction management features that enforce transaction atomicity and consistency
  - Transactions must be successfully completed or their modifications are undone

Basic Transaction Statement Definitions

BEGIN TRANSACTION

```
BEGIN { TRAN | TRANSACTION } [
Transaction_name | @tran_name_variable ]
```

```
BEGIN TRAN T1;
UPDATE table1 ...;
```

COMMIT TRANSACTION

```
COMMIT { TRAN | TRANSACTION } [
transaction_name | @tran_name_variable ]
```

```
COMMIT TRAN T1;
```

ROLLBACK TRANSACTION

```
ROLLBACK { TRAN | TRANSACTION } [
transaction_name | @tran_name_variable |
savepoint_name | @savepoint_variable ]
```

```
ROLLBACK TRAN T1;
```

Using Nested Transactions

- Explicit transactions can be nested to support transactions in stored procedures

```
CREATE TABLE TestTrans(Cola INT PRIMARY KEY,
                       Colb CHAR(3) NOT NULL);
GO
CREATE PROCEDURE TransProc @PriKey INT, @CharCol CHAR(3) AS
BEGIN TRANSACTION InProc
INSERT INTO TestTrans VALUES (@PriKey, @CharCol)
INSERT INTO TestTrans VALUES (@PriKey + 1, @CharCol)
COMMIT TRANSACTION InProc;
GO
BEGIN TRANSACTION OutOfProc; /* Starts a transaction */
GO
EXEC TransProc 1, 'aaa';
GO
ROLLBACK TRANSACTION OutOfProc; /* Rolls back the outer
                                   transaction */
GO
EXECUTE TransProc 3,'bbb';
GO
SELECT * FROM TestTrans;
GO
```

```
Cola   Colb
-------------
1      bb
2      bb
```

@@trancount

- *@@trancount* is a global variable that keeps track of the nesting level
  - **begin tran** increments *@@trancount* by 1
  - **commit tran** decrements *@@trancount* by 1
  - **rollback tran** sets *@@trancount* to 0
  - **save tran** and **rollback** *savepoint_name* do not change *@@trancount*

Introducing Functions

- Types of Functions
- What Is a Scalar Function?
- What Is an Inline Table-Valued Function?
- What Is a Multi-Statement Table-Valued Function?

What Are User-Defined Functions?

A User-Defined Function is a routine that accepts parameters, performs an action, and returns the result of that action as a value.

Benefits of using User-Defined Functions

- Modular programming for reusable logic.
- Complex operations can be optimized for faster execution.
- Logic performed in database reduces network traffic

What Are T-SQL Functions?

Functions	Notes
Rowset	• Return objects that can be used as table references
Examples: CONTAINSTABLE, OPENDATASOURCE, OPENQUERY	
Aggregate	• Operate on a collection but returns a single value
Examples: AVG, CHECKSUM_AGG, SUM, COUN	
Ranking	• Return a ranking value for each row in a partition
Examples: RANK, DENSE_RANK	
Scalar	• Operate on a single value and then return a single value
Examples: CREATE FUNCTION dbo.ufn_CubicVolume	

Types of Functions

Types of Functions

- Scalar Functions
- Inline Table-Valued Functions
- Multi-Statement Table-Valued Functions
- Built-in Functions

How To Implement Different Types Of User-Defined Functions

Types	Usage
Scalar-valued	• Scalar is specified in the RETURNS clause • Can be defined with multiple T-SQL statements
Inline table-valued	• TABLE is specified in the RETURNS clause • Does not have associated return variables • *select_stmt* is the single SELECT statement that defines the return value
Multi-statement table-valued	• TABLE is specified in the RETURNS clause • *function_body* is used as a series of T-SQL statements that populate a TABLE return variable • *@return_variable* is used to store and accumulate rows that are returned as the value

What Is a Scalar Function?

Scalar Functions:

- Return a single data value
- Can be either inline or multi-statement
- Can return any data type except for text, ntext, image, cursor, and timestamps

```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ]
parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS return_data_type
```

What Is an Inline Table-Valued Function?

Inline Table-Valued Function:

- Returns a TABLE data-type
- Has no function body
- Is comprised of a single result set

```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ]
parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)
RETURNS TABLE
```

What Is a Multi-Statement Table-Valued Function?

Multi-statement Table-Valued Function:

- Returns a TABLE data-type
- Has a function body defined by BEGIN and END blocks
- Defines a table-type variable and schema
- Inserts rows from multiple Transact-SQL statements into the returned table

```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ]
parameter_data_type
    [ = default ] [READONLY] }
    [ ,...n ]] )
RETURNS @return_variable TABLE <table_type_definition>
```