variables

- Every language requires variables to temporarily store values in memory
- T-SQL variables are created with the DECLARE command.
- The DECLARE command is followed by the variable name and data type
- The available data types are similar to those used to create tables

variables

```
DECLARE @x INT=0;
```

- Must start with @
- The scope, or application and duration, of the variable extends only to the current batch
- Newly declared variables default to NULL

Batch

- A batch is a group of one or more Transact-SQL statements sent at the same time from an application to SQL Server for execution. SQL Server compiles the statements of a batch into a single executable unit, called an execution plan. The statements in the execution plan are then executed one at a time.

Batch(Example)

- Select * from student

  select * from instructor

  insert into student (Id,Name) values (5,’ahmed’)

go

Note: this is a batch

variables

- Both the SET command and the SELECT command can assign the value of an expression to a variable

```
Declare @x int=0;
Set @x=(select max(salary) from  instructor)
```

```
Declare @x int=0;
Select  @x=(select max(salary) from  instructor)
```

```
Declare @x int=0;
Select @x=max(salary) from instructor
```

variables

- Never use a SELECT to populate a variable unless you're sure that it will return only a single row.

```
select @test2=salary from Instructor
Will get last salary
```

Variables

```
SET @x += 5;
```

```
SET @x *= 2
```

```
Using variables within SQL queries

DECLARE @ProductCode CHAR(10);
SET @ProductCode = '1001';
SELECT ProductName
FROM Product
WHERE Code = @ProductCode;
```

Global variables

- A global variable is a named location in memory, initialized and updated by Server
  - Names start with “@@”
  - Cannot be created or assigned by users
- *@@error*
  - Returns the error number generated by the last statement
- *@@rowcount*
  - Returns the number of rows affected by the last statement

Global variables

- *@@identity*
  - Returns the value last inserted into an IDENTITY column
- *@@Language*
- The language, by name, used by the current connection

Global variables

- @@ServerName : Name of the current server
- @@version: SQL Server edition, version, and service pack

Data types conversion

- Data is moved to, compared, or combined with other data
- Data is moved from a result column, return code, or output parameter into a program variable
- There are two types:

-Implicit Conversion

-Explicit Conversion

Implicit Conversion

- Implicit conversions are not visible to the user
- SQL Server automatically converts the data from one data type to another

```
declare @x int=1000;
select * from Instructor where Salary=@x
Convert implicitly int to money
```

Explicit Conversion

- Explicit conversions use the CAST or CONVERT functions

```
Select * from student
Where convert(nvarchar(50),nationalid) like '%150042%'
Search specific pattern in int
```

```
Select * from student
Where cast(nationalid as varchar(50)) like '%150042%'
Search specific pattern in int
```

Procedural Flow(control of flow)

- IF else
- While
- Begin end

- Go to
- Continue
- Break

IF esle

```
IF  Condition
Statement;
Else
statement
```

```
IF Condition
  Begin;
  Multiple lines;
  End;
Else if  (condition)
  Begin
  End
Else
  begin
end
```

While

- The WHILE command is used to loop through code while a condition is still true.
- To control a full block of commands, BEGIN/END is used.

```
DECLARE @Temp INT;
SET @Temp = 0;
WHILE @Temp < 3
BEGIN;
PRINT 'tested condition' + STR(@Temp);
SET @Temp = @Temp + 1;
END;
```

Case statement

- determine the value of an expression based on a condition.

Simple case

```
SELECT  CustomerTypeName,
CASE IsDefault
WHEN 1 THEN ‘default type’
WHEN 0 THEN ‘possible’
ELSE ‘-’
END AS AssignStatus
FROM CustomerType;
```

Case statement

Boolean case

```
select case
when salary+isnull(bonus,0)>1000 then 'good salary'
when salary+isnull(bonus,0)>2000 then 'bad salary'
else 'medium'
end as 'indicator'
from  [dbo].[Instructor]
```

Message statement

- Three statements can return text information to users:
  - **select**
  - **print**
  - **raiserror**

Raiserror()

- Useful for returning error messages in the message window

```
RAISERROR (
message or number, severity, state, optional arguments
) WITH LOG;
```

```
RAISERROR('this is error',14,-1)
```

sp_addmessage

- To manage messages in code, use the sp_addmessage system stored procedure:

```
EXEC   sp_addmessage   50001, 16, 'Unable to update ';
```

```
Raiserror(50001,16,-1)
```

- To drop error message

```
sp_dropmessage message_number
```

Exception handling

- TRY...CATCH is a standard method of trapping and handling errors

```
declare @x int=5;
BEGIN TRY
   -- Generate divide-by-zero error.
if (@x<10)
begin
raiserror(60000,17,-1)
end
select 80
END TRY
BEGIN CATCH
   -- Execute error retrieval routine.
   select 50
END CATCH;
```

Transactions

- A transaction is a sequence of tasks that together constitute a logical unit of work

- All the tasks must complete or fail as a single unit example: bank transactions

Why we use 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 by locking

Transactions

- The following three commands appear simple:
- BEGIN TRANSACTION
- COMMIT TRANSACTION
- ROLLBACK TRANSACTION

Transactions

- There are two types of transactions:

1- Explicit transaction

2- implicit transaction

Explicit transaction

```
BEGIN TRY;
BEGIN TRANSACTION;
UPDATE Production.ProductInventory
SET Quantity-= 100
WHERE ProductID = 527 AND LocationID = 6 -- misc storage AND Shelf = 'B'
AND Bin = 4;
UPDATE Production.ProductInventory
SET Quantity+= 100
WHERE ProductID = 527
AND LocationID = 50 -- subassembly area
AND Shelf = 'F'
AND Bin = 11;
COMMIT TRANSACTION;
END TRY
BEGIN CATCH;
ROLLBACK TRANSACTION;
RAISERROR('Inventory Transaction Error', 16, 1);
RETURN;
END CATCH;
```

Implicit transaction

- By default the implicit transaction is off

```
SET Implicit_Transactions on;
INSERT INTO [dbo].[Instructor] values (25,'yous','master',5000,20,5,10)
select @@TRANCOUNT

SET Implicit_Transactions off;
INSERT INTO [dbo].[Instructor] values (25,'yous','master',5000,20,5,10)
select @@TRANCOUNT
```

- @@transcount variable show how many pending transaction level awaiting a COMMIT or rollback

Functions

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

Functions

- Why we use functions:

1-modular programming fro reusable logic

2-complex operations can be optimized for faster execution

3-logic performed in database reduces network traffic

Types of functions

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

Scalar functions

- A scalar function is one that returns a single specific value
- The function can accept multiple parameters
- the value is passed back through the function by means of a RETURN command.

Scalar functions

```
create function ScFn(@a int,@b int)
returns nvarchar(50)
as
begin
    declare @x nvarchar(50)
    if(@a>@b)
    begin
    set @x='greater than'
    end
    else if(@a<@b)
    begin
    set @x='less than'
    end
    else set @x='equal'
    return @x
end
```

Scalar functions

- Can have default parameters

```
CREATE FUNCTION dbo.fsMultiply (@A INT, @B INT=3)
RETURNS INT
AS
BEGIN;
RETURN @A * @B;
END;
Go

SELECT dbo.fsMultiply (3,4),
dbo.fsMultiply (7,DEFAULT);
```

Inline tabled valued function

- is very similar to a view
- An inline table-valued user-defined function retains the benefits of a view, and adds parameters.
- The inline table-valued user-defined function has no BEGIN/END body

Inline tabled valued function

```
create function InFn(@id int)
returns table
as
return(select * from Instructor
where Ins_Id=@id)
```

Multi-Statement Table-Valued Functions

- The multi-statement table-valued user-defined function combines the scalar function's ability to contain
- complex code with the inline table-valued function's ability to return a result set.
- Returns a TABLE data-type
- Has a function body defined by BEGIN and END blocks
- Inserts rows from multiple Transact-SQL statements into the returned table

Multi-Statement Table-Valued Functions

```
CREATE FUNCTION FunctionName (InputParamenters)
RETURNS @TableName TABLE (Columns)
AS
BEGIN;
Code to populate table variable
RETURN;
END;
```

Restrictions on user defined functions

- UDF can’t update data

Soln: use stored procedure

- They cannot return BLOB (binary large object) data such as text, ntext, timestamp, and image data-type variables
- Can’t perform transactions

Stored procedure

- stored procedure is a batch that has been stored with a name so it can be easily called, and its query execution plan saved in memory

Why we use stored procedure

- Promotes modular programming
- Provides security attributes and permission chaining
- Allows code reuse
- Reduces network traffic because of Implementing the business logic in database server

Types of stored procedure

- User-defined Stored Procedures
- Extended Stored Procedures
- System Stored Procedures

User defined stored procedure

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name [ ; number
]
                [ { @parameter [ type_schema_name. ] data_type }
                 [ VARYING ] [ = default ] [ OUT | OUTPUT ] [READONLY]
                ] [ ,...n ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS { <sql_statement> [;][ ...n ] | <method_specifier> }
[;]
<procedure_option> ::=
                [ ENCRYPTION ]
                [ RECOMPILE ]
                [ EXECUTE_AS_Clause ]

<sql_statement> ::=
{ [ BEGIN ] statements [ END ] }
<method_specifier> ::=
EXTERNAL NAME assembly_name.class_name.method_name
```

Simple stored procedure

```
CREATE PROCEDURE HumanResources.usp_GetEmployeesName
@NamePrefix char(1)
AS
BEGIN
SELECT BusinessEntityID, FirstName, LastName,
EmailAddress
FROM HumanResources.vEmployee
WHERE FirstName LIKE @NamePrefix + '%'
ORDER BY FirstName
END
```

Stored procedure calling

- When calling a stored procedure within a SQL batch
- If the stored-procedure call is the first line of a batch, the stored-procedure call doesn’t require the EXEC command.

```
Execute  dbo.sp_ViewInstructor


dbo.sp_ViewInstructor
--If the stored-procedure call is the first line of a batch
```

Parameterized Stored Procedures

- Stored Procedure Parameters
- Table-valued Parameters

Stored procedure parameters

- Input parameters

```
Create procedure dbo.sp_ViewInstuctor (@id int)
as
begin
select * from Instructor where ins_id=@id
end
```

- Output parameters

```
alter procedure dbo.sp_ViewInstuctor (@id int,@salary
money output)
as
begin
set @salary=(select salary from Instructor where Ins_Id=@id)
end

declare @x money
exec sp_ViewInstuctor 2,@x output
select @x
```

- Return values

Input parameters

```
alter procedure dbo.sp_ViewInstuctor (@id int,@name
nvarchar(50),@salary money)
as
begin
insert into Instructor (Ins_Id,Ins_Name,Salary) values
(@id,@name,@salary)
return @@rowcount
end
```

Firing stored procedure

- Passing by parameter position

```
sp_ViewInstuctor 4544,'oop',410
```

- Passing by parameter name

```
sp_ViewInstuctor @salary=40000,@id=421,@name='ppp'
```

Default parameters

```
alter procedure dbo.sp_ViewInstuctor (@id int,@name nvarchar(50),@salary
money=8999)


sp_ViewInstuctor @id=423,@name='ppp'
```

Return

- A RETURN command unconditionally terminates the procedure and returns an integer value to the calling batch or client.
- EXEC @LocalVariable=StoredProcedureName;

Output parameters

- Output parameters enable a stored procedure to return data to the calling client procedure.
- The keyword OUTPUT is required both when the procedure is created and when it is called.

```
alter procedure dbo.sp_ViewInstuctor (@id int,@name
nvarchar(50),@salary money=8999,@x int output)



declare @y int
 exec sp_ViewInstuctor @id=488,@name='ppp',@x=@y
output
 select @y
```

Triggers

- special stored procedures attached to table events. They can’t be directly executed; they fire only in response to an INSERT , UPDATE, or DELETE event on a table.

Triggers types

- AFTER triggers execute after an INSERT, UPDATE, or DELETE statement

- INSTEAD OF triggers execute instead of an INSERT, UPDATE, or DELETE statement

Triggers benefits

- Cascading modifications through related tables
- Rolling back changes that violate data integrity
- Enforcing restrictions that are too complex for rules or constraints
- Maintaining duplicate data
- Maintaining columns with derived data
- Performing custom recording(audit)

Triggers rules

- Triggers can:
  - Declare local variables
  - Invoke non trigger stored procedures
- Triggers cannot:
  - Be called directly
  - Use parameters
  - Be defined on temporary tables or views
  - Create permanent database objects
- Minimally logged operations (such as **truncate table**) do not cause triggers to fire

Creating triggers

```
CREATE TRIGGER Schema.TriggerName ON Schema.TableName
AFTER | INSTEAD OF [Insert, Update, (and or) Delete]
AS
Trigger Code;
```

After trigger

- Complex data validation
- Enforcing complex business rules
- Writing data-audit trails
- Maintaining modified date columns
- Enforcing custom referential-integrity checks and cascading deletes

After Trigger

```
CREATE TRIGGER [delCategory] ON [Categories]
AFTER DELETE AS
BEGIN
  UPDATE P SET [Discontinued] = 1
  FROM [Products] P INNER JOIN deleted as d
  ON
  P.[CategoryID] = d.[CategoryID]
END;
```

Instead of triggers

- INSTEAD OF triggers execute ‘‘instead of’’ (as a substitute for) the submitted transaction

```
CREATE TRIGGER dbo.TriggerTwo ON dbo.Person
INSTEAD OFINSERT
AS
PRINT‘In the Instead of Trigger’;
go
```

Disabling triggers

```
ALTER TABLE dbo.Person
DISABLE TRIGGER TriggerOne;
```

Working with the Transaction

- Update() : which returns true for a single column if that column is affected by the DML
- Selected table
- Deleted table

DML Statement	Inserted Table	Deleted Table
Insert	Rows being inserted	Empty
Update	Rows in the database after the update	Rows in the database before
Delete	Empty	Rows being deleted