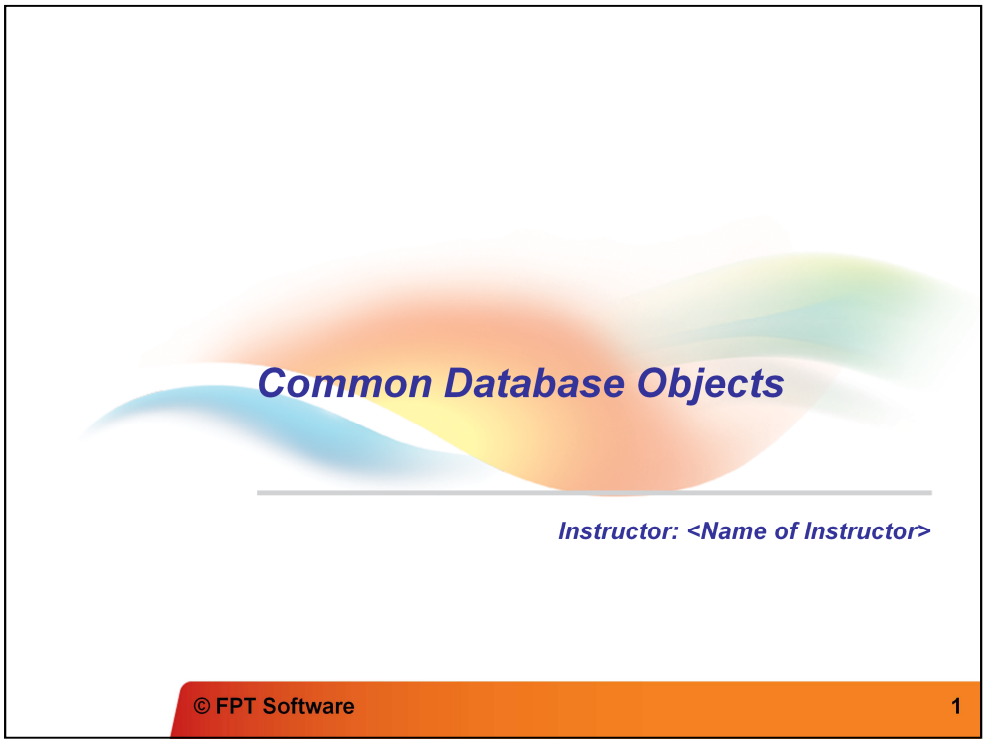


Latest updated by: HanhTT1

Agenda

- SQL Language Elements
- Stored Procedure
- User Defined Function
- Trigger
- SQL Code Practices

SQL Language Elements

- Comments
- Identifiers
- Variables
- Control-of-flow

SQL Language Elements
Comments

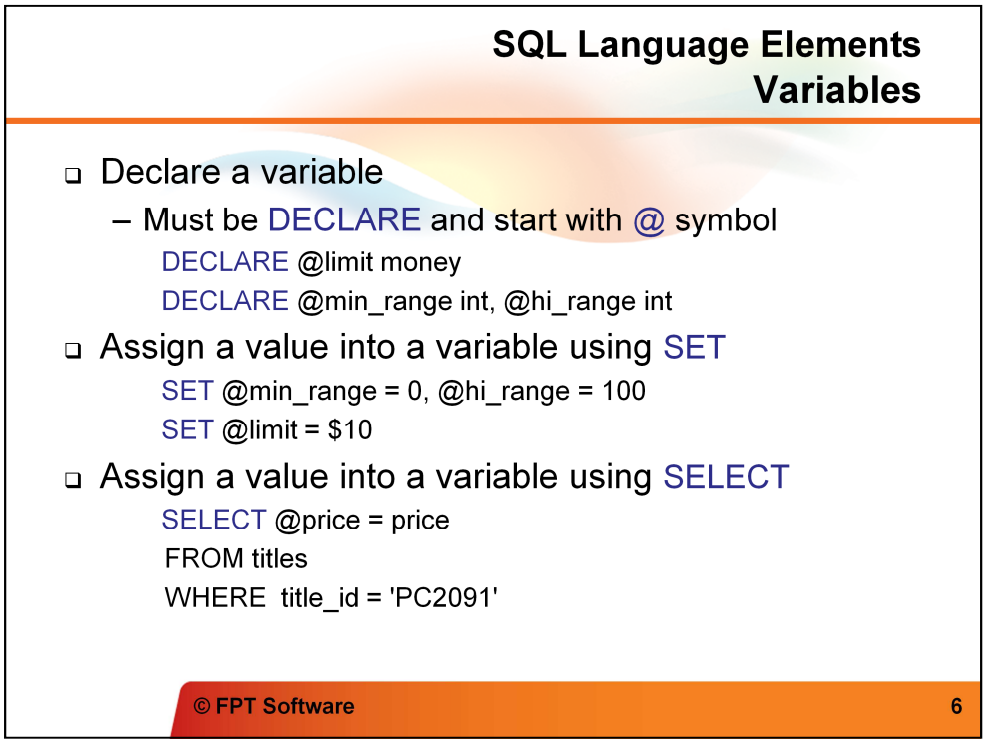
- Indicates user-provided text
  - Block Comment
    /*Multi-line comments here*/
  - Double Dash
    SELECT OrderId, OrderName FROM Orders -- This is a comment

SQL Language Elements
Identifiers

- The database object name is referred to as its identifier.
  - An object identifier is created when the object is defined
  - The identifier is used to reference the object
  - There are 2 types of Identifiers
  - Regular Identifiers
    - For instance: Orders, Customers, Employee…
  - Delimited Identifiers: Are enclosed in double quotation marks (") or brackets ([ ])
    - [My Table]
    - [1Person]

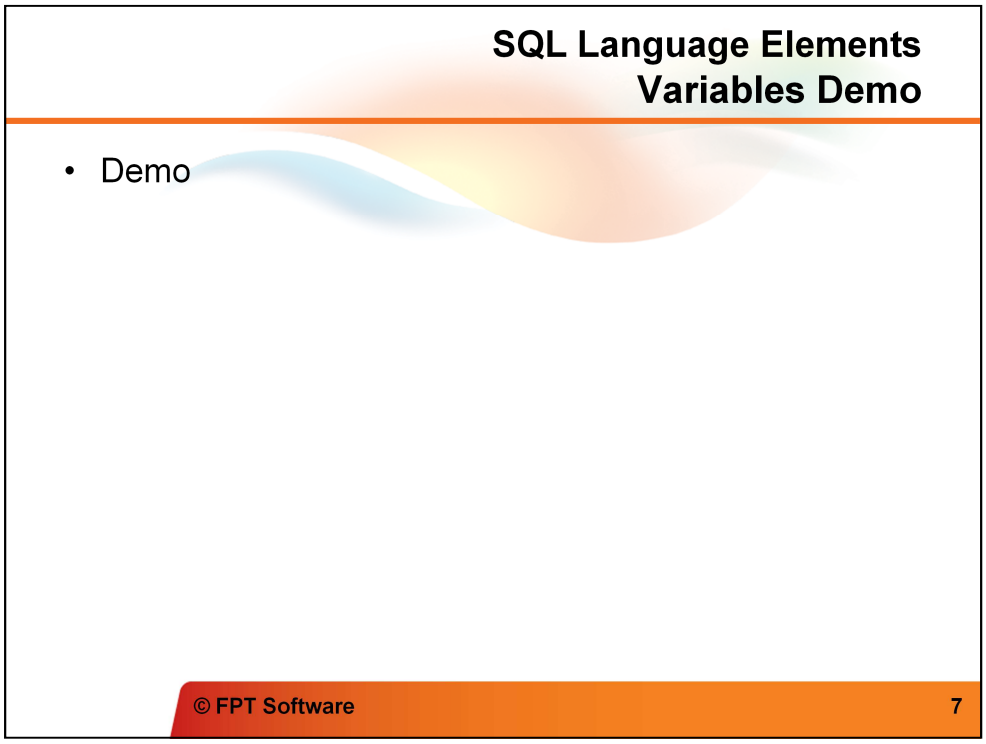
Relevant Queries:

```
select object_id('products')
select * from sysobjects
```


SQL Language Elements
Variables
Declare a variable
– Must be DECLARE and start with @ symbol
DECLARE @limit money
DECLARE @min_range int, @hi_range int
Assign a value into a variable using SET
SET @min_range = 0, @hi_range = 100
SET @limit = $10
Assign a value into a variable using SELECT
SELECT @price = price
FROM titles
WHERE title_id = 'PC2091'
© FPT Software
6

See also:

SET



See “Variable Demo” in Day2_Demo.docx

SQL Language Elements
Control of flow

- The T-SQL control-of-flow language keywords are:
  - BEGIN…END
  - IF…ELSE
  - CASE … WHEN
  - TRY…CATCH
  - WHILE
  - BREAK / CONTINUE
  - GOTO
  - RETURN

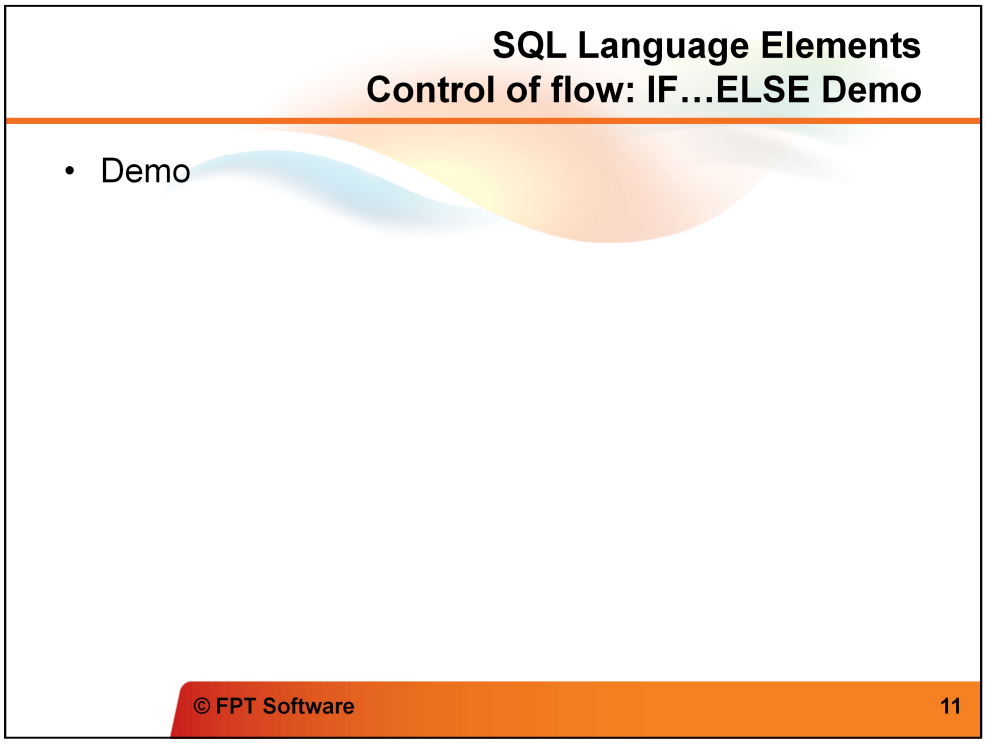
SQL Language Elements
Control of flow: BEGIN…END

- Defines a statement block
- Other Programming Languages:
  - C#, Java, C: **{ … }**
  - Pascal, Delphi: **BEGIN … END**

SQL Language Elements
Control of flow: IF…ELSE

- Defines conditional and, optionally, alternate execution when a condition is false
- Syntax:

```
IF Boolean_expression
   SQL_statement | block_of_statements
[ELSE
   SQL_statement | block_of_statements ]
```

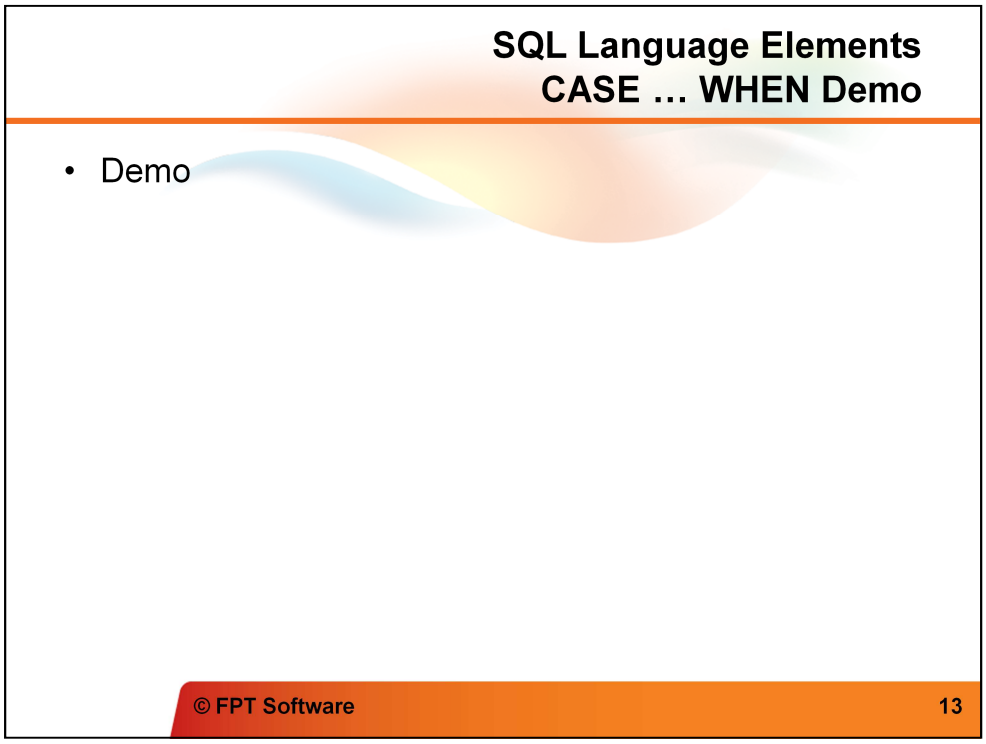


See “Control flow demo” in Day2_Demo.docx

SQL Language Elements
Control of flow: CASE … WHEN

- Evaluates a list of conditions and returns one of multiple possible result expressions
- Syntax:

```
CASE input_expression
   WHEN  when_expression  THEN result_expression
   [WHEN when_expression  THEN result_expression…n]
[ELSE else_result_expression ]
END
```



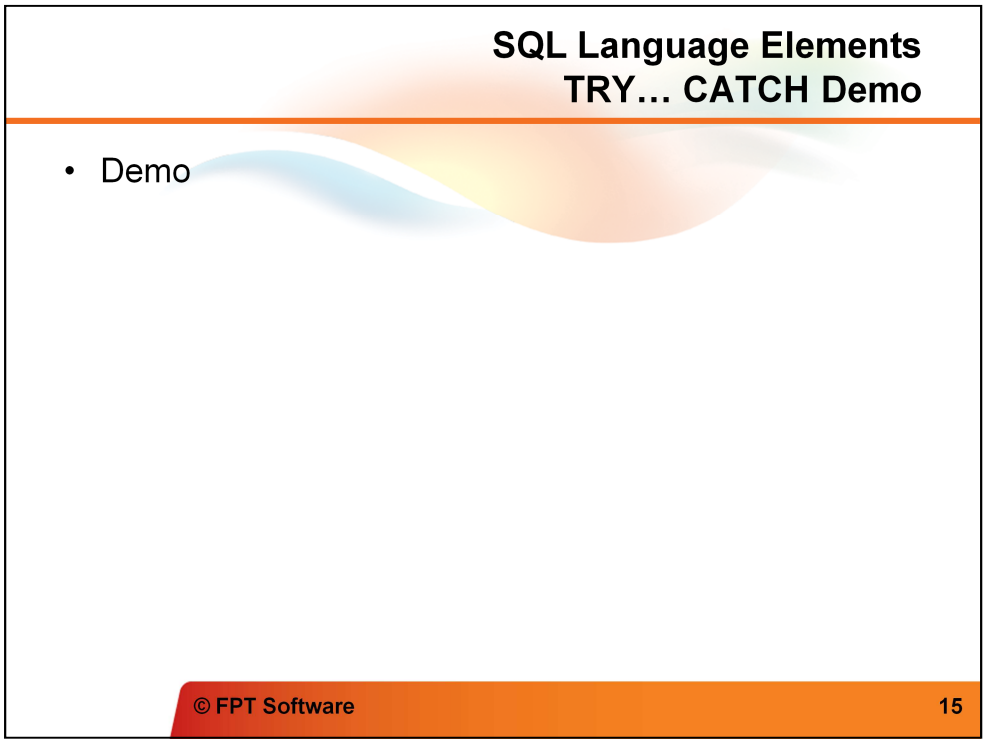
See “CASE …. WHEN” in Day2_Demo.docx

SQL Language Elements
Control of flow: TRY… CATCH

- Provides error handling for T-SQL that is similar to the exception handling in the C# / Java
- Syntax:

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


See “Control flow demo” in Day2_Demo.docx

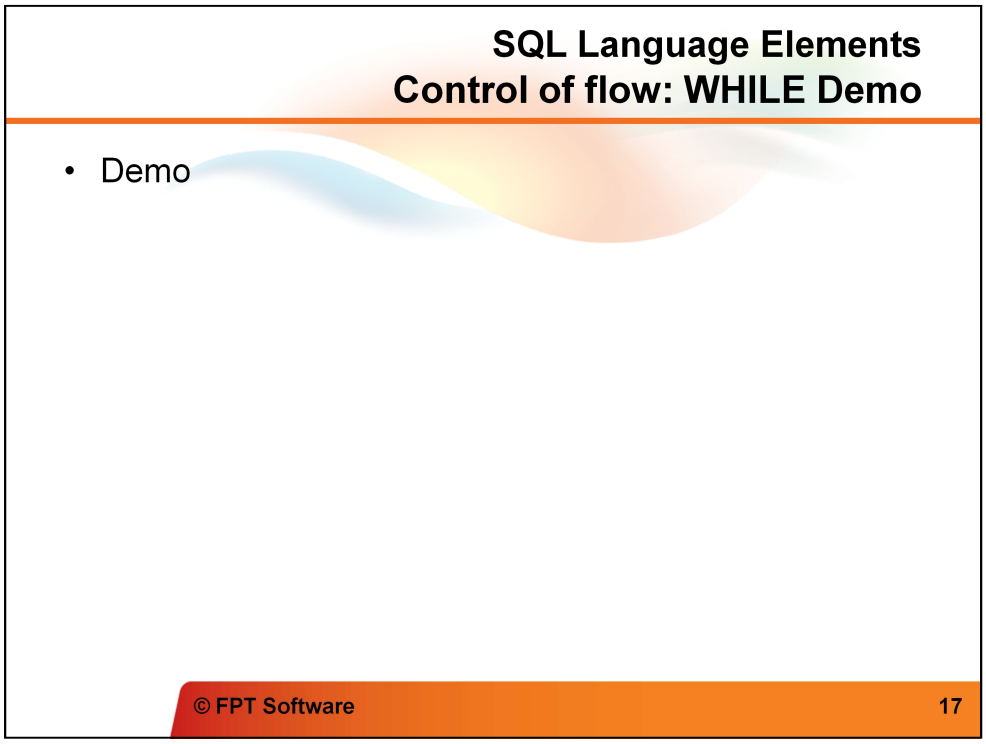


See “TRY …. CATCH” in Day2_Demo.docx

SQL Language Elements
Control of flow: WHILE

- Sets a condition for the repeated execution of an statement block
  - The statements are executed repeatedly as long as the specified condition is true
  - The execution of statements in the WHILE loop can be controlled from inside the loop with the BREAK and CONTINUE keywords
    - Syntax

```
WHILE Boolean_expression
{ sql_statement | statement_block | BREAK | CONTINUE }
```



See “Control flow demo” in Day2_Demo.docx

SQL Language Elements
Control of flow: GOTO

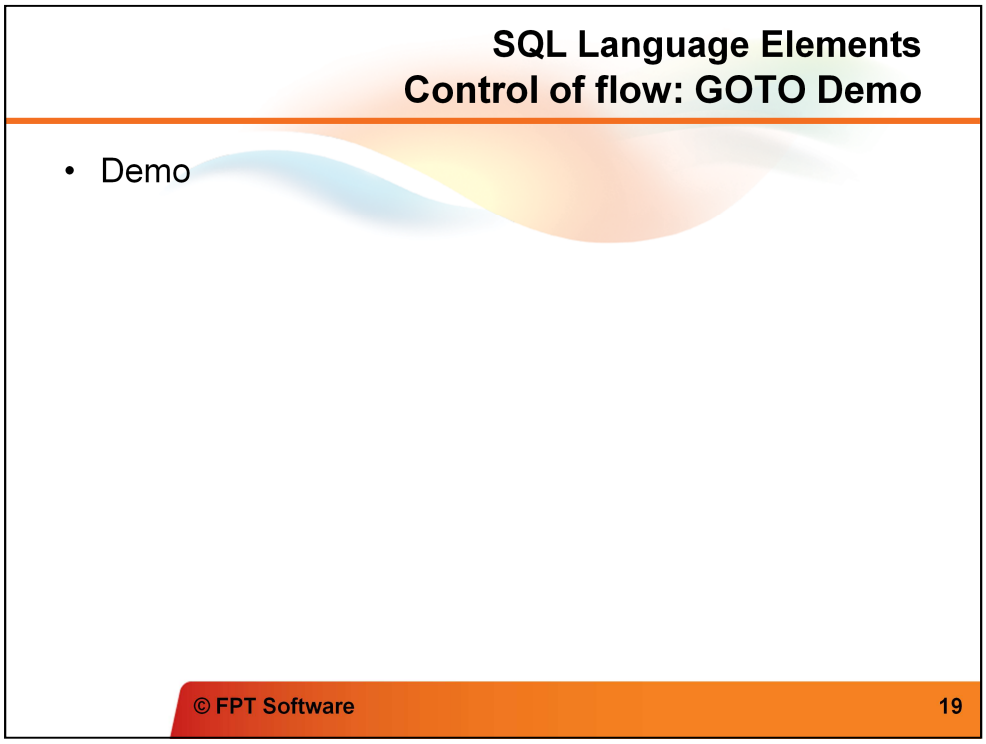
- Alters the flow of execution to a label. The Transact-SQL statement or statements that follow GOTO are skipped and processing continues at the label
- Syntax:

Define the label:

label :

Alter the execution:

GOTO label



See “Control flow demo” in Day2_Demo.docx

SQL Language Elements
Control of flow: RETURN

- Exits unconditionally from a query or procedure
- This will be discussed more detail in Stored Procedure section.
- Syntax

```
RETURN [ integer_expression ]
```

Stored Procedure Overview 1/2

- A stored procedure (SP) is a collection of SQL statements that SQL Server compiles into a single execution plan.
- It can accept input parameters, return output values as parameters, or return success or failure status messages

Stored Procedure Overview 2/2

- Stored procedures return data in four ways:
  - Output parameters, which can return either data (such as an integer or character value) or a cursor variable (cursors are result sets that can be retrieved one row at a time).
  - Return codes, which are always an integer value.
  - A result set for each SELECT statement contained in the stored procedure or any other stored procedures called by the stored procedure.
  - A global cursor that can be referenced outside the stored procedure.

Stored Procedure
Benefit of Using SP 1/2

- Benefit of Using SP
  - Reduced server/client network traffic:
    - Only the call to execute the procedure is sent across the network
  - Stronger security
    - When calling a procedure over the network, only the call to execute the procedure is visible. Therefore, malicious users cannot see table and database object names, embed Transact-SQL statements of their own, or search for critical data

Stored Procedure
Benefit of Using SP 2/2

- Benefit of Using SP
  - Reuse of code:
    - The code for any repetitious database operation is the perfect candidate for encapsulation in procedure (for instance, UPDATE data on a table)
  - Improve Performance:
    - Procedure is stored in cache area of memory when the stored procedure is first executed so that it can be used repeatedly. SQL Server does not have to recompile it every time the stored procedure is run.

Stored Procedure
Stored Procedure vs. SQL Statement

SQL Statement	*Stored Procedure*
	Creating - *Check syntax* - *Compile*
First Time - *Check syntax* - *Compile* - *Execute* - *Return data*	**First Time** - *Execute* - *Return data*
Second Time - *Check syntax* - *Compile* - *Execute* - *Return data*	**Second Time** - *Execute* - *Return data*

Stored Procedure
Create a SP- Syntax

- Create / Modify a SP

```
CREATE PROC[EDURE] procedure_name
   [ @parameter_name data_type] [= default]
   OUTPUT][,...,n]
AS
   SQL_statement_block
```

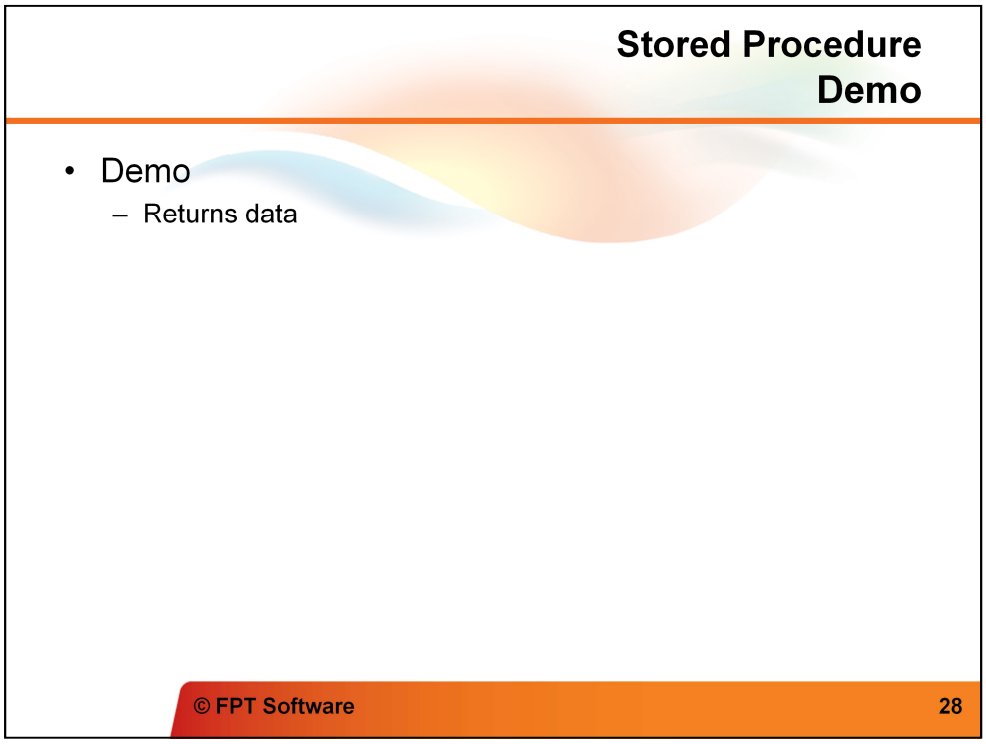
Stored Procedure
Exec, Update, Delete a SP- Syntax

- Execute a Procedure:
  - EXEC[UTE] procedure_name
- Update a Procedure

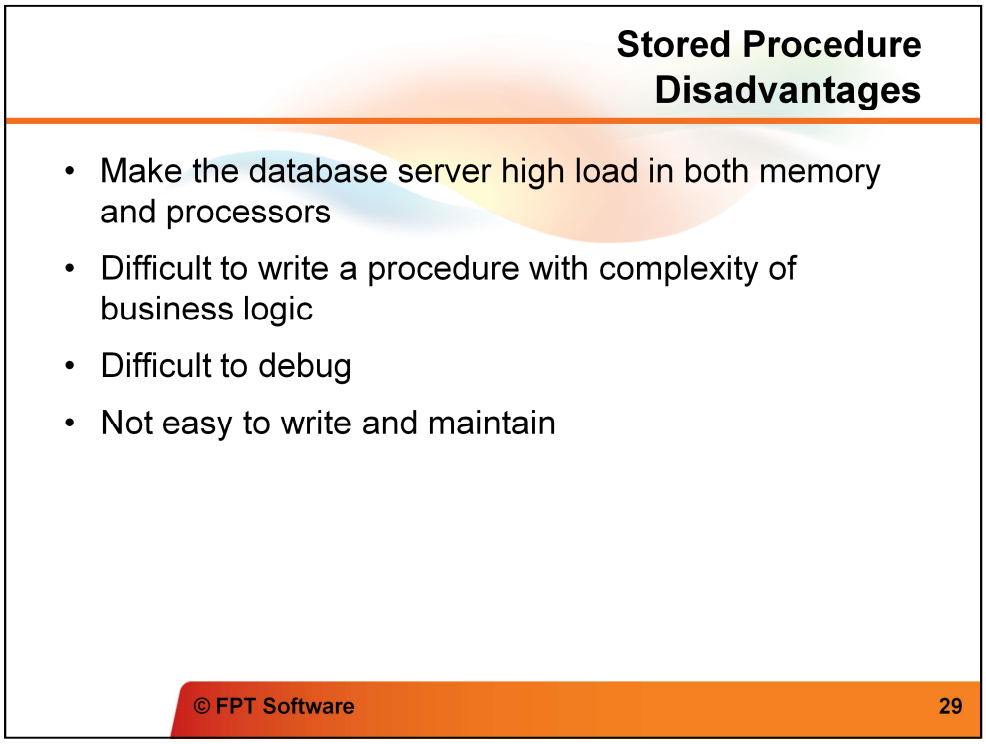
```
ALTER PROC[EDURE] procedure_name
   [ @parameter_name data_type]
     [= default] [OUTPUT]
   [,...,n]
AS
   SQL_statement(s)
```

- Delete a Procedure

```
DROP PROC[EDURE]  procedure_name
```



See “Stored Procedure” in Day2_Demo.docx



Stored procedures make the database server high load in both memory and processors. Instead of being focused on the storing and retrieving data, you could be asking the database server to perform a number of logical operations or a complex of business logic which is not the well designed in database server.

Stored procedure only contains declarative SQL so it is very difficult to write a procedure with complexity of business logic like other languages in application layer such as Java, C#, C++…

Stored procedure is difficult to debug. You cannot debug stored procedure in almost RDMBSs and in MySQL also. There are some workarounds on this problem but it still not easy enough to do so.

Store procedure is not easy to write and maintain. Writing and maintaining stored procedure is usually required specialized skill set that not all developers possess. This may introduced problems in both application development and maintain phase.

User-Defined Function (UDF)
What is a Function?

- UDF are routines that accept parameters, perform an action and return the result of that action as a value. The return value can be a single scalar value or a result set
  - Function cannot perform permanent environmental changes to SQL Server as Insert, Update, Delete on the real table
- UDF's types:
  - Scalar functions
  - Table-valued functions
    - Inline Table-valued Functions
    - Multi-statement Table-Valued Functions

User-Defined Function
Scalar Function Syntax

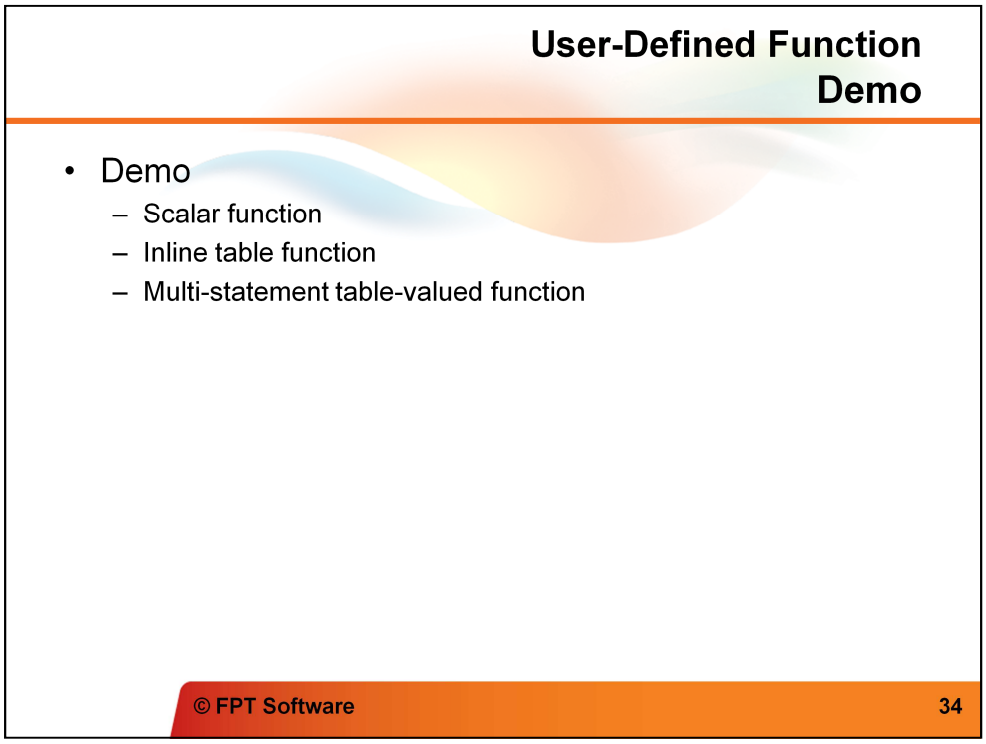
```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name data_type [ = default ] [ READONLY ]
   } [ ,...n ] ] )
RETURNS return_data_type
[ AS ]
BEGIN
   function_body
   RETURN scalar_expression
END
```

User-Defined Function
Inline Table-valued Functions Syntax

```
CREATE FUNCTION [schema_name.]function_name
( [ { @parameter_name data_type  [ = default ] } [ ,...n ] ] )
RETURNS TABLE
[ WITH < function_option > [ ,...n ] ]
[ AS ]
RETURN [ ( ] select_statement [ ) ]
```

User-Defined Function
Multi-statement Table-Valued Functions Syntax

```
CREATE FUNCTION [ schema_name. ] function_name
(( [ { @parameter_name data_type [ = default ] [ READONLY
   ] } [ ,...n ] ] )
)
RETURNS @return_variable TABLE <table_type_definition>
   [ WITH <function_option> [ ,...n ] ]
   [ AS ]
   BEGIN
      function_body
      RETURN
   END
```



See “Function demo” in Day2_Demo.docx

Trigger
What is a Trigger?

- A trigger is a special type of stored procedure that is executed automatically as part of a data modification.
- A trigger is created on a table and associated with one or more actions linked with a data modification (INSERT, UPDATE, or DELETE).
- When one of the actions for which the trigger is defined occurs, the trigger fires automatically
- Following are some examples of trigger uses:
  - Maintenance of duplicate and derived data
  - Complex column constraints
  - Cascading referential integrity
  - Complex defaults
  - Inter-database referential integrity

Trigger
Trigger Types

- We focus on two types of Trigger:
  - DML triggers (Standart triggers)
    - Raising whenever user change data on table or view (INSERT, UPDATE, DELETE).
    - Can be used to enforce business rules and data integrity
  - DDL triggers implemented whenever changing structure view, table, … (CREATE, ALTER, DROP…).

Trigger
DML Trigger

- DML Trigger includes:
  - AFTER Trigger: raising after changing data implemented successfully. AFTER is default and cannot use for view.
  - INSTEAD OF Trigger: implemented instead of SQL statements cause the trigger. INSTEAD OF trigger use for table and view.
    - Used to replace SQL statements interact with data.
    - Very useful when changing data on view that cannot implement in common way.

Trigger
DML Trigger Syntax

CREATE TRIGGER Trigger_name
ON *table* | *view*
[WITH ENCRYPTION]
{ FOR | AFTER | INSTEAD OF }
{[DELETE] [,] [INSERT] [,] [UPDATE] }
AS Sql_statement

ALTER TRIGGER trigger_name
ON (table | view)
[WITH ENCRYPTION] { { (FOR | AFTER | INSTEAD OF)
{ [DELETE] [,] [INSERT] [,] [UPDATE] }
[NOT FOR REPLICATION] AS sql_statement [...n]

DROP TRIGGER { *trigger_name* }

Trigger
Disable/Enable syntax

- Disable syntax

  Disable trigger <trigger_name> ON <table_name>

- Enable syntax

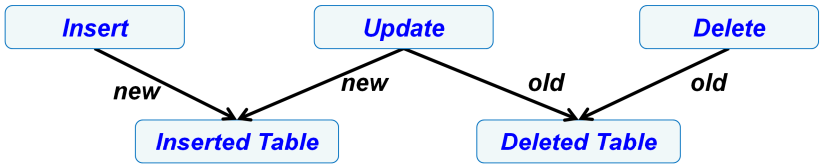
  Enable trigger <trigger_name> ON <table_name>

Trigger
Uses of triggers

- No change of front end code is required to perform:
  - Automation
  - Notification
  - Logging/Auditing
  - Maintaining de-normalized data

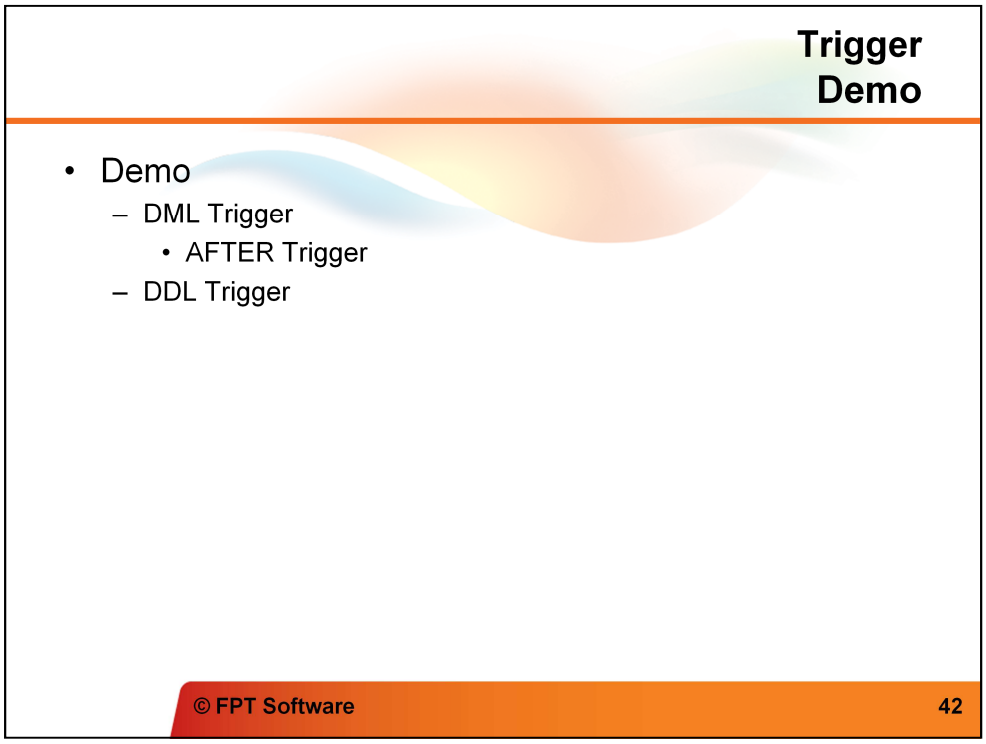
Trigger
Deleted and Inserted tables

- When you create a trigger, you have access to two temporary tables (the **deleted** and **inserted** tables). They are referred to as tables, but they are different from true database tables. They are stored in memory—not on disk.
- When the insert, update or delete statement is executed. All data will be copied into these tables with the same structure.



- The values in the inserted and deleted tables are accessible only within the trigger. Once the trigger is completed, these tables are no longer accessible.

This is the most important slide in Trigger section. Teacher should explain this very carefully.



See “Trigger demo” in Day2_Demo.docx

SQL Code Practice

- Should use Transaction for DELETE/ INSERT/ UPDATE action.
- Transaction will help Atomicity, Consistency, Isolation, Durability.

SQL Code Practices
Transaction

- A transaction is a method through which developers can define a unit of work logically or physically that, when it completes, leaves the database in a consistent state
- Transaction's properties (ACID):
  - **A**tomicity: All data modifications within the transaction must be both accepted and inserted successfully into the database, or none of the modifications will be performed.
  - **C**onsistency: Once the data has been successfully applied, or rolled back to the original state, all the data must remain in a consistent state, and the data must still maintain its integrity.
  - **I**solation: Any modification in one transaction must be isolated from any modifications in any other transaction
  - **D**urability: Any system failure (hardware or software) will not remove any changes applied

SQL Code Practice
Stored Procedure

- Include SET NOCOUNT ON statement: This will decrease network traffic
- For example:

```
CREATE PROC dbo.ProcName
AS
SET NOCOUNT ON;
--Procedure code here
GO
```

SQL Code Practice
Stored Procedure

- **Lining up parameter names, data types, and default values**

```
CREATE PROCEDURE dbo.User_Update
   @CustomerID      INT,
   @FirstName       VARCHAR(32)      = NULL,
   @LastName        VARCHAR(32)      = NULL,
   @Password        VARCHAR(16)      = NULL,
   @EmailAddress    VARCHAR(320)     = NULL,
   @Active          BIT              = 1,
   @LastLogin       SMALLDATETIME    = NULL
AS
BEGIN
```

SQL Code Practice
Stored Procedure

- Always try to declare variable at the beginning of SP : This will prevent recompiled and will improve performance
- Use IF EXISTS (SELECT 1 …) instead of IF EXISTS (SELECT * …)
- Avoid using WHILE: Loop in SP will cause performance problem.

SQL Code Practice

- **HAVING clause is used to filter the rows after all the rows are selected. It is just like a filter. Do not use HAVING clause for any other purposes.**
  Using:

```
SELECT subject, count(subject)
FROM student_details
WHERE subject != 'Science'  AND subject != 'Maths'
GROUP BY subject;
```

  Instead of:

```
SELECT subject, count(subject)
FROM student_details
GROUP BY subject
HAVING subject!= 'Science' AND subject!= 'Maths';
```

SQL Code Practices

- **Sometimes you may have more than one subqueries in your main query. Try to minimize the number of subquery block in your query.**

- **For example,**

  Using:

```
SELECT name FROM employee
WHERE (salary, age) = (SELECT MAX (salary), MAX (age)
FROM employee_details)  AND dept = 'Electronics';
```

  Instead of:

```
SELECT name FROM employee
WHERE salary = (SELECT MAX(salary) FROM employee_details)
AND age = (SELECT MAX(age) FROM employee_details)
AND emp_dept = 'Electronics';
```

SQL Code Practice

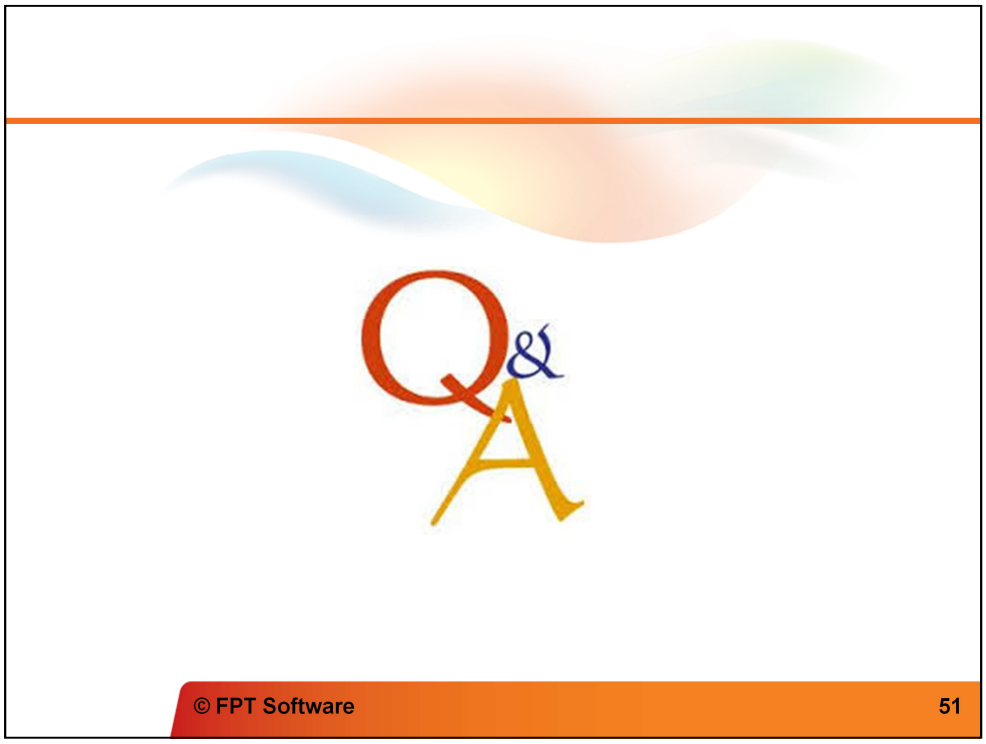
- **Use non-column expression on one side of the query because it will be processed earlier.**

Using:

```
SELECT id, name, salary
FROM employee
WHERE salary < 25000;
```

Instead of:

```
SELECT id, name, salary
FROM employee
WHERE salary + 10000 < 35000;
```


Q&A