

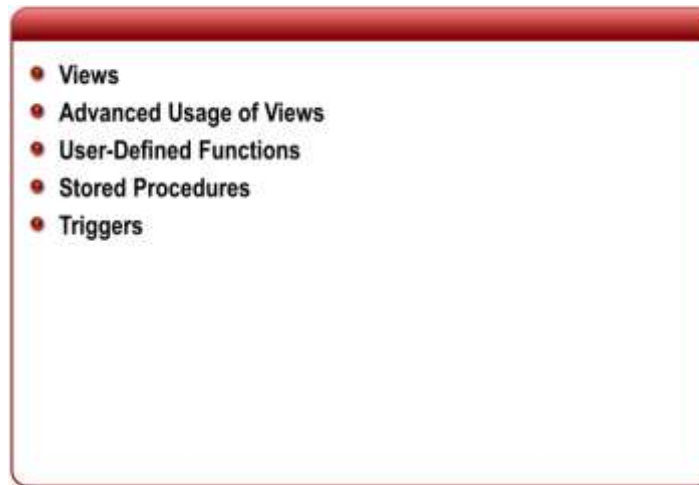
Module 7:

SQL Server Objects for Data Access

Contents

Module Overview	1
Lesson 1: Views.....	3
What Are Views?	4
Creating and Managing a View	5
Considerations When Creating Views	7
Restrictions for Modifying Data by Using Views	8
Indexed Views	9
Practice: Working with Views.....	10
Lesson 2: Advanced Usage of Views	12
Partitioned Views	13
Writing Distributed Queries	14
Working with Heterogeneous Data.....	16
Practice: Advanced Usage of Views	18
Lesson 3: User-Defined Functions.....	20
What Are User-Defined Functions?	21
Creating and Managing User-Defined Functions	22
Creating a Table Valued User-Defined Function	24
Restrictions When Creating User-Defined Functions.....	26
Practice: Working with User-Defined Functions	27
Lesson 4: Stored Procedures.....	29
What Are Stored Procedures?.....	30
How to Create Stored Procedures?.....	31
Details about Stored Procedure Execution	33
Stored Procedure Best Practices	35
Practice: Working with Stored Procedures	37
Lesson 5: Triggers.....	39
How Triggers Work?	40
How to Create Triggers?.....	41
Trigger Types and Limitations	43
Practice: Working with Triggers	44
Summary	46

Module Overview



A Microsoft SQL Server 2012 Database Engine relational database offers more functionality to access the data than just direct access to the tables. To enhance the data-logic layer you can create views, user-defined functions (UDF), stored procedures (SP), and triggers. With these data access objects you can enable a secure, and more effective method of working with the database.

By creating views you can mask complex data structures as normalized tables, hide confidential data from users, and support business logic objects without enable access directly to the tables.

User-defined functions offers standardized methods of interpret data and encapsulate complex formulas. As UDFs can return both scalar values as well as result sets, they offer a wide selection of uses.

With stored procedures in the database you can have control of the manipulating access to the tables. SPs offer a method for encapsulate all the inserts, updates, and deletes. This enables a secure, and effective, method for the application to access, store, change, and delete its data in a normalized data model.

Triggers enable access to data when it appears in the tables. A trigger can be activated after, or instead of, an insert, update, or a delete has been executed against a table. This can help logging access to tables, and advanced constraints on values, and offer automatic extraction of data into new data structures. A trigger is like a stored procedure, but connected to a table, and activated on changes of the content of the table.

User-defined functions and types, stored procedures, and triggers are objects that also can be defined as .Net CLR objects. .Net CLR extensibility is not covered in this module.

Objectives

After completing this module, you will be able to:

- Know how to create and use views.
- Benefit from the advanced usage of views.

- Know how to create and use user-defined functions.
- Know how to create and use stored procedures.
- Know how to create and use triggers.

Lesson 1: Views



A view is a data access object that is accessed as a table and you can execute SELECT, INSERT, UPDATE, and DELETE statements against it. Even if manipulating data through views are supported there are some restrictions. The content of a view is a SELECT statement that returns data in a result set.

To create, change, or delete a view we use DDL statements; CREATE VIEW, ALTER VIEW, and DELETE VIEW. The full definition of the view must be contained when running these statements.

While a view is not a table, there are still benefits of creating indexes on a view. Queries against tables that the view consists of may use the indexes of the view.

Objectives

After completing this lesson, you will be able to:

- Understand what a view is
- Create and manage views
- Understand when to use views
- Understand the restrictions for modifying data by using views
- Benefit from using indexed views

What Are Views?



Even if a view is accessed in the same manner as a table, a view is not a table and does not contain data. A view is more of a named query that can be referenced. The body of a view is a SELECT statement, or a WITH...SELECT.

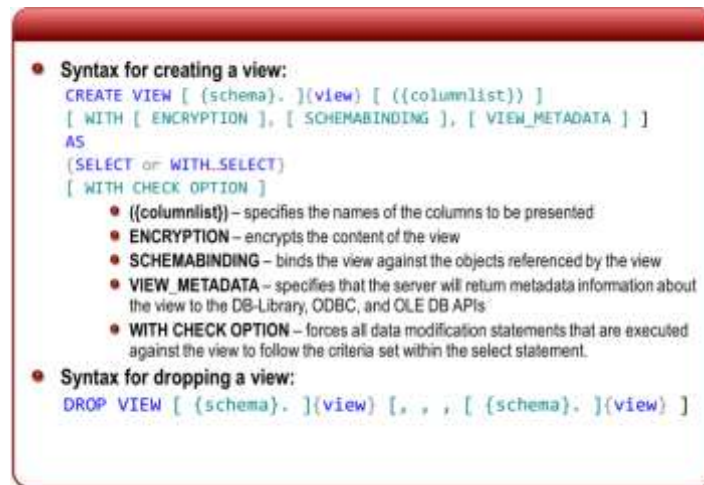
- “Why should I use views?”

A view can mask complex data models, separate the application from direct table access, target relevant data to a user, enhance the security of the database, and/or potentially offer a logical layer between the application and the tables that allow a DBA (DataBase Administrator) to optimize the database without changing the application.

- “Will a view automatically make my database more secure?”

A view contains a query against other objects within the database. As the view is accessed instead of the tables, the users do not need to have permissions to the tables directly.

Creating and Managing a View



In order to create a view we need to respect a set of syntax rules. Therefore the following syntax is used for creating a view:

```
CREATE VIEW [ {schema}. ](view) [ ({columnlist}) ]
[ WITH [ ENCRYPTION ], [ SCHEMABINDING ], [ VIEW_METADATA ] ]
AS
{SELECT or WITH...SELECT}
[ WITH CHECK OPTION ]
```

({columnlist}) – specifies the names of the columns to be presented. Column names are required if the query itself doesn't specify the column names.

ENCRYPTION – encrypts the content of the view in sys.syscomments and for sp_helptext.

SCHEMABINDING – binds the view against the objects referenced by the view. This prohibits changes of the objects that the view depends on. It is similar to a referencing constraint between columns, but on the objects instead.

VIEW_METADATA – specifies that the server will return metadata information about the view to the DB-Library, ODBC, and OLE DB APIs, instead of the base table or tables when browse-mode metadata is requested for a query referencing the view.

WITH CHECK OPTION – forces all data modification statements that are executed against the view to follow the criteria set within the select statement.

Example:

```
ALTER VIEW Sales.myView
WITH SCHEMABINDING
AS
SELECT p.LastName + ', ' + p.FirstName as Name, a.City
FROM Person.Person p
```

```
INNER JOIN Person.BusinessEntityAddress pa
ON p.BusinessEntityID = pa.BusinessEntityID
INNER JOIN Person.AddressType at ON pa.AddressTypeID =
at.AddressTypeID
INNER JOIN Person.Address a ON pa.AddressID = a.AddressID
WHERE at.Name = 'Home'
GO
SELECT * FROM Sales.myView
```

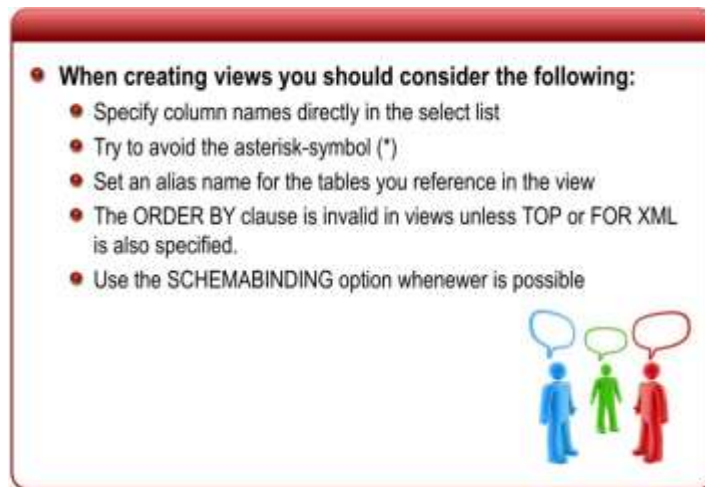
Syntax for changing a view

```
ALTER VIEW [ {schema}. ]{view} [ ({columnlist}) ]
[ WITH [ ENCRYPTION ], [ SCHEMABINDING ], [ VIEW_METADATA ] ]
AS
{SELECT or WITH...SELECT}
[ WITH CHECK OPTION ]
```

Syntax for dropping a view

```
DROP VIEW [ {schema}. ]{view} [, , , [ {schema}. ]{view} ]
```


Considerations When Creating Views



When creating a view it is recommended that you specify all the column names directly in the select list of the query. Every column in the result set must have a name and no two columns can have the same name.

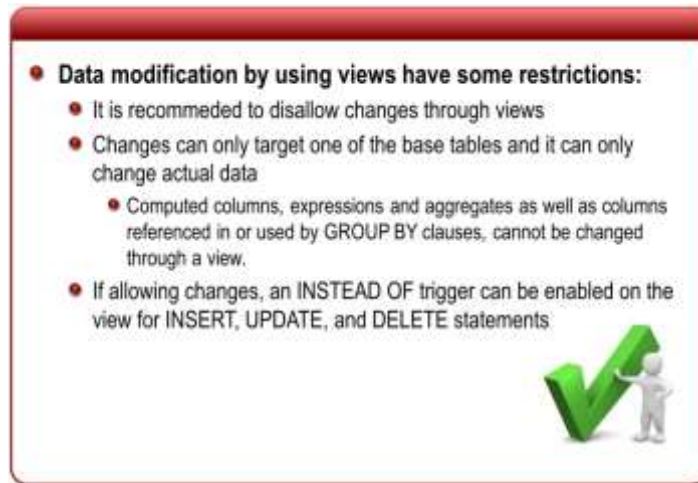
Never use the asterisk-symbol (*) instead of specifying the column names. If a column is dropped from a table and then recreated, the column order will change. This may create problems when an application uses the view.

Always set an alias name for the tables you reference in the view and always use this alias when referencing columns. This will prevent the “ambiguous column name”-error (Message 209) that occurs if a table that is referenced by the view has a column with the same name of another referenced table.

A view cannot contain the ORDER BY clause unless the SELECT statement uses the TOP, OFFSET (specifies the number of rows to skip, before starting to return rows from the query) or FOR XML (returns query results as XML elements) clause is specified.

It is always a good choice to use the SCHEMABINDING option.

Restrictions for Modifying Data by Using Views



Even if a view supports changes of data, e.g. INSERT, UPDATE, and DELETE statements, the common recommendation is to disallow changes through views.

If executing inserts, updates, and deletes against a view, the change can only target one of the base tables and it can only change actual data. Computed columns, expressions and aggregates as well as columns referenced in or used by GROUP BY clauses, cannot be changed through a view.

If allowing changes through a view, an INSTEAD OF trigger can be enabled on the view for INSERT, UPDATE, and DELETE statements that can extract data and inserted to insert, update and delete data in the actual base tables. This method means capturing the change and re-applying it against the tables.

Indexed Views

- **Indexed views can improve query performance**
 - View with at least one unique clustered index is called indexed view
 - After a unique clustered index is created you can create additional non clustered indexes
- **Indexed views can also decrease data modification performance**
- **Indexed view must be created with the SCHEMABINDING option**

```
CREATE UNIQUE CLUSTERED INDEX cix_myView ON
Sales.myView(BusinessEntityID)
GO
CREATE NONCLUSTERED INDEX nix_myView_Country ON
Sales.myView(Country)
GO
```

An indexed view is a view with at least one unique clustered index. After a unique clustered index is created on the view, additional non clustered indexes can be created. Indexed views can improve query performance as it actually stores the data returned by the view in the same way as data is stored for a table with a clustered index.

The Query Optimizer can utilize from indexes on views even if the query itself references the view. Indexed views can also decrease data modification performance as the indexes for the view also must be updated as the tables' data changes.

To create an indexed view, the view must be created with the SCHEMABINDING option. All specific SET options must be correct. The view definition must be deterministic.

Example:

```
CREATE UNIQUE CLUSTERED INDEX cix_myView ON
Sales.myView(BusinessEntityID)
GO
CREATE NONCLUSTERED INDEX nix_myView_Country ON
Sales.myView(Country)
GO
CREATE NONCLUSTERED INDEX nix_myView_StateProvince ON
Sales.myView(StateProvince)
GO
CREATE NONCLUSTERED INDEX nix_myView_City ON
Sales.myView(City)
GO
CREATE NONCLUSTERED INDEX nix_myView_FullName ON
Sales.myView(FullName)
```

Practice: Working with Views

In this practice, you will:

- Create a View called personDetailInfo which returns following rows: BusinessEntityID, title, last name, first name, middle name, city, state province name, country name for all persons who have Home as adress type

```
CREATE VIEW Sales.personDetailInfo
AS
SELECT p.BusinessEntityID ....
```

- Create a unique clustered index called cix_personDetailInfo
- Resolve following error message:
Msg 1939, Level 16, State 1, Line 1
Cannot create index on view 'personDetailInfo' because the view is not schema bound.

In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 1 from Module 7.

To successfully complete the exercise you need the following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
 - <http://msftdbprodsamples.codeplex.com/>

Exercise 1: Working with Views

1. Click **Start**→**Microsoft SQL Server 2012**→**SQL Server Management Studio**
2. On the Connect to Server dialog windows, under Server name, type the name of your local instance.
 - a. If it is default then just type (local)
3. Use Windows Authentication
4. Open **File** menu →**New** →**Database Engine Query**

Query Scenario

Create a View called personDetailInfo which returns following rows: BusinessEntityID, title, last name, first name, middle name, city, state province name, country name for all persons who have Home as adress type.

5. Type following T-SQL code to create a view:

```
CREATE VIEW Sales.personDetailInfo
AS
SELECT p.BusinessEntityID, ISNULL(p.Title + ' ', '') +
p.LastName + ', ' +
p.FirstName + ISNULL(' ' + p.MiddleName, '') AS FullName,
```

```
a.City,  
sp.Name AS StateProvince,  
cr.name AS Country  
FROM Person.Person p  
INNER JOIN Person.BusinessEntityAddress pa ON  
p.BusinessEntityID = pa.BusinessEntityID  
INNER JOIN Person.AddressType at ON pa.AddressTypeID =  
at.AddressTypeID  
INNER JOIN Person.Address a ON pa.AddressID = a.AddressID  
INNER JOIN Person.StateProvince sp ON a.StateProvinceID =  
sp.StateProvinceID  
INNER JOIN Person.CountryRegion cr ON sp.CountryRegionCode =  
cr.CountryRegionCode  
WHERE at.Name = 'Home'  
GO
```

6. Execute a view:

```
SELECT * FROM Sales.personDetailInfo
```

7. Create a unique clustered index called `cix_personDetailInfo`

```
CREATE UNIQUE CLUSTERED INDEX cix_personDetailInfo ON  
Sales.personDetailInfo (BusinessEntityID)
```

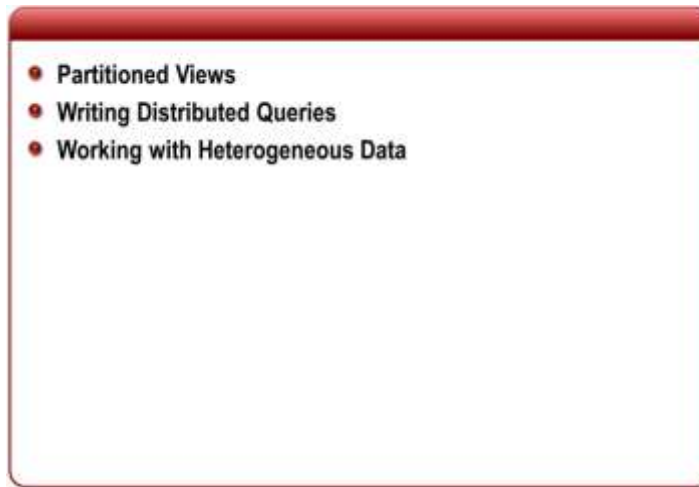
8. Why we are not able to create an unique clustered index for view `personDetailInfo`

```
Msg 1939, Level 16, State 1, Line 1  
Cannot create index on view 'personDetailInfo' because the  
view is not schema bound.
```

9. Alter view and add SCHEMABINDING

```
ALTER VIEW Sales.personDetailInfo  
WITH SCHEMABINDING  
AS  
...
```

Lesson 2: Advanced Usage of Views



If more than one table has the same structure, you can create a partitioned view that combines the result sets into a single result. For example, sales data that is divided into tables per year. The reason for dividing data in such a manner could be based on optimizing different types of data and to have smaller sized tables and indexes that need to be managed.

When working in larger environments, the data you need for a particular query could be located on another server, or in other database structures, for example, on another instance of SQL Server, on an Oracle database, in an Excel-file or in an Access database. Data from multiple sources can be accessed either by creating Linked Server objects or by using the `OPENDATASOURCE()`, or `OPENROWSET()` functions.

Depending on where the data resides and how the tables are addressed, the view is either partitioned or partition distributed.

Objectives

After completing this lesson, you will be able to:

- Create a Partitioned View
- Write a Distributed Query
- Work with Heterogeneous Data

Partitioned Views

- Combines two or more base tables of the same structure with the UNION ALL clause when all columns from all tables are:
 - referenced
 - have the same data types
 - have same ordinal position

```
CREATE VIEW Sales.mySalesAllYears
AS
SELECT
StoreID, SalesOrderID, OrderDate, TotalSalesAmount
FROM Sales.SalesYear2012
UNION ALL
SELECT
StoreID, SalesOrderID, OrderDate, TotalSalesAmount
FROM Sales.SalesYear2011
UNION ALL
SELECT
StoreID, SalesOrderID, OrderDate, TotalSalesAmount
FROM Sales.SalesYear2010
```

A partitioned view is defined as a view that combines two or more base tables of the same structure with the UNION ALL clause, and when all columns from all tables are referenced, and the columns have the same data types, and same ordinal position. Constraints on the tables should also be constructed in a manner that overlapping data should not be possible if insert, update, and delete are allowed. The same column name cannot appear multiple times in the column list.

The partitioning column should be part of the table's primary key and it should not be: computed, an identity column, default, or timestamp.

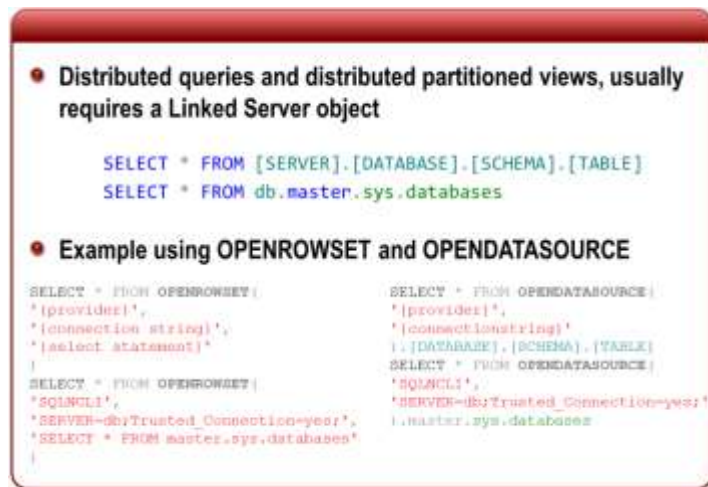
The tables can either be local tables, tables on other SQL Servers addressed by its four-part name [SERVER].[DATABASE].[SCHEMA].[TABLE], OPENROWSET() or OPENDATASOURCE() functions.

If the tables are local in the database, the view is partitioned. If the tables are on different SQL Server instances, the view is distributed and partitioned.

Examples:

```
CREATE VIEW Sales.mySalesAllYears
AS
SELECT
StoreID, SalesOrderID, OrderDate, TotalSalesAmount
FROM Sales.SalesYear2012
UNION ALL
SELECT
StoreID, SalesOrderID, OrderDate, TotalSalesAmount
FROM Sales.SalesYear2011
UNION ALL
SELECT
StoreID, SalesOrderID, OrderDate, TotalSalesAmount
FROM Sales.SalesYear2010
```

Writing Distributed Queries



Working with distributed queries and distributed partitioned views, usually requires a Linked Server object. Linked servers are created by executing the following procedures: `sp_addlinkedserver`, `sp_serveroption`, and `sp_addlinkedsrvlogin`. This creates a stored connection string from this server to the remote data source, configuring connection properties and security for that connection.

Functions **OPENROWSET** and **OPENDATASOURCE** both return result sets and support OLE DB data sources as targets such as Linked Servers. They should only be referenced infrequently. If the remote data sources is used often a Linked Server object is recommended.

Remember that using Windows Authentication is much more secure than using SQL Server Authentication when working with Linked Servers, **OPENROWSET** and **OPENDATASOURCE**.

Example using Linked Servers

```
SELECT * FROM [SERVER].[DATABASE].[SCHEMA].[TABLE]
SELECT * FROM db.master.sys.databases
```

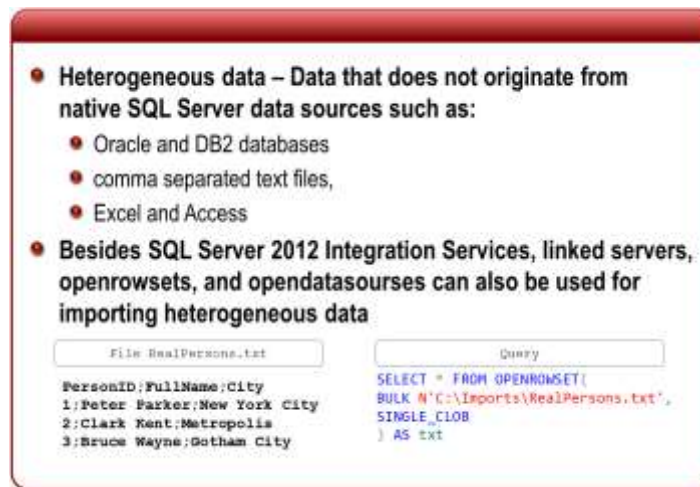
Example using OPENROWSET

```
SELECT * FROM OPENROWSET (
    '{provider}',
    '{connection string}',
    '{select statement}'
)
SELECT * FROM OPENROWSET (
    'SQLNCLI',
    'SERVER=db;Trusted_Connection=yes;',
    'SELECT * FROM master.sys.databases'
)
```

Example using OPENDATASOURCE


```
SELECT * FROM OPENDATASOURCE (
    '{provider}',
    '{connectionstring}'
) . [DATABASE] . [SCHEMA] . [TABLE]
SELECT * FROM OPENDATASOURCE (
    'SQLNCLI',
    'SERVER=db;Trusted_Connection=yes;'
) .master.sys.databases
```

Working with Heterogeneous Data



Data that does not originate from native SQL Server data sources are usually called Heterogeneous data. This can be anything from Oracle and DB2 databases, as well as comma separated text files, Excel, and Access databases. There are almost no limitation from where SQL Server can fetch, and process data.

Even if the preferable service for importing heterogeneous data is Microsoft SQL Server 2012 Integration Services, the ETL-platform (Extract, Transform and Load) of SQL Server, linked servers, openrowsets, and opendatasources are also of great use.

Example:

File:

C:\Imports\RealPersons.txt

Content:

```
PersonID;FullName;City
1;Peter Parker;New York City
2;Clark Kent;Metropolis
3;Bruce Wayne;Gotham City
```

Query:

```
SELECT * FROM OPENROWSET (
BULK N'C:\Imports\RealPersons.txt',
SINGLE_CLOB
) AS txt
```

Result set:

```
BulkColumn
-----
PersonID;FullName;City
1;Peter Parker;New York City
2;Clark Kent;Metropolis
```

```
3;Bruce Wayne;Gotham City
```

```
(1 row(s) affected)
```

Practice: Advanced Usage of Views

In this practice, you will:

- <action verb> Title of Exercise 1
- <action verb> Title of Exercise 2
- <action verb> Title of Exercise 3

***Practice Slide
(Placeholder)***

In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 2 from Module 7.

To successfully complete the exercise you need the following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
 - <http://msftdbprodsamples.codeplex.com/>

Exercise 2: Advanced Usage of Views

10. Click **Start**→**Microsoft SQL Server 2012**→**SQL Server Management Studio**
11. On the Connect to Server dialog windows, under Server name, type the name of your local instance.
 - a. If it is default then just type (local)
12. Use Windows Authentication
13. Open **File** menu →**New** →**Database Engine Query**

Query Scenario

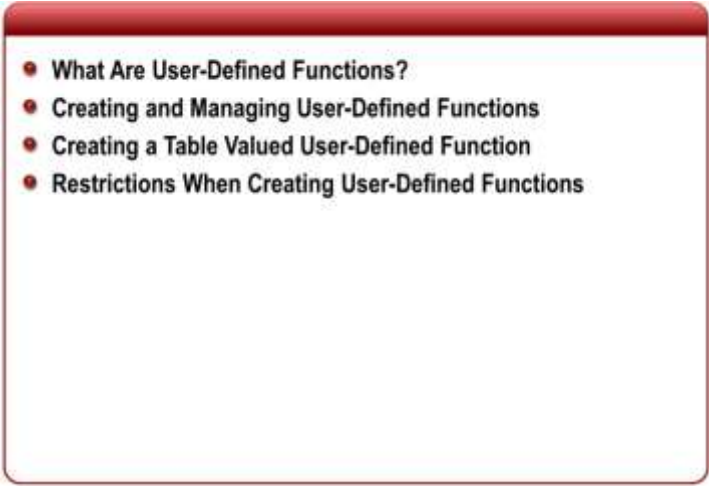
Create a View called personDetailInfo which returns following rows: BusinessEntityID, title, last name, first name, middle name, city, state province name, country name for all persons who have Home as adress type.

14. Type following T-SQL code to create a view:

```
CREATE TABLE dbo.SUPPLY1 (
    SupplyID INT PRIMARY KEY CHECK (SupplyID BETWEEN 1 and 150),
    Supplier CHAR(50)
);
CREATE TABLE dbo.SUPPLY2 (
    SupplyID INT PRIMARY KEY CHECK (SupplyID BETWEEN 151 and 300),
    Supplier CHAR(50)
);
CREATE TABLE dbo.SUPPLY3 (
    SupplyID INT PRIMARY KEY CHECK (SupplyID BETWEEN 301 and 450),
    Supplier CHAR(50)
);
CREATE TABLE dbo.SUPPLY4 (
    SupplyID INT PRIMARY KEY CHECK (SupplyID BETWEEN 451 and 600),
    Supplier CHAR(50)
);
GO
INSERT dbo.SUPPLY1 VALUES ('1', 'CaliforniaCorp'), ('5',
'BraziliaLtd');
INSERT dbo.SUPPLY2 VALUES ('231', 'FarEast'), ('280', 'NZ');
INSERT dbo.SUPPLY3 VALUES ('321', 'EuroGroup'), ('442',
'UKArchip');
INSERT dbo.SUPPLY4 VALUES ('475', 'India'), ('521',
'Afrique');
GO

CREATE VIEW dbo.all_Supplier_view
WITH SCHEMABINDING
AS
SELECT SupplyID, Supplier
FROM dbo.SUPPLY1
UNION ALL
SELECT SupplyID, Supplier
FROM dbo.SUPPLY2
UNION ALL
SELECT SupplyID, Supplier
FROM dbo.SUPPLY3
UNION ALL
SELECT SupplyID, Supplier
FROM dbo.SUPPLY4;
```

Lesson 3: User-Defined Functions

- 
- What Are User-Defined Functions?
 - Creating and Managing User-Defined Functions
 - Creating a Table Valued User-Defined Function
 - Restrictions When Creating User-Defined Functions

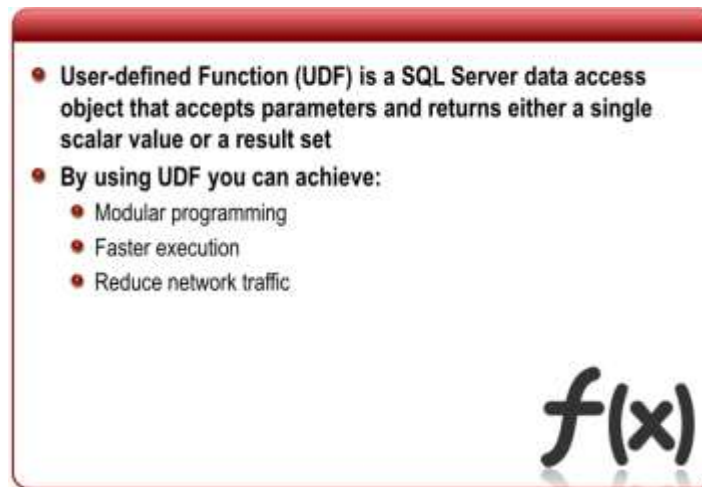
Databases have many varied needs for calculations, and expressions. In this context user-defined functions can encapsulate these complex formulas into objects that accept parameters and return either a single scalar value or a result set.

Objectives

After completing this lesson, you will be able to:

- Describe a user-defined function
- Create and manage user-defined functions
- Create and manage table valued user-defined functions
- Understand the use and limitations of user-defined functions

What Are User-Defined Functions?



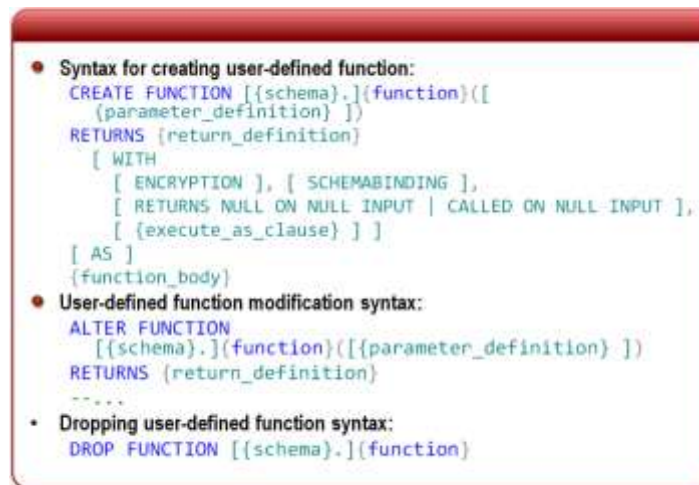
A User-defined Function (UDF) is a SQL Server data access object that accepts parameters and returns either a single scalar value or a result set. Their functionality offers a wide selection of uses. For example, they can contain expressions to validate certain data or dynamically build up a result set based in parameters supplied when executing the UDF.

Modular programming - a UDF contain the expression which you write once and then re-use it all over the database. If the expression needs a change, you have only one object that has to be altered. This offers consistency in your database.

Faster execution - a UDF is a specialized unit of work that does only one thing and its execution plan is cached and re-used. This means that using a UDF instead of an expression will give you better performance as the UDF doesn't need to be re-parsed and re-optimized every time it's executed.

Reduce network traffic - some operations that filter data on complex constraints cannot always be expressed as a single scalar expression. They can better benefit from UDFs as they can be invoked in the WHERE-clause of the query. They can, with a single parameter, generate a set of data that filters the data and reduces the rows on the server-side before sending the result back to the client.

Creating and Managing User-Defined Functions



In order to create a user-defined function use the following syntax:

```

CREATE FUNCTION[{schema}.]{function} ([{parameter_definition} ])
    RETURNS {return_definition}
    [ WITH
        [ ENCRYPTION ], [ SCHEMABINDING ],
        [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ],
        [ {execute_as_clause} ] ]
    [ AS ]
    {function_body}
  
```

Parameter definition – a parameter is defined with its name, data type and default behavior. For example @parameter int = 1 READONLY creates a parameter called @parameter of the data type integer with a default value of 1 and it is read-only within the function. A read-only parameter cannot be changed during the execution because it acts like it was a constant. If the parameter should be changeable within the function just leave out the word READONLY.

Return definition – depending if the function is scalar, in-line or multi-statement, the return is either defined by its data type or a return parameter is declared. A scalar function contains a function body that ends with a RETURN statement.

Execute as clause – you can control the execution context by adding an EXECUTE AS [CALLER | SELF | OWNER | 'database_user']. This is useful if the user running the UDF should not have access to the underlying sources directly.

Function body – contains the body of the UDF.

Example:

```

CREATE FUNCTION Person.CountVowels(@string nvarchar(max) =
N'')
    RETURNS int -- data type to be returned to the caller
    AS
    BEGIN -- Start of the function body
  
```



```

DECLARE
    @return int = 0,
-- return value containing the number of vowels
    @position int = 1,
-- position from where to read a single character from string
    @length int
SET @string = LOWER(@string)
-- convert to lower case in case of case sensitivity
SET @length = LEN(@string) -- get the length of the string
WHILE @position <= @length
-- iterate through the string character by character
BEGIN
    IF SUBSTRING(@string, @position, 1) LIKE N'[a, e, i, o,
u, y]'
```

```

        -- examines if the character is a vowel
        BEGIN
            SET @return = @return + 1
-- add one to the value of the return value
        END
        SET @position = @position + 1 -- next character
    END
    RETURN @return -- return the value to the caller
END -- End of the function body
GO
SELECT
    BusinessEntityID,
    LastName,
    Person.CountVowels(LastName) AS NumberOfVowels
FROM Person.Person
```

Modification of the user-defined function allows following syntax:

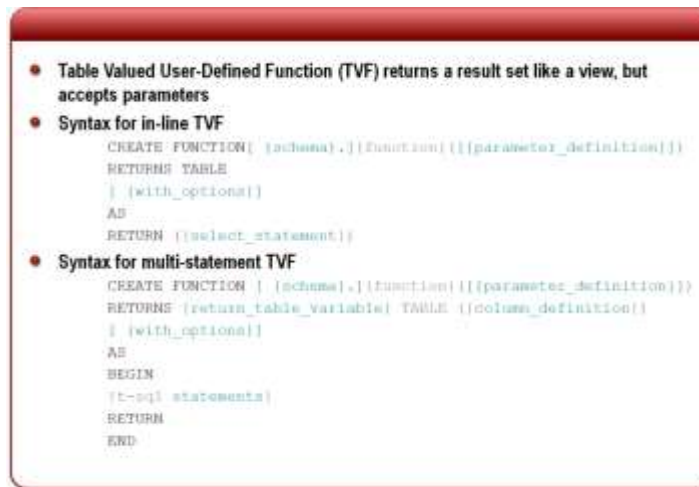
```

ALTER FUNCTION [{schema}.]{function}([{parameter_definition} ])
RETURNS {return_definition}
[ WITH
    [ ENCRYPTION ], [ SCHEMABINDING ],
    [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ],
    [ {execute_as_clause} ] ]
[ AS ]
{function_body}
```

In order to drop a user-defined function use following syntax:

```
DROP FUNCTION [{schema}.]{function}
```

Creating a Table Valued User-Defined Function



A Table Valued User-Defined Function (TVF) returns a result set like a view, but accepts parameters. TVFs can be either in-line or multi-statement.

The in-line TVF is defined with TABLE as the return type and contains only single RETURN with a single SELECT statement.

```
CREATE FUNCTION [ {schema}. ] {function} ( [{parameter_definition}] )
RETURNS TABLE
[ {with_options} ]
AS
RETURN ( {select_statement} )
```

Example:

```
CREATE FUNCTION Sales.SalesPerYearAndTerritory(@year int =
NULL)
RETURNS TABLE
AS
RETURN (
SELECT
    Year(h.OrderDate) AS SalesYear,
    t.[Group] + ': ' + t.Name AS SalesTerritory,
    SUM(d.UnitPrice * d.OrderQty) AS SalesTotal
FROM Sales.SalesTerritory t
    INNER JOIN Sales.SalesOrderHeader h
    ON t.TerritoryID = h.TerritoryID
    INNER JOIN Sales.SalesOrderDetail d
    ON h.SalesOrderID = d.SalesOrderID
WHERE
    (
        h.OrderDate >= CAST(CAST(@year as varchar(12)) + '-01-01'
as datetime2) AND
        h.OrderDate < DATEADD(year, 1, CAST(CAST(@year as
```

```

        varchar(12))+'-01-01' as datetime2))
    ) OR
    @year IS NULL
GROUP BY
    Year(h.OrderDate),
    t.[Group],
    t.Name
)
GO
SELECT * FROM Sales.SalesPerYearAndTerritory(2005)

```

The multi-statement TVF has a table variable as the return type and may contains inserts, updates, and deletes against the return table variable.

```

CREATE FUNCTION [ {schema}. ] {function} ([{parameter_definition}])
    RETURNS {return_table_variable} TABLE ({column_definition})
    [ {with_options}]
AS
BEGIN
    {t-sql statements}
RETURN
END

```

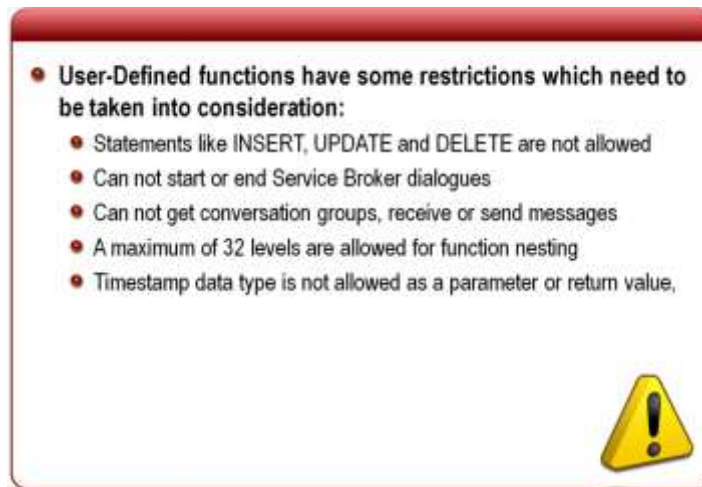
Example:

```

CREATE FUNCTION dbo.RangeOfNumbersWithStep(@start int,@end int,
                                           @step int)
    RETURNS @nums TABLE (num int)
AS
BEGIN
    WHILE @start <= @end
    BEGIN
        INSERT INTO @nums (num) VALUES (@start)
        SET @start = @start + @step
    END
    RETURN
END
GO
SELECT * FROM dbo.RangeOfNumbersWithStep(1, 1000, 11)

```

Restrictions When Creating User-Defined Functions



A function cannot perform actions that change the state of data. Statements like INSERT, UPDATE and DELETE against tables are not allowed. It can therefore not start or end Service Broker dialogues or get conversation groups, receive or send messages.

A maximum of 32 levels are allowed for function nesting. Exceeding this limit will cause the full nesting chain to fail.

You can execute stored procedures within a UDF and TVF as long as the procedure does not change the data.

All data types are allowed for a parameter and return value, except the timestamp data type.

Practice: Working with User-Defined Functions

In this practice, you will:

- <action verb> Title of Exercise 1
- <action verb> Title of Exercise 2
- <action verb> Title of Exercise 3

***Practice Slide
(Placeholder)***

In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 3 from Module 7.

To successfully complete the exercise you need the following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
 - <http://msftdbprodsamples.codeplex.com/>

Exercise 3: Working with User-Defined Functions

15. Click **Start→Microsoft SQL Server 2012→SQL Server Management Studio**
16. On the Connect to Server dialog windows, under Server name, type the name of your local instance.
 - a. If it is default then just type (local)
17. Use Windows Authentication
18. Open **File** menu →**New** →**Database Engine Query**

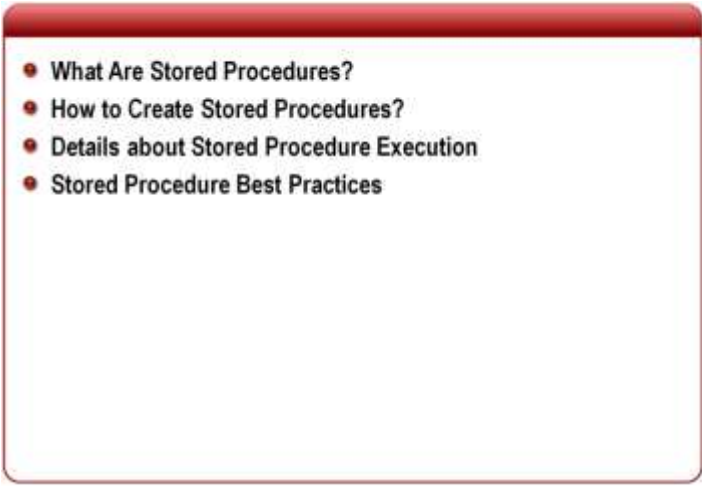
Query Scenario

Create a View called personDetailInfo which returns following rows: BusinessEntityID, title, last name, first name, middle name, city, state province name, country name for all persons who have Home as adress type.

19. Type following T-SQL code to create a view:

```
CREATE FUNCTION dbo.ufnGetInventoryStock (@ProductID int)
RETURNS int
AS
BEGIN
    DECLARE @ret int;
    SELECT @ret = SUM(p.Quantity)
    FROM Production.ProductInventory p
    WHERE p.ProductID = @ProductID
        AND p.LocationID = '6';
    IF (@ret IS NULL)
        SET @ret = 0;
    RETURN @ret;
END;
```

Lesson 4: Stored Procedures

- 
- What Are Stored Procedures?
 - How to Create Stored Procedures?
 - Details about Stored Procedure Execution
 - Stored Procedure Best Practices

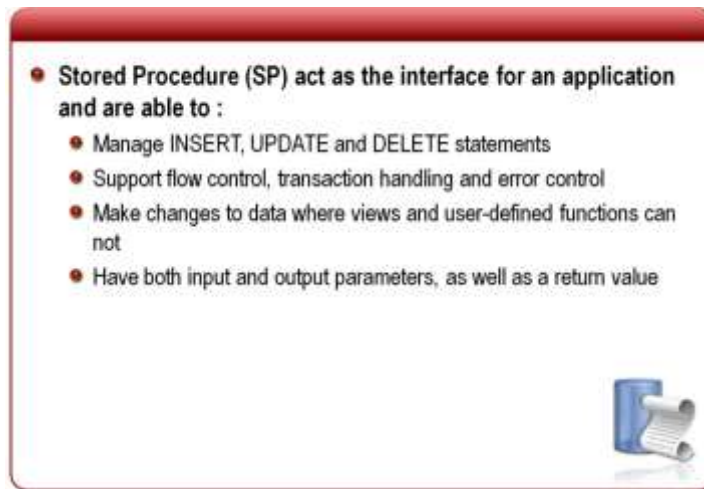
A Stored Procedure (SP) is a SQL Server data access object that can execute almost any T-SQL statement. Stored procedures are useful for controlling INSERT, UPDATE, and DELETE statements in a database as they support full flow-control, transactions and error control.

Objectives

After completing this lesson, you will be able to:

- Describe what a stored procedure is
- Create stored procedures
- Understand how a stored procedure executes
- Know some of the stored procedure best practices

What Are Stored Procedures?

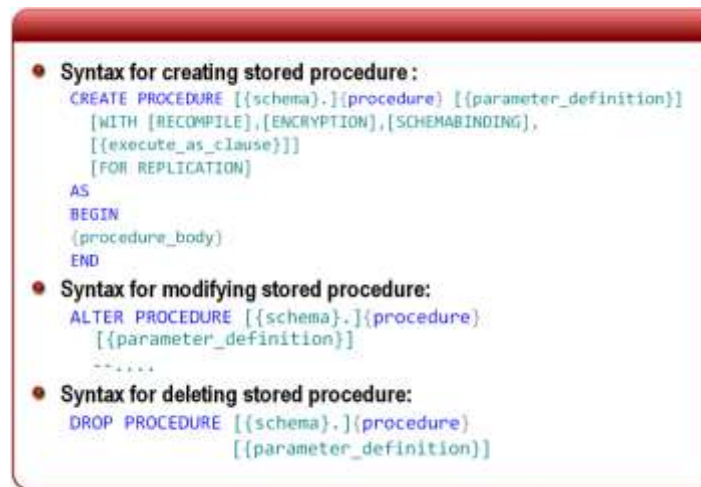


Stored Procedure (SP) act as the interface for an application to manage all the INSERT, UPDATE and DELETE statements. As they are programmable objects they support flow control, transaction handling and error control. Store procedures allow changes to data where views and user-defined functions do not.

SPs can have both input and output parameters, as well as a return value. Procedure parameters can have TABLE as a type, usually called a Table-Valued Parameter (TVP).

In a normalized data model, SCHEMAS can represent the objects in an application, e.g. the Person-object can be represented as the schema [Person]. In the same manner a view can represent an entity or collection of objects, in this scenario the view [Person].[Persons]. Stored procedures will become methods of the object, for example; procedure [Person].[addPerson] could represent the method Person.Add in the application layer. A function in the database acts as functions in the application layer.

How to Create Stored Procedures?



In order to create a stored procedure use the following syntax:

```

CREATE PROCEDURE [{schema}.]{procedure} [{parameter definition}]
    [WITH [RECOMPILE], [ENCRYPTION], [SCHEMABINDING],
    [{execute_as_clause}]]
    [FOR REPLICATION]
AS
BEGIN
    {procedure_body}
END
  
```

Parameter definition – every parameter has a name and a data type, @parameter int and can hold a default value, be an output parameter, OUTPUT, and be read-only, READONLY. The parameter can also be a Table Valued Parameter; in which case a user-defined data type must be created ahead of the procedure.

Recompile – the procedure will be parsed, optimized and compiled for every execution. This is a good option for procedures that are executed on rare occasions.

Procedure body – is the body of the procedure and consists of T-SQL statements, transaction control, control flow statements and error control.

For replication – the procedure is created for replication and cannot be executed on the subscriber. If this option is supplied, the parameters are not allowed.

Example:

```

CREATE PROCEDURE
    Person.addPerson
    @FirstName nvarchar(400) = '',
    @LastName nvarchar(400) = '',
    @City nvarchar(400) = '',
    @PersonID int OUTPUT
  
```

```

AS
BEGIN
-- This procedure only examines input parameters and return
-- mock up result set.
-- Depending on the examination it returns a return value
IF LEN(@FirstName) != 0 AND LEN(@LastName) != 0
BEGIN
SET @PersonID = 1
SELECT
    @PersonID AS [PersonID], @FirstName AS [FirstName],
    @LastName AS [LastName],
    @City AS [City], getdate() AS [AddedDate]
Return 0
END
ELSE
BEGIN
-- If the required parameter is empty or not valid, print
--the message below
PRINT 'Missing required values'
-- If the required parameter is empty, print the message
--above
RETURN -1
END
END
GO
-- If the required parameter is empty, print the message below
DECLARE @NewPersonID int, @ReturnValue int
EXECUTE
    @ReturnValue = Person.addPerson
    @FirstName = 'Mattias', @LastName = 'Lind', @City =
    'Ludvika'
    , @PersonID = @NewPersonID OUTPUT
SELECT @ReturnValue AS [ReturnValue], @NewPersonID AS
[NewPersonID]

```

Stored procedure modification requires following syntax:

```

ALTER PROCEDURE [{schema}.]{procedure} [{parameter_definition}]
    [WITH [RECOMPILE], [ENCRYPTION], [SCHEMABINDING],
    [{execute as clause}]]
    [FOR REPLICATION]
AS
BEGIN
    {procedure_body}
END

```

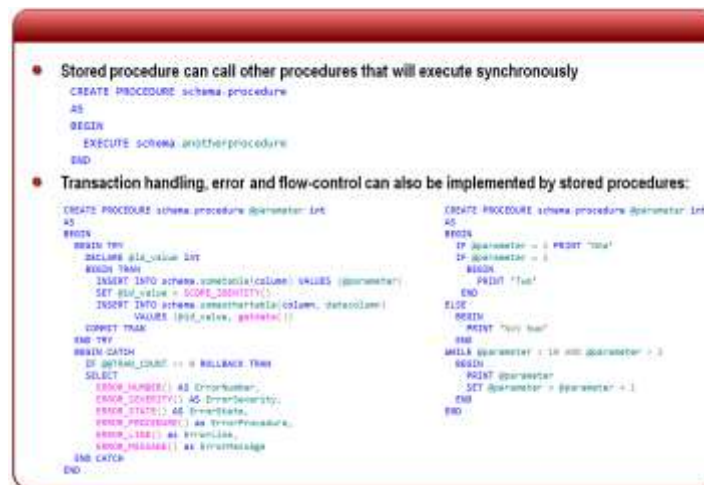
In order to drop a stored procedure use the following syntax

```

DROP PROCEDURE [{schema}.]{procedure} [{parameter_definition}]

```

Details about Stored Procedure Execution



A stored procedure is executed as one batch but can call other procedures that will execute synchronously. You can implement control flow, transaction handling and error control within the body of the procedure. In the next example you can see how you can execute a procedure within a procedure:

```
CREATE PROCEDURE schema.procedure
AS
BEGIN
    EXECUTE schema.anotherprocedure
END
```

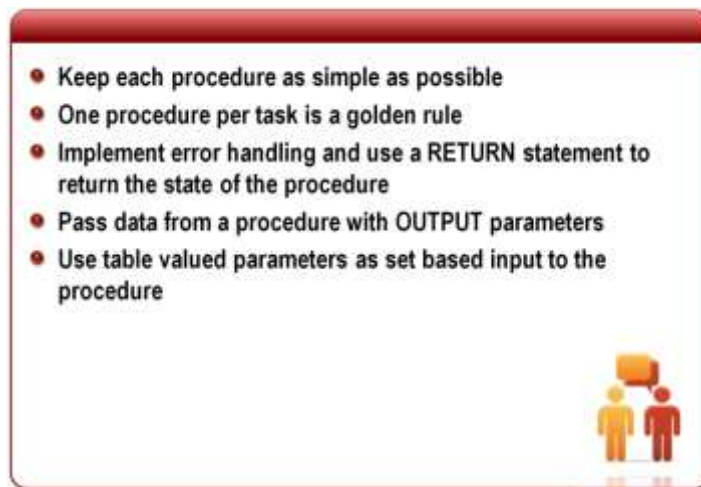
For achieving flow-control in a procedure you can use following syntax

```
CREATE PROCEDURE schema.procedure @parameter int
AS
BEGIN
    IF @parameter = 1 PRINT 'One'
    IF @parameter = 2
        BEGIN
            PRINT 'Two'
        END
    ELSE
        BEGIN
            PRINT 'Not two'
        END
    WHILE @parameter < 10 AND @parameter > 2
        BEGIN
            PRINT @parameter
            SET @parameter = @parameter + 1
        END
END
```

Stored procedures can also be used for transaction handling and error control

```
CREATE PROCEDURE schema.procedure @parameter int
AS
BEGIN
    BEGIN TRY
        DECLARE @id_value int
        BEGIN TRAN
            INSERT INTO schema.sometable(column) VALUES (@parameter)
            SET @id_value = SCOPE_IDENTITY()
            INSERT INTO schema.someothertable(column, datecolumn)
                VALUES (@id_value, getdate())
        COMMIT TRAN
    END TRY
    BEGIN CATCH
        IF @@TRAN_COUNT <> 0 ROLLBACK TRAN
        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
END
```

Stored Procedure Best Practices



When creating stored procedures, a good rule of thumb is to keep each procedure as simple as possible. If it takes more than five minutes to read through the procedure and understand its logic, it is probably too complex and can be split up into additional procedures.

One procedure per task is another golden rule.

Always implement error handling in the procedure and use a RETURN statement to return the state of the procedure.

Pass data from a procedure with OUTPUT parameters. For example Person.addPerson should have an OUTPUT parameter that returns the id for the new record.

Use table valued parameters as set based input to the procedure.

Example:

```
CREATE TYPE dbo.myTableType -- Create the table-valued type
AS TABLE (
    RowNum int NOT NULL IDENTITY(1,1),
    Value nvarchar(max)
)
GO
CREATE PROCEDURE dbo.myTableValuedProcedure
    @NumberOfRows int OUTPUT,
    @TableValuedParameter dbo.myTableType READONLY
-- Table-valued parameter as READONLY
AS
BEGIN
    SET NOCOUNT ON;
    DECLARE @Values nvarchar(max) = '',
            @ValuesOrdered nvarchar(max) = ''
    SELECT
        @values = @values + Value + ', '
    FROM @TableValuedParameter
```

```
ORDER BY RowNum ASC
SELECT
    @valuesOrdered = @valuesOrdered + Value + ', '
FROM @TableValuedParameter
ORDER BY Value ASC
SELECT
    @NumberOfRows = MAX(RowNum)
FROM @TableValuedParameter
SELECT
    LEFT(@values, LEN(@values) - 1) AS ValueList,
    LEFT(@valuesOrdered, LEN(@valuesOrdered) - 1)
        AS ValueListOrdered
END
GO -- Now use it
DECLARE
    @TableValuedVariable dbo.myTableType,
    @RowsAdded int
INSERT INTO @TableValuedVariable (Value)
SELECT name FROM sys.databases
EXECUTE dbo.myTableValuedProcedure
    @RowsAdded OUTPUT,
    @TableValuedVariable
SELECT @RowsAdded
```

Practice: Working with Stored Procedures

In this practice, you will:

- <action verb> Title of Exercise 1
- <action verb> Title of Exercise 2
- <action verb> Title of Exercise 3

***Practice Slide
(Placeholder)***

In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 4 from Module 7.

To successfully complete the exercise you need the following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
 - <http://msftdbprodsamples.codeplex.com/>

Exercise 4: Working with Stored Procedures

20. Click **Start→Microsoft SQL Server 2012→SQL Server Management Studio**
21. On the Connect to Server dialog windows, under Server name, type the name of your local instance.
 - a. If it is default then just type (local)
22. Use Windows Authentication
23. Open **File** menu →**New** →**Database Engine Query**

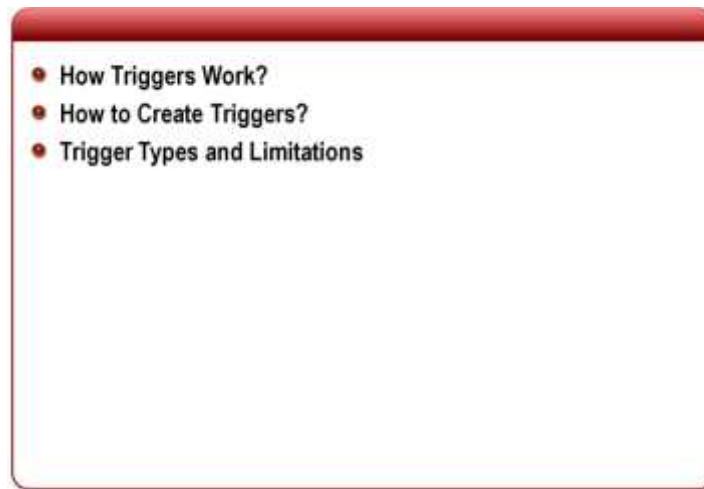
Query Scenario

Create a View called personDetailInfo which returns following rows: BusinessEntityID, title, last name, first name, middle name, city, state province name, country name for all persons who have Home as adress type.

24. Type following T-SQL code to create a view:

```
CREATE PROCEDURE HumanResources.uspGetEmployees
    @LastName nvarchar(50) = 'D%',
    @FirstName nvarchar(50) = '%'
AS
    SET NOCOUNT ON;
    SELECT FirstName, LastName, Department
    FROM HumanResources.vEmployeeDepartmentHistory
    WHERE FirstName LIKE @FirstName AND LastName LIKE
@LastName;
GO
```


Lesson 5: Triggers



A trigger is a database object that is attached to a table and in many ways is similar to a stored procedure. The primary difference between a trigger and a stored procedure is that the trigger is attached to a table and is activated when an INSERT, UPDATE or DELETE occurs. You specify the modification action(s) that activates the trigger when it is created. Within their body triggers can benefit from reading, and interpreting, both previous (DELETED) and new (INSERTED) data.

Triggers can also aid in validating data, log actions and much more. It is common to see triggers that carry out actions that are supposed to be part of stored procedures, e.g. splitting data into multiple tables or calculate data such as aggregating sales. In most cases, triggers are not the best method for such of actions.

Objectives

After completing this lesson, you will be able to:

- Understand how triggers work
- Know how to create a trigger
- Understand trigger types and limitations

How Triggers Work?

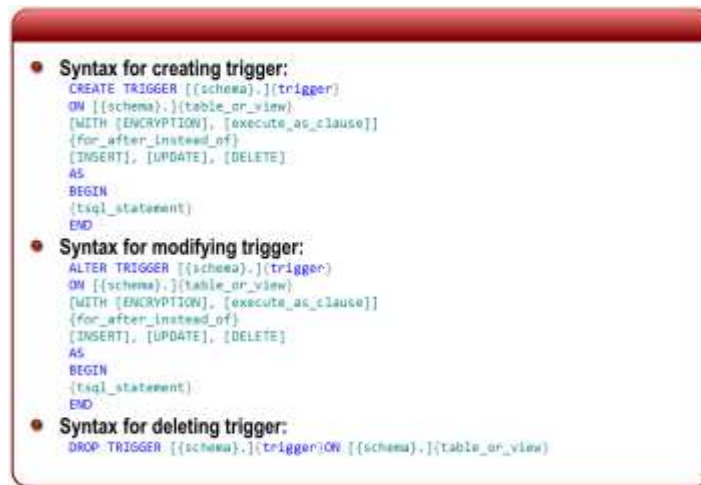
- **Trigger is database object attached to a table and is activated after or instead an INSERT, UPDATE or DELETE occurs**
 - AFTER trigger is executed within the calling transaction
 - Inserted and deleted are two tables accessible within the trigger
 - An update statement uses both tables
- **INSTEAD OF trigger prevents the INSERT, UPDATE and/or DELETE**
 - Recommended for views that de-normalize highly normalized data models

A trigger is similar to a stored procedure, but is activated after or instead of an insert, update, or delete statement is launched against a table or a view.

An AFTER trigger is executed within the calling transaction and is executed synchronously. It is activated on INSERT, UPDATE and/or DELETE against its table or view. It can read the data available in transaction. There are two special tables accessible within the trigger called inserted and deleted. An insert statement uses the inserted table and a delete statement uses the deleted table. An update statement uses both tables. The initiating transaction continues when the after trigger is complete.

An INSTEAD OF trigger prevents the INSERT, UPDATE and/or DELETE. This means that no data is changed. INSTEAD OF triggers make especially good use on views that de-normalize highly normalized data models. They can manage the data changing statements that need to be carried out in the normalized tables automatically when the statements are executed against the view. Then can, for example, utilize procedures with table valued parameters that are fed from both inserted and deleted tables.

How to Create Triggers?



In order to create a trigger use the following syntax:

```

CREATE TRIGGER [{schema}.]{trigger}
ON [{schema}.]{table_or_view}
[WITH [ENCRYPTION], [execute_as_clause]]
{for_after_instead_of}
[INSERT], [UPDATE], [DELETE]
AS
BEGIN
{tsql_statement}
END

```

[FOR] AFTER – the trigger is executed after the statement triggering the trigger. That means the change will be done, the trigger executes, and then control is returned to the originating statement.

[FOR] INSTEAD OF – instead of doing the change, the trigger executes, and then the control is returned to the originating statement. This type of trigger is especially useful on views to simulate a changing statement against a table.

INSERT – the trigger executes on an INSERT statement. The special table inserted contains the inserted rows.

UPDATE – the trigger executes on an UPDATE statement. Both the inserted and deleted special tables contain both the new and previous rows.

DELETE – the trigger executes on a DELETE statement. The special table deleted contains the deleted rows.

Trigger modification requires following syntax:

```

ALTER TRIGGER [{schema}.]{trigger}
ON [{schema}.]{table_or_view}
[WITH [ENCRYPTION], [execute_as_clause]]
{for_after_instead_of}
[INSERT], [UPDATE], [DELETE]

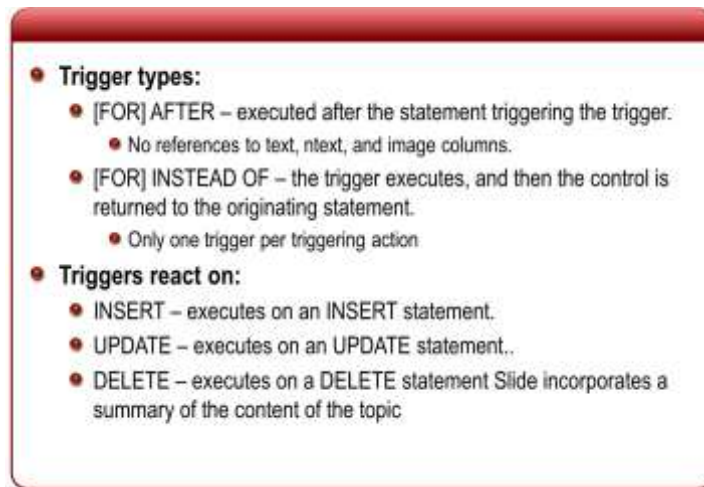
```

```
AS  
BEGIN  
    {tsql_statement}  
END
```

In order to remove a trigger use the following syntax

```
DROP TRIGGER [{schema}.]{trigger} ON [{schema}.]{table_or_view}
```

Trigger Types and Limitations



There are two main types of triggers; AFTER and INSTEAD OF. An AFTER trigger is executed after the change has been made and can validate and/or change data referenced by the originating statement. The INSTEAD OF trigger executes before the change is made instead of changing the data and can be very useful if the database is highly normalized and the statement is directed against a view de-normalizing the tables.

Triggers react on three kinds of statements; INSERT, UPDATE and/or DELETE. This means the trigger can capture all changing activity against the tables and views on a content level. Triggers can use both inserted and deleted special tables. These tables contain its changing rows within the statement, both new (inserted) and previous (deleted) versions. This offers a means to evaluate, validate and change data automatically in the changing statement.

AFTER triggers can be applied to tables and you can have more than one per triggering action(INSERT, UPDATE, and/or DELETE). They support cascading referential integrity, are executed after constraint processing, declarative referential actions, the creation of inserted and deleted special tables. Triggers can also have the first and last executions specified. They allow column references to the varchar(max), nvarchar(max) and varbinary(max) in the inserted and deleted special tables but no references to text, ntext, and image columns.

INSTEAD OF triggers can be applied to both tables and views but allow only one trigger per triggering action. They cannot be created on tables that are targets of cascaded referential integrity constraints. They are executed before processing constraints instead of the triggering action after the creation of the inserted and deleted special tables. They support varchar(max), nvarchar(max), and varbinary(max), as well as text, ntext, and image, as column references in the inserted and deleted special tables.

Practice: Working with Triggers

In this practice, you will:

- <action verb> Title of Exercise 1
- <action verb> Title of Exercise 2
- <action verb> Title of Exercise 3

***Practice Slide
(Placeholder)***

In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 5 from Module 7.

To successfully complete the exercise you need the following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
 - <http://msftdbprodsamples.codeplex.com/>

Exercise 4: Working with Triggers

25. Click **Start**→**Microsoft SQL Server 2012**→**SQL Server Management Studio**
26. On the Connect to Server dialog windows, under Server name, type the name of your local instance.
 - a. If it is default then just type (local)
27. Use Windows Authentication
28. Open **File** menu →**New** →**Database Engine Query**

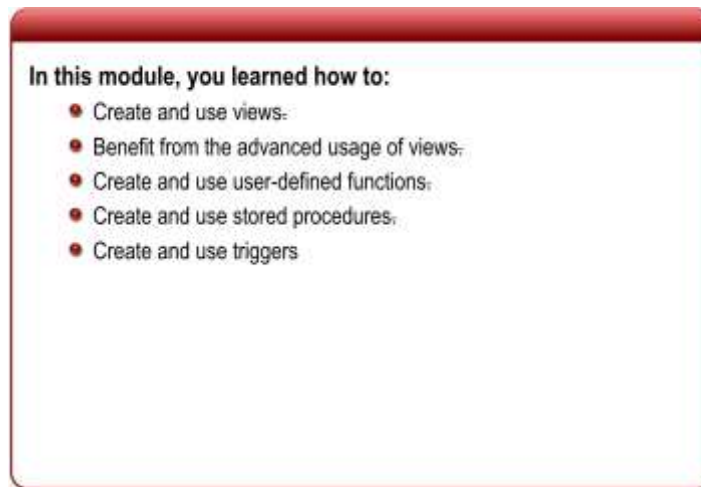
Query Scenario

Create a View called personDetailInfo which returns following rows: BusinessEntityID, title, last name, first name, middle name, city, state province name, country name for all persons who have Home as adress type.

29. Type following T-SQL code to create a view:

```
CREATE TRIGGER NewPODetail
ON Purchasing.PurchaseOrderDetail
AFTER INSERT AS
    UPDATE PurchaseOrderHeader
    SET SubTotal = SubTotal + LineTotal
    FROM inserted
    WHERE PurchaseOrderHeader.PurchaseOrderID =
inserted.PurchaseOrderID
```

Summary



In this module you learned about programmability options in Microsoft SQL Server 2012, especially how views, user-defined functions, procedures, and triggers work through their native T-SQL support. UDFs, SPs, and triggers can also be defined by using .Net CLR functionality which will be covered in one of the following modules.

Objectives

After completing this module, you learned:

- How to create and use views
- How to benefit from the advanced usage of views
- Know how to create and use user-defined functions
- Know how to create and use stored procedures
- Know how to create and use triggers