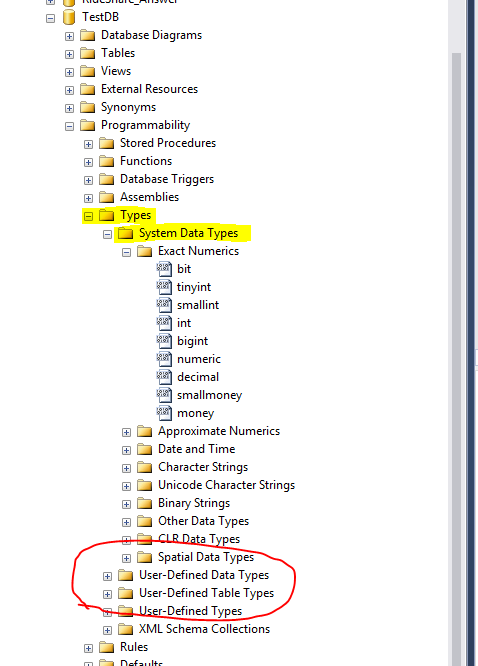
Table Value Functions & User Datatypes

# Creating Datatypes

## System Types

We have seen that the SQL Server metadata tables store almost everything about our databases. We’ve seen our tables, columns, indexes, constraints, procedures, and more all in the object explorer, which is our most common view into the metadata tables.

If we explore the “Programmability” folder a bit more, we can see a few more interesting things:



SQL Server stores all its data types in the metadata. We could explore all the built-in data types right within SSMS. We also see folders for User-Defined Data Types and User-Defined Table types, which we are going to explore in this lesson.

**Best Practice**

**Avoid Deprecated Data Types**

The following datatypes are deprecated and should not be used: TEXT; NTEXT; IMAGE; SQL\_VARIANT.

## Create Type

<https://technet.microsoft.com/en-us/library/ms175007(v=sql.105).aspx>

CREATE TYPE [ schema\_name. ] type\_name

{

FROM base\_type

[ ( precision [ , scale ] ) ]

[ NULL | NOT NULL ]

| EXTERNAL NAME assembly\_name [ .class\_name ]

| AS TABLE ( { <column\_definition> | <computed\_column\_definition> }

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

} [ ; ]

The CREATE TYPE statement is used to create a new datatype based on one of the existing datatypes. Let’s look at some interesting details:

* Types are created in a schema, just like any other object. We should be using explicit schemas.
* We define a new datatype using a built-in base type.
* We can build nullability right into the datatype.
* There is the possibility to create datatypes that are defined as tables, which we will be exploring soon.

Let’s create our first user defined datatype.

We are working on a system with many lookup tables. These lookup tables frequently have a short, readable name, and a longer description. We would like to standardize the size of these fields, so we are going to use user defined data types.

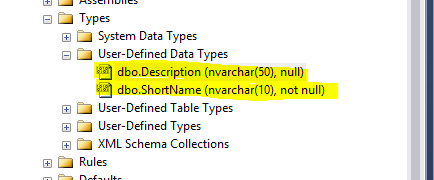
USE TestDB;

GO

CREATE TYPE dbo.ShortName FROM NVARCHAR(10) NOT NULL;

CREATE TYPE dbo.Description FROM NVARCHAR(50) NULL;

Using the code above, we have created two new user defined data types, which we can now see in the object explorer.



We can now use these data types as we would any other datatype. We can use these to define columns, parameters, and variables. Let’s create a quick test table:

CREATE TABLE dbo.Province (

ProvinceID INT IDENTITY PRIMARY KEY,

ProvinceName dbo.ShortName,

ProvinceDesc dbo.Description

);

We can create new columns using these data types, just like we would any other. Let’s test the boundaries of these new types:

INSERT INTO dbo.Province ( ProvinceName, ProvinceDesc )

VALUES ( 'ON', 'Ontario' ); -- Normal insert

INSERT INTO dbo.Province ( ProvinceName, ProvinceDesc )

VALUES ( '12345678901', NULL ); -- Long character insert

INSERT INTO dbo.Province ( ProvinceName, ProvinceDesc )

VALUES ( NULL, NULL ); -- NULL insert

Did these data types behave the way you expected?

Notice that we defined ShortName as not null, so our ProvinceName column is not nullable. This does not mean that columns based on ShortName are never nullable. The nullability of a data type only defines its default. We can override this by explicitly defining the nullability at the column level.

CREATE TABLE dbo.Province (

ProvinceID INT IDENTITY PRIMARY KEY,

ProvinceName dbo.ShortName NULL,

ProvinceDesc dbo.Description

);

INSERT INTO dbo.Province ( ProvinceName, ProvinceDesc )

VALUES ( NULL, NULL ); -- NULL insert

**Data Types Cannot Be Altered**

It is not possible to alter a user defined data type. You can only drop and recreate it. This requires you to drop all objects that are dependent on that data type first. Keep in mind that user defined data types are not necessarily suitable if you want to be able to quickly and easily change the definition for a large group of columns.

# User Defined Table Types

## Creating a Table Type

In SQL Server, it is possible to define a type that behaves like a table. It can store multiple rows and columns instead of a single value, giving us more flexibility.

CREATE TYPE dbo.ProvinceType AS TABLE (

ProvinceID INT IDENTITY PRIMARY KEY,

ProvinceName dbo.ShortName DEFAULT 'Unknown',

ProvinceDesc dbo.Description

);

There are a couple of interesting things here worth noting:

* Just like any other object, our name must be unique.
* We are able to define primary and unique keys, which can help with performance in some cases.
* Many of the constraints that are available to us in normal tables are available to us here. In this example, we’ve created a default constraint and an identity.
* We can use user defined data types as part of our definition for the table type.

It is not possible to use table types as the datatype for table columns, but we can still use them for variables and parameters.

This can give our stored procedures a lot more flexibility.

## Table Type Input Parameters

As we know, it’s much better to work with data sets instead of RBAR processing. Let’s create a stored procedure that will insert a block of new provinces, all at once.

CREATE PROCEDURE dbo.CreateProvinces (

@Provinces dbo.ProvinceType READONLY

)

AS

BEGIN;

SET NOCOUNT ON;

SET XACT\_ABORT ON;

INSERT INTO dbo.Province ( ProvinceName, ProvinceDesc )

SELECT ProvinceName, ProvinceDesc

FROM @Provinces;

END;

This procedure takes a list of provinces and inserts it into the Province table. There are a few notable things happening here:

* We have created the @Provinces parameter just like any other parameter
* When a parameter is defined as a table type, that parameter must be defined as READONLY. It is not possible to use them as a target of an INSERT, UPDATE or DELETE statement.
* We are able to select from @Provinces, just like we would any other table.

To execute this procedure, let’s create a new variable, fill it, then pass it to this procedure. When we’re done, we’ll select from the table to make sure it worked:

--Reset the province table

TRUNCATE TABLE dbo.Province;

DECLARE @Provinces dbo.ProvinceType;

INSERT INTO @Provinces ( ProvinceName, ProvinceDesc )

VALUES ( 'BC', 'British Columbia' ),

( 'AB', 'Alberta' ),

( 'SK', 'Saskatchewan' ),

( 'MB', 'Manitoba' ),

( 'ON', 'Ontario' ),

( 'QC', 'Quebec' ),

( 'NB', 'New Brunswick' ),

( 'PE', 'Prince Edward Island' ),

( 'NS', 'Nova Scotia' ),

( 'NL', 'Newfoundland' );

EXECUTE dbo.CreateProvinces @Provinces;

SELECT \* FROM dbo.Province;

# Table Valued Functions

## Inline Table Valued Functions

We are able to define functions that return tables instead of scalar values. When we do that, we create a table-valued function.

Let’s create a new function that will get the province IDs for a list of provinces. The first way we can do that is by not setting a specific return object and just returning to TABLE. When we do this, we create an **inline table valued function**.

CREATE FUNCTION dbo.GetProvinceIDs (

@Provinces dbo.ProvinceType READONLY

)

RETURNS TABLE

AS

RETURN (

SELECT pr.ProvinceID, pr.ProvinceName, pr.ProvinceDesc

FROM dbo.Province pr

JOIN @Provinces pr\_in

ON pr.ProvinceName = pr\_in.ProvinceName );

GO

If you still have your table populated from before, we can run this and get our list. Notice that when we call the function, we can use the function the same way we would use a table.

DECLARE @Provinces dbo.ProvinceType;

INSERT INTO @Provinces ( ProvinceName, ProvinceDesc )

VALUES ( 'ON', 'Ontario' ),

( 'QC', 'Quebec' ),

( 'NB', 'New Brunswick' ),

( 'NS', 'Nova Scotia' ),

( 'NL', 'Newfoundland' );

SELECT \*

FROM dbo.GetProvinceIDs(@Provinces);

Did that return what you expected?

Notice that we are limited to a return statement only, we cannot create a block of code that executes multiple statements.

## Multi-statement Table Valued Functions

We will create a new function, but this time use a multi-statement table valued function.

DROP FUNCTION dbo.GetProvinceIDs;

GO

CREATE FUNCTION dbo.GetProvinceIDs (

@Provinces dbo.ProvinceType READONLY

)

RETURNS @retProvinces TABLE (

ProvinceID INT,

ProvinceName dbo.ShortName,

ProvinceDesc dbo.Description

)

AS

BEGIN

INSERT INTO @retProvinces

SELECT pr.ProvinceID, pr.ProvinceName, pr.ProvinceDesc

FROM dbo.Province pr

JOIN @Provinces pr\_in

ON pr.ProvinceName = pr\_in.ProvinceName;

RETURN;

END;

GO

Notice there are a few quirks here:

* We have to give the return value a parameter name. We insert into that parameter before we return.
* We must give the return parameter a definition. It is not possible to simply use our user defined table type.
* We can insert into our return parameter multiple times. The function will return whatever is in the parameter at the end of the function.
* The RETURN keyword is still mandatory, but this time is called on its own.

If you want to test our new function, we can use the same code from above:

DECLARE @Provinces dbo.ProvinceType;

INSERT INTO @Provinces ( ProvinceName, ProvinceDesc )

VALUES ( 'ON', 'Ontario' ),

( 'QC', 'Quebec' ),

( 'NB', 'New Brunswick' ),

( 'NS', 'Nova Scotia' ),

( 'NL', 'Newfoundland' );

SELECT \*

FROM dbo.GetProvinceIDs(@Provinces);

# Homework

## Understand

1. Create a user defined data type that might be appropriate for a Social Insurance Number.
2. Create a user defined data type that might be appropriate for a first or last name.
3. Create a user defined data type that might be appropriate for a Fanshawe student number.
4. Create a user defined table type that stores SIN, first name, last name, and student number for a student, using the user defined data types created above.
5. Create a new table that can store StudentID, SIN, first name, last name, and student number. Create a stored procedure that can insert a group of new rows into this table, if passed a table parameter using the user defined table type you created above.
6. Create an inline table valued function that will return the StudentID and SIN of all students that match a first and last name passed in as a parameter.
7. Recreate the function from step 6, but as a multi-line table valued function.