**FUNCTION**

**CREATE FUNCTION (Transact-SQL)**

Creates a user-defined function. This is a saved Transact-SQL or common language runtime (CLR) routine that returns a value. User-defined functions cannot be used to perform actions that modify the database state. User-defined functions, like system functions, can be invoked from a query. Scalar functions can be executed by using an EXECUTE statement like stored procedures.

|  |
| --- |
| **ms186755.note(en-us,SQL.100).gifNote:** |
| You cannot create a user-defined function with an OUTPUT INTO clause that has a table as its target. |

User-defined functions are modified by using [ALTER FUNCTION](http://msdn.microsoft.com/en-us/library/ms186967.aspx) and dropped by using [DROP FUNCTION](http://msdn.microsoft.com/en-us/library/ms190290.aspx).

Topic link icon[Transact-SQL Syntax Conventions](http://msdn.microsoft.com/en-us/library/ms177563.aspx)

http://i.msdn.microsoft.com/Global/Images/clear.gif Syntax

Scalar Functions

CREATE FUNCTION [ schema\_name. ] function\_name

( [ { @parameter\_name [ AS ][ type\_schema\_name. ] parameter\_data\_type

    [ = default ] [ READONLY ] }

    [ ,...n ]

  ]

)

RETURNS return\_data\_type

    [ WITH <function\_option> [ ,...n ] ]

    [ AS ]

    BEGIN

                function\_body

        RETURN scalar\_expression

    END

[ ; ]

Inline Table-Valued Functions

CREATE FUNCTION [ schema\_name. ] function\_name

( [ { @parameter\_name [ AS ] [ type\_schema\_name. ] parameter\_data\_type

    [ = default ] [ READONLY ] }

    [ ,...n ]

  ]

)

RETURNS TABLE

    [ WITH <function\_option> [ ,...n ] ]

    [ AS ]

    RETURN [ ( ] select\_stmt [ ) ]

[ ; ]

Multistatement Table-valued Functions

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>

    [ WITH <function\_option> [ ,...n ] ]

    [ AS ]

    BEGIN

                function\_body

        RETURN

    END

[ ; ]

CLR Scalar Functions

CREATE FUNCTION [ schema\_name. ] function\_name

( { @parameter\_name [AS] [ type\_schema\_name. ] parameter\_data\_type

    [ = default ] }

    [ ,...n ]

)

RETURNS { return\_data\_type }

    [ WITH <clr\_function\_option> [ ,...n ] ]

    [ AS ] EXTERNAL NAME <method\_specifier>

[ ; ]

CLR Table-Valued Functions

CREATE FUNCTION [ schema\_name. ] function\_name

( { @parameter\_name [AS] [ type\_schema\_name. ] parameter\_data\_type

    [ = default ] }

    [ ,...n ]

)

RETURNS TABLE <clr\_table\_type\_definition>

    [ WITH <clr\_function\_option> [ ,...n ] ]

    [ ORDER( <order\_clause> )]

    [ AS ] EXTERNAL NAME <method\_specifier>

[ ; ]

<order\_clause> ::=

{

<column\_name\_in\_clr\_table\_type\_definition>

[ ASC | DESC ]

} [ ,...n]

Method Specifier

<method\_specifier>::=

    assembly\_name.class\_name.method\_name

Function Options

<function\_option>::=

{

    [ ENCRYPTION ]

  | [ SCHEMABINDING ]

  | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]

  | [ EXECUTE\_AS\_Clause ]

}

<clr\_function\_option>::=

}

    [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]

  | [ EXECUTE\_AS\_Clause ]

}

Table Type Definitions

<table\_type\_definition>:: =

( { <column\_definition> <column\_constraint>

  | <computed\_column\_definition> }

        [ <table\_constraint> ] [ ,...n ]

)

<clr\_table\_type\_definition>::=

( { column\_name data\_type } [ ,...n ] )

<column\_definition>::=

{

    { column\_name data\_type }

    [ [ DEFAULT constant\_expression ]

      [ COLLATE collation\_name ] | [ ROWGUIDCOL ]

    ]

    | [ IDENTITY [ (seed , increment ) ] ]

    [ <column\_constraint> [ ...n ] ]

}

<column\_constraint>::=

{

    [ NULL | NOT NULL ]

    { PRIMARY KEY | UNIQUE }

      [ CLUSTERED | NONCLUSTERED ]

        [ WITH FILLFACTOR = fillfactor

        | WITH ( < index\_option > [ , ...n ] )

      [ ON { filegroup | "default" } ]

  | [ CHECK ( logical\_expression ) ] [ ,...n ]

}

<computed\_column\_definition>::=

column\_name AS computed\_column\_expression

<table\_constraint>::=

{

    { PRIMARY KEY | UNIQUE }

      [ CLUSTERED | NONCLUSTERED ]

            ( column\_name [ ASC | DESC ] [ ,...n ] )

        [ WITH FILLFACTOR = fillfactor

        | WITH ( <index\_option> [ , ...n ] )

  | [ CHECK ( logical\_expression ) ] [ ,...n ]

}

<index\_option>::=

{

    PAD\_INDEX = { ON | OFF }

  | FILLFACTOR = fillfactor

  | IGNORE\_DUP\_KEY = { ON | OFF }

  | STATISTICS\_NORECOMPUTE = { ON | OFF }

  | ALLOW\_ROW\_LOCKS = { ON | OFF }

  | ALLOW\_PAGE\_LOCKS ={ ON | OFF }

}

http://i.msdn.microsoft.com/Global/Images/clear.gif Arguments

*schema\_name*

Is the name of the schema to which the user-defined function belongs.

*function\_name*

Is the name of the user-defined function. Function names must comply with the rules for [identifiers](http://msdn.microsoft.com/en-us/library/ms175874.aspx) and must be unique within the database and to its schema.

|  |
| --- |
| **ms186755.note(en-us,SQL.100).gifNote:** |
| Parentheses are required after the function name even if a parameter is not specified. |

**@** *parameter\_name*

Is a parameter in the user-defined function. One or more parameters can be declared.

A function can have a maximum of 2,100 parameters. The value of each declared parameter must be supplied by the user when the function is executed, unless a default for the parameter is defined.

Specify a parameter name by using an at sign (**@**) as the first character. The parameter name must comply with the rules for identifiers. Parameters are local to the function; the same parameter names can be used in other functions. Parameters can take the place only of constants; they cannot be used instead of table names, column names, or the names of other database objects.

|  |
| --- |
| **ms186755.note(en-us,SQL.100).gifNote:** |
| ANSI\_WARNINGS is not honored when you pass parameters in a stored procedure, user-defined function, or when you declare and set variables in a batch statement. For example, if a variable is defined as **char(3)**, and then set to a value larger than three characters, the data is truncated to the defined size and the INSERT or UPDATE statement succeeds. |

[ *type\_schema\_name***.** ] *parameter\_data\_type*

Is the parameter data type, and optionally the schema to which it belongs. For Transact-SQL functions, all data types, including CLR user-defined types and user-defined table types, are allowed except the **timestamp** data type. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, user-defined table types and **timestamp** data types. The nonscalar types, **cursor** and **table**, cannot be specified as a parameter data type in either Transact-SQL or CLR functions.

If *type\_schema\_name* is not specified, the Database Engine looks for the *scalar\_parameter\_data\_type* in the following order:

* The schema that contains the names of SQL Server system data types.
* The default schema of the current user in the current database.
* The **dbo** schema in the current database.

[ **=** *default* ]

Is a default value for the parameter. If a *default* value is defined, the function can be executed without specifying a value for that parameter.

|  |
| --- |
| **ms186755.note(en-us,SQL.100).gifNote:** |
| Default parameter values can be specified for CLR functions except for the **varchar(max)** and **varbinary(max)** data types. |

When a parameter of the function has a default value, the keyword DEFAULT must be specified when the function is called to retrieve the default value. This behavior is different from using parameters with default values in stored procedures in which omitting the parameter also implies the default value.

READONLY

Indicates that the parameter cannot be updated or modified within the definition of the function. If the parameter type is a user-defined table type, READONLY should be specified.

*return\_data\_type*

Is the return value of a scalar user-defined function. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except the **timestamp** data type. For CLR functions, all data types, including CLR user-defined types, are allowed except the **text**, **ntext**, **image**, and **timestamp** data types. The nonscalar types, **cursor** and **table**, cannot be specified as a return data type in either Transact-SQL or CLR functions.

*function\_body*

Specifies that a series of Transact-SQL statements, which together do not produce a side effect such as modifying a table, define the value of the function. *function\_body* is used only in scalar functions and multistatement table-valued functions.

In scalar functions, *function\_body* is a series of Transact-SQL statements that together evaluate to a scalar value.

In multistatement table-valued functions, *function\_body* is a series of Transact-SQL statements that populate a TABLE return variable.

*scalar\_expression*

Specifies the scalar value that the scalar function returns.

TABLE

Specifies that the return value of the table-valued function is a table. Only constants and **@***local\_variables* can be passed to table-valued functions.

In inline table-valued functions, the TABLE return value is defined through a single SELECT statement. Inline functions do not have associated return variables.

In multistatement table-valued functions, **@***return\_variable* is a TABLE variable, used to store and accumulate the rows that should be returned as the value of the function. **@***return\_variable* can be specified only for Transact-SQL functions and not for CLR functions.

*select\_stmt*

Is the single SELECT statement that defines the return value of an inline table-valued function.

ORDER (<order\_clause>)

Specifies the order in which results are being returned from the table-valued function. For more information, see the section, "Guidance on Using Sort Order," later in this topic.

EXTERNAL NAME <method\_specifier> *assembly\_name***.***class\_name***.***method\_name*

Specifies the method of an assembly to bind with the function. *assembly\_name* must match an existing assembly in SQL Server in the current database with visibility on. *class\_name* must be a valid SQL Server identifier and must exist as a class in the assembly. If the class has a namespace-qualified name that uses a period (**.**) to separate namespace parts, the class name must be delimited by using brackets (**[** **]**) or quotation marks (**"** **"**). *method\_name* must be a valid SQL Server identifier and must exist as a static method in the specified class.

|  |
| --- |
| **ms186755.note(en-us,SQL.100).gifNote:** |
| By default, SQL Server cannot execute CLR code. You can create, modify, and drop database objects that reference common language runtime modules; however, you cannot execute these references in SQL Server until you enable the [clr enabled option](http://msdn.microsoft.com/en-us/library/ms175193.aspx). To enable this option, use [sp\_configure](http://msdn.microsoft.com/en-us/library/ms188787.aspx). |

*<*table\_type\_definition*>* **(** { <column\_definition> <column\_constraint>   | <computed\_column\_definition> }   [ <table\_constraint> ] [ **,**...*n* ] **)**

Defines the table data type for a Transact-SQL function. The table declaration includes column definitions and column or table constraints. The table is always put in the primary filegroup.

< clr\_table\_type\_definition > **(** { *column\_name* *data\_type* } [ **,**...*n* ] **)**

Defines the table data types for a CLR function. The table declaration includes only column names and data types. The table is always put in the primary filegroup.

**<function\_option>::= and <clr\_function\_option>::=**

Specifies that the function will have one or more of the following options.

ENCRYPTION

Indicates that the Database Engine will convert the original text of the CREATE FUNCTION statement to an obfuscated format. The output of the obfuscation is not directly visible in any catalog views. Users that have no access to system tables or database files cannot retrieve the obfuscated text. However, the text will be available to privileged users that can either access system tables over the [DAC port](http://msdn.microsoft.com/en-us/library/ms189595.aspx) or directly access database files. Also, users that can attach a debugger to the server process can retrieve the original procedure from memory at runtime. For more information about accessing system metadata, see [Metadata Visibility Configuration](http://msdn.microsoft.com/en-us/library/ms187113.aspx).

Using this option prevents the function from being published as part of SQL Server replication. This option cannot be specified for CLR functions.

SCHEMABINDING

Specifies that the function is bound to the database objects that it references. This condition will prevent changes to the function if other schema-bound objects are referencing it.

The binding of the function to the objects it references is removed only when one of the following actions occurs:

* The function is dropped.
* The function is modified by using the ALTER statement with the SCHEMABINDING option not specified.

A function can be schema bound only if the following conditions are true:

* The function is a Transact-SQL function.
* The user-defined functions and views referenced by the function are also schema-bound.
* The objects referenced by the function are referenced using a two-part name.
* The function and the objects it references belong to the same database.
* The user who executed the CREATE FUNCTION statement has REFERENCES permission on the database objects that the function references.

SCHEMABINDING cannot be specified for CLR functions or functions that reference alias data types.

RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT

Specifies the **OnNULLCall** attribute of a scalar-valued function. If not specified, CALLED ON NULL INPUT is implied by default. This means that the function body executes even if NULL is passed as an argument.

If RETURNS NULL ON NULL INPUT is specified in a CLR function, it indicates that SQL Server can return NULL when any of the arguments it receives is NULL, without actually invoking the body of the function. If the method of a CLR function specified in <method\_specifier> already has a custom attribute that indicates RETURNS NULL ON NULL INPUT, but the CREATE FUNCTION statement indicates CALLED ON NULL INPUT, the CREATE FUNCTION statement takes precedence. The **OnNULLCall** attribute cannot be specified for CLR table-valued functions.

EXECUTE AS Clause

Specifies the security context under which the user-defined function is executed. Therefore, you can control which user account SQL Server uses to validate permissions on any database objects that are referenced by the function.

|  |
| --- |
| **ms186755.note(en-us,SQL.100).gifNote:** |
| EXECUTE AS cannot be specified for inline user-defined functions. |

For more information, see [EXECUTE AS Clause (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188354.aspx).

**< column\_definition >::=**

Defines the table data type. The table declaration includes column definitions and constraints. For CLR functions, only *column\_name* and *data\_type* can be specified.

*column\_name*

Is the name of a column in the table. Column names must comply with the rules for identifiers and must be unique in the table. *column\_name* can consist of 1 through 128 characters.

*data\_type*

Specifies the column data type. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except **timestamp**. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, **char**, **varchar**, **varchar(max)**, and **timestamp**.The nonscalar type **cursor** cannot be specified as a column data type in either Transact-SQL or CLR functions.

DEFAULT *constant\_expression*

Specifies the value provided for the column when a value is not explicitly supplied during an insert. *constant\_expression* is a constant, NULL, or a system function value. DEFAULT definitions can be applied to any column except those that have the IDENTITY property. DEFAULT cannot be specified for CLR table-valued functions.

COLLATE *collation\_name*

Specifies the collation for the column. If not specified, the column is assigned the default collation of the database. Collation name can be either a Windows collation name or a SQL collation name. For a list of and more information about collations, see [Windows Collation Name (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188046.aspx) and [SQL Server Collation Name (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms180175.aspx).

The COLLATE clause can be used to change the collations only of columns of the **char**, **varchar**, **nchar**, and **nvarchar** data types.

COLLATE cannot be specified for CLR table-valued functions.

ROWGUIDCOL

Indicates that the new column is a row globally unique identifier column. Only one **uniqueidentifier** column per table can be designated as the ROWGUIDCOL column. The ROWGUIDCOL property can be assigned only to a **uniqueidentifier** column.

The ROWGUIDCOL property does not enforce uniqueness of the values stored in the column. It also does not automatically generate values for new rows inserted into the table. To generate unique values for each column, use the NEWID function on INSERT statements. A default value can be specified; however, NEWID cannot be specified as the default.

IDENTITY

Indicates that the new column is an identity column. When a new row is added to the table, SQL Server provides a unique, incremental value for the column. Identity columns are typically used together with PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to **tinyint**, **smallint**, **int**, **bigint**, **decimal(p,0)**, or **numeric(p,0)** columns. Only one identity column can be created per table. Bound defaults and DEFAULT constraints cannot be used with an identity column. You must specify both the *seed* and *increment* or neither. If neither is specified, the default is (1,1).

IDENTITY cannot be specified for CLR table-valued functions.

*seed*

Is the integer value to be assigned to the first row in the table.

*increment*

Is the integer value to add to the *seed* value for successive rows in the table.

**< column\_constraint >::= and < table\_constraint>::=**

Defines the constraint for a specified column or table. For CLR functions, the only constraint type allowed is NULL. Named constraints are not allowed.

NULL | NOT NULL

Determines whether null values are allowed in the column. NULL is not strictly a constraint but can be specified just like NOT NULL. NOT NULL cannot be specified for CLR table-valued functions.

PRIMARY KEY

Is a constraint that enforces entity integrity for a specified column through a unique index. In table-valued user-defined functions, the PRIMARY KEY constraint can be created on only one column per table. PRIMARY KEY cannot be specified for CLR table-valued functions.

UNIQUE

Is a constraint that provides entity integrity for a specified column or columns through a unique index. A table can have multiple UNIQUE constraints. UNIQUE cannot be specified for CLR table-valued functions.

CLUSTERED | NONCLUSTERED

Indicate that a clustered or a nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints use CLUSTERED, and UNIQUE constraints use NONCLUSTERED.

CLUSTERED can be specified for only one constraint. If CLUSTERED is specified for a UNIQUE constraint and a PRIMARY KEY constraint is also specified, the PRIMARY KEY uses NONCLUSTERED.

CLUSTERED and NONCLUSTERED cannot be specified for CLR table-valued functions.

CHECK

Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns. CHECK constraints cannot be specified for CLR table-valued functions.

*logical\_expression*

Is a logical expression that returns TRUE or FALSE.

**<computed\_column\_definition>::=**

Specifies a computed column. For more information about computed columns, see [CREATE TABLE (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms174979.aspx).

*column\_name*

Is the name of the computed column.

*computed\_column\_expression*

Is an expression that defines the value of a computed column.

**<index\_option>::=**

Specifies the index options for the PRIMARY KEY or UNIQUE index. For more information about index options, see [CREATE INDEX (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188783.aspx).

PAD\_INDEX = { ON | OFF }

Specifies index padding. The default is OFF.

FILLFACTOR = *fillfactor*

Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or change. *fillfactor* must be an integer value from 1 to 100. The default is 0.

IGNORE\_DUP\_KEY = { ON | OFF }

Specifies the error response when an insert operation attempts to insert duplicate key values into a unique index. The IGNORE\_DUP\_KEY option applies only to insert operations after the index is created or rebuilt. The default is OFF.

STATISTICS\_NORECOMPUTE = { ON | OFF }

Specifies whether distribution statistics are recomputed. The default is OFF.

ALLOW\_ROW\_LOCKS = { ON | OFF }

Specifies whether row locks are allowed. The default is ON.

ALLOW\_PAGE\_LOCKS = { ON | OFF }

Specifies whether page locks are allowed. The default is ON.

http://i.msdn.microsoft.com/Global/Images/clear.gif Remarks

User-defined functions are either scalar-valued or table-valued. Functions are scalar-valued if the RETURNS clause specified one of the scalar data types. Scalar-valued functions can be defined by using multiple Transact-SQL statements.

Functions are table-valued if the RETURNS clause specified TABLE. Depending on how the body of the function is defined, table-valued functions can be classified as inline or multi-statement functions. For more information, see [Table-Valued User-Defined Functions](http://msdn.microsoft.com/en-us/library/ms191165.aspx).

The following statements are valid in a function:

* Assignment statements.
* Control-of-Flow statements except TRY...CATCH statements.
* DECLARE statements defining local data variables and local cursors.
* SELECT statements that contain select lists with expressions that assign values to local variables.
* Cursor operations referencing local cursors that are declared, opened, closed, and deallocated in the function. Only FETCH statements that assign values to local variables using the INTO clause are allowed; FETCH statements that return data to the client are not allowed.
* INSERT, UPDATE, and DELETE statements modifying local **table** variables.
* EXECUTE statements calling extended stored procedures.
* For more information, see [Creating User-Defined Functions (Database Engine)](http://msdn.microsoft.com/en-us/library/ms191320.aspx).

**Nesting User-defined Functions**

User-defined functions can be nested; that is, one user-defined function can call another. The nesting level is incremented when the called function starts execution, and decremented when the called function finishes execution. User-defined functions can be nested up to 32 levels. Exceeding the maximum levels of nesting causes the whole calling function chain to fail.

|  |
| --- |
| **ms186755.note(en-us,SQL.100).gifNote:** |
| Any reference to managed code from a Transact-SQL user-defined function counts as one level against the 32-level nesting limit. Methods invoked from within managed code do not count against this limit. |

**Function Properties**

In earlier versions of SQL Server, functions are categorized only as deterministic or nondeterministic. In SQL Server 2005 and later, functions have the following properties. The values of these properties determine whether functions can be used in computed columns that can be persisted or indexed.

|  |  |  |
| --- | --- | --- |
| **Property** | **Description** | **Notes** |
| **IsDeterministic** | Function is deterministic or nondeterministic. | Local data access is allowed in deterministic functions. For example, functions that always return the same result any time they are called by using a specific set of input values and with the same state of the database would be labeled deterministic. |
| **IsPrecise** | Function is precise or imprecise. | Imprecise functions contain operations such as floating point operations. |
| **IsSystemVerified** | The precision and determinism properties of the function can be verified by SQL Server. |  |
| **SystemDataAccess** | Function accesses system data (system catalogs or virtual system tables) in the local instance of SQL Server. |  |
| **UserDataAccess** | Function accesses user data in the local instance of SQL Server. | Includes user-defined tables and temp tables, but not table-variables. |

The precision and determinism properties of Transact-SQL functions are determined automatically by SQL Server. For more information, see [User-Defined Function Design Guidelines](http://msdn.microsoft.com/en-us/library/ms187440.aspx). The data access and determinism properties of CLR functions can be specified by the user. For more information, see [Overview of CLR Integration Custom Attributes](http://msdn.microsoft.com/en-us/library/ms131098.aspx).

To display the current values for these properties, use [OBJECTPROPERTYEX](http://msdn.microsoft.com/en-us/library/ms188390.aspx).

**Indexing Computed Columns That Invoke a User-Defined Function**

A computed column that invokes a user-defined function can be used in an index when the user-defined function has the following property values:

* **IsDeterministic** = true
* **IsSystemVerified** = true (unless the computed column is persisted)
* **UserDataAccess** = false
* **SystemDataAccess** = false

For more information, see [Creating Indexes on Computed Columns](http://msdn.microsoft.com/en-us/library/ms189292.aspx).

**Calling Extended Stored Procedures from Functions**

The extended stored procedure, when it is called from inside a function, cannot return result sets to the client. Any ODS APIs that return result sets to the client will return FAIL. The extended stored procedure could connect back to an instance of SQL Server; however, it should not try to join the same transaction as the function that invoked the extended stored procedure.

Similar to invocations from a batch or stored procedure, the extended stored procedure will be executed in the context of the Windows security account under which SQL Server is running. The owner of the stored procedure should consider this when giving EXECUTE permission on it to users.

**Function Invocation**

Scalar-valued functions can be invoked where scalar expressions are used. This includes computed columns and CHECK constraint definitions. Scalar-valued functions can also be executed by using the [EXECUTE](http://msdn.microsoft.com/en-us/library/ms188332.aspx) statement. Scalar-valued functions must be invoked by using at least the two-part name of the function. For more information about multipart names, see [Transact-SQL Syntax Conventions (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms177563.aspx). Table-valued functions can be invoked where table expressions are allowed in the FROM clause of SELECT, INSERT, UPDATE, or DELETE statements. For more information, see [Executing User-Defined Functions (Database Engine)](http://msdn.microsoft.com/en-us/library/ms175562.aspx).

**Using Parameters and Return Values in CLR Functions**

If parameters are specified in a CLR function, they should be SQL Server types as defined previously for *scalar\_parameter\_data\_type*. For information about comparing SQL Server system data types to CLR integration data types or .NET Framework common language runtime data types, see [Mapping CLR Parameter Data](http://msdn.microsoft.com/en-us/library/ms131092.aspx).

For SQL Server to reference the correct method when it is overloaded in a class, the method indicated in <method\_specifier> must have the following characteristics:

* Receive the same number of parameters as specified in [ **,**...*n* ].
* Receive all the parameters by value, not by reference.
* Use parameter types that are compatible with those specified in the SQL Server function.

If the return data type of the CLR function specifies a table type (RETURNS TABLE), the return data type of the method in <method\_specifier> should be of type **IEnumerator** or **IEnumerable**, and it is assumed that the interface is implemented by the creator of the function. Unlike Transact-SQL functions, CLR functions cannot include PRIMARY KEY, UNIQUE, or CHECK constraints in <table\_type\_definition>. The data types of columns specified in <table\_type\_definition> must match the types of the corresponding columns of the result set returned by the method in <method\_specifier> at execution time. This type-checking is not performed at the time the function is created.

For more information about how to program CLR functions, see [CLR User-Defined Functions](http://msdn.microsoft.com/en-us/library/ms131077.aspx).

**Disallowed SQL Statements**

The following Service Broker statements cannot be included in the definition of a Transact-SQL user-defined function:

* BEGIN DIALOG CONVERSATION
* END CONVERSATION
* GET CONVERSATION GROUP
* MOVE CONVERSATION
* RECEIVE
* SEND

**Viewing Information About Functions**

To display the definition of Transact-SQL user-defined functions, use the [sys.sql\_modules](http://msdn.microsoft.com/en-us/library/ms175081.aspx) catalog view in the database in which the function exists.

For example:

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl77other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl77other');)

USE AdventureWorks;

GO

SELECT definition, type

FROM sys.sql\_modules AS m

JOIN sys.objects AS o ON m.object\_id = o.object\_id

AND type IN ('FN', 'IF', 'TF');

GO

|  |
| --- |
| **ms186755.note(en-us,SQL.100).gifNote:** |
| The definition of functions created by using the ENCRYPTION option cannot be viewed by using **sys.sql\_modules**; however, other information about the encrypted functions is displayed. |

To display information about CLR user-defined functions, use the [sys.assembly\_modules](http://msdn.microsoft.com/en-us/library/ms180052.aspx) catalog view in the database in which the function exists.

To display information about the parameters defined in user-defined functions, use the [sys.parameters](http://msdn.microsoft.com/en-us/library/ms176074.aspx) catalog view in the database in which the function exists.

For a report on the objects referenced by a function, use [sys.sql\_expression\_dependencies](http://msdn.microsoft.com/en-us/library/bb677315.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif Permissions

Requires CREATE FUNCTION permission in the database and ALTER permission on the schema in which the function is being created. If the function specifies a user-defined type, requires EXECUTE permission on the type.

http://i.msdn.microsoft.com/Global/Images/clear.gif Guidance on Using Sort Order

When using the ORDER clause in CLR table-valued functions, follow these guidelines:

* You must ensure that results are always ordered in the specified order. If the results are not in the specified order, SQL Server will generate an error message when the query is executed.
* If an ORDER clause is specified, the output of the table-valued function must be sorted according to the collation of the column (explicit or implicit). For example, if the column collation is Chinese (either specified in the DDL for the table-valued function or obtained from the database collation), the returned results must be sorted according to Chinese sorting rules.
* The ORDER clause, if specified, is always verified by SQL Server while returning results, whether or not it is used by the query processor to perform further optimizations. Only use the ORDER clause if you know it is useful to the query processor.
* The SQL Server query processor takes advantage of the ORDER clause automatically in following cases:
  + Insert queries where the ORDER clause is compatible with an index.
  + ORDER BY clauses that are compatible with the ORDER clause.
  + Aggregates, where GROUP BY is compatible with ORDER clause.
  + DISTINCT aggregates where the distinct columns are compatible with the ORDER clause.

The ORDER clause does not guarantee ordered results when a SELECT query is executed, unless ORDER BY is also specified in the query. See [sys.function\_order\_columns (Transact-SQL)](http://msdn.microsoft.com/en-us/library/bb933868.aspx) for information on how to query for columns included in the sort-order for table-valued functions.

http://i.msdn.microsoft.com/Global/Images/clear.gif Examples

**A. Using a scalar-valued user-defined function that calculates the ISO week**

The following example creates the user-defined function ISOweek. This function takes a date argument and calculates the ISO week number. For this function to calculate correctly, SET DATEFIRST 1 must be invoked before the function is called.

The example also shows using the [EXECUTE AS](http://msdn.microsoft.com/en-us/library/ms188354.aspx) clause to specify the security context in which a stored procedure can be executed. In the example, the option CALLER specifies that the procedure will be executed in the context of the user that calls it. The other options that you can specify are SELF, OWNER, and *user\_name*.

Here is the function call. Notice that DATEFIRST is set to 1.

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl102other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl102other');)

USE AdventureWorks;

GO

IF OBJECT\_ID (N'dbo.ISOweek', N'FN') IS NOT NULL

DROP FUNCTION dbo.ISOweek;

GO

CREATE FUNCTION dbo.ISOweek (@DATE datetime)

RETURNS int

WITH EXECUTE AS CALLER

AS

BEGIN

DECLARE @ISOweek int;

SET @ISOweek= DATEPART(wk,@DATE)+1

-DATEPART(wk,CAST(DATEPART(yy,@DATE) as CHAR(4))+'0104');

--Special cases: Jan 1-3 may belong to the previous year

IF (@ISOweek=0)

SET @ISOweek=dbo.ISOweek(CAST(DATEPART(yy,@DATE)-1

AS CHAR(4))+'12'+ CAST(24+DATEPART(DAY,@DATE) AS CHAR(2)))+1;

--Special case: Dec 29-31 may belong to the next year

IF ((DATEPART(mm,@DATE)=12) AND

((DATEPART(dd,@DATE)-DATEPART(dw,@DATE))>= 28))

SET @ISOweek=1;

RETURN(@ISOweek);

END;

GO

SET DATEFIRST 1;

SELECT dbo.ISOweek(CONVERT(DATETIME,'12/26/2004',101)) AS 'ISO Week';

Here is the result set.

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl103other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl103other');)

ISO Week

----------------

52

**B. Creating an inline table-valued function**

The following example returns an inline table-valued function. It returns three columns ProductID, Name and the aggregate of year-to-date totals by store as YTD Total for each product sold to the store.

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl104other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl104other');)

USE AdventureWorks;

GO

IF OBJECT\_ID (N'Sales.ufn\_SalesByStore', N'IF') IS NOT NULL

DROP FUNCTION Sales.ufn\_SalesByStore;

GO

CREATE FUNCTION Sales.ufn\_SalesByStore (@storeid int)

RETURNS TABLE

AS

RETURN

(

SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS 'YTD Total'

FROM Production.Product AS P

JOIN Sales.SalesOrderDetail AS SD ON SD.ProductID = P.ProductID

JOIN Sales.SalesOrderHeader AS SH ON SH.SalesOrderID = SD.SalesOrderID

WHERE SH.CustomerID = @storeid

GROUP BY P.ProductID, P.Name

);

GO

To invoke the function, run this query.

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl105other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl105other');)

SELECT \* FROM Sales.ufn\_SalesByStore (602);

**C. Creating a multi-statement table-valued function**

The following example creates the table-valued function fn\_FindReports(InEmpID). When supplied with a valid employee ID, the function returns a table that corresponds to all the employees that report to the employee either directly or indirectly. The function uses a recursive common table expression (CTE) to produce the hierarchical list of employees. For more information about recursive CTEs, see [WITH common\_table\_expression (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms175972.aspx).

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl107other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl107other');)

USE AdventureWorks;

GO

IF OBJECT\_ID (N'dbo.ufn\_FindReports', N'TF') IS NOT NULL

DROP FUNCTION dbo.ufn\_FindReports;

GO

CREATE FUNCTION dbo.ufn\_FindReports (@InEmpID INTEGER)

RETURNS @retFindReports TABLE

(

EmployeeID int primary key NOT NULL,

Name nvarchar(255) NOT NULL,

Title nvarchar(50) NOT NULL,

EmployeeLevel int NOT NULL,

Sort nvarchar (255) NOT NULL

)

--Returns a result set that lists all the employees who report to the

--specific employee directly or indirectly.\*/

AS

BEGIN

WITH DirectReports(Name, Title, EmployeeID, EmployeeLevel, Sort) AS

(SELECT CONVERT(Varchar(255), c.FirstName + ' ' + c.LastName),

e.Title,

e.EmployeeID,

1,

CONVERT(Varchar(255), c.FirstName + ' ' + c.LastName)

FROM HumanResources.Employee AS e

JOIN Person.Contact AS c ON e.ContactID = c.ContactID

WHERE e.EmployeeID = @InEmpID

UNION ALL

SELECT CONVERT(Varchar(255), REPLICATE ('| ' , EmployeeLevel) +

c.FirstName + ' ' + c.LastName),

e.Title,

e.EmployeeID,

EmployeeLevel + 1,

CONVERT (Varchar(255), RTRIM(Sort) + '| ' + FirstName + ' ' +

LastName)

FROM HumanResources.Employee as e

JOIN Person.Contact AS c ON e.ContactID = c.ContactID

JOIN DirectReports AS d ON e.ManagerID = d.EmployeeID

)

-- copy the required columns to the result of the function

INSERT @retFindReports

SELECT EmployeeID, Name, Title, EmployeeLevel, Sort

FROM DirectReports

RETURN

END;

GO

-- Example invocation

SELECT EmployeeID, Name, Title, EmployeeLevel

FROM dbo.ufn\_FindReports(109)

ORDER BY Sort;

GO

**D. Creating a CLR function**

The following example assumes that the [SQL Server Database Engine Samples](http://msdn.microsoft.com/en-us/library/ms160958.aspx) are installed in the default location of the local computer and the StringManipulate.csproj sample application is compiled. For more information, see [Readme\_Supplementary-Aware String Manipulation Sample](http://msdn.microsoft.com/en-us/library/ms160903.aspx).

The example creates CLR function len\_s. Before the function is created, the assembly SurrogateStringFunction.dll is registered in the local database.

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl110other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl110other');)

DECLARE @SamplesPath nvarchar(1024);

-- You may have to modify the value of this variable if you have

-- installed the sample in a location other than the default location.

SELECT @SamplesPath = REPLACE(physical\_name, 'Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\master.mdf', 'Microsoft SQL Server\90\Samples\Engine\Programmability\CLR\')

    FROM master.sys.database\_files

    WHERE name = 'master';

CREATE ASSEMBLY [SurrogateStringFunction]

FROM @SamplesPath + 'StringManipulate\CS\StringManipulate\bin\debug\SurrogateStringFunction.dll'

WITH PERMISSION\_SET = EXTERNAL\_ACCESS;

GO

CREATE FUNCTION [dbo].[len\_s] (@str nvarchar(4000))

RETURNS bigint

AS EXTERNAL NAME [SurrogateStringFunction].[Microsoft.Samples.SqlServer.SurrogateStringFunction].[LenS];

GO

For an example of how to create a CLR table-valued function, see [CLR Table-Valued Functions](http://msdn.microsoft.com/en-us/library/ms131103.aspx).

**ALTER FUNCTION (Transact-SQL)**

Alters an existing Transact-SQL or CLR function that was previously created by executing the CREATE FUNCTION statement, without changing permissions and without affecting any dependent functions, stored procedures, or triggers.

Topic link icon[Transact-SQL Syntax Conventions](http://msdn.microsoft.com/en-us/library/ms177563.aspx)

 Syntax

Scalar Functions

ALTER FUNCTION [ schema\_name. ] function\_name

( [ { @parameter\_name [ AS ][ type\_schema\_name. ] parameter\_data\_type

    [ = default ] }

    [ ,...n ]

  ]

)

RETURNS return\_data\_type

    [ WITH <function\_option> [ ,...n ] ]

    [ AS ]

    BEGIN

                function\_body

        RETURN scalar\_expression

    END

[ ; ]

Inline Table-valued Functions

ALTER FUNCTION [ schema\_name. ] function\_name

( [ { @parameter\_name [ AS ] [ type\_schema\_name. ] parameter\_data\_type

    [ = default ] }

    [ ,...n ]

  ]

)

RETURNS TABLE

    [ WITH <function\_option> [ ,...n ] ]

    [ AS ]

    RETURN [ ( ] select\_stmt [ ) ]

[ ; ]

Multistatement Table-valued Functions

ALTER FUNCTION [ schema\_name. ] function\_name

( [ { @parameter\_name [ AS ] [ type\_schema\_name. ] parameter\_data\_type

    [ = default ] }

    [ ,...n ]

  ]

)

RETURNS @return\_variable TABLE <table\_type\_definition>

    [ WITH <function\_option> [ ,...n ] ]

    [ AS ]

    BEGIN

                function\_body

        RETURN

    END

[ ; ]

CLR Functions

ALTER FUNCTION [ schema\_name. ] function\_name

( { @parameter\_name [AS] [ type\_schema\_name. ] parameter\_data\_type

        [ = default ] }

    [ ,...n ]

)

RETURNS { return\_data\_type | TABLE <clr\_table\_type\_definition> }

    [ WITH <clr\_function\_option> [ ,...n ] ]

    [ AS ] EXTERNAL NAME <method\_specifier>

[ ; ]

<method\_specifier>::=

        assembly\_name.class\_name.method\_name

Function Options

<function\_option>::=

{

    [ ENCRYPTION ]

  | [ SCHEMABINDING ]

  | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]

  | [ EXECUTE\_AS\_Clause ]

}

<clr\_function\_option>::=

}

    [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]

  | [ EXECUTE\_AS\_Clause ]

}

Table Type Definitions

<table\_type\_definition>:: =

( { <column\_definition> <column\_constraint>

  | <computed\_column\_definition> }

        [ <table\_constraint> ] [ ,...n ]

)

<clr\_table\_type\_definition>:: =

( { column\_name data\_type } [ ,...n ] )

<column\_definition>::=

{

    { column\_name data\_type }

    [ [ DEFAULT constant\_expression ]

      [ COLLATE collation\_name ] | [ ROWGUIDCOL ]

    ]

    | [ IDENTITY [ (seed , increment ) ] ]

    [ <column\_constraint> [ ...n ] ]

}

<column\_constraint>::=

{

    [ NULL | NOT NULL ]

    { PRIMARY KEY | UNIQUE }

      [ CLUSTERED | NONCLUSTERED ]

        [ WITH FILLFACTOR = fillfactor

        | WITH ( < index\_option > [ , ...n ] )

            [ ON { filegroup | "default" } ]

  | [ CHECK ( logical\_expression ) ] [ ,...n ]

}

<computed\_column\_definition>::=

column\_name AS computed\_column\_expression

<table\_constraint>::=

{

    { PRIMARY KEY | UNIQUE }

      [ CLUSTERED | NONCLUSTERED ]

            ( column\_name [ ASC | DESC ] [ ,...n ] )

        [ WITH FILLFACTOR = fillfactor

        | WITH ( <index\_option> [ , ...n ] )

  | [ CHECK ( logical\_expression ) ] [ ,...n ]

}

<index\_option>::=

{

       PAD\_INDEX = { ON | OFF }

  | FILLFACTOR = fillfactor

  | IGNORE\_DUP\_KEY = { ON | OFF }

  | STATISTICS\_NORECOMPUTE = { ON | OFF }

  | ALLOW\_ROW\_LOCKS = { ON | OFF }

  | ALLOW\_PAGE\_LOCKS ={ ON | OFF }

}

 Arguments

*schema\_name*

Is the name of the schema to which the user-defined function belongs.

*function\_name*

Is the user-defined function to be changed.

|  |
| --- |
| **Note:** |
| Parentheses are required after the function name even if a parameter is not specified. |

**@** *parameter\_name*

Is a parameter in the user-defined function. One or more parameters can be declared.

A function can have a maximum of 2,100 parameters. The value of each declared parameter must be supplied by the user when the function is executed, unless a default for the parameter is defined.

Specify a parameter name by using an at sign (**@**) as the first character. The parameter name must comply with the rules for [identifiers](http://msdn.microsoft.com/en-us/library/ms175874.aspx). Parameters are local to the function; the same parameter names can be used in other functions. Parameters can take the place only of constants; they cannot be used instead of table names, column names, or the names of other database objects.

|  |
| --- |
| **Note:** |
| ANSI\_WARNINGS is not honored when passing parameters in a stored procedure, user-defined function, or when declaring and setting variables in a batch statement. For example, if a variable is defined as **char(3)**, and then set to a value larger than three characters, the data is truncated to the defined size and the INSERT or UPDATE statement succeeds. |

[ *type\_schema\_name.* ] *parameter\_data\_type*

Is the parameter data type and optionally, the schema to which it belongs. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except the **timestamp** data type. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, and **timestamp** data types. The nonscalar types **cursor** and **table** cannot be specified as a parameter data type in either Transact-SQL or CLR functions.

If *type\_schema\_name* is not specified, the SQL Server 2005 Database Engine looks for the *parameter\_data\_type* in the following order:

* The schema that contains the names of SQL Server system data types.
* The default schema of the current user in the current database.
* The **dbo** schema in the current database.

[ **=** *default* ]

Is a default value for the parameter. If a *default* value is defined, the function can be executed without specifying a value for that parameter.

|  |
| --- |
| **Note:** |
| Default parameter values can be specified for CLR functions except for **varchar(max)** and **varbinary(max)** data types. |

When a parameter of the function has a default value, the keyword DEFAULT must be specified when calling the function to retrieve the default value. This behavior is different from using parameters with default values in stored procedures in which omitting the parameter also implies the default value.

*return\_data\_type*

Is the return value of a scalar user-defined function. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except the **timestamp** data type. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, and **timestamp** data types. The nonscalar types **cursor** and **table** cannot be specified as a return data type in either Transact-SQL or CLR functions.

*function\_body*

Specifies that a series of Transact-SQL statements, which together do not produce a side effect such as modifying a table, define the value of the function. *function\_body* is used only in scalar functions and multistatement table-valued functions.

In scalar functions, *function\_body* is a series of Transact-SQL statements that together evaluate to a scalar value.

In multistatement table-valued functions, *function\_body* is a series of Transact-SQL statements that populate a TABLE return variable.

*scalar\_expression*

Specifies that the scalar function returns a scalar value.

TABLE

Specifies that the return value of the table-valued function is a table. Only constants and **@***local\_variables* can be passed to table-valued functions.

In inline table-valued functions, the TABLE return value is defined through a single SELECT statement. Inline functions do not have associated return variables.

In multistatement table-valued functions, **@***return\_variable* is a TABLE variable used to store and accumulate the rows that should be returned as the value of the function. **@***return\_variable* can be specified only for Transact-SQL functions and not for CLR functions.

*select-stmt*

Is the single SELECT statement that defines the return value of an inline table-valued function.

EXTERNAL NAME <method\_specifier>*assembly\_name.class\_name*.*method\_name*

Specifies the method of an assembly to bind with the function. *assembly\_name* must match an existing assembly in SQL Server in the current database with visibility on. *class\_name* must be a valid SQL Server identifier and must exist as a class in the assembly. If the class has a namespace-qualified name that uses a period (**.**) to separate namespace parts, the class name must be delimited by using brackets (**[** **]**) or quotation marks (**"** **"**). *method\_name* must be a valid SQL Server identifier and must exist as a static method in the specified class.

|  |
| --- |
| **Note:** |
| By default, SQL Server cannot execute CLR code. You can create, modify, and drop database objects that reference common language runtime modules; however, you cannot execute these references in SQL Server until you enable the [clr enabled option](http://msdn.microsoft.com/en-us/library/ms175193.aspx). To enable the option, use [sp\_configure](http://msdn.microsoft.com/en-us/library/ms188787.aspx). |

*<*table\_type\_definition*>***(** { <column\_definition> <column\_constraint>   | <computed\_column\_definition> }   [ <table\_constraint> ] [ **,**...*n* ]**)**

Defines the table data type for a Transact-SQL function. The table declaration includes column definitions and column or table constraints.

< clr\_table\_type\_definition > **(** { *column\_name* *data\_type* } [ **,**...*n* ] **)**

Defines the table data types for a CLR function. The table declaration includes only column names and data types.

**<function\_option>::= and <clr\_function\_option>::=**

Specifies the function will have one or more of the following options.

ENCRYPTION

Indicates that the Database Engine encrypts the catalog view columns that contains the text of the ALTER FUNCTION statement. Using ENCRYPTION prevents the function from being published as part of SQL Server replication. ENCRYPTION cannot be specified for CLR functions.

SCHEMABINDING

Specifies that the function is bound to the database objects that it references. This condition will prevent changes to the function if other schema bound objects are referencing it.

The binding of the function to the objects it references is removed only when one of the following actions occurs:

* The function is dropped.
* The function is modified by using the ALTER statement with the SCHEMABINDING option not specified.

For a list of conditions that must be met before a function can be schema bound, see [CREATE FUNCTION (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms186755.aspx).

SCHEMABINDING cannot be specified for CLR functions or functions that reference alias data types.

RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT

Specifies the **OnNULLCall** attribute of a scalar-valued function. If not specified, CALLED ON NULL INPUT is implied by default. This means that the function body executes even if NULL is passed as an argument.

If RETURNS NULL ON NULL INPUT is specified in a CLR function, it indicates that SQL Server can return NULL when any of the arguments it receives is NULL, without actually invoking the body of the function. If the method specified in <method\_specifier> already has a custom attribute that indicates RETURNS NULL ON NULL INPUT, but the ALTER FUNCTION statement indicates CALLED ON NULL INPUT, the ALTER FUNCTION statement takes precedence. The **OnNULLCall** attribute cannot be specified for CLR table-valued functions.

EXECUTE AS Clause

Specifies the security context under which the user-defined function is executed. Therefore, you can control which user account SQL Server uses to validate permissions on any database objects referenced by the function.

|  |
| --- |
| **Note:** |
| EXECUTE AS cannot be specified for inline user-defined functions. |

For more information, see [EXECUTE AS Clause (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188354.aspx).

**< column\_definition >::=**

Defines the table data type. The table declaration includes column definitions and constraints. For CLR functions, only *column\_name* and *data\_type* can be specified.

*column\_name*

Is the name of a column in the table. Column names must comply with the rules for identifiers and must be unique in the table. *column\_name* can consist of 1 through 128 characters.

*data\_type*

Specifies the column data type. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except **timestamp**. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, **char**, **varchar**, **varchar(max)**, and **timestamp**.The nonscalar type **cursor** cannot be specified as a column data type in either Transact-SQL or CLR functions.

DEFAULT *constant\_expression*

Specifies the value provided for the column when a value is not explicitly supplied during an insert. *constant\_expression* is a constant, NULL, or a system function value. DEFAULT definitions can be applied to any column except those that have the IDENTITY property. DEFAULT cannot be specified for CLR table-valued functions.

COLLATE *collation\_name*

Specifies the collation for the column. If not specified, the column is assigned the default collation of the database. Collation name can be either a Windows collation name or a SQL collation name. For a list of and more information, see [Windows Collation Name (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188046.aspx) and [SQL Server Collation Name (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms180175.aspx).

The COLLATE clause can be used to change the collations only of columns of the **char**, **varchar**, **nchar**, and **nvarchar** data types.

COLLATE cannot be specified for CLR table-valued functions.

ROWGUIDCOL

Indicates that the new column is a row global unique identifier column. Only one **uniqueidentifier** column per table can be designated as the ROWGUIDCOL column. The ROWGUIDCOL property can be assigned only to a **uniqueidentifier** column.

The ROWGUIDCOL property does not enforce uniqueness of the values stored in the column. It also does not automatically generate values for new rows inserted into the table. To generate unique values for each column, use the NEWID function on INSERT statements. A default value can be specified; however, NEWID cannot be specified as the default.

IDENTITY

Indicates that the new column is an identity column. When a new row is added to the table, SQL Server provides a unique, incremental value for the column. Identity columns are typically used together with PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to **tinyint**, **smallint**, **int**, **bigint**, **decimal(p,0)**, or **numeric(p,0)** columns. Only one identity column can be created per table. Bound defaults and DEFAULT constraints cannot be used with an identity column. You must specify both the *seed* and *increment* or neither. If neither is specified, the default is (1,1).

IDENTITY cannot be specified for CLR table-valued functions.

*seed*

Is the integer value to be assigned to the first row in the table.

*increment*

Is the integer value to add to the *seed* value for successive rows in the table.

**< column\_constraint >::= and < table\_constraint>::=**

Defines the constraint for a specified column or table. For CLR functions, the only constraint type allowed is NULL. Named constraints are not allowed.

NULL | NOT NULL

Determines whether null values are allowed in the column. NULL is not strictly a constraint but can be specified just like NOT NULL. NOT NULL cannot be specified for CLR table-valued functions.

PRIMARY KEY

Is a constraint that enforces entity integrity for a specified column through a unique index. In table-valued user-defined functions, the PRIMARY KEY constraint can be created on only one column per table. PRIMARY KEY cannot be specified for CLR table-valued functions.

UNIQUE

Is a constraint that provides entity integrity for a specified column or columns through a unique index. A table can have multiple UNIQUE constraints. UNIQUE cannot be specified for CLR table-valued functions.

CLUSTERED | NONCLUSTERED

Indicate that a clustered or a nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints use CLUSTERED, and UNIQUE constraints use NONCLUSTERED.

CLUSTERED can be specified for only one constraint. If CLUSTERED is specified for a UNIQUE constraint and a PRIMARY KEY constraint is also specified, the PRIMARY KEY uses NONCLUSTERED.

CLUSTERED and NONCLUSTERED cannot be specified for CLR table-valued functions.

CHECK

Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns. CHECK constraints cannot be specified for CLR table-valued functions.

*logical\_expression*

Is a logical expression that returns TRUE or FALSE.

**<computed\_column\_definition>::=**

Specifies a computed column. For more information about computed columns, see [CREATE TABLE (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms174979.aspx).

*column\_name*

Is the name of the computed column.

*computed\_column\_expression*

Is an expression that defines the value of a computed column.

**<index\_option>::=**

Specifies the index options for the PRIMARY KEY or UNIQUE index. For more information about index options, see [CREATE INDEX (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188783.aspx).

PAD\_INDEX = { ON | OFF }

Specifies index padding. The default is OFF.

FILLFACTOR = *fillfactor*

Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or change. *fillfactor* must be an integer value from 1 to 100. The default is 0.

IGNORE\_DUP\_KEY = { ON | OFF }

Specifies the error response when an insert operation attempts to insert duplicate key values into a unique index. The IGNORE\_DUP\_KEY option applies only to insert operations after the index is created or rebuilt. The default is OFF.

STATISTICS\_NORECOMPUTE = { ON | OFF }

Specifies whether distribution statistics are recomputed. The default is OFF.

ALLOW\_ROW\_LOCKS = { ON | OFF }

Specifies whether row locks are allowed. The default is ON.

ALLOW\_PAGE\_LOCKS = { ON | OFF }

Specifies whether page locks are allowed. The default is ON.

 Remarks

ALTER FUNCTION cannot be used to change a scalar-valued function to a table-valued function, or vice versa. Also, ALTER FUNCTION cannot be used to change an inline function to a multistatement function, or vice versa. ALTER FUNCTION cannot be used to change a Transact-SQL function to a CLR function or vice-versa.

The following Service Broker statements cannot be included in the definition of a Transact-SQL user-defined function:

* BEGIN DIALOG CONVERSATION
* END CONVERSATION
* GET CONVERSATION GROUP
* MOVE CONVERSATION
* RECEIVE
* SEND

 Permissions

Requires ALTER permission on the function or on the schema. If the function specifies a user-defined type, requires EXECUTE permission on the type.

**DROP FUNCTION (Transact-SQL)**

Removes one or more user-defined functions from the current database. User-defined functions are created by using [CREATE FUNCTION](http://msdn.microsoft.com/en-us/library/ms186755.aspx) and modified by using [ALTER FUNCTION](http://msdn.microsoft.com/en-us/library/ms186967.aspx).

Topic link icon[Transact-SQL Syntax Conventions](http://msdn.microsoft.com/en-us/library/ms177563.aspx)

 Syntax

DROP FUNCTION { [ schema\_name. ] function\_name } [ ,...n ]

 Arguments

*schema\_name*

Is the name of the schema to which the user-defined function belongs.

*function\_name*

Is the name of the user-defined function or functions to be removed. Specifying the schema name is optional. The server name and database name cannot be specified.

 Remarks

DROP FUNCTION will fail if there are Transact-SQL functions or views in the database that reference this function and were created by using SCHEMABINDING, or if there are computed columns, CHECK constraints, or DEFAULT constraints that reference the function.

DROP FUNCTION will fail if there are computed columns that reference this function and have been indexed.

 Permissions

To execute DROP FUNCTION, at a minimum, a user must have ALTER permission on the schema to which the function belongs, or CONTROL permission on the function.

 Examples

**A. Dropping a function**

The following example drops the fn\_SalesByStore user-defined function from the Sales schema in the AdventureWorks sample database. To create this function, see Example B in [CREATE FUNCTION (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms186755.aspx).

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl20other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl20other');)

USE AdventureWorks;

GO

IF OBJECT\_ID (N'Sales.fn\_SalesByStore', N'IF') IS NOT NULL

DROP FUNCTION Sales.fn\_SalesByStore;

GO

**Creating User-Defined Functions (Database Engine)**

User-defined functions are created using the CREATE FUNCTION statement, modified using the ALTER FUNCTION statement, and removed using the DROP FUNCTION statement. Each fully qualified user-defined function name (*schema\_name.function\_name*) must be unique.

 Guidelines

Transact-SQL errors that cause a statement to be canceled and continue with the next statement in the module (such as triggers or stored procedures) are treated differently inside a function. In functions, such errors cause the execution of the function to stop. This in turn causes the statement that invoked the function to be canceled.

The statements in a BEGIN...END block cannot have any side effects. Function side effects are any permanent changes to the state of a resource that has a scope outside the function such as a modification to a database table. The only changes that can be made by the statements in the function are changes to objects local to the function, such as local cursors or variables. Modifications to database tables, operations on cursors that are not local to the function, sending e-mail, attempting a catalog modification, and generating a result set that is returned to the user are examples of actions that cannot be performed in a function.

|  |
| --- |
| **Note:** |
| If a CREATE FUNCTION statement produces side effects against resources that do not exist when the CREATE FUNCTION statement is issued, SQL Server executes the statement. However, SQL Server does not execute the function when it is invoked. |

The number of times that a function specified in a query is actually executed can vary between execution plans built by the optimizer. An example is a function invoked by a subquery in a WHERE clause. The number of times the subquery and its function is executed can vary with different access paths chosen by the optimizer.

 Valid Statements in a Function

The types of statements that are valid in a function include:

* DECLARE statements can be used to define data variables and cursors that are local to the function.
* Assignments of values to objects local to the function, such as using SET to assign values to scalar and table local variables.
* Cursor operations that reference local cursors that are declared, opened, closed, and deallocated in the function. FETCH statements that return data to the client are not allowed. Only FETCH statements that assign values to local variables using the INTO clause are allowed.
* Control-of-flow statements except TRY...CATCH statements.
* SELECT statements containing select lists with expressions that assign values to variables that are local to the function.
* UPDATE, INSERT, and DELETE statements modifying table variables that are local to the function.
* EXECUTE statements calling an extended stored procedure.

**Built-in System Functions**

The following nondeterministic built-in functions can be used in Transact-SQL user-defined functions.

|  |  |
| --- | --- |
| CURRENT\_TIMESTAMP | @@MAX\_CONNECTIONS |
| GET\_TRANSMISSION\_STATUS | @@PACK\_RECEIVED |
| GETDATE | @@PACK\_SENT |
| GETUTCDATE | @@PACKET\_ERRORS |
| @@CONNECTIONS | @@TIMETICKS |
| @@CPU\_BUSY | @@TOTAL\_ERRORS |
| @@DBTS | @@TOTAL\_READ |
| @@IDLE | @@TOTAL\_WRITE |
| @@IO\_BUSY |  |

The following nondeterministic built-in functions cannot be used in Transact-SQL user-defined functions.

|  |  |
| --- | --- |
| NEWID | RAND |
| NEWSEQUENTIALID | TEXTPTR |

For a list of deterministic and nondeterministic built-in system functions, see [Deterministic and Nondeterministic Functions](http://msdn.microsoft.com/en-us/library/ms178091.aspx).

 Schema-Bound Functions

CREATE FUNCTION supports a SCHEMABINDING clause that binds the function to the schema of any objects it references, such as tables, views, and other user-defined functions. An attempt to alter or drop any object referenced by a schema-bound function fails.

These conditions must be met before you can specify SCHEMABINDING in CREATE FUNCTION:

* All views and user-defined functions referenced by the function must be schema-bound.
* All objects referenced by the function must be in the same database as the function. The objects must be referenced using either one-part or two-part names.
* You must have REFERENCES permission on all objects (tables, views, and user-defined functions) referenced in the function.

You can use ALTER FUNCTION to remove the schema binding. The ALTER FUNCTION statement should redefine the function without specifying WITH SCHEMABINDING.

 Specifying Parameters

A user-defined function takes zero or more input parameters and returns either a scalar value or a table. A function can have a maximum of 1024 input parameters. When a parameter of the function has a default value, the keyword DEFAULT must be specified when calling the function to get the default value. This behavior is different from parameters with default values in user-defined stored procedures in which omitting the parameter also implies the default value. User-defined functions do not support output parameters.

**Executing User-Defined Functions (Database Engine)**

User-defined functions can be invoked in queries or in other statements or expressions such as computed columns or string expressions. Scalar-valued functions can be executed using the EXECUTE statement.

 Invoking User-defined Functions That Return a Scalar Value

You can invoke a user-defined function that returns a scalar value anywhere that a scalar expression of the same data type is allowed in Transact-SQL statements. Scalar-valued functions must be invoked by using at least the two-part name of the function. For more information about multipart names, see [Transact-SQL Syntax Conventions (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms177563.aspx).

**Queries**

User-defined functions that return scalar values are allowed in these locations:

* As an *expression* in the *select\_list* of a SELECT statement:

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl08other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl08other');)

USE AdventureWorks;

GO

SELECT ProductID, ListPrice, dbo.ufnGetProductDealerPrice(ProductID, StartDate) AS DealerPrice,

StartDate, EndDate

FROM Production.ProductListPriceHistory

WHERE ListPrice > .0000

ORDER BY ProductID, StartDate;

GO

* As an *expression* or *string\_expression* in a WHERE or HAVING clause predicate:

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl09other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl09other');)

USE AdventureWorks;

GO

SELECT ProductID, ListPrice, StartDate, EndDate

FROM Production.ProductListPriceHistory

WHERE dbo.ufnGetProductDealerPrice(ProductID, StartDate) > .0000

ORDER BY ProductID, StartDate;

GO

* As a *group\_by\_expression* in a GROUP BY clause.
* As an *order\_by\_expression* in an ORDER BY clause.
* As an *expression* in the SET clause in an UPDATE statement:

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl10other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl10other');)

USE AdventureWorks;

GO

UPDATE Production.ProductListPriceHistory

SET ListPrice = dbo.ufnGetProductDealerPrice(ProductID, StartDate)

WHERE ProductID > 900;

GO

* As an *expression* in the VALUES clause of an INSERT statement:   
  User-defined functions referenced in these locations are logically executed one time per row.

**CHECK Constraints**

User-defined functions that return scalar values can be invoked in CHECK constraints if the argument values passed to the function reference columns only in the table or constants. Every time the query processor checks the constraint, the query processor calls the function with the argument values associated with the current row being checked. The owner of a table must also be the owner of the user-defined function invoked by a CHECK constraint on the table.

**DEFAULT Definitions**

User-defined functions can be invoked as the *constant\_expression* of DEFAULT definitions if the argument values passed to the function only contain constants. The owner of the table must also be the owner of the user-defined function invoked by a DEFAULT definition on the table.

**Computed Columns**

Functions can be invoked by computed columns if the argument values passed to the function reference only columns in the table or constants. The owner of the table must also be the owner of the user-defined function invoked by a computed column in the table.

**Assignment Operators**

Assignment operators (*left\_operand* = *right\_operand*) can invoke user-defined functions that return a scalar value in the expression specified as the right operand.

**Control-of-Flow Statements**

User-defined functions that return scalar values can be invoked by control-of-flow statements in their Boolean expressions.

**CASE Expressions**

User-defined functions that return a scalar value can be invoked in any of the CASE expressions.

**PRINT Statements**

User-defined functions that return a character string can be invoked as the *string\_expr* expression of PRINT statements.

**Functions and Stored Procedures**

* Function arguments can also be a reference to a user-defined function that returns a scalar value.
* RETURN *integer\_expression* statements in stored procedures can invoke user-defined functions that return an integer as the *integer\_expression*.
* RETURN *return\_type\_spec* statements in user-defined functions can invoke user-defined functions that return a scalar data type such as the *return\_type\_spec*, provided the value returned by the invoked user-defined function can be implicitly converted to the return data type of the invoking function.

 Executing User-defined Functions That Return a Scalar Value

You can execute user-defined functions that return scalar values in the same manner as stored procedures. When executing a user-defined function that returns a scalar value, the parameters are specified in the same way they are for stored procedures:

* The argument values are not enclosed in parentheses.
* Parameter names can be specified.
* If parameter names are specified, the argument values do not have to be in the same sequence as the parameters.

The following example creates a user-defined function that returns a decimal scalar value.

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl17other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl17other');)

IF OBJECT\_ID(N'dbo.ufn\_CubicVolume', N'FN') IS NOT NULL

DROP FUNCTION dbo.ufn\_CubicVolume;

GO

CREATE FUNCTION dbo.ufn\_CubicVolume

-- Input dimensions in centimeters.

(@CubeLength decimal(4,1), @CubeWidth decimal(4,1),

@CubeHeight decimal(4,1) )

RETURNS decimal(12,3) -- Cubic Centimeters.

WITH SCHEMABINDING

AS

BEGIN

RETURN ( @CubeLength \* @CubeWidth \* @CubeHeight )

END;

GO

The following example executes the dbo.ufn\_CubicVolume function. Using the Transact-SQL EXECUTE statement, the arguments are identified in an order different from the parameters in the function definition:

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl18other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl18other');)

DECLARE @MyDecimalVar decimal(12,3);

EXEC @MyDecimalVar = dbo.ufn\_CubicVolume @CubeLength = 12.3,

@CubeHeight = 4.5, @CubeWidth = 4.5;

SELECT @MyDecimalVar;

GO

The following example executes the dbo.ufn\_CubicVolume function without specifying the parameter names:

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl19other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl19other');)

DECLARE @MyDecimalVar decimal(12,3);

EXEC @MyDecimalVar = dbo.ufn\_CubicVolume 12.3, 4.5, 4.5;

SELECT @MyDecimalVar;

GO

You can also use the ODBC CALL syntax to execute the dbo.ufn\_CubicVolume function from OLE DB or ODBC applications:

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl20other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl20other');)

-- First use SQLBindParam to bind the return value parameter marker

-- to a program variable of the appropriate type

SQLExecDirect(hstmt,

"{ CALL ? = dbo.ufn\_CubicVolume(12.3, 4.5, 4.5) }",

SQL\_NTS);

 Invoking User-defined Functions That Return a table Data Type

You can invoke a user-defined function that returns a **table** where table expressions are allowed in the FROM clause of SELECT, INSERT, UPDATE, or DELETE statements. An invocation of a user-defined function that returns a table can be followed by an optional table alias. The following example illustrates calling the table-valued function dbo.ufnGetContactInformation in the FROM clause of a SELECT statement.

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl24other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl24other');)

USE AdventureWorks;

GO

SELECT ContactID, FirstName, LastName, JobTitle, ContactType

FROM dbo.ufnGetContactInformation(2200);

GO

SELECT ContactID, FirstName, LastName, JobTitle, ContactType

FROM dbo.ufnGetContactInformation(5);

GO

When a user-defined function that returns a table is invoked in the FROM clause of a subquery, the function arguments cannot reference any columns from the outer query.

Static, read-only cursors are the only type of cursor that can be opened on a SELECT statement whose FROM clause refers to a user-defined function that returns a table.

A SELECT statement that references a user-defined function that returns a **table** invokes the function one time.

 Invoking Built-in Table-valued Functions

There are several built-in table-valued functions that return a table value. The invocation of these built-in user-defined functions can be either unqualifed or can use the **sys** schema qualifier. You should use the **sys** schema qualifier for built-in table-valued functions, because it prevents conflicts with user-defined functions of the same name. The following example shows how to invoke the system built-in function fn\_helpcollations.

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl28other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl28other');)

SELECT \*

FROM sys.fn\_helpcollations();

GO

 Using Hints with Table-valued Functions

When you create a user-defined function, you can apply a table hint in any queries that form the function definition. Hints applied to views that reference Transact-SQL table-valued functions are also applied to the functions. These functions can conflict with the hints in the function definition. For more information, see [View Resolution](http://msdn.microsoft.com/en-us/library/ms190237.aspx).

You cannot apply hints on views that reference CLR table-valued functions.

|  |
| --- |
| **Note:** |
| The ability of the Database Engine to apply hints on views to multi-statement table-valued functions that are part of the view definition will be removed in a future version of SQL Server. |

You cannot apply a table hint to the result of any table-valued function in the FROM clause of a query.

**Viewing User-Defined Functions**

Several system stored procedures and catalog views provide information about stored procedures. Using these, you can:

* See the definition of the function. That is, the Transact-SQL statements used to create a user-defined function. This can be useful if you do not have the Transact-SQL script files used to create the function.
* Get information about a function such as its schema, when it was created, and its parameters.
* List the objects used by the specified function, and the objects that use the specified function. This information can be used to identify the functions affected by the changing or removal of an object in the database.

**To view the definition of a user-defined function**

* [sys.sql\_modules (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms175081.aspx)
* [OBJECT\_DEFINITION (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms176090.aspx)
* [sp\_helptext (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms176112.aspx)

**To view information about a user-defined function**

* [sys.objects (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms190324.aspx)
* [sys.parameters (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms176074.aspx)
* [sp\_help (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms187335.aspx)

**To view the dependencies of a user-defined function**

* [sys.sql\_expression\_dependencies (Transact-SQL)](http://msdn.microsoft.com/en-us/library/bb677315.aspx)
* [sys.dm\_sql\_referenced\_entities (Transact-SQL)](http://msdn.microsoft.com/en-us/library/bb677185.aspx)
* [sys.dm\_sql\_referencing\_entities (Transact-SQL)](http://msdn.microsoft.com/en-us/library/bb630351.aspx)

 Examples

**A. Using system catalog views to return user-defined function information**

The following examples use the catalog views sys.objects and sys.parameters to return information about user-defined functions and their parameters.

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl15other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl15other');)

-- Display metadata about the user-defined functions in AdventureWorks.

USE AdventureWorks;

GO

SELECT \*

FROM sys.objects

WHERE type IN ('IF','TF','FN','FS','FT');

GO

-- Return parameters associated with the functions

SELECT o.name AS FunctionName, p.\*

FROM sys.objects AS o

JOIN sys.parameters AS p ON o.object\_id = p.object\_ID

WHERE type IN ('IF','TF','FN','FS','FT');

GO

**B. Using OBJECT\_DEFINITION**

The following example uses the system function OBJECT\_DEFINITION to return the definition of user-defined function dbo.ufnGetContactInformation.

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl16other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl16other');)

USE AdventureWorks;

GO

SELECT OBJECT\_DEFINITION(OBJECT\_ID('dbo.ufnGetContactInformation'));

GO

**C. Using sys.sql\_expression\_dependencies**

The following example uses the sys.sql\_expression\_dependencies and sys.columns catalog views to return the table and column names on which the user-defined function dbo.ufnGetContactInformation depends.

[[](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl17other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl17other');)

USE AdventureWorks;

GO

SELECT OBJECT\_NAME(d.referencing\_id) AS referencing\_entity,

OBJECT\_NAME(referenced\_id) AS referenced\_entity,

referenced\_minor\_id AS referenced\_column\_id,

c.name AS referenced\_column

FROM sys.sql\_expression\_dependencies AS d

JOIN sys.columns AS c ON c.object\_id = d.referenced\_id

AND c.column\_id = d.referenced\_minor\_id

WHERE d.referencing\_id = OBJECT\_ID(N'AdventureWorks.dbo.ufnGetContactInformation');

**Creating CLR Functions**

You can create a database object inside an instance of SQL Server that is programmed in an assembly created in the Microsoft .NET Framework common language runtime (CLR). Database objects that can leverage the rich programming model provided by the common language runtime include aggregate functions, functions, stored procedures, triggers, and types.

Creating a CLR function in SQL Server involves the following steps:

* Define the function as a static method of a class in a language supported by the .NET Framework. For more information about how to program functions in the common language runtime, see [CLR User-Defined Functions](http://msdn.microsoft.com/en-us/library/ms131077.aspx). Then, compile the class to build an assembly in the .NET Framework by using the appropriate language compiler.
* Register the assembly in SQL Server by using the CREATE ASSEMBLY statement. For more information about assemblies in SQL Server, see [Assemblies (Database Engine)](http://msdn.microsoft.com/en-us/library/ms186221.aspx).
* Create the function that references the registered assembly by using the [CREATE FUNCTION](http://msdn.microsoft.com/en-us/library/ms186755.aspx) statement.

|  |
| --- |
| **Note:** |
| Deploying a SQL Server Project in Microsoft Visual Studio registers an assembly in the database that was specified for the project. Deploying the project also creates CLR functions in the database for all methods annotated with the **SqlFunction** attribute. For more information, see [Deploying CLR Database Objects](http://msdn.microsoft.com/en-us/library/ms345099.aspx). |
| **Note:** |
| The ability of SQL Server to execute CLR code is off by default. You can create, alter, and drop database objects that reference managed code modules, but these references will not execute in SQL Server unless the [clr enabled Option](http://msdn.microsoft.com/en-us/library/ms175193.aspx) is enabled by using [sp\_configure (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188787.aspx). |

 Accessing External Resources

CLR functions can be used to access external resources such as files, network resources, Web Services, other databases (including remote instances of SQL Server). This can be achieved by using various classes in the .NET Framework, such as System.IO, System.WebServices, System.Sql, and so on. The assembly that contains such functions should at least be configured with the EXTERNAL\_ACCESS permission set for this purpose. For more information, see [CREATE ASSEMBLY (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms189524.aspx). The SQL Client Managed Provider can be used to access remote instances of SQL Server. However, loopback connections to the originating server are not supported in CLR functions.

**To create, modify, or drop assemblies in SQL Server**

* [CREATE ASSEMBLY (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms189524.aspx)
* [ALTER ASSEMBLY (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms186711.aspx)
* [DROP ASSEMBLY (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms177514.aspx)

**To create a CLR function**

* [CREATE FUNCTION (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms186755.aspx)

**Creating User-Defined Aggregates**

You can create a database object inside SQL Server that is programmed in a CLR assembly. Database objects that can leverage the rich programming model provided by the CLR include triggers, stored procedures, functions, aggregate functions, and types.

Like the built-in aggregate functions provided in Transact-SQL, user-defined aggregate functions perform a calculation on a set of values and return a single value.

Creating a user-defined aggregate function in SQL Server involves the following steps:

* Define the user-defined aggregate function as a class in a Microsoft .NET Framework-supported language. For more information about how to program user-defined aggregates in the CLR, see [CLR User-Defined Aggregates](http://msdn.microsoft.com/en-us/library/ms131057.aspx). Compile this class to build a CLR assembly using the appropriate language compiler.
* Register the assembly in SQL Server using the CREATE ASSEMBLY statement. For more information about assemblies in SQL Server, see [Assemblies (Database Engine)](http://msdn.microsoft.com/en-us/library/ms186221.aspx).
* Create the user-defined aggregate that references the registered assembly using the CREATE AGGREGATE statement.

|  |
| --- |
| **Note:** |
| Deploying a SQL Server Project in Microsoft Visual Studio registers an assembly in the database that was specified for the project. Deploying the project also creates a user-defined aggregate in the database for all class definitions annotated with the **SqlUserDefinedAggregate** attribute. For more information, see [Deploying CLR Database Objects](http://msdn.microsoft.com/en-us/library/ms345099.aspx). |
| **Note:** |
| The ability of SQL Server to execute CLR code is off by default. You can create, alter and drop database objects that reference managed code modules, but these references will not execute in SQL Server unless the [clr enabled Option](http://msdn.microsoft.com/en-us/library/ms175193.aspx) is enabled using [sp\_configure (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms188787.aspx). |

**To create, modify, or drop an assembly**

* [CREATE ASSEMBLY (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms189524.aspx)
* [ALTER ASSEMBLY (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms186711.aspx)
* [DROP ASSEMBLY (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms177514.aspx)

**To create a user-defined aggregate**

* [CREATE AGGREGATE (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms182741.aspx)