

Lab 2 - Introduction to Transact-SQL

Content:

- Introduction to Transact-SQL
- Introduction to the SQL command window in Microsoft SQL Server 2014
- Practice of simple T-SQL commands

Duration: 4 teaching periods

Learning outcome:

- Familiar with T-SQL, how to edit and execute SQL commands in Query Editor of SQL Server 2014
- Knowledge of Data Definition Commands in T-SQL
- Practice of Simple Data Query with SELECT

Part 1: Introduction to Transact-SQL

Transact-SQL, sometimes abbreviated T-SQL, is proprietary extension to the SQL language and central to using SQL Server. All applications that communicate with an instance of SQL Server do so by sending Transact-SQL statements to the server, regardless of the user interface of the application.

The following is a list of the kinds of applications that can generate Transact-SQL:

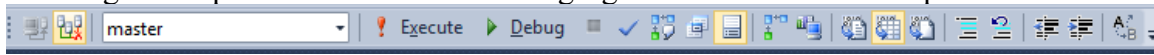
- General office productivity applications.
- Applications that use a graphical user interface (GUI) to let users select the tables and columns from which they want to see data.
- Applications that use general language sentences to determine what data a user wants to see.
- Line of business applications that store their data in SQL Server databases. These applications can include both applications written by vendors and applications written in-house.
- Applications created by using development systems such as Microsoft Visual C++, Microsoft Visual Basic, that use database APIs such as ADO, OLE DB, and ODBC.
- Web pages that extract data from SQL Server databases.





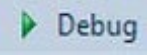



Part 2: Query Editor in Microsoft SQL Server 2014













As you progress through the labs, the creation of objects, the manipulation of data, and the execution of code will be shown either by using the graphical interface and options that Object Explorer provides or by writing code using T-SQL. To write code, we need a free-form text editor so that we can type anything we need. Luckily, SSMS provides just such an editor as a tabbed screen within the document view on the right-hand side. This is

known as Query Editor, and it can be found when you click New Query of the main toolbar or by selecting File → New → Database Engine Query.

We discussed some of the options that affect Query Editor, such as how text is entered and how results from running the T-SQL code are displayed. There is not a great deal to say about the editor itself, as it really is a free-form method of entering commands and statements for SQL Server to execute. However, Query Editor has a toolbar that is worth covering at this point in time. The following figure shows this toolbar options.



Tool	Tool Name	Description
	Connect	Requests a connection to the server if one doesn't currently exist
	Change Connection	Allows you to change the current connection by disconnecting and reconnecting the connection you are using
	Available Databases	A combobox that lists all the databases in the server you are currently connected to; if you want to run a query against a different database you can select the database here. The code will execute against the database displayed provided that you have permission to run against it.
	Execute	Executes the code in the query window or if some of the code is highlighted will execute only the highlighted code.
	Debug	Will debug and allow you to step through the code; you will see this in Chapter 14.
	Cancel Executing Query	Sends a cancel signal to the server to request the query that is executing to be stopped as soon as the server can; a cancel is not always immediate. There will be a delay in sending the command while SQL Server "pauses" to receive the command.
	Parse	Parse the query code looking for syntactical errors
	Display Estimated Execution Plan	Helps you to analyze your T-SQL query for optimization; displays the

		query plan SQL Server estimates would execute if you were to run the code
	Query Options	The connection options that will be used when executing the code
	IntelliSense Enabled	The ability to allow SSMS to help you write your code; when you start typing code if IntelliSense is enabled then SSMS will attempt to intelligently sense what you are wanting to achieve and suggest column names and syntax keywords and so forth to help you write your code more quickly than if you were to type it all out by hand.
	Include Actual Execution Plan	Then your code is executed you can also output the steps SQL Server executed between the start and end of your query 4this will be placed in a separate tab to the results and any messages.
	Include Client Statistics	Displays statistics about the query; this will be placed in a separate tab to the results and any messages.
	Results To Text	Displays the results in text format
	Results To Grid	Displays the results in a grid format, the default
	Results To File	Places the results in an output file rather than in SQL Server Management Studio
	Comment out the selected lines	Takes the lines that are highlighted and comments them out so that they don't execute.
	Uncomment out the selected lines	Takes the lines that are highlighted and uncomments them out so that they don't execute.
	Decrease Indent	Decreases the indentation of the highlighted code
	Increase Indent	Increases the indentation of the highlighted code
	Specify Values For Template Parameters	SSMS has templates for many actions. You will see these used

		throughout the book then a template is used, pressing this button will allow you to enter values in the template options.
--	--	---

Practice: Self-exploring Query Editor.

Part 3: Data Definition Command in T-SQL

Practice: Defining a Table Through Query Editor

1. Ensure that you are pointing to the ApressFinancial database in Query Editor, as in the following table:



For preparation, it is necessary to open CreateDatabase_Table.sql file, double check the file path in the script, and then run it.

2. In Query Editor, enter the following code:

```
CREATE TABLE TransactionDetails.Transactions
(
  TransactionId bigint IDENTITY(1,1) NOT NULL,
  CustomerId bigint NOT NULL,
  TransactionType int NOT NULL,
  DateEntered datetime NOT NULL,
  Amount numeric(18, 5) NOT NULL,
  ReferenceDetails nvarchar(50) NULL,
  Notes nvarchar(max) NULL,
  RelatedShareId bigint NULL,
  RelatedProductId bigint NOT NULL)

```

3. Execute the code by either pressing Ctrl+E or F5, or clicking the toolbar Execute button.

4. You should now see the following message in the results pane:

The command(s) completed successfully.

5. However, you may have received an error message instead. This could be for a number of reasons, from a typing mistake through to not having the authority to create tables.

6. Now move to Object Explorer. If it is already open, you will have to refresh the Details pane (by right-clicking the Tables node and selecting Refresh). You should then see the TransactionDetails.Transactions table alongside the CustomerDetails.Customers table created previously.

7. Similarly, you create the TransactionTypes as follows:

```
USE ApressFinancial
GO
IF OBJECT_ID('TransactionDetails.TransactionTypes',
'U') IS NOT NULL
DROP TABLE TransactionDetails.TransactionTypes

```

```
GO
CREATE TABLE TransactionDetails.TransactionTypes(
TransactionTypeId int IDENTITY(1,1) NOT NULL,
TransactionDescription nvarchar(30) NOT NULL,
CreditType bit NOT NULL
)
GO
```

The basic syntax for creating a table is as follows:

```
CREATE TABLE [database_name].[schema_name].table_name
(column_name data_type [length] [IDENTITY(seed,
increment)] [NULL/NOT NULL])
```

Further reading: Schemas on page 122 in book *Beginning SQL Server 2012 for Developers*

The ALTER TABLE Statement

The ALTER TABLE statement allows restrictive alterations to a table layout but keeps the contents. Columns can be added, removed, or modified using the ALTER TABLE statement. Removing a column will simply remove the data within that column, but careful thought has to take place before adding, removing, or altering a column.

Add a column:

1. First of all, open up Query Editor and ensure that you are pointing to the ApressFinancial database. Then write the code to alter the TransactionDetails.TransactionTypes table to add the new column. The format is very simple. We specify the table prefixed by the **schema name** we want to alter after the ALTER TABLE statement. Next we use a comma-delimited list of the columns we wish to add. We define the name, the data type, the length if required, and finally whether we allow NULLs or not. As we don't want the existing data to have any default values, we will have to define the column to allow NULL values.

```
ALTER TABLE TransactionDetails.TransactionTypes
ADD AffectCashBalance bit NULL
GO
```

2. Once we've altered the data as required, we then want to **remove the ability for further rows of data to have a NULL value**. This new column will take a value of 0 or 1. Again, we use the ALTER TABLE statement, but this time we'll add the ALTER COLUMN statement with the name of the column we wish to alter. After this statement, the results are the alterations we wish to make. Although we are not altering the data type, it is a mandatory requirement to redefine the data type and data length. After this, we can inform SQL server that the column will not allow NULL values.

```
ALTER TABLE TransactionDetails.TransactionTypes
ALTER COLUMN AffectCashBalance bit NOT NULL
GO
```

3. Execute the preceding code to make the TransactionDetails.TransactionTypes table correct.

Creating the Remaining Tables

We need to create the remaining four tables. We will do this as code placed in Query Editor. There is nothing specifically new to cover in this next section, and therefore only the code is listed. Enter the following code and then execute it as before. You can then move into SQL Server and refresh it, after which you should be able to see the new tables.

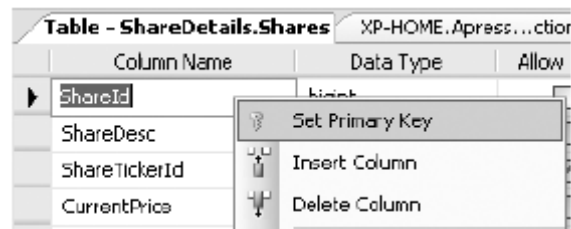
```
USE ApressFinancial
GO
CREATE TABLE CustomerDetails.CustomerProducts(
    CustomerFinancialProductId bigint NOT NULL,
    CustomerId bigint NOT NULL,
    FinancialProductId bigint NOT NULL,
    AmountToCollect money NOT NULL,
    Frequency smallint NOT NULL,
    LastCollected datetime NOT NULL,
    LastCollection datetime NOT NULL,
    Renewable bit NOT NULL
)
ON [PRIMARY]
GO
CREATE TABLE CustomerDetails.FinancialProducts(
    ProductId bigint NOT NULL,
    ProductName nvarchar(50) NOT NULL
) ON [PRIMARY]
GO
CREATE TABLE ShareDetails.SharePrices(
    SharePriceId bigint IDENTITY(1,1) NOT NULL,
    ShareId bigint NOT NULL,
    Price numeric(18, 5) NOT NULL,
    PriceDate datetime NOT NULL
) ON [PRIMARY]
GO
CREATE TABLE ShareDetails.Shares(
    ShareId bigint IDENTITY(1,1) NOT NULL,
    ShareDesc nvarchar(50) NOT NULL,
    ShareTickerId nvarchar(50) NULL,
    CurrentPrice numeric(18, 5) NOT NULL
) ON [PRIMARY]
GO
```

Practice: Defining a Primary Key


This section will demonstrate how to set a primary key of a table over the interface of SQL server.

1. In SQL Server, you have navigated to the ApressFinancial database. Find the ShareDetails.Shares table, and right-click and select Design. Once in the Table

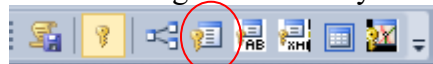
Designer, select the **ShareId** column. This will be the column we are setting the primary key for. Right-click to bring up the popup menu shown in the following figure.



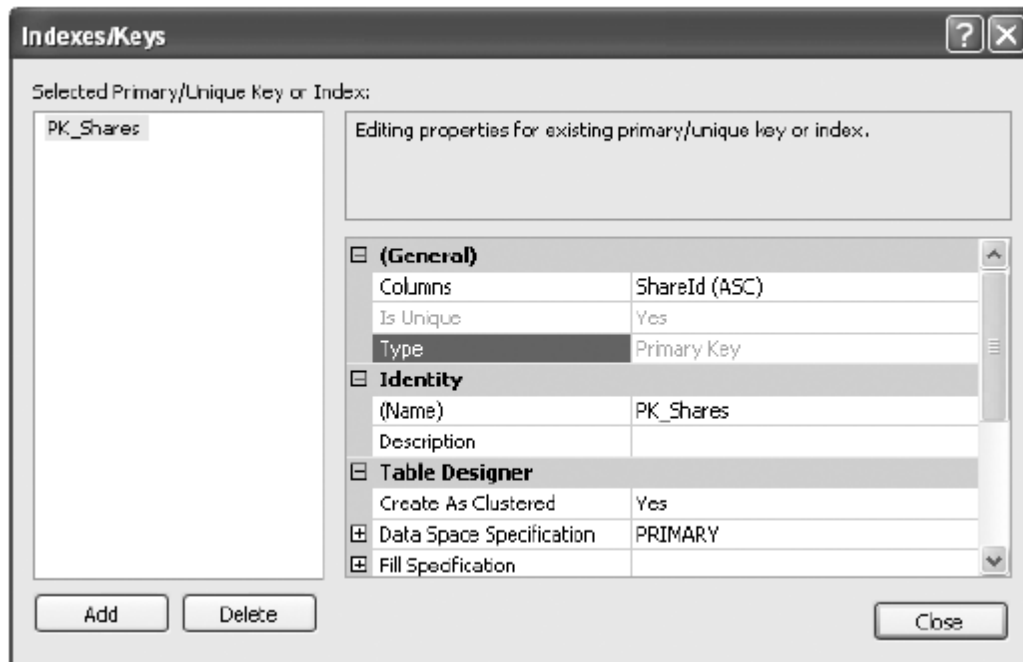
2. Select the **Set Primary Key** option from the pop-up menu. This will then change the display to place a small key in the leftmost column details. Only one column has been defined as the primary key, as you see in the following figure.

Table - ShareDetails.Shares*			
Column Name	Data Type	Allow Nulls	
ShareId	bigint	<input type="checkbox"/>	
ShareDesc	nvarchar(50)	<input type="checkbox"/>	
ShareTickerId	nvarchar(50)	<input checked="" type="checkbox"/>	
CurrentPrice	numeric(18, 5)	<input type="checkbox"/>	
		<input type="checkbox"/>	

3. However, this is not all that happens, as you will see. Save the table modifications by clicking the **Save** button. Click the **Manage Indexes/Keys** button on the toolbar.



This brings up the dialog box shown in the following figure.



Look at Type, the third option down in the General section. It says Primary Key. Notice that a key definition has been created for you, with a name and the selected column, informing you that the index is unique and clustered (more on indexes and their relation to primary keys will be introduced in later labs).

That's all there is to creating and setting a primary key. A primary key has now been set up on the ShareDetails.Shares table. In this instance, any record added to this table will ensure that the data will be kept in ShareId ascending order, and it is impossible to insert a duplicate row of data. This key can then be used to link to other tables within the database at a later stage.

Creating a Relationship between Tables

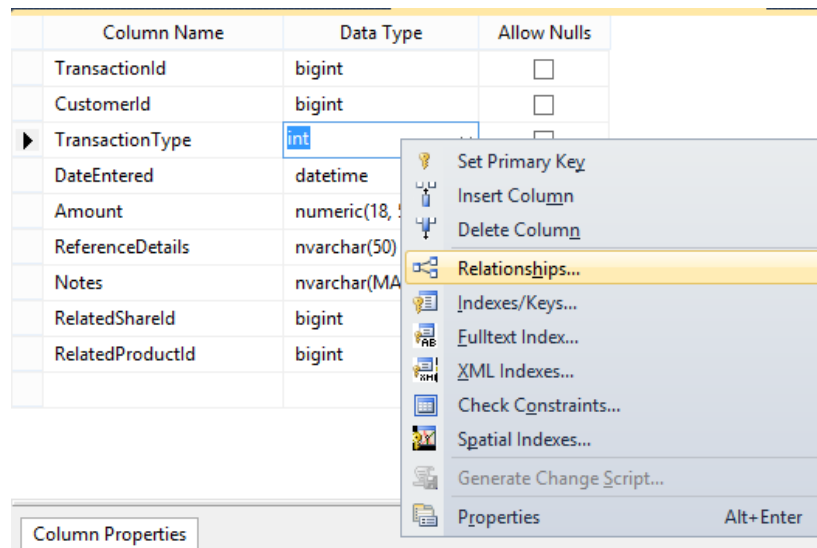
In a database schema, there are relationships, each links two tables. In this section, the **first relationship** that we create will be between the **customer** and **customer transactions** tables. This will be a one-to-many relationship where there is one customer record to many transaction records. Keep in mind that although a customer may have several customer records, one for each product he or she has bought, the relationship is a combination of customer and product to transactions because a new CustomerId will be generated for each product the customer buys. We will now build that first relationship.

1. In SQL Server environment, we have the ApressFinancial database be selected and expanded. We need to add a primary key to CustomerDetails.Customers. Enter the code that follows and then execute it:

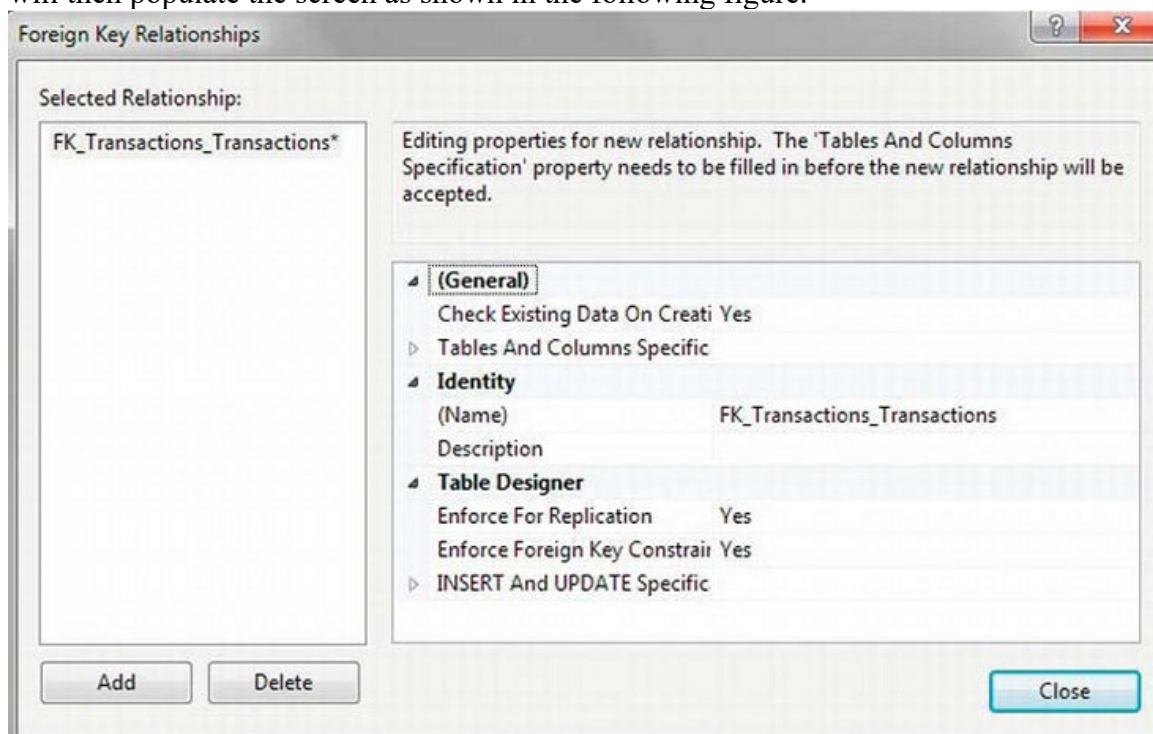
```
ALTER TABLE CustomerDetails.Customers
ADD CONSTRAINT
    PK_Customers PRIMARY KEY NONCLUSTERED
    (
        CustomerId
    )
WITH (STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
ON [PRIMARY]
GO
```

2. Find and select the TransactionDetails.Transactions table, and then right-click. Select Design to invoke the Table Designer.

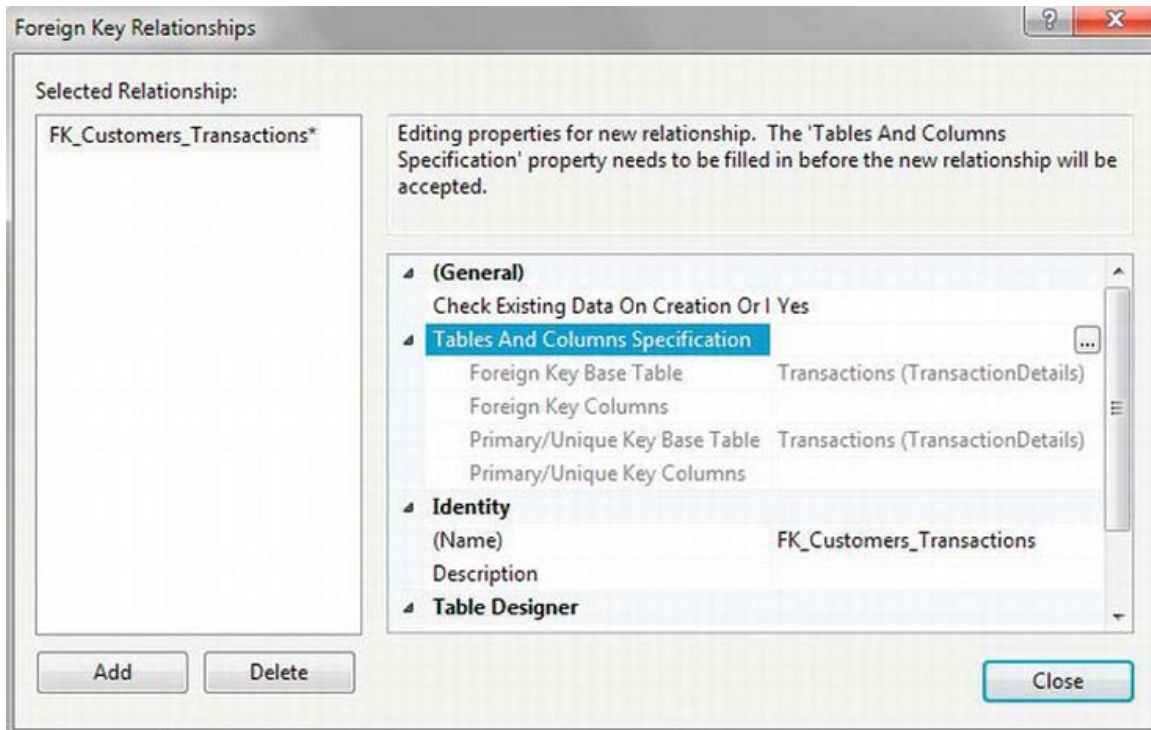
3. Once in the Table Designer, right-click and select Relationships from the pop-up menu shown in the following figure. Or click the Relationships button on the Table Designer toolbar.



4. This brings up the Relationship Designer. As it is empty, you need to click Add. This will then populate the screen as shown in the following figure.



5. Expand the Tables and Columns Specified node, which will allow the relationship to be built. Notice that there is now an ellipse button on the right, as shown in the following figure. To create the relationship, click the ellipse button (...).



6. The first requirement is to change the name to make it more meaningful. Quite often you will find that naming the key `FK_ParentTable_ChildTable` is the best method, so in this case change it to `FK_Customers_Transactions` as the `CustomerDetails.Customers` table will be the master table for this foreign key. We also need to define the column in each table that is the link. We are linking every one customer record to many transaction records, and we can do so via the `CustomerId`. So select that column for both tables, as shown in the following figure. Now click OK.

Relationship name:
FK_Customers_Transactions

Primary key table:
Customers (CustomerDetails) ▼
CustomerId

Foreign key table:
Transactions (TransactionDetails)
CustomerId ▼

OK Cancel

Note: In this instance, both columns have the same name, but this is not mandatory. The only requirement is that the information in the two columns be the same.

7. This brings us back to the Foreign Key Relationships definition screen, shown in the following figure.

(General)
Check Existing Data On Creation Or Re-Enab Yes

Tables And Columns Specification

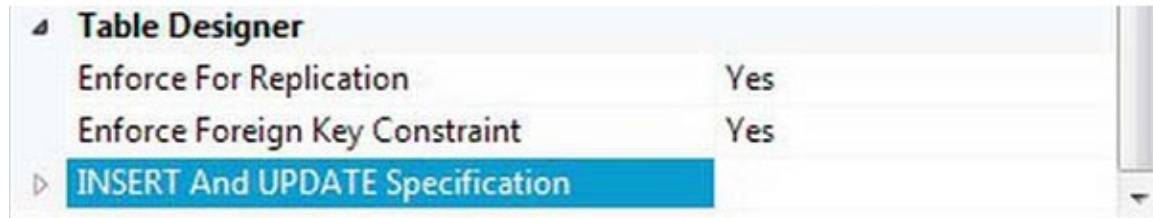
Foreign Key Base Table	Transactions (TransactionD
Foreign Key Columns	CustomerId
Primary/Unique Key Base Table	Customers (CustomerDetai
Primary/Unique Key Columns	CustomerId

Identity
(Name) FK_Customers_Transaction
Description

Table Designer

Notice that at the top of the list items in the grayed-out area you can see the details of the foreign key we just defined. Within the Identity section there is now also a description of the foreign key. Ignore the option Enforce for Replication, which you'll see is in the Table Designer section if you scroll down in the Foreign Key Relationships screen.

8. There are three other options we are interested in that are displayed at the bottom of the dialog box, as shown in the following figure. Leave the options at the defaults.



9. Closing this dialog box does not save the changes. Save your changes before closing the Table Designer.

Check Existing Data on Creation

If there is data in either of the tables, by setting this option to Yes we instruct SQL Server that when the time comes to physically add the relationship, the data within the tables is to be checked. If the data meets the definition of the relationship, the relationship is successfully inserted into the table. However, if any data fails the relationship test, the relationship is not applied to the database. An example of this would be when it is necessary to ensure that there is a customer record for all transactions, but there are customer transactions records that don't have a corresponding customer record, which would cause the relationship to fail. Obviously, if you come across this, you have a decision to make. Either correct the data by adding master records or altering the old records, and then reapply the relationship, or revisit the relationship to ensure it is what you want.

By creating the relationship, you want the data within the relationship to work, therefore you would select No if you were going to go back and fix the data after the additions. What if you still miss rows? Would this be a problem? In our scenario, there should be no transaction records without customer records. But you may still wish to add the relationship to stop further anomalies going forward.

Enforce Foreign Key Constraints

Once the relationship has been created, it is possible to prevent the relationship from being broken. If you set Check Existing Data on Creation from higher up in the dialog box to Yes, you are more than likely hoping to keep the integrity of the data intact. That option will only check the existing data. It does nothing for further additions, deletions, etc., on the data. However, by setting the Enforce Foreign Key Constraints option to Yes, we will ensure that any addition, modification, or deletion of the data will not break the relationship. It doesn't stop changing or removing data providing that the integrity of the

database is kept in sync. For example, it would be possible to change the customer number of transactions, providing that the new customer number also exists with the CustomerDetails.Customers table.

Delete Rule/Update Rule

If a deletion or an update is performed, it is possible for one of four actions to then occur on the related data, based on the following options:

No Action: Nothing happens to any related data.

Cascade: If you delete a customer, all of the transaction rows for that customer will also be deleted.

Set Null: If you delete a customer, then if the CustomerId column in the TransactionDetails.Transactions table could accept NULL as a value, the value would be set to NULL. In the customers/transactions scenario, we have specified the column cannot accept NULL values. The danger with this is that you are leaving “unlinked” rows behind, a scenario that can be valid, but do take care.

Set Default: When defining the table, the column could be defined so that a default value is placed in it. On setting the option to this value, you are saying that the column will revert to this default value. Again a dangerous setting, but potentially a less dangerous option than Set Null, as at least there is a meaningful value within the column.

Part 4: Simple Data Query in T-SQL

Insert One Row of Data by the T-SQL INSERT Statement

```
INSERT [INTO]
    {table_name|view_name}
    [{(column_name,column_name,...)}]
    {VALUES (expression, expression, ...)}
```

INTO: optional, for readable purposes

table_name|view_name: where to insert data

column_name: the name of column where data is inserted

{VALUES (expression, expression, ...)}: values will be inserted to respectively listed columns

Example of INSERT:

1. Log in an account which can modify the ApressFinancial database, open a Query Editor window, connect to the ApressFinancial database.
2. Right-click against the ShareDetails.Shares table, select Script Table As → INSERT To → New Query Editor Window.
3. The following code appears:

```

INSERT INTO [ApressFinancial].[ShareDetails].[Shares]
    ([ShareDesc]
    , [ShareTickerId]
    , [CurrentPrice])
VALUES
    (<ShareDesc, nvarchar(50),>
    , <ShareTickerId, nvarchar(50),>
    , <CurrentPrice, numeric,>)

```

4. Modify the code to:

```

SET QUOTED_IDENTIFIER OFF
GO
INSERT INTO [ApressFinancial].[ShareDetails].[Shares]
    ([ShareDesc]
    , [ShareTickerId]
    , [CurrentPrice])
VALUES
    ("ACME'S HOMEBAKE COOKIES INC",
    'AHC1',
    2.34125)

```

The first line allows quotation masks be used in strings.

5. Execute the code by pressing F5 or Ctrl+E, or clicking the Execute button on the toolbar. The following result appears:

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

Default Values

Default values are used when a large number of INSERTs for a column would have the same value entered each time. For such column, the value does not need to appear in the INSERT command.

When creating the CustomerDetails.Customers table, column DateAdded is set up to be populated with a default value which is set by function GETDATE() in SQL Server. When a new row is added, SQL Server calls the function and assigns the returned value to the column. (Note: double check if this column is set to a default value)

Using NULL Values

The next method for avoiding having to fill in data for every column is to allow NULL values in the columns. When defining the tables, set up column's Allow Nulls option checked by value "true" for required columns. As in the following figure, one of the columns in the ShareDetails.Shares table, ShareTickerId, does allow a NULL value to be entered into it.

SANG-WIN.ApressFi...hareDetails.Shares X SQLQuery5.sql - SA...Admin			
	Column Name	Data Type	Allow Nulls
▶	ShareId	bigint	<input type="checkbox"/>
	ShareDesc	nvarchar(50)	<input type="checkbox"/>
	ShareTickerId	nvarchar(50)	<input checked="" type="checkbox"/>
	CurrentPrice	numeric(18, 5)	<input type="checkbox"/>

In this table, ShareID is an IDENTITY column and is auto-filled. Thus, only two rows required values in the INSERT command. Comparing to the previous example, the code is reduced to:

```
INSERT INTO [ApressFinancial].[ShareDetails].[Shares]
    ([ShareDesc]
    , [CurrentPrice])
VALUES
    ("ACME'S HOMEBAKE COOKIES INC",
    2.34125)
```

After executing this command, when opening the table, the result is as:

	ShareId	Description	StockExchangeTicker	CurrentPrice
1	1	ACME'S HOMEBAKE COOKIES INC	AHCI	2.34125
2	2	ACME'S HOMEBAKE COOKIES INC	NULL	2.34125

The value of NULL requires special handling within SQL Server or applications that will be viewing this data. What this value actually means is that the information within the column is unknown; it is not a numeric or an alphanumeric value. Therefore, because you don't know if it is numeric or alphanumeric, you cannot compare the value of a column that has a setting of NULL to the value of any other column, and this includes another NULL column.

Column Constraints

A constraint is essentially a check that SQL Server places on a column to ensure that the data to be entered in the column meets specific conditions. This will keep out data that is not satisfying the condition and therefore avoid data inconsistencies. Constraints are used to keep database integrity by ensuring that a column only receives data within certain parameters.

```
[DateAdded] [datetime] NULL CONSTRAINT
[DF_Customers_DateAdded] DEFAULT (getdate()),
```

Constraints are used to not only insert default values, but also validate data as well as primary keys. However, when using constraints within SQL Server, you do have to look at the whole picture, which is the user graphical system with the SQL Server database in the background. If you are using a constraint for data validation, some people will argue that perhaps it is better to check the values inserted within the user front-end application

rather than in SQL Server. This has some merit, but what also has to be kept in mind is that you may have several points of entry to your database. This could be from the user application, a web-based solution, or other applications if you are building a central database. Many people will say that all validation, no matter what the overall picture is, should always be placed in one central place, which is the SQL Server database. Then there is only one set of code to alter if anything changes. It is a difficult choice and one that you need to look at carefully.

Practice: Add constraint

1. Ensure that Query Editor is running. Although all the examples deal with the CustomerDetails.CustomerProducts table, each constraint being added to the table will be created one at a time, which will allow a discussion for each point to take place. In the Query Editor pane, enter the following code, which will add a primary key to the CustomerProducts table. This will place the CustomerFinancialProductId column within the key, which will be clustered.

```
2.  
USE ApressFinancial  
GO  
ALTER TABLE CustomerDetails.CustomerProducts  
ADD CONSTRAINT PK_CustomerProducts  
PRIMARY KEY CLUSTERED  
(CustomerFinancialProductId) ON [PRIMARY]  
GO
```

2. Next we add a CHECK constraint on the AmountToCollect column. The CustomerDetails.CustomerProducts table is once again altered, and a new constraint added called CK_CustProds_AmtCheck. This constraint will ensure that for all rows inserted into the CustomerDetails.CustomerProducts table from this point on, the score must be greater than 0. Notice as well that the NOCHECK option is mentioned, detailing that any rows already inserted will not be checked for this constraint. If they have invalid data, which they don't, then the constraint would ignore them and still be added.

```
ALTER TABLE CustomerDetails.CustomerProducts  
WITH NOCHECK  
ADD CONSTRAINT CK_CustProds_AmtCheck  
CHECK ((AmountToCollect > 0))  
GO
```

3. Moving on to the third constraint to add to the CustomerDetails.CustomerProducts table, we have a DEFAULT value constraint. In other words, this will insert a value of 0 to the Renewable column if no value is entered specifically into this column. This signifies that the premium collected is a one-off collection.

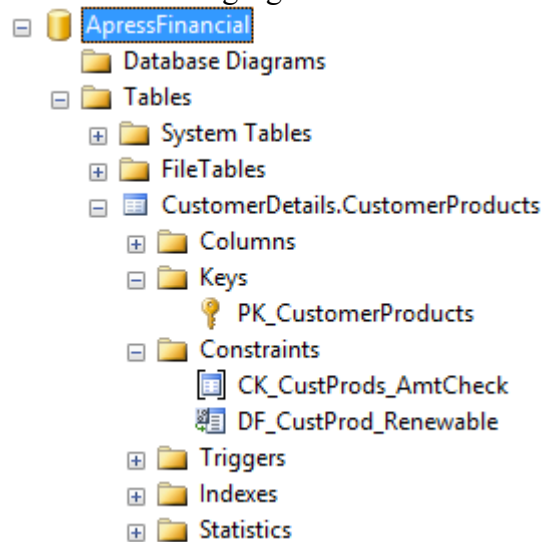
```
ALTER TABLE CustomerDetails.CustomerProducts WITH NOCHECK  
ADD CONSTRAINT DF_CustProd_Renewable  
DEFAULT (0)
```


FOR Renewable

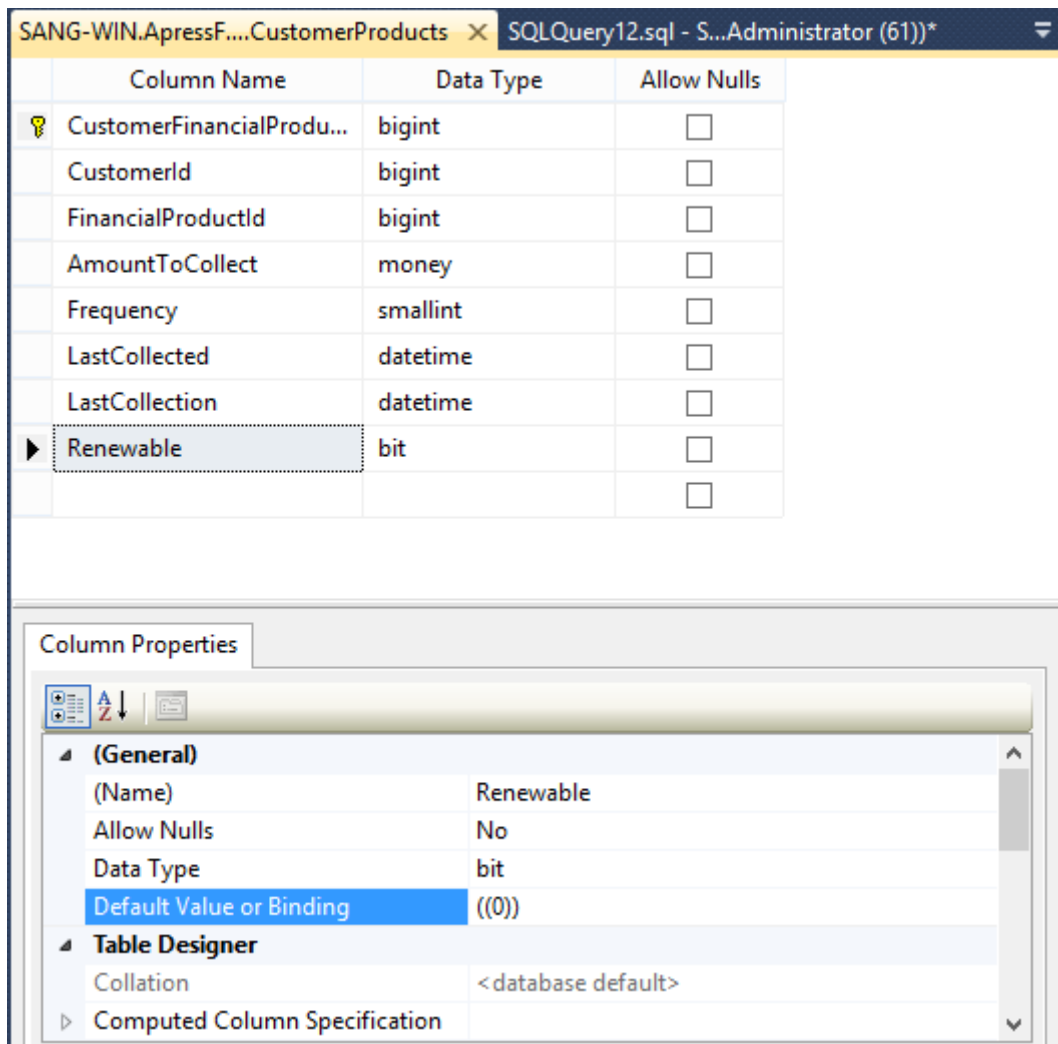
4. Execute the three batches of work by pressing F5 or Ctrl+E, or clicking the Execute button on the toolbar. You should then see the following result:

The statement(s) completed successfully.

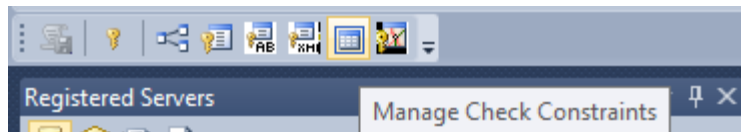
5. There are two methods to check that the code has worked before adding in any data. Move to Object Explorer in Query Editor. This isn't refreshed automatically, so you do need to refresh it. You should then see the three new constraints added, two under the Constraints node and one under the Keys node, as well as a display change in the Columns node, as shown in the following figure.



6. Another method is to move to SQL Server, find the CustomerDetails.CustomerProducts table, right-click it, and select Design. This brings us into the Table Designer, where we can navigate to the necessary column to check out the default value, in this case Renewable. Also notice the yellow key against the CustomerFinancialProductId signifying that this is now a primary key, as shown in the following figure.



7. Move to the Table Designer toolbar and click the Manage Check Constraints button, shown here:



8. This will display the Check Constraints dialog box, shown in the following figure, where we will see the AmountToCollect column constraint displayed. We can add a further constraint by clicking the Add button. Do so now.

Check Constraints

Selected Check Constraint:

CK_CustProds_AmtCheck

Editing properties for existing check constraint.

(General)

Expression	([AmountToCollect]>(0))
------------	-------------------------

Identity

(Name)	CK_CustProds_AmtCheck
Description	

Table Designer

Check Existing Data On Creation	No
Enforce For INSERTs And UPDATEs	Yes
Enforce For Replication	Yes

Add Delete Close

9. This will alter the Check Constraints dialog box to allow a new check constraint to be added, as you see in the following figure. This check will ensure that the LastCollection date is greater than the value entered in another column. Here we want to ensure that the LastCollection date is equal to or after the LastCollected date. Recall that LastCollection defines when we last took the payment, and LastCollected defines when the last payment should be taken.

Check Constraints

Selected Check Constraint:

CK_CustomerProducts*
CK_CustProds_AmtCheck

Editing properties for new check constraint. The 'Expression' property needs to be filled in before the new check constraint will be accepted.

(General)

Expression	
------------	--

Identity

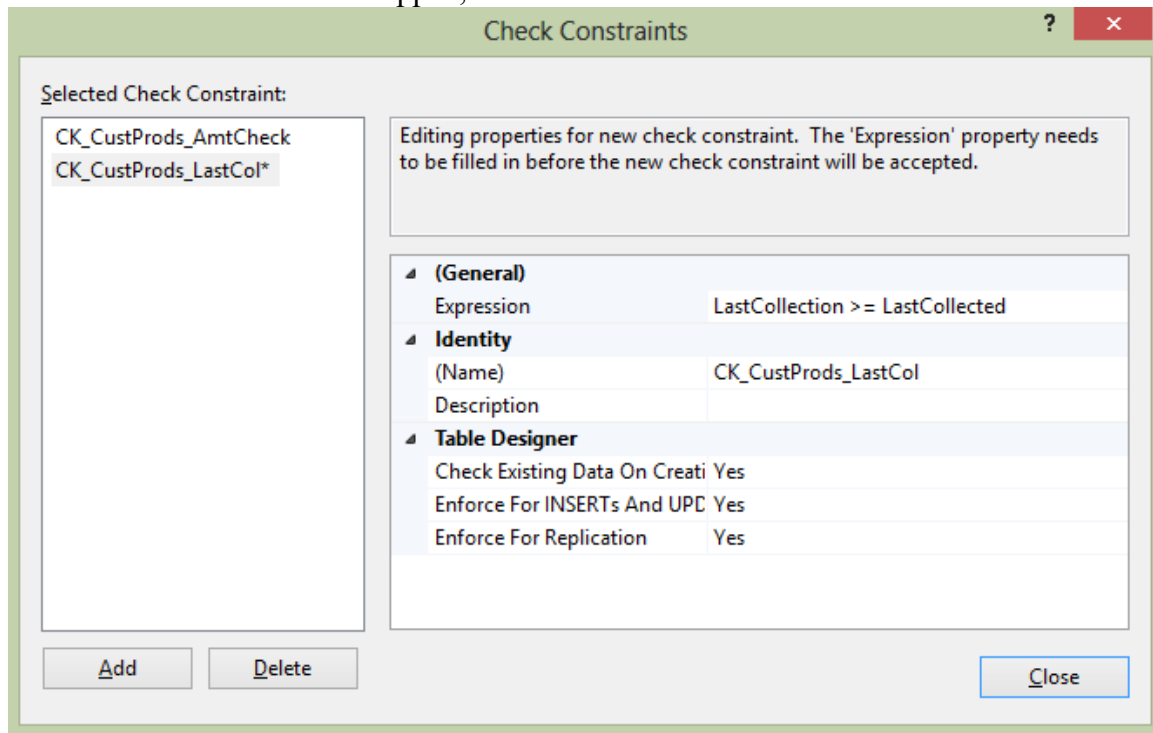
(Name)	CK_CustomerProducts
Description	

Table Designer

Check Existing Data On Creation	Yes
Enforce For INSERTs And UPDATEs	Yes
Enforce For Replication	Yes

Add Delete Close

10. The expression we want to add, which is the test the constraint is to perform, is not a value nor a system function like GETDATE(), but a test between two columns from a table, albeit the same table we are working with. This is as simple as naming the columns and the test you wish to perform. Also at the same time, change the name of the constraint to something meaningful. Your check constraint should look something like what appears in the following figure. Afterwards, click Close, which will add the constraint to the list, although it has not yet been added to the table. It is not until the table is closed that this will happen, so do that now.



11. Now it's time to test the constraints to ensure that they work. First of all, we want to check the AmountToCollect constraint. Enter the following code, which will fail as the amount to collect is a negative amount.

```
INSERT INTO CustomerDetails.CustomerProducts
(CustomerFinancialProductId, CustomerId, FinancialProductId
, AmountToCollect, Frequency,
LastCollected, LastCollection, Renewable)
VALUES (1,1,1,-100,0, '24 Aug 2005', '24 Aug 2005',0)
```

12. When you execute the code in Query Editor, you will see the following result. Instantly you can see that the constraint check (CK_CustProds_AmtCheck) has cut in and the row has not been inserted.

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint
```

```
"CK_CustProds_AmtCheck". The conflict occurred in
database
"ApressFinancial", table
"CustomerDetails.CustomerProducts",
column 'AmountToCollect'.
The statement has been terminated.
```

13. We alter this now to have a positive amount, but change the LastCollection so that we break the CK_CustProd_LastColl constraint. Enter the following code:

```
INSERT INTO CustomerDetails.CustomerProducts
(CustomerFinancialProductId, CustomerId, FinancialProductId
, AmountToCollect, Frequency,
LastCollected, LastCollection)
VALUES (1, 1, 1, 100, 0, '24 Aug 2005', '23 Aug 2005')
```

14. When the preceding code is executed, you will see the following error message:

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint
"CK_CustProd_LastColl". The conflict occurred in database
"ApressFinancial", table
"CustomerDetails.CustomerProducts".
The statement has been terminated.
```

Dealing with Several Rows at Once

In Query Editor, you can input a sequence of SQL commands and execute them as a batch. Notice that between these commands, you have to place command GO. Each command will be treated as a single unit of work, which either completes or fails.

Practice: Insert Several Records at Once

1. Ensure that SQL Server Query Editor is up and running. In the Query Editor window, enter the following code. In this example, several customers will be added through only one INSERT statement.

```
INSERT INTO CustomerDetails.Customers
(CustomerTitleId, CustomerFirstName, CustomerOtherInitials,
CustomerLastName, AddressId, AccountNumber, AccountTypeId,
ClearedBalance, UnclearedBalance)
VALUES
(3, 'Bernie', 'I', 'McGee', 314, 65368765, 1, 6653.11, 0.00),
(2, 'Julie', 'A', 'Dewson', 2134, 81625422, 1, 53.32, -12.21),
(1, 'Kirsty', NULL, 'Hull', 4312, 96565334, 1, 1266.00, 10.32)
```

2. Now just execute the code in the usual way. You will see the following output in the results pane. This indicates that three rows of information have been inserted into the database, one at a time.

(3 row(s) affected)

Practice: Retrieve Data in SQL Server environment

1. Ensure that SQL Server environment is running. Navigate to the ApressFinancial database and click the Tables node; this should then list all the tables in the right-hand pane. Find the CustomerDetails.Customers table, right-click it to bring up the pop-up menu you have seen a number of times before, and select Select Top 1000 Rows. This instantly opens up a new Query Editor pane like the one in the following figure, which shows all the rows that are in the CustomerDetails.Customers table. But how did SQL Server get this data? Let's find out.

The screenshot shows the SQL Server Enterprise Manager interface. The top pane displays a query in the Query Editor:

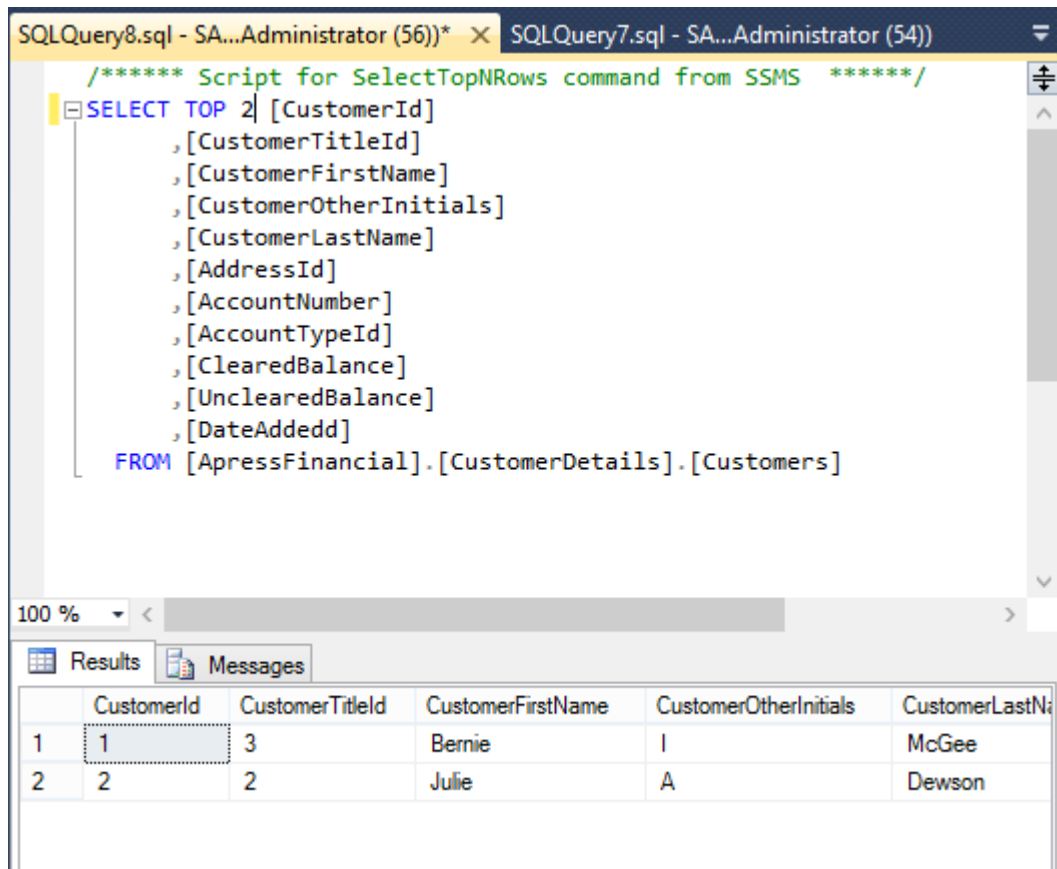
```
/****** Script for SelectTopNRows command from SSMS *****/  
SELECT TOP 1000 [CustomerId]  
    , [CustomerTitleId]  
    , [CustomerFirstName]  
    , [CustomerOtherInitials]  
    , [CustomerLastName]  
    , [AddressId]  
    , [AccountNumber]  
    , [AccountTypeId]  
    , [ClearedBalance]  
    , [UnclearedBalance]  
    , [DateAddedd]  
FROM [ApressFinancial].[CustomerDetails].[Customers]
```

The bottom pane shows the Results tab with a table of data:

	CustomerId	CustomerTitleId	CustomerFirstName	CustomerOtherInitials	CustomerLastName
1	1	3	Bernie	I	McGee
2	2	2	Julie	A	Dewson
3	3	1	Kirsty	NULL	Hull

2. Above the results, you will see the SELECT T-SQL statement used to return the data. The SELECT statement is returning the top 1,000 rows and then defining the columns to display.

3. You can alter the number next to the top clause if you want to return a smaller or a greater number of rows, and then re-execute the query. For this first time, alter this to 2, and you should see something similar to the following figure. This will return a maximum of two rows.



The SELECT Statement

Simple syntax for a SELECT statement:

```
SELECT [ ALL | DISTINCT ]
[ TOP expression [ PERCENT ] [ WITH TIES ] ]
{
  *
  | { table_name | view_name | alias_name }. *
  | { column_name | [ ] expression | $IDENTITY |
  $ROWGUID }
  [ [ AS ] column_alias ]
  | column_alias = expression
} [ ,...n ]
FROM table_name | view_name alias_name
WHERE filter_criteria
ORDER BY ordering_criteria
```

The following list breaks down the SELECT syntax, explaining each option.

SELECT: Required—this informs SQL Server that a SELECT instruction is being performed; in other words, we just want to return a set of columns and rows to view.

ALL | DISTINCT: Optional—we want to return either all of the rows or only distinct, or unique, rows. Normally, you do not specify either of these options.

TOP expression/PERCENT/WITH TIES: Optional—you can return the top number of rows as defined by either the order of the data in the clustered index or, if the result is ordered by an

ORDER BY clause, the top number from that order sequence. If there is no clustered index or no ordering, the rows will be returned in an arbitrary order. You can also add the word PERCENT to the end: this will mean that the top n percent of rows will be returned. If PERCENT is not specified, all the rows will be returned (unless specific column names are given). WITH TIES can only be used with an ORDER BY. If you specify you want to return TOP 10 rows, and the 11th row has the same value as the 10th row on those columns that have been defined in the ORDER BY, then the 11th row will also be returned. Same for subsequent rows, until you get to the point that the values differ.

*****: Optional—by using the asterisk, you are instructing SSE to return all the columns from all the tables included in the query. This is not an option that should be used on large amounts of data or over a network, especially if it is busy. By using this, we are bringing back more information than is required. Wherever possible we should name the columns instead.

table_name.* | view_name.* | alias_name.*: Optional—similar to *, but you are defining which table, if the SELECT is working on more than one table. When working with more than one table, this is known as a JOIN.

column_name: Optional but recommended; not required if * is used—this option is where we name the columns that we wish to return from a table. When naming the columns, it is always a good idea to prefix the column names with their corresponding table name. This becomes mandatory when we are using more than one table in our SELECT statement and instances where there may be columns within different tables that share the same name.

expression: Optional—we don't have to return columns of rows within a SELECT. We can return a value, a variable, or an expression.

\$IDENTITY: Optional—will return the value from the IDENTITY column.

\$ROWGUID: Optional—will return the value from the ROWGUID column.

AS: Optional—we can change the column header name when displaying the results by using the AS option.

FROM table_name | view_name: Required—we have to inform SSE where the information is coming from.

WHERE filter_clause: Optional—if we want to retrieve rows that meet specific criteria, we need to have a WHERE clause specifying the criteria to use to return the data. The WHERE clause tends to contain the name of a column on the left-hand side of a comparison operator (like =, <, or >) and either another column within the same table, or another table, a variable, or a static value. There are other options that the WHERE statement can contain, where more advanced searching is required, but on the whole these comparison operators will be the main constituents of the clause.

ORDER BY ordering_criteria: Optional—the data will be returned arbitrarily from the table if no ORDER BY clause is specified, which, if you have a clustered index built on the table, will be in that order; otherwise, it will be in the order in which they were inserted. However, you can alter the ordering by using the ORDER BY clause, which will determine the order of the rows returned, and you can specify whether each column is returned in ascending or descending order. Ascending, ASC, or descending, DESC, is defined for each column, not defined just once for all the columns within the ORDER BY. Sorting is completed once the data has been retrieved from SQL Server but before any statement like TOP.

Practice: The First Set of Searches

1. Ensure that Query Editor is running and that you are within the ApressFinancial database. In the Query Editor pane, enter the following SQL code:

```
SELECT * FROM CustomerDetails.Customers
```

2. Execute the code using Ctrl+E, F5, or the Execute button on the toolbar. You should then see something like the results shown in the following figure.

Results		Messages			
	CustomerId	CustomerTitleId	CustomerFirstName	CustomerOtherInitials	CustomerLastName
1	1	3	Bernie	I	McGee
2	2	2	Julie	A	Dewson
3	3	1	Kirsty	NULL	Hull

3. This is a simple SELECT statement returning all the columns and all the rows from the CustomerDetails.Customers table. Let's now take it to the next stage where specific column names will be defined in the query, which is a much cleaner solution. In this instance from the CustomerDetails.Customers table, we would like to return a customer's first name, last name, and current account balances. This would mean naming CustomerFirstName, CustomerLastName, and ClearedBalance as the column names in the query. The code will read as follows:

```
SELECT CustomerFirstName, CustomerLastName, ClearedBalance
FROM CustomerDetails.Customers
```

4. Now execute this code, which will return the results shown in the following figure. As you can see, not every column is returned.

	CustomerFirstName	CustomerLastName	ClearedBalance
1	Bernie	McGee	6653.11
2	Julie	Dewson	53.32
3	Kirsty	Hull	1266.00

5. As you have seen from the examples so far, the column names, although well named from a design viewpoint, are not exactly suitable if we had to give this to a set of users. Using the same query as before, a couple of minor modifications are required to give the columns aliases. The first alias name is in quotes as it contains a space. Notice the last column also does not have AS specified because this keyword is optional.

```
SELECT CustomerFirstName As 'First Name',
CustomerLastName AS 'Surname',
ClearedBalance Balance
FROM CustomerDetails.Customers
```

6. Execute this and the displayed output changes—much more friendly column names, as you see in the following figure.

	First Name	Surname	Balance
1	Bernie	McGee	6653.11
2	Julie	Dewson	53.32
3	Kirsty	Hull	1266.00

Practice: Limiting the Search: The Use of WHERE

1. First of all to use a different table, let's enter some more rows in to the ShareDetails.Shares table. Enter and execute the following code:

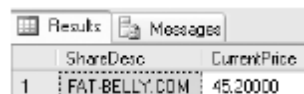
```
INSERT INTO ShareDetails.Shares
(ShareDesc, ShareTickerId, CurrentPrice)
VALUES ('FAT-BELLY.COM', 'FBC', 45.20)
INSERT INTO ShareDetails.Shares
(ShareDesc, ShareTickerId, CurrentPrice)
VALUES ('NetRadio Inc', 'NRI', 29.79)
INSERT INTO ShareDetails.Shares
(ShareDesc, ShareTickerId, CurrentPrice)
VALUES ('Texas Oil Industries', 'TOI', 0.455)
```

```
INSERT INTO ShareDetails.Shares
(ShareDesc, ShareTickerId,CurrentPrice)
VALUES ('London Bridge Club','LBC',1.46)
```

2. The requirement for this section is to find the current share price for FAT-BELLY.COM. We restrict the SELECT statement so that only the specific row comes back by using the WHERE statement, as can be seen in the following code:

```
SELECT ShareDesc,CurrentPrice
FROM ShareDetails.Shares
WHERE ShareDesc = 'FAT-BELLY.COM'
```

3. Execute this code, and you will see that the single row for FAT-BELLY.COM is returned, as shown in the following figure.



	ShareDesc	CurrentPrice
1	FAT-BELLY.COM	45.20000

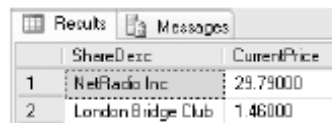
4. To prove that we are working within an installation that is not case sensitive from a data perspective (unless you installed a different collation sequence to that described in Chapter 1), if you perform the following query, you will get the same results as displayed in the previous figure.

```
SELECT ShareDesc,CurrentPrice
FROM ShareDetails.Shares
WHERE ShareDesc = 'FAT-BELLY.COM'
```

5. You have seen the WHERE in action using the equals sign; it is also possible to use the other relational operations in the WHERE statement. The next query demonstrates how SSE takes the WHERE condition and starts returning rows after the given point. This query provides an interesting set of results. Enter the code as detailed here:

```
SELECT ShareDesc,CurrentPrice
FROM ShareDetails.Shares
WHERE ShareDesc > 'FAT-BELLY.COM'
AND ShareDesc < 'TEXAS OIL INDUSTRIES'
```

6. Once done, execute the code and check the results, which should resemble the following figure.



	ShareDesc	CurrentPrice
1	NetRadio Inc	29.79000
2	London Bridge Club	1.46000

7. Let's now bring in another option in the WHERE statement that allows us to avoid returning specific rows. This can be achieved in one of two ways: the first is by using the less than and greater than signs; the second is by using the NOT operator. Enter the

following code, which will return all rows except FAT-BELLY.COM. Run both sets of code at once. This will return two sets of output, known as multiple result sets.

```
SELECT ShareDesc,CurrentPrice
FROM ShareDetails.Shares
WHERE ShareDesc <> 'FAT-BELLY.COM'
SELECT ShareDesc,CurrentPrice
FROM ShareDetails.Shares
WHERE NOT ShareDesc = 'FAT-BELLY.COM'
```

8. Executing this code will produce the output shown in the following figure. Notice how in neither set of output FAT-BELLY.COM has been listed.

Results			Messages	
	ShareDesc	CurrentPrice		
1	ACME'S HOMEBAKE COOKIES INC	2.34125		
2	ACME'S HOMEBAKE COOKIES INC	2.34125		
3	NetRadio Inc	29.79000		
4	Texas Oil Industries	0.45500		
5	London Bridge Club	1.46000		
	ShareDesc	CurrentPrice		
1	ACME'S HOMEBAKE COOKIES INC	2.34125		
2	ACME'S HOMEBAKE COOKIES INC	2.34125		
3	NetRadio Inc	29.79000		
4	Texas Oil Industries	0.45500		
5	London Bridge Club	1.46000		

Creating Data: SELECT INTO

You can use the result of a SELECT command to create a new table. The syntax is as follow:

```
SELECT *|column1,column2,...
INTO new_tablename
FROM tablename
```

End of Lab 2