Høgskolen i Telemark

Telemark University College
Department of Electrical Engineering, Information Technology and Cybernetics

Tutorial

Structured Query Language

HANS-PETTER HALVORSEN, 2012.04.13

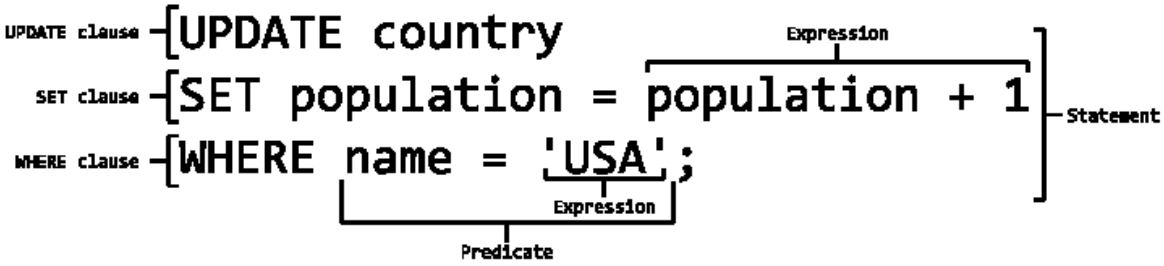


Table of Contents

Introduction to SQL

SQL (Structured Query Language) is a database computer language designed for managing data in relational database management systems (RDBMS).

SQL, is a standardized computer language that was originally developed by IBM for querying, altering and defining relational databases, using declarative statements.

SQL is pronounced /ˌɛs kjuː ˈɛl/ (letter by letter) or /ˈsiːkwəl/ (as a word).

SQL – Structured Query language

A Database Computer Language designed for Managing Data in Relational Database Management Systems (RDBMS)

Query Examples:

```
• insert into STUDENT (Name , Number, SchoolId)
  values ('John Smith', '100005', 1)

• select SchoolId, Name from SCHOOL

• select * from SCHOOL where SchoolId > 100

• update STUDENT set Name='John Wayne' where StudentId=2

• delete from STUDENT where SchoolId=3
```

We have 4 different Query Types: **INSERT**, **SELECT**, **UPDATE** and **DELETE**

What can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database

- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

Even if SQL is a standard, many of the database systems that exist today implement their own version of the SQL language. In this document we will use the Microsoft SQL Server as an example.

There are lots of different database systems, or DBMS – Database Management Systems, such as:

- **Microsoft SQL Server**
  - Enterprise, Developer versions, etc.
  - Express version is free of charge
- **Oracle**
- **MySQL** (Oracle, previously Sun Microsystems) - MySQL can be used free of charge (open source license), Web sites that use MySQL: YouTube, Wikipedia, Facebook
- **Microsoft Access**
- **IBM DB2**
- **Sybase**
- … lots of other systems

In this Tutorial we will focus on Microsoft SQL Server. SQL Server uses T-SQL (Transact-SQL). T-SQL is Microsoft's proprietary extension to SQL. T-SQL is very similar to standard SQL, but in addition it supports some extra functionality, built-in functions, etc.

Other useful Tutorials about databases:

- **Introduction to Database Systems**
- **Database Communication in LabVIEW**

These Tutorials are located at: **http://home.hit.no/~hansha**

Data Definition Language (DDL)

The **Data Definition Language (DDL)** manages table and index structure. The most basic items of DDL are the CREATE, ALTER, RENAME and DROP statements:

- **CREATE** creates an object (a table, for example) in the database.
- **DROP** deletes an object in the database, usually irretrievably.

- **ALTER** modifies the structure an existing object in various ways—for example, adding a column to an existing table.

Data Manipulation Language (DML)

The **Data Manipulation Language (DML)** is the subset of SQL used to add, update and delete data.

The acronym **CRUD** refers to all of the major functions that need to be implemented in a relational database application to consider it complete. Each letter in the acronym can be mapped to a standard SQL statement:

Operation	SQL	Description
Create	INSERT INTO	inserts new data into a database
Read (Retrieve)	SELECT	extracts data from a database
Update	UPDATE	updates data in a database
Delete (Destroy)	DELETE	deletes data from a database

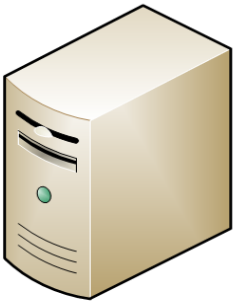
Introduction to SQL Server

Microsoft is the vendor of SQL Server. The newest version is “SQL Server 2012”.

We have different editions of SQL Server, where SQL Server Express is free to download and use.

SQL Server uses T-SQL (Transact-SQL). T-SQL is Microsoft's proprietary extension to SQL. T-SQL is very similar to standard SQL, but in addition it supports some extra functionality, built-in functions, etc. T-SQL expands on the SQL standard to include procedural programming, local variables, various support functions for string processing, date processing, mathematics, etc.

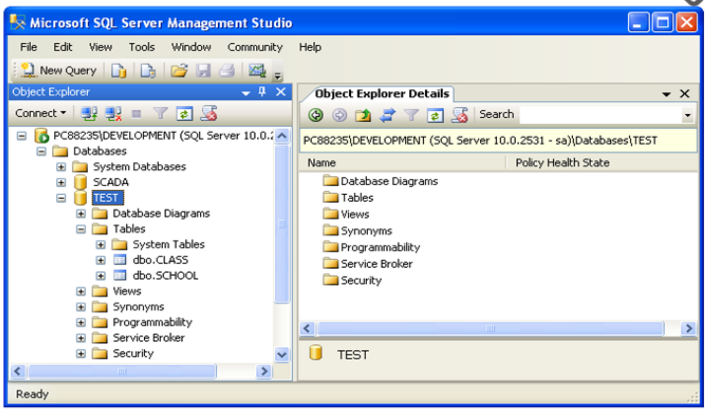
SQL Server consists of a **Database Engine** and a **Management Studio** (and lots of other stuff which we will not mention here). The Database engine has no graphical interface - it is just a service running in the background of your computer (preferable on the server). The Management Studio is graphical tool for configuring and viewing the information in the database. It can be installed on the server or on the client (or both).



Database Engine

A Service running on the computer in the background

Management Studio

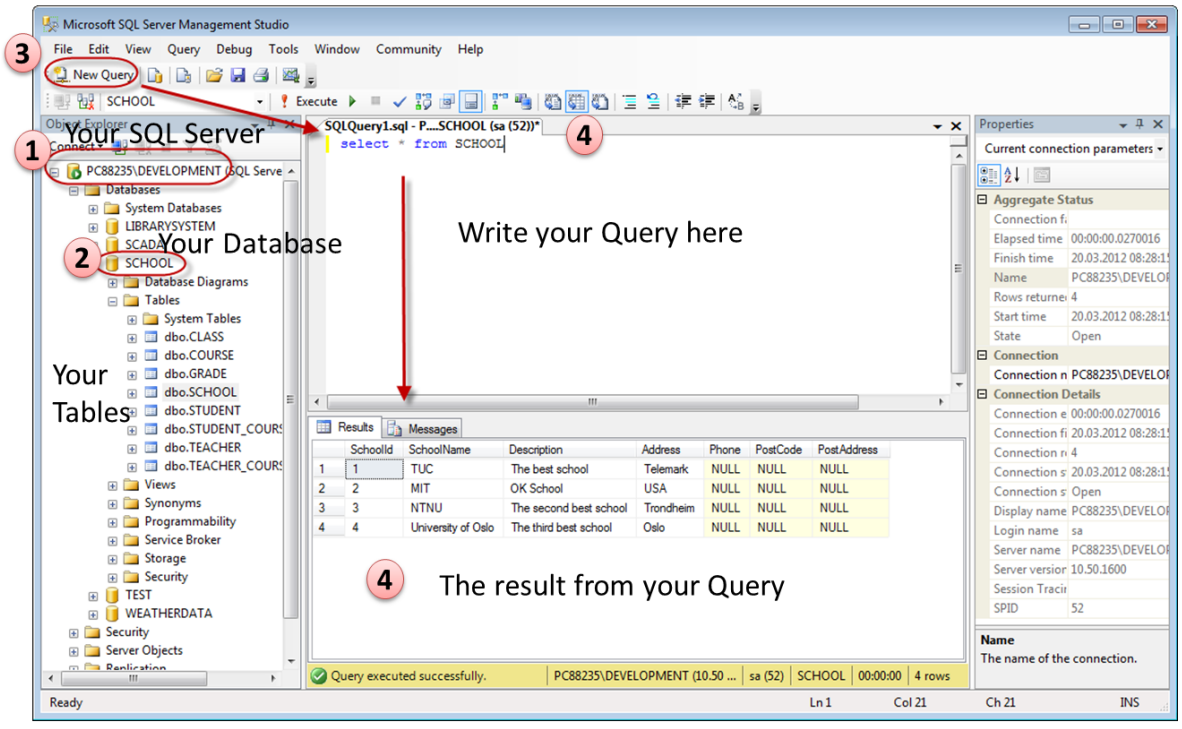


A Graphical User Interface to the database used for configuration and management of the database

SQL Server Management Studio

SQL Server Management Studio is a GUI tool included with SQL Server for configuring, managing, and administering all components within Microsoft SQL Server. The tool includes both script editors and graphical tools that work with objects and features of the server. As mentioned earlier, version of SQL Server Management Studio is also available for SQL Server Express Edition, for which it is known as SQL Server Management Studio Express.

A central feature of SQL Server Management Studio is the Object Explorer, which allows the user to browse, select, and act upon any of the objects within the server. It can be used to visually observe and analyze query plans and optimize the database performance, among others. SQL Server Management Studio can also be used to create a new database, alter any existing database schema by adding or modifying tables and indexes, or analyze performance. It includes the query windows which provide a GUI based interface to write and execute queries.

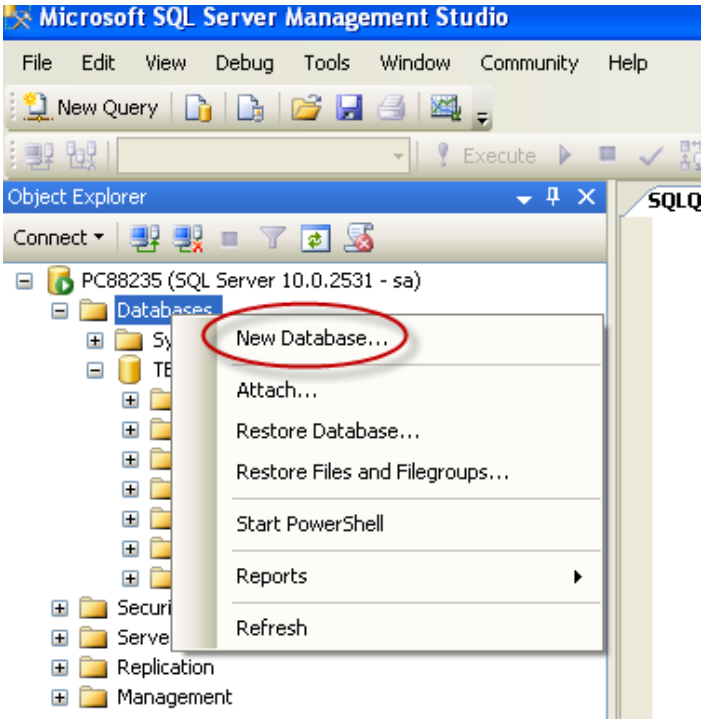


When creating SQL commands and queries, the “Query Editor” (select “New Query” from the Toolbar) is used (shown in the figure above).

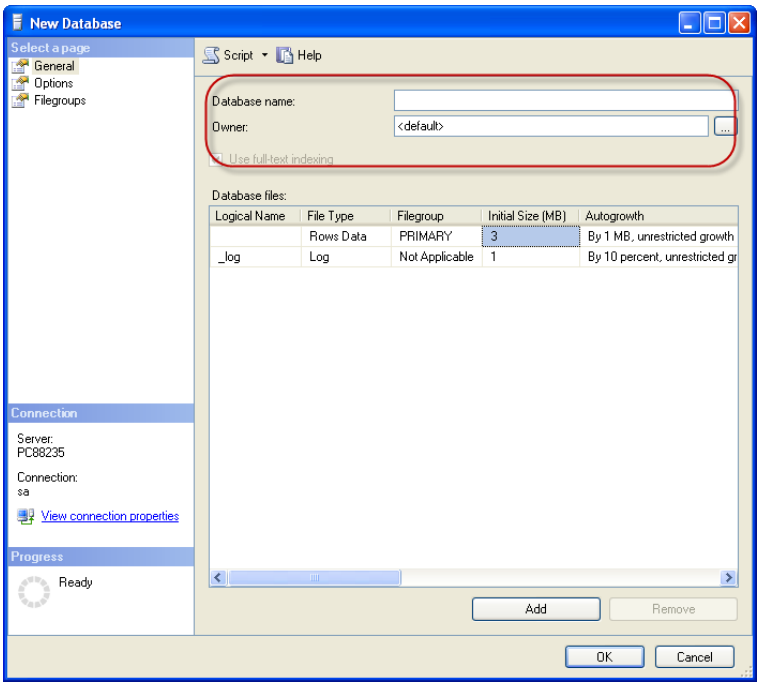
With SQL and the “Query Editor” we can do almost everything with code, but sometimes it is also a good idea to use the different Designer tools in SQL to help us do the work without coding (so much).

Create a new Database

It is quite simple to create a new database in Microsoft SQL Server. Just right-click on the "Databases" node and select "New Database…"



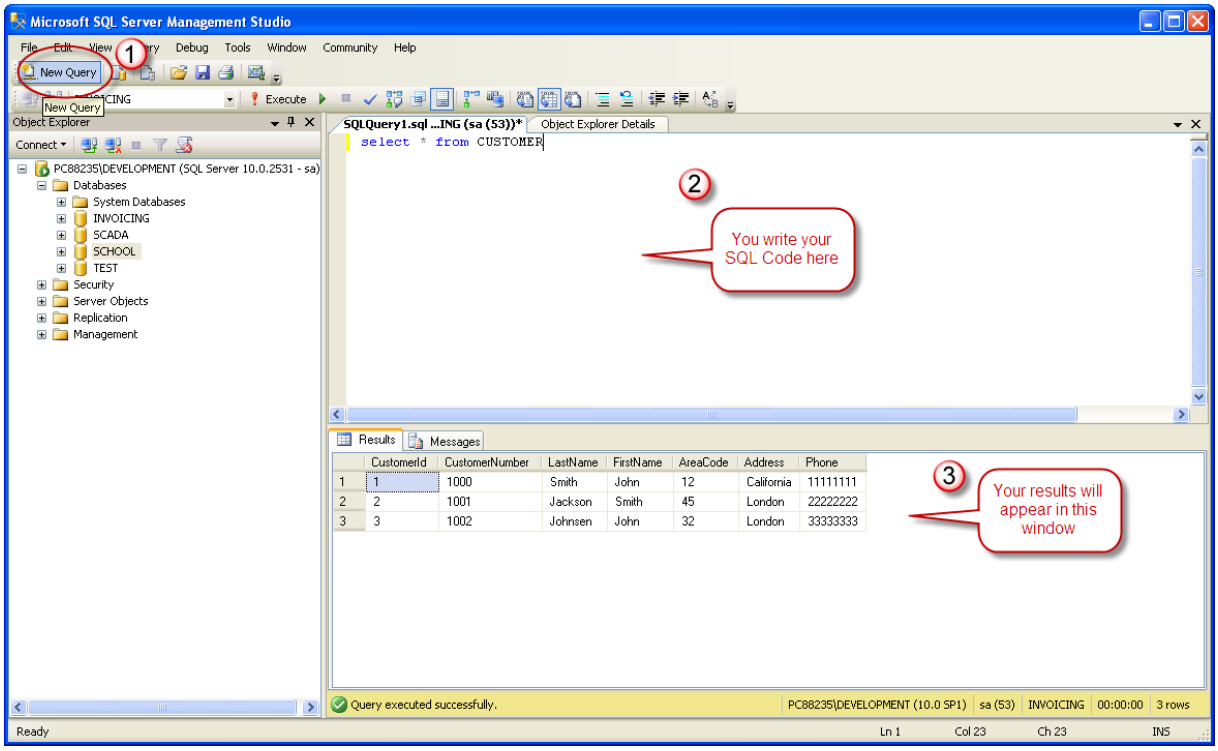
There are lots of settings you may set regarding your database, but the only information you must fill in is the name of your database:



You may also use the SQL language to create a new database, but sometimes it is easier to just use the built-in features in the Management Studio.

Queries

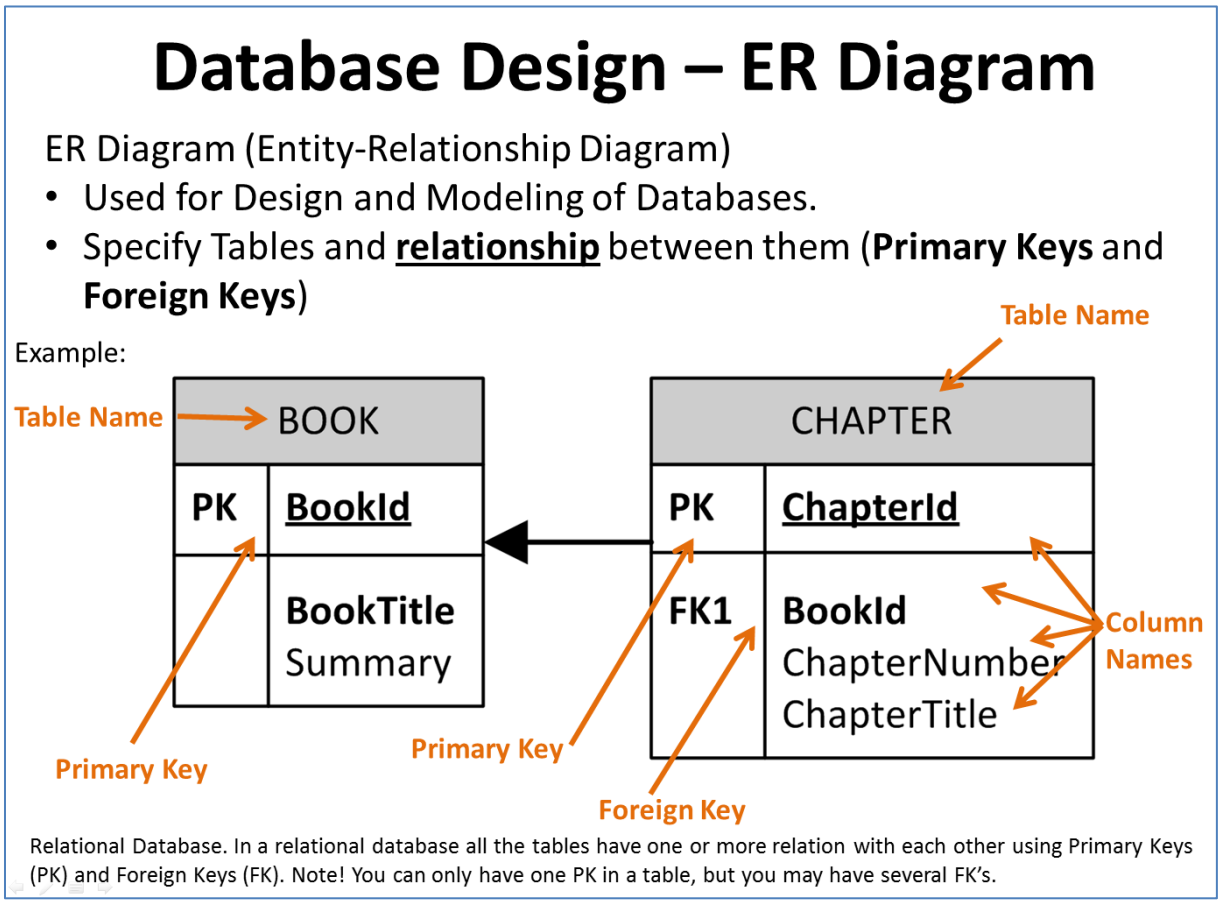
In order to make a new SQL query, select the “New Query” button from the Toolbar.



Here we can write any kind of queries that is supported by the SQL language.

CREATE TABLE

Before you start implementing your tables in the database, you should always spend some time design your tables properly using a design tool like, e.g., ERwin, Toad Data Modeler or Visio. This is called Database Modeling.



The **CREATE TABLE** statement is used to create a table in a database.

Syntax:

```
CREATE TABLE table_name
(
column_name1 data_type,
column_name2 data_type,
column_name3 data_type,
....
)
```

The data type specifies what type of data the column can hold.

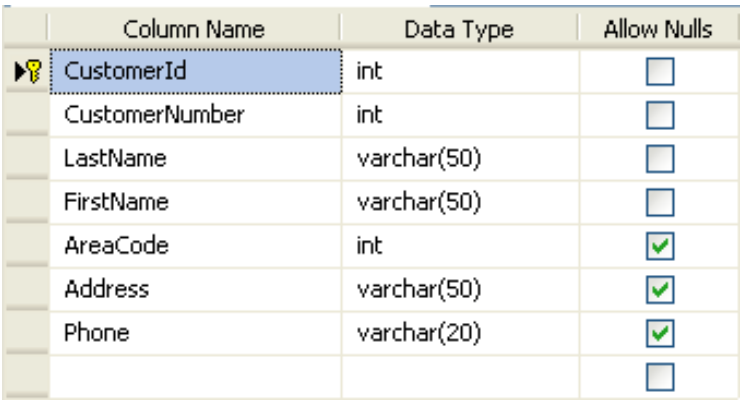
You have special data types for numbers, text dates, etc.

Examples:

- Numbers: **int**, **float**
- Text/Stings: **varchar(X)** – where X is the length of the string
- Dates: **datetime**
- etc.

Example:

We want to create a table called "CUSTOMER" which has the following columns and data types:



Column Name	Data Type	Allow Nulls
CustomerId	int	☐
CustomerNumber	int	☐
LastName	varchar(50)	☐
FirstName	varchar(50)	☐
AreaCode	int	☑
Address	varchar(50)	☑
Phone	varchar(20)	☑
		☐

```
CREATE TABLE CUSTOMER
(
      CustomerId int IDENTITY(1,1) PRIMARY KEY,
      CustomerNumber int NOT NULL UNIQUE,
      LastName varchar(50) NOT NULL,
      FirstName varchar(50) NOT NULL,
      AreaCode int NULL,
      Address varchar(50) NULL,
      Phone varchar(50) NULL,
)
GO
```

Best practice:

When creating tables you should consider following these guidelines:

- **Tables**: Use upper case and singular form in table names – not plural, e.g., "CUSTOMER". Never use spaces inside the table names.
- **Columns**: Use Pascal notation, e.g., "CustomerId". Never use spaces inside the column names.
- **Primary Keys**: Use Integer and **Identity(1,1)** for Primary Keys. Then the Primary Keys will just be a running number that is automatically inserted by the system. It is also a

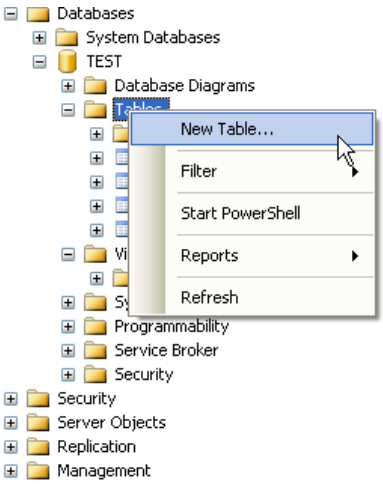
good idea to name the Primary Key column the same as the table name + Id, e.g., if the table name is “CUSTOMER” (upper case), then the Primary Key column should be named “CustomerId” (Pascal notation).

Create Tables using the Designer Tools

Even if you can do “everything” using the SQL language, it is sometimes easier to do it in the designer tools in the Management Studio in SQL Server.

Instead of creating a script you may as well easily use the designer for creating tables.

Step1: Select “New Table …”:



Step2: Next, the table designer pops up where you can add columns, data types, etc.

Column Name	Data Type	Allow Nulls
CustomerId	int	☐
CustomerNumber	int	☐
LastName	varchar(50)	☐
FirstName	varchar(50)	☐
AreaCode	int	☑
Address	varchar(50)	☑
Phone	varchar(20)	☑
		☐

In this designer we may also specify Column Names, Data Types, etc.

Step 3: Save the table by clicking the Save button.

SQL Constraints

Constraints are used to limit the type of data that can go into a table.

Constraints can be specified when a table is created (with the CREATE TABLE statement) or after the table is created (with the ALTER TABLE statement).

Here are the most important constraints:

- PRIMARY KEY
- NOT NULL
- UNIQUE
- FOREIGN KEY
- CHECK
- DEFAULT
- IDENTITY

In the sections below we will explain some of these in detail.

PRIMARY KEY

The PRIMARY KEY constraint uniquely identifies each record in a database table.

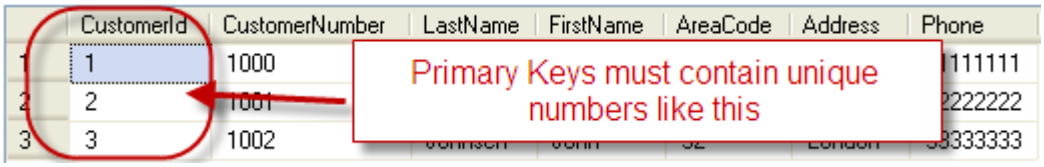
Primary keys must contain unique values. It is normal to just use running numbers, like 1, 2, 3, 4, 5, … as values in Primary Key column. It is a good idea to let the system handle this for you by specifying that the Primary Key should be set to **identity(1,1)**. IDENTITY(1,1) means the first value will be 1 and then it will increment by 1.

Each table should have a primary key, and each table can have only ONE primary key.

If we take a closer look at the CUSTOMER table created earlier:

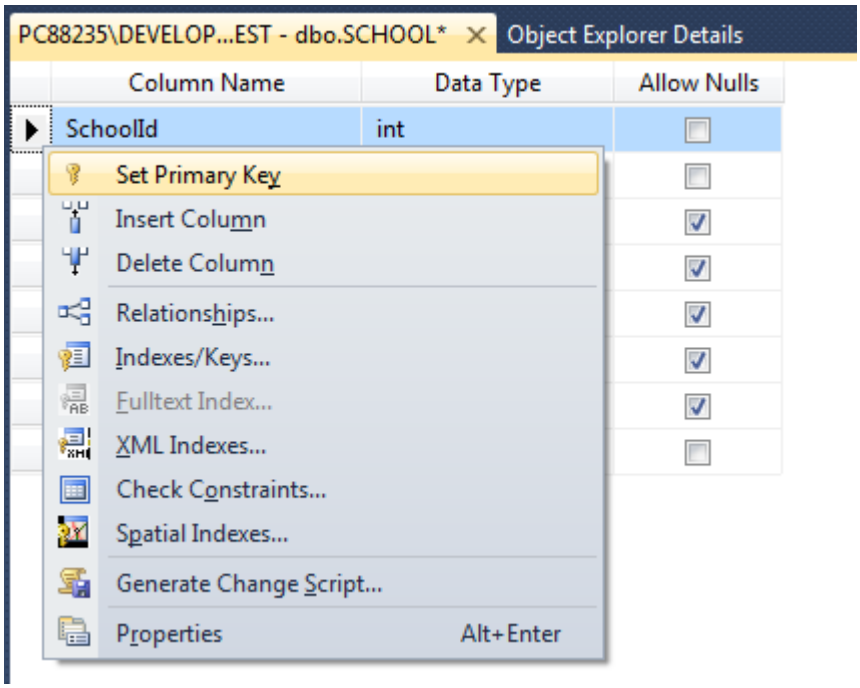
```
CREATE TABLE [CUSTOMER]
(
      CustomerId int IDENTITY(1,1) PRIMARY KEY,
      CustomerNumber int NOT NULL UNIQUE,
      LastName varchar(50) NOT NULL,
      FirstName varchar(50) NOT NULL,
      AreaCode int NULL,
      Address varchar(50) NULL,
      Phone varchar(50) NULL,
)
GO
```

As you see we use the “Primary Key” keyword to specify that a column should be the Primary Key.



Setting Primary Keys in the Designer Tools:

If you use the Designer tools in SQL Server you can easily set the primary Key in a table just by right-click and select “Set primary Key”.

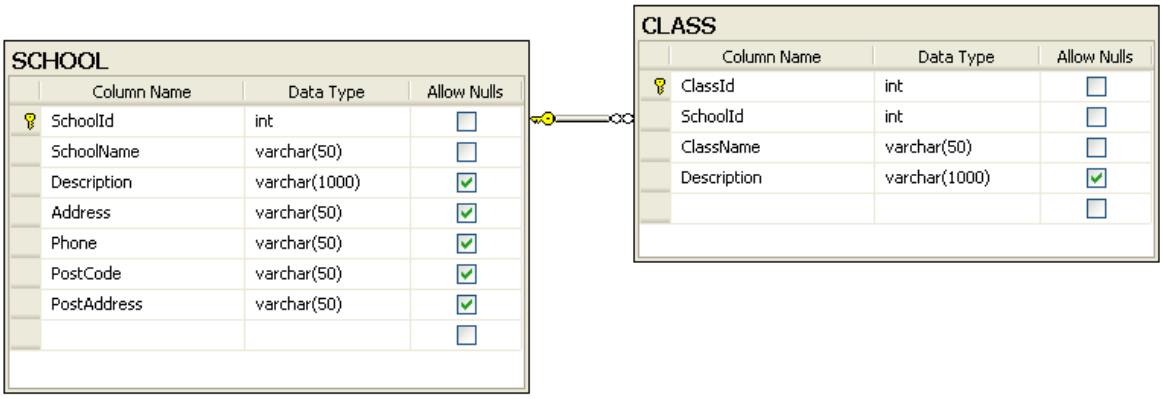


The primary Key column will then have a small key in front to illustrate that this column is a Primary Key.

FOREIGN KEY

A FOREIGN KEY in one table points to a PRIMARY KEY in another table.

Example:



We will create a CREATE TABLE script for these tables:

SCHOOL:

```
CREATE TABLE SCHOOL
(
      SchoolId int IDENTITY(1,1) PRIMARY KEY,
      SchoolName varchar(50) NOT NULL UNIQUE,
      Description varchar(1000) NULL,
      Address varchar(50) NULL,
      Phone varchar(50) NULL,
      PostCode varchar(50) NULL,
      PostAddress varchar(50) NULL,
)
GO
```

CLASS:

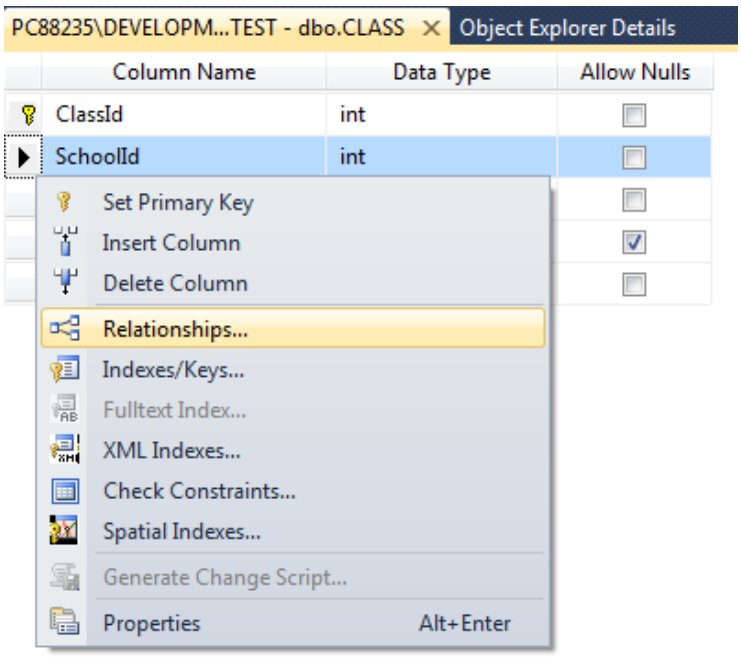
```
CREATE TABLE CLASS
(
      ClassId int IDENTITY(1,1) PRIMARY KEY,
      SchoolId int NOT NULL FOREIGN KEY REFERENCES SCHOOL (SchoolId),
      ClassName varchar(50) NOT NULL UNIQUE,
      Description varchar(1000) NULL,
)
GO
```

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

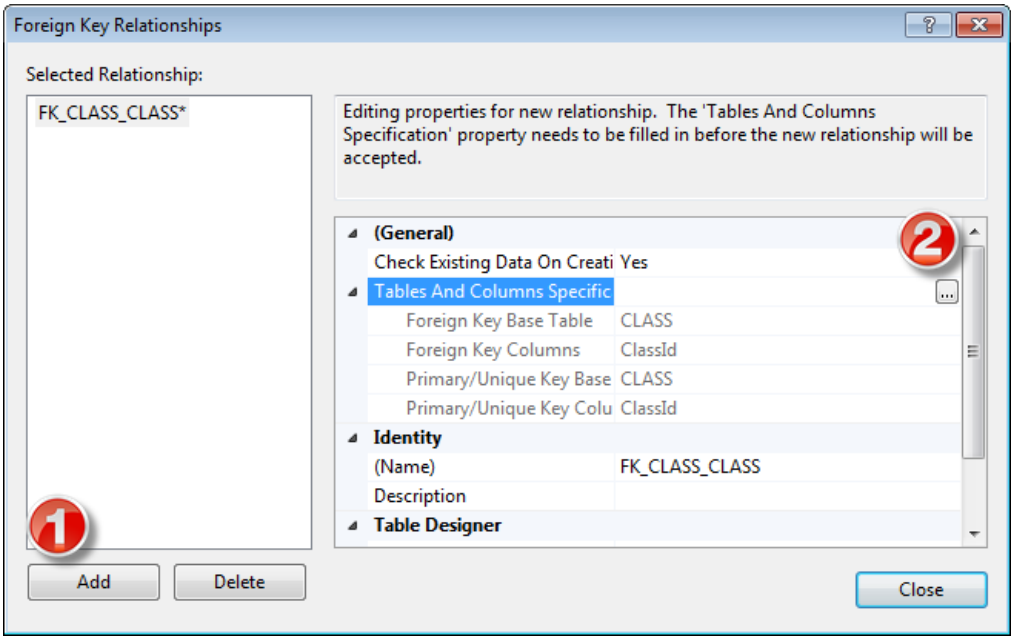
The FOREIGN KEY constraint also prevents that invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

Setting Foreign Keys in the Designer Tools:

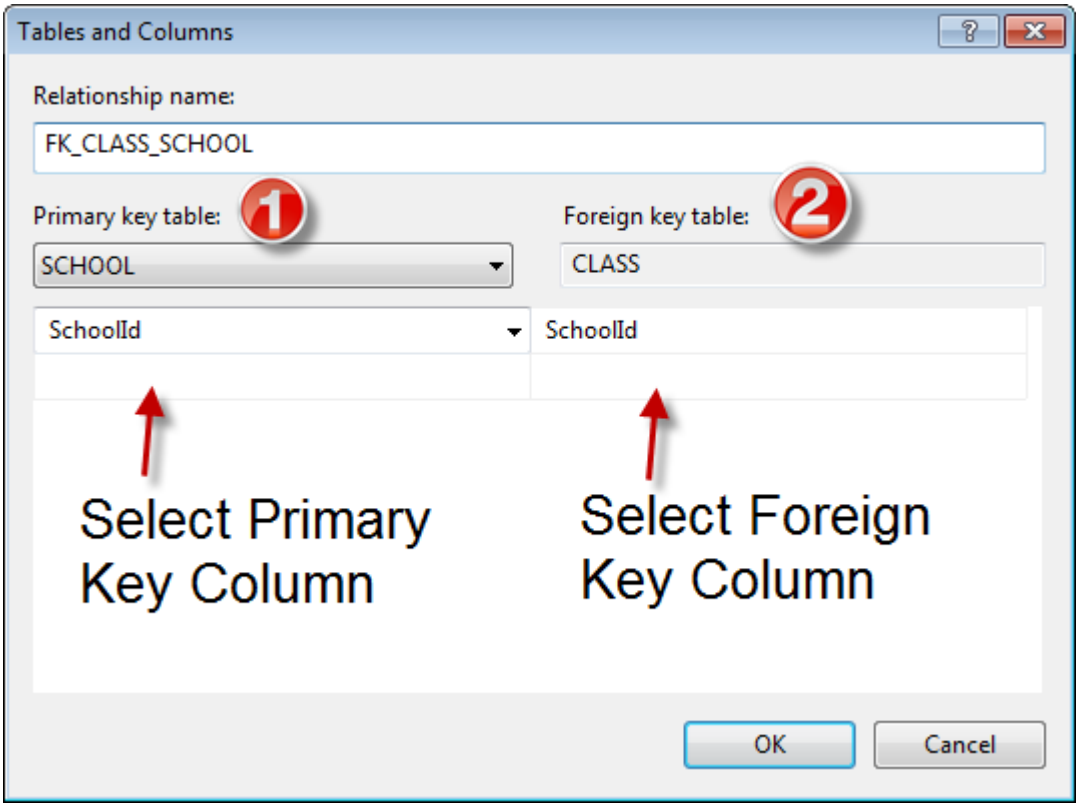
If you want to use the designer, right-click on the column that you want to be the Foreign Key and select "**Relationships…**":



The following window pops up (Foreign Key Relationships):



Click on the “Add” button and then click on the small “…” button. Then the following window pops up (Tables and Columns):



Here you specify the primary Key Column in the Primary Key table and the Foreign Key Column in the Foreign Key table.

NOT NULL / Required Columns

The NOT NULL constraint enforces a column to NOT accept NULL values.

The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to this field.

If we take a closer look at the CUSTOMER table created earlier:

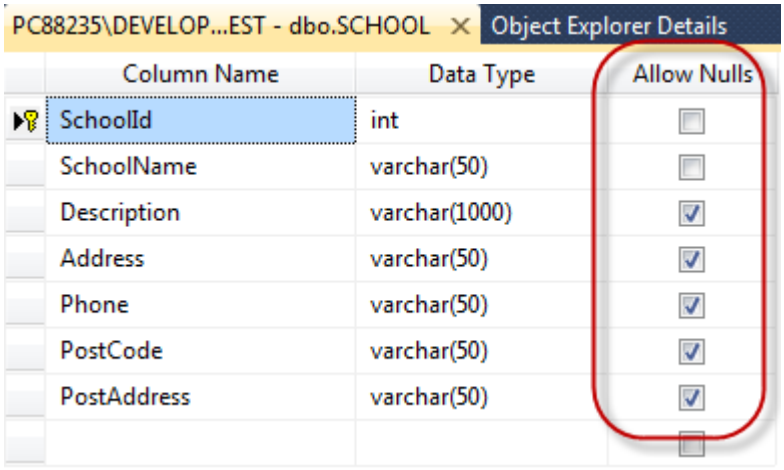
```
CREATE TABLE [CUSTOMER]
(
      CustomerId int IDENTITY(1,1) PRIMARY KEY,
      CustomerNumber int NOT NULL UNIQUE,
      LastName varchar(50) NOT NULL,
      FirstName varchar(50) NOT NULL,
      AreaCode int NULL,
      Address varchar(50) NULL,
      Phone varchar(50) NULL,
)
GO
```

We see that “CustomerNumber”, “LastName” and “FirstName” is set to “NOT NULL”, this means these columns needs to contain data. While “AreaCode”, “Address” and “Phone” may be left empty, i.e, they don’t need to filled out.

Note! A primary key column cannot contain NULL values.

Setting NULL/NOT NULL in the Designer Tools:

In the Table Designer you can easily set which columns that should allow NULL or not:



UNIQUE

The **UNIQUE** constraint uniquely identifies each record in a database table. The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.

Note! You can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

If we take a closer look at the CUSTOMER table created earlier:

```
CREATE TABLE [CUSTOMER]
(
      CustomerId int IDENTITY(1,1) PRIMARY KEY,
      CustomerNumber int NOT NULL UNIQUE,
      LastName varchar(50) NOT NULL,
      FirstName varchar(50) NOT NULL,
      AreaCode int NULL,
      Address varchar(50) NULL,
      Phone varchar(50) NULL,
)
```

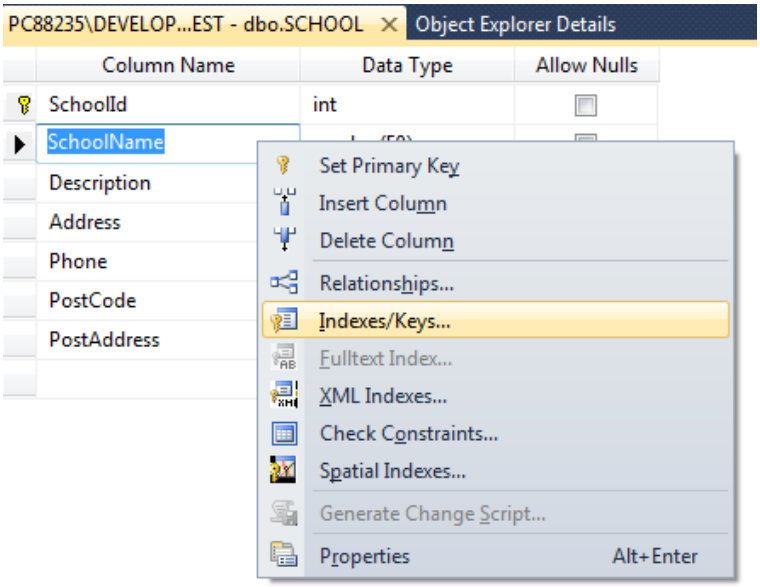
```
GO
```

We see that the “CustomerNumber” is set to UNIQUE, meaning each customer must have a unique Customer Number. Example:

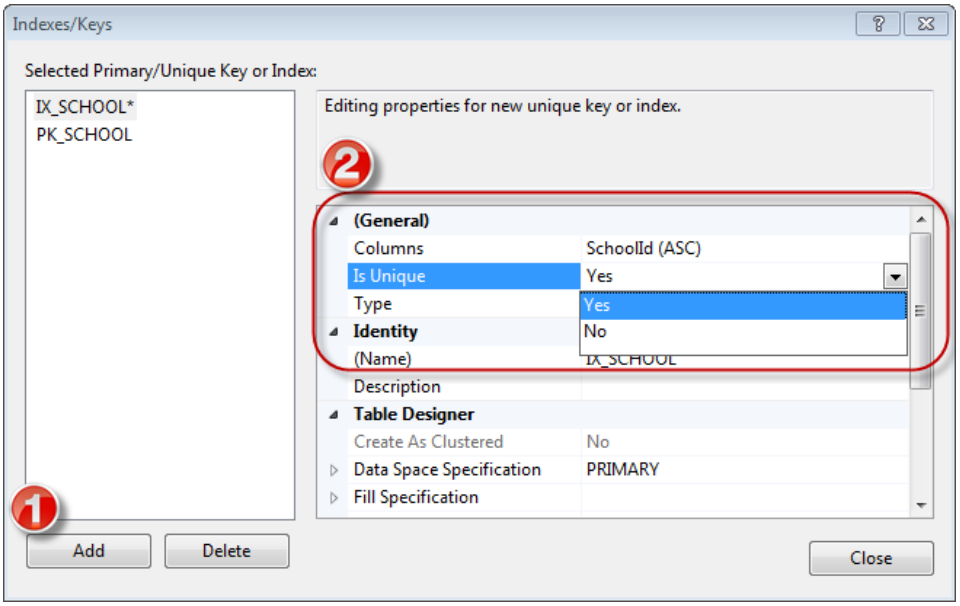
	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

Setting UNIQUE in the Designer Tools:

If you want to use the designer, right-click on the column that you want to be UNIQUE and select “**Indexes/Keys…**”:



Then click “Add” and then set the “Is Unique” property to “Yes”:



CHECK

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a single column it allows only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

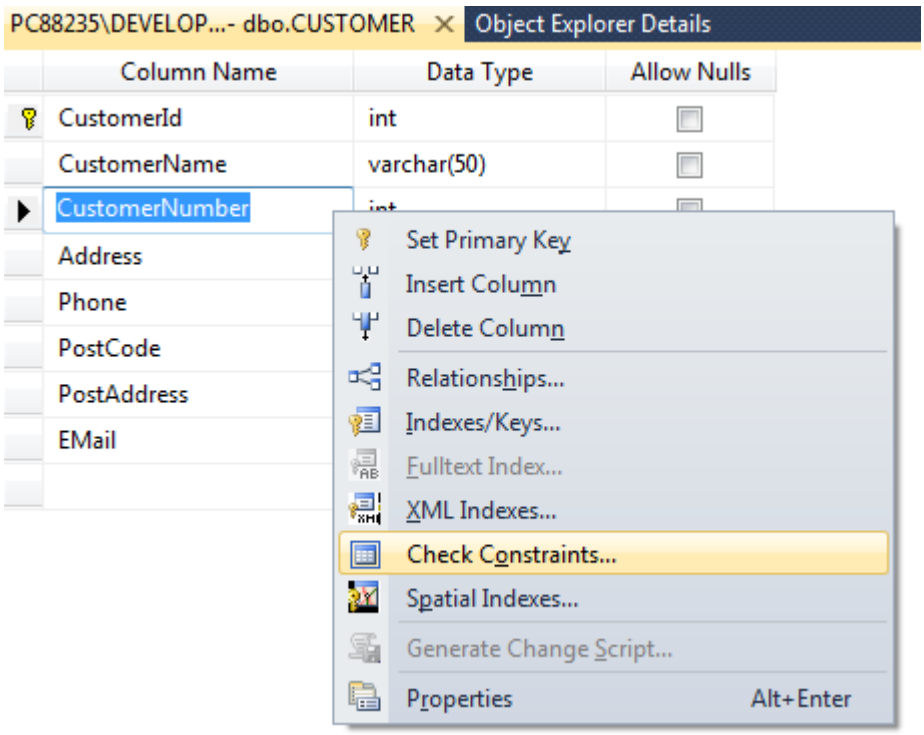
Example:

```
CREATE TABLE [CUSTOMER]
(
      CustomerId int IDENTITY(1,1) PRIMARY KEY,
      CustomerNumber int NOT NULL UNIQUE CHECK(CustomerNumber>0),
      LastName varchar(50) NOT NULL,
      FirstName varchar(50) NOT NULL,
      AreaCode int NULL,
      Address varchar(50) NULL,
      Phone varchar(50) NULL,
)
GO
```

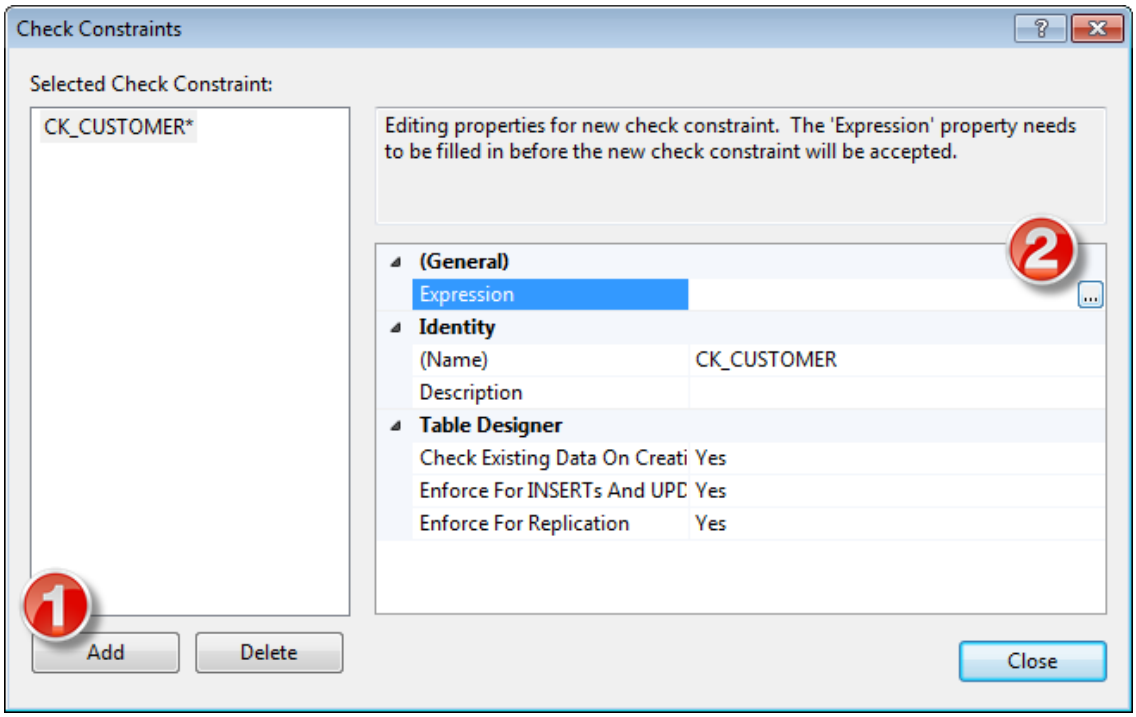
In this case, when we try to insert a Customer Number less than zero we will get an error message.

Setting CHECK constraints in the Designer Tools:

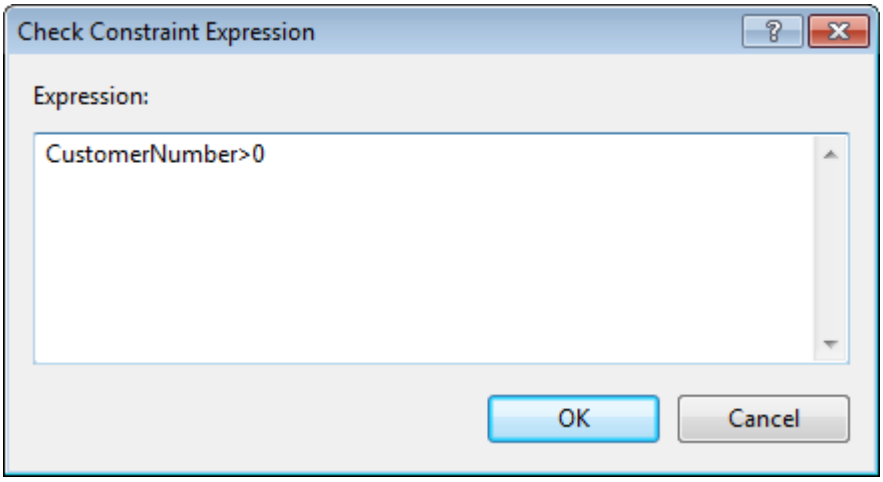
If you want to use the designer, right-click on the column where you want to set the constraints and select "**Check Constraints…**":



Then click “Add” and then click “…” in order to open the Expression window:



In the Expression window you can type in the expression you want to use:



DEFAULT

The DEFAULT constraint is used to insert a default value into a column.

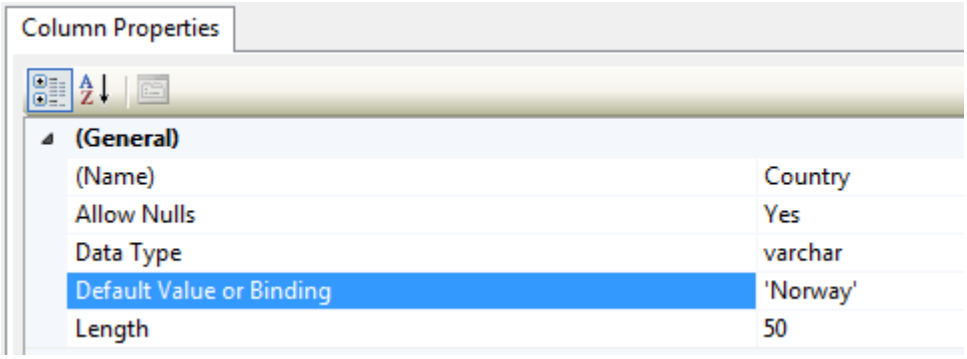
The default value will be added to all new records, if no other value is specified.

Example:

```
CREATE TABLE [CUSTOMER]
(
      CustomerId int IDENTITY(1,1) PRIMARY KEY,
      CustomerNumber int NOT NULL UNIQUE,
      LastName varchar(50) NOT NULL,
      FirstName varchar(50) NOT NULL,
      Country varchar(20) DEFAULT 'Norway',
      AreaCode int NULL,
      Address varchar(50) NULL,
      Phone varchar(50) NULL,
)
GO
```

Setting DEFAULT values in the Designer Tools:

Select the column and go into the “Column Properties”:



AUTO INCREMENT or IDENTITY

Very often we would like the value of the primary key field to be created automatically every time a new record is inserted.

Example:

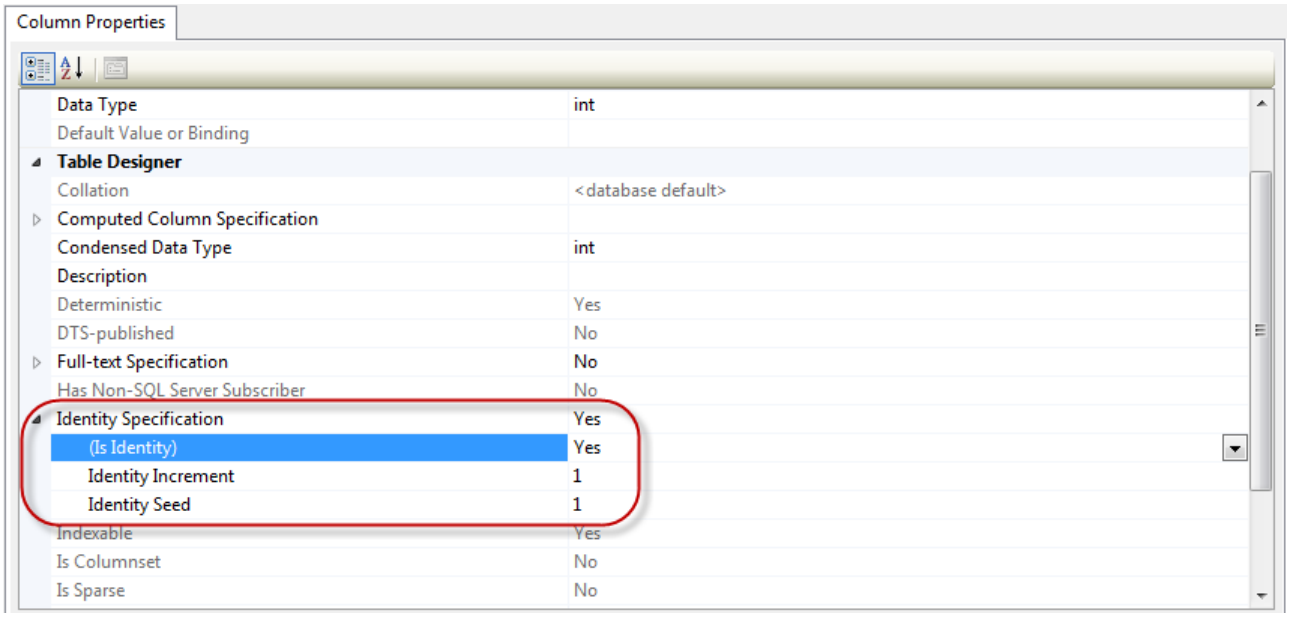
```
CREATE TABLE CUSTOMER
(
      CustomerId int IDENTITY(1,1) PRIMARY KEY,
      CustomerNumber int NOT NULL UNIQUE,
      LastName varchar(50) NOT NULL,
      FirstName varchar(50) NOT NULL,
      AreaCode int NULL,
      Address varchar(50) NULL,
      Phone varchar(50) NULL,
)
GO
```

As shown below, we use the IDENTITY() for this. IDENTITY(1,1) means the first value will be 1 and then it will increment by 1.

Setting identity(1,1) in the Designer Tools:

We can use the designer tools to specify that a Primary Key should be an identity column that is automatically generated by the system when we insert data in to the table.

Click on the column in the designer and go into the Column Properties window:



ALTER TABLE

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

To add a column in a table, use the following syntax:

```
ALTER TABLE table_name
ADD column_name datatype
```

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

```
ALTER TABLE table_name
DROP COLUMN column_name
```

To change the data type of a column in a table, use the following syntax:

```
ALTER TABLE table_name
ALTER COLUMN column_name datatype
```

If we use CREATE TABLE and the table already exists in the table we will get an error message, so if we combine CREATE TABLE and ALTER TABLE we can create robust database scripts that gives no errors, as the example shown below:

```
if not exists (select * from dbo.sysobjects where id = object id(N'[CUSTOMER]') and
OBJECTPROPERTY(id, N'IsUserTable') = 1)
CREATE TABLE CUSTOMER
(
        CustomerId int PRIMARY KEY,
        CustomerNumber int NOT NULL UNIQUE,
        LastName varchar(50) NOT NULL,
        FirstName varchar(50) NOT NULL,
        AreaCode int NULL,
        Address varchar(50) NULL,
        Phone varchar(50) NULL,
)
GO


if exists(select * from dbo.syscolumns where id = object id(N'[CUSTOMER]') and OBJECTPROPERTY(id,
N'IsUserTable') = 1 and name = 'CustomerId')
ALTER TABLE CUSTOMER ALTER COLUMN CustomerId int
Else
ALTER TABLE CUSTOMER ADD CustomerId int
GO


if exists(select * from dbo.syscolumns where id = object_id(N'[CUSTOMER]') and OBJECTPROPERTY(id,
N'IsUserTable') = 1 and name = 'CustomerNumber')
ALTER TABLE CUSTOMER ALTER COLUMN CustomerNumber int
```

```
Else
ALTER TABLE CUSTOMER ADD CustomerNumber int
GO

...
```

INSERT INTO

The INSERT INTO statement is used to insert a new row in a table.

It is possible to write the INSERT INTO statement in two forms.

The first form doesn't specify the column names where the data will be inserted, only their values:

```
INSERT INTO table_name
VALUES (value1, value2, value3,...)
```

Example:

```
INSERT INTO CUSTOMER VALUES ('1000', 'Smith', 'John', 12, 'California',
'11111111')
```

The second form specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3,...)
VALUES (value1, value2, value3,...)
```

This form is recommended!

Example:

```
INSERT INTO CUSTOMER (CustomerNumber, LastName, FirstName, AreaCode,
Address, Phone)
VALUES ('1000', 'Smith', 'John', 12, 'California', '11111111')
```

Insert Data Only in Specified Columns:

It is also possible to only add data in specific columns.

Example:

```
INSERT INTO CUSTOMER (CustomerNumber, LastName, FirstName)
VALUES ('1000', 'Smith', 'John')
```

Note! You need at least to include all columns that cannot be NULL.

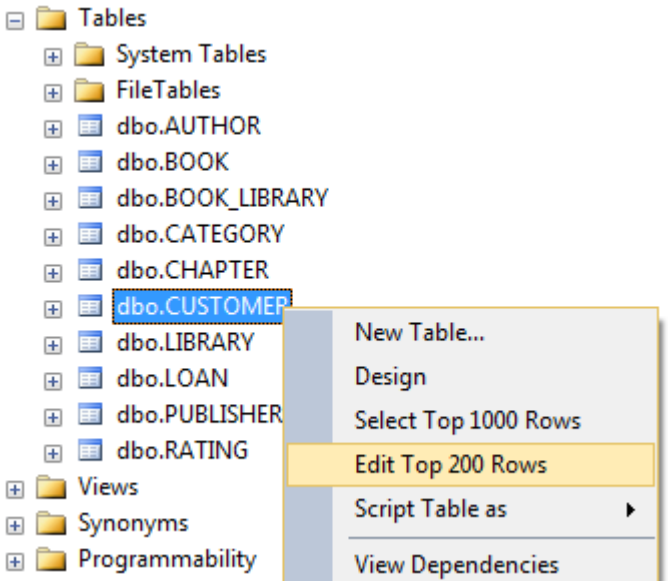
We remember the table definition for the CUSTOMER table:

Column Name	Data Type	Allow Nulls
CustomerId	int	☐
CustomerNumber	int	☐
LastName	varchar(50)	☐
FirstName	varchar(50)	☐
AreaCode	int	☑
Address	varchar(50)	☑
Phone	varchar(20)	☑
		☐

i.e., we need to include at least “CustomerNumber”, “LastName” and “FirstName”. “CustomerId” is set to “identity(1,1)” and therefore values for this column are generated by the system.

Insert Data in the Designer Tools:

When you have created the tables you can easily insert data into them using the designer tools. Right-click on the specific table and select “Edit Top 200 Rows”:



Then you can enter data in a table format, similar to, e.g., MS Excel:

PC88235\DEVELOP...- dbo.CUSTOMER | Object Explorer Details

CustomerId	CustomerName	CustomerNu...	Address	Phone	PostCode	PostAddress	EMail	Country
1	Bill Clinton	1000	NULL	NULL	NULL	NULL	NULL	NULL
2	Jens Stoltenberg	1001	NULL	NULL	NULL	NULL	NULL	NULL
3	Barak Obama	1002	NULL	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

UPDATE

The UPDATE statement is used to update existing records in a table.

The syntax is as follows:

```
UPDATE table_name
SET column1=value, column2=value2,...
WHERE some_column=some_value
```

Note! Notice the WHERE clause in the UPDATE syntax. The WHERE clause specifies which record or records that should be updated. If you omit the WHERE clause, all records will be updated!

Example:

```
update CUSTOMER set AreaCode=46 where CustomerId=2
```

Before update:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

After update:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	46	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

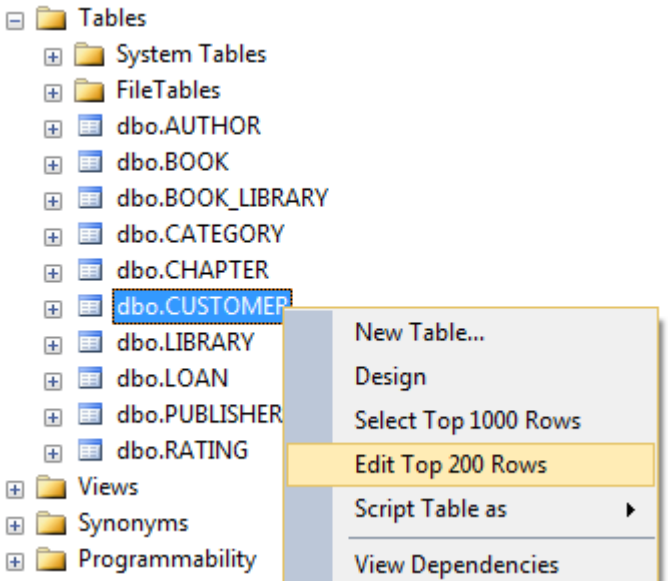
If you don't include the WHERE clause the result becomes:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	46	California	11111111
2	2	1001	Jackson	Smith	46	London	22222222
3	3	1002	Johnsen	John	46	London	33333333

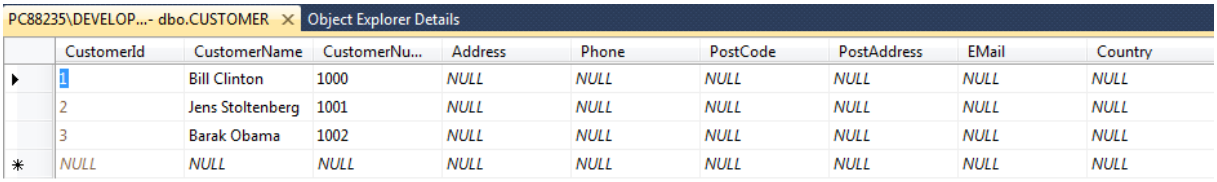
→ So make sure to include the WHERE clause when using the UPDATE command!

Update Data in the Designer Tools:

The same way you insert data you can also update the data. Right-click on the specific table and select “Edit Top 200 Rows”:



Then you can change your data:



CustomerId	CustomerName	CustomerNu...	Address	Phone	PostCode	PostAddress	EMail	Country
1	Bill Clinton	1000	NULL	NULL	NULL	NULL	NULL	NULL
2	Jens Stoltenberg	1001	NULL	NULL	NULL	NULL	NULL	NULL
3	Barak Obama	1002	NULL	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

DELETE

The DELETE statement is used to delete rows in a table.

Syntax:

```
DELETE FROM table_name
WHERE some_column=some_value
```

Note! Notice the WHERE clause in the DELETE syntax. The WHERE clause specifies which record or records that should be deleted. If you omit the WHERE clause, all records will be deleted!

Example:

```
delete from CUSTOMER where CustomerId=2
```

Before delete:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

After delete:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	3	1002	Johnsen	John	32	London	33333333

Delete All Rows:

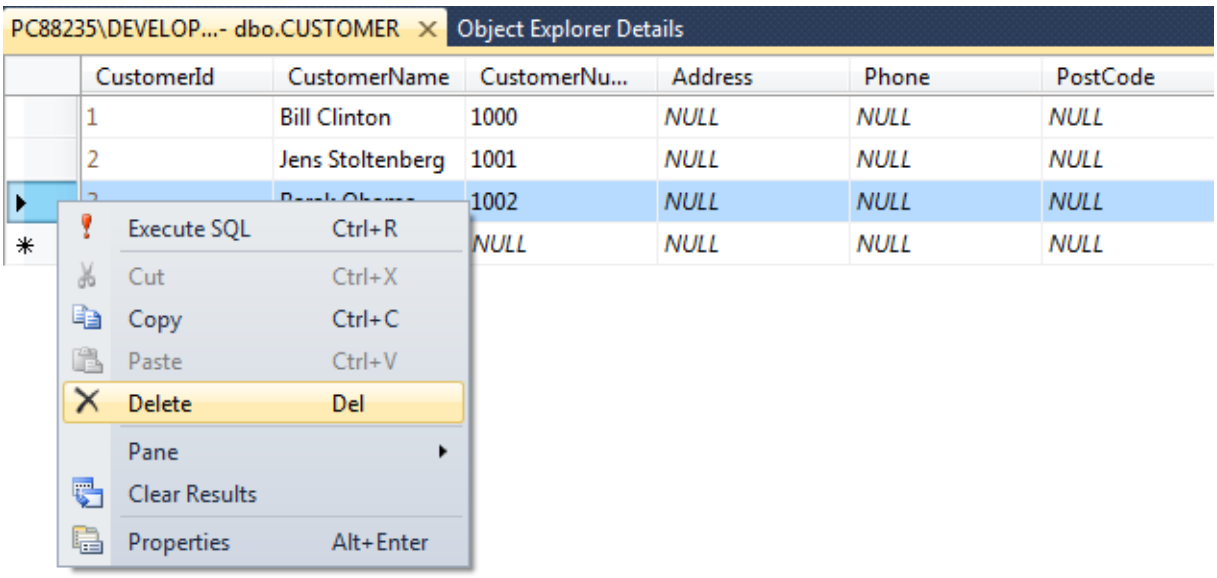
It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

```
DELETE FROM table_name
```

Note! Make sure to do this only when you really mean it! You cannot UNDO this statement!

Delete Data in the Designer Tools:

You delete data in the designer by right-click on the row and select “Delete”:


PC88235\DEVELOP...- dbo.CUSTOMER
Object Explorer Details
CustomerId
CustomerName
CustomerNu...
Address
Phone
PostCode
1
Bill Clinton
1000
NULL
NULL
NULL
2
Jens Stoltenberg
1001
NULL
NULL
NULL
1002
NULL
NULL
NULL
NULL
NULL
NULL
NULL
Execute SQL Ctrl+R
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Delete Del
Pane
Clear Results
Properties Alt+Enter

SELECT

The SELECT statement is probably the most used SQL command. The SELECT statement is used for retrieving rows from the database and enables the selection of one or many rows or columns from one or many tables in the database.

We will use the CUSTOMER table as an example.

The CUSTOMER table has the following columns:

Column Name	Data Type	Allow Nulls
CustomerId	int	☐
CustomerNumber	varchar(20)	☐
LastName	varchar(50)	☐
FirstName	varchar(50)	☐
AreaCode	int	☑
Address	varchar(50)	☑
Phone	varchar(20)	☑

The CUSTOMER table contains the following data:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

Example:

```
select * from CUSTOMER
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

This simple example gets all the data in the table CUSTOMER. The symbol “*” is used when you want to get all the columns in the table.

If you only want a few columns, you may specify the names of the columns you want to retrieve, example:

```
select CustomerId, LastName, FirstName from CUSTOMER
```

	CustomerId	LastName	FirstName
1	1	Smith	John
2	2	Jackson	Smith
3	3	Johnsen	John

So in the simplest form we can use the SELECT statement as follows:

```
select <column_names> from <table_names>
```

If we want all columns, we use the symbol "*"

Note! SQL is not case sensitive. SELECT is the same as select.

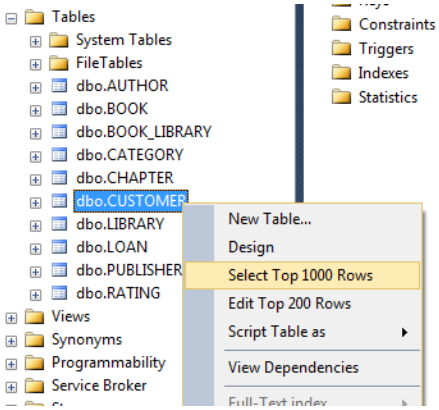
The full syntax of the SELECT statement is complex, but the main clauses can be summarized as:

```
SELECT
[ ALL | DISTINCT ]
    [TOP ( expression ) [PERCENT] [ WITH TIES ] ]
select_list [ INTO new_table ]
[ FROM table_source ] [ WHERE search_condition ]
[ GROUP BY group_by_expression ]
[ HAVING search_condition ]
[ ORDER BY order_expression [ ASC | DESC ] ]
```

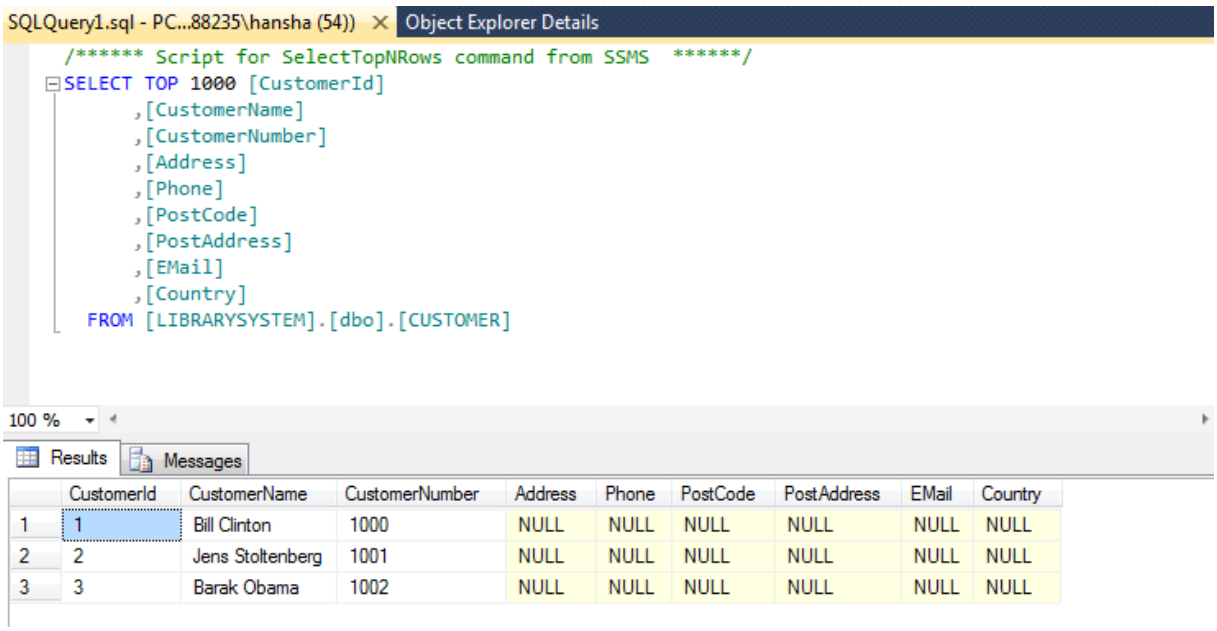
It seems complex, but we will take the different parts step by step in the next sections.

Select Data in the Designer Tools:

Right-click on a table and select "Select Top 1000 Rows":



The following will appear:



A Select query is automatically created for you which you can edit if you want to.

The ORDER BY Keyword

If you want the data to appear in a specific order you need to use the “order by” keyword.

Example:

```
select * from CUSTOMER order by LastName
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	2	1001	Jackson	Smith	45	London	22222222
2	3	1002	Johnsen	John	32	London	33333333
3	1	1000	Smith	John	12	California	11111111

You may also sort by several columns, e.g. like this:

```
select * from CUSTOMER order by Address, LastName
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

If you use the “order by” keyword, the default order is ascending (“asc”). If you want the order to be opposite, i.e., descending, then you need to use the “desc” keyword.

```
select * from CUSTOMER order by LastName desc
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	3	1002	Johnsen	John	32	London	33333333
3	2	1001	Jackson	Smith	45	London	22222222

SELECT DISTINCT

In a table, some of the columns may contain duplicate values. This is not a problem, however, sometimes you will want to list only the different (distinct) values in a table.

The DISTINCT keyword can be used to return only distinct (different) values.

The syntax is as follows:

```
select distinct <column_names> from <table_names>
```

Example:

```
select distinct FirstName from CUSTOMER
```

	FirstName
1	John
2	Smith

The WHERE Clause

The WHERE clause is used to extract only those records that fulfill a specified criterion.

The syntax is as follows:

```
select <column_names>
from <table_name>
where <column_name> operator value
```

Example:

```
select * from CUSTOMER where CustomerNumber='1001'
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	2	1001	Jackson	Smith	45	London	22222222

Note! SQL uses single quotes around text values, as shown in the example above.

Operators

With the WHERE clause, the following operators can be used:

Operator	Description
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	If you know the exact value you want to return for at least one of the columns

Examples:

```
select * from CUSTOMER where AreaCode>30
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	2	1001	Jackson	Smith	45	London	22222222
2	3	1002	Johnsen	John	32	London	33333333

LIKE Operator

The LIKE operator is used to search for a specified pattern in a column.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name LIKE pattern
```

Example:

```
select * from CUSTOMER where LastName like 'J%'
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	2	1001	Jackson	Smith	45	London	22222222
2	3	1002	Johnsen	John	32	London	33333333

Note! The "%" sign can be used to define wildcards (missing letters in the pattern) both before and after the pattern.

```
select * from CUSTOMER where LastName like '%a%'
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	2	1001	Jackson	Smith	45	London	22222222

You may also combine with the NOT keyword, example:

```
select * from CUSTOMER where LastName not like '%a%'
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	3	1002	Johnsen	John	32	London	33333333

IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1,value2,...)
```

BETWEEN Operator

The BETWEEN operator selects a range of data between two values. The values can be numbers, text, or dates.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name
BETWEEN value1 AND value2
```

Wildcards

SQL wildcards can substitute for one or more characters when searching for data in a database.

Note! SQL wildcards must be used with the SQL LIKE operator.

With SQL, the following wildcards can be used:

Wildcard	Description
%	A substitute for zero or more characters
_	A substitute for exactly one character
[charlist]	Any single character in charlist
[^charlist] or **[!charlist]**	Any single character not in charlist

Examples:

```
SELECT * FROM CUSTOMER WHERE LastName LIKE 'J_cks_n'
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	2	1001	Jackson	Smith	45	London	22222222

```
SELECT * FROM CUSTOMER WHERE CustomerNumber LIKE '[10]%'
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

AND & OR Operators

The AND operator displays a record if both the first condition and the second condition is true.

The OR operator displays a record if either the first condition or the second condition is true.

Examples:

```
select * from CUSTOMER where LastName='Smith' and FirstName='John'
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111

```
select * from CUSTOMER where LastName='Smith' or FirstName='John'
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	3	1002	Johnsen	John	32	London	33333333

Combining AND & OR:

You can also combine AND and OR (use parenthesis to form complex expressions).

Example:

```
select * from CUSTOMER
where LastName='Smith' and (FirstName='John' or FirstName='Smith')
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111

SELECT TOP Clause

The TOP clause is used to specify the number of records to return.

The TOP clause can be very useful on large tables with thousands of records. Returning a large number of records can impact on performance.

Syntax:

```
SELECT TOP number|percent column_name(s)
FROM table_name
```

Examples:

```
select TOP 1 * from CUSTOMER
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111

You can also specify in percent:

```
select TOP 60 percent * from CUSTOMER
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222

This is very useful for large tables with thousands of records

Alias

You can give a table or a column another name by using an alias. This can be a good thing to do if you have very long or complex table names or column names.

An alias name could be anything, but usually it is short.

SQL Alias Syntax for Tables:

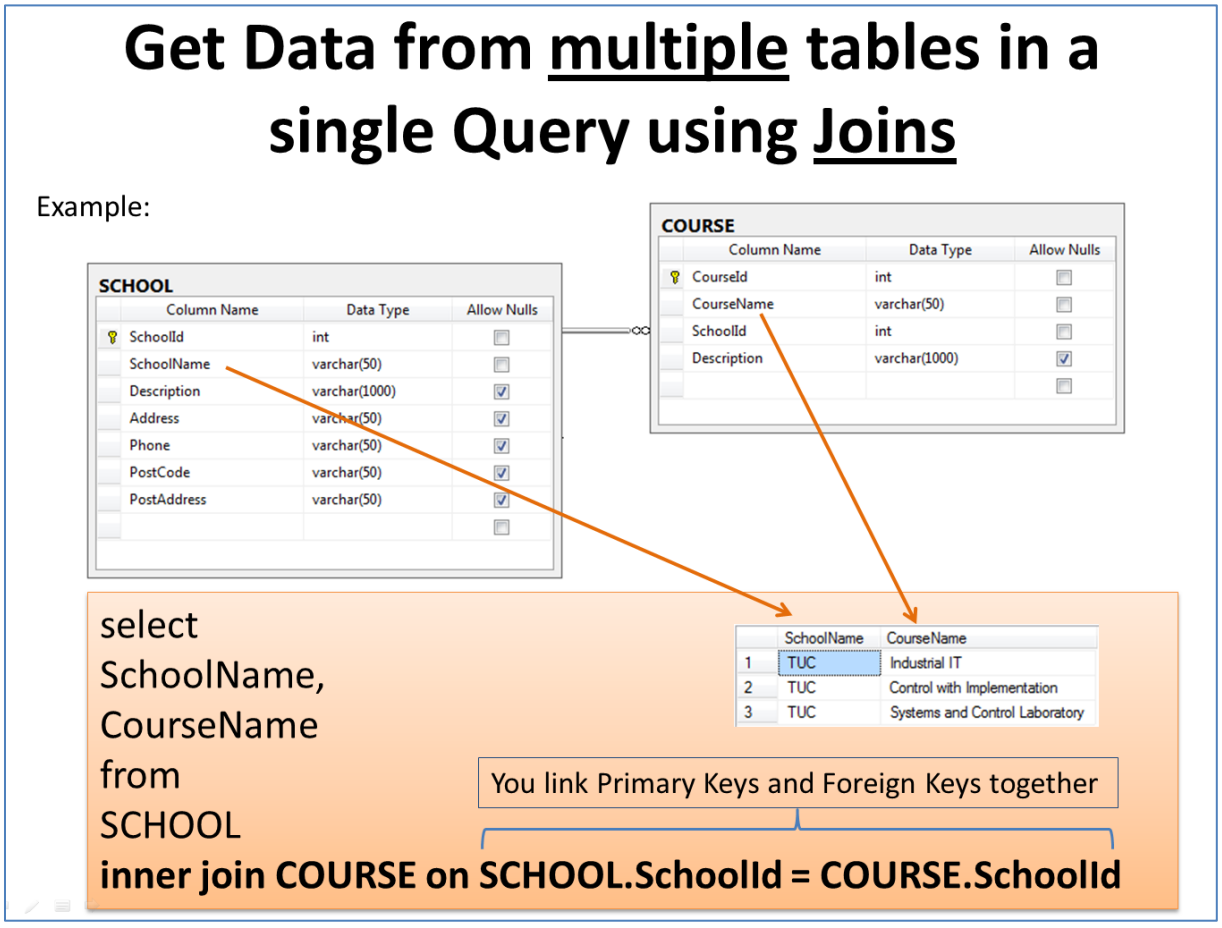
```
SELECT column_name(s)
FROM table_name
AS alias_name
```

SQL Alias Syntax for Columns:

```
SELECT column_name AS alias_name
FROM table_name
```

Joins

SQL joins are used to query data from two or more tables, based on a relationship between certain columns in these tables.



Different SQL JOINs

Before we continue with examples, we will list the types of JOIN you can use, and the differences between them.

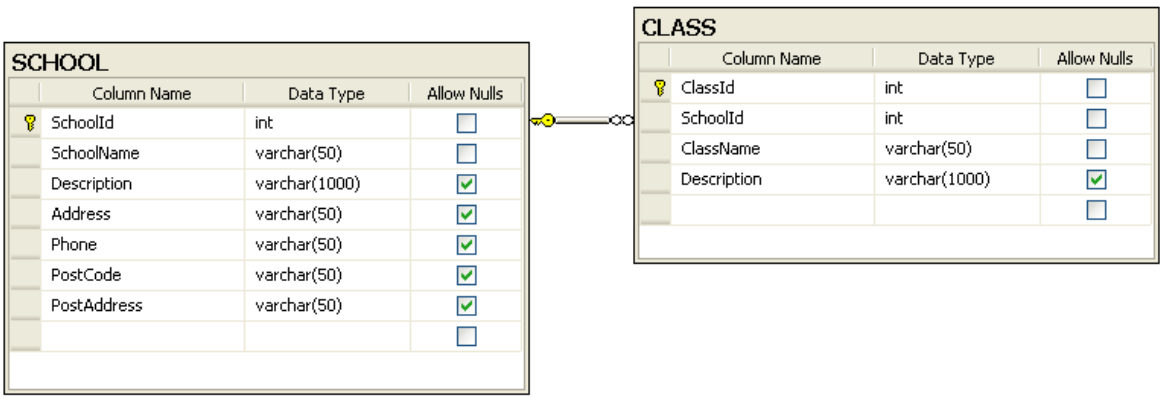
- JOIN: Return rows when there is at least one match in both tables
- LEFT JOIN: Return all rows from the left table, even if there are no matches in the right table
- RIGHT JOIN: Return all rows from the right table, even if there are no matches in the left table
- FULL JOIN: Return rows when there is a match in one of the tables

Example:

Given 2 tables:

- SCHOOL
- CLASS

The diagram is shown below:



We want to get the following information using a query:

SchoolName	ClassName
…	…
…	…

In order to get information from more than one table we need to use the JOIN. The JOIN is used to join the primary key in one table with the foreign key in another table.

```
select
SCHOOL.SchoolName,
CLASS.ClassName
from
SCHOOL
INNER JOIN CLASS ON SCHOOL.SchoolId = CLASS.SchoolId
```

	SchoolName	ClassName
1	TUC	SCE1
2	TUC	SCE2
3	TUC	PT1
4	TUC	PT2
5	NTNU	A1
6	NTNU	A2

SQL Scripts

A SQL script is a collection of SQL statements that you can execute in one operation. You can use any kind of SQL commands, such as insert, select, delete, update, etc. In addition you can define and use variables, and you may also use program flow like If-Else, etc. You may also add comments to make the script easier to read and understand.

Using Comments

Using comments in you SQL script is important to make the script easier to read and understand.

In SQL we can use 2 different kinds of comments:

- Single-line comment
- Multiple-line comment

Single-line comment

We can comment one line at the time using "--" before the text you want to comment out.

Syntax:

```
-- text_of_comment
```

Multiple-line comment

We can comment several line using "/*" in the start of the comment and "*/" in the end of the comment.

Syntax:

```
/*
text_of_comment
text_of_comment
*/
```

Variables

The ability to using variables in SQL is a powerful feature. You need to use the keyword **DECLARE** when you want to define the variables. Local variables must have the the symbol "@" as a prefix. You also need to specify a data type for your variable (int, varchar(x), etc.).

Syntax for declaring variables:

```
declare @local_variable data_type
```

If you have more than one variable you want to declare:

```
declare
@myvariable1 data_type,
@myvariable2 data_type,
…
```

When you want to assign values to the variable, you must use either a **SET** or a **SELECT** statement.

Example:

```
declare @myvariable int

set @myvariable=4
```

If you want to see the value for a variable, you can e.g., use the **PRINT** command like this:

```
declare @myvariable int

set @myvariable=4

print @myvariable
```

The following will be shown in SQL Server:

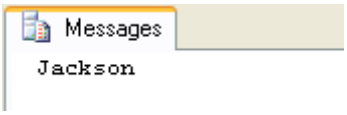
Assigning variables with a value from a SELECT statement is very useful.

We use the CUSTOMER table as an example:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

You can assign a value to the variable from a select statement like this:

```
declare @mylastname varchar(50)

select @mylastname=LastName from CUSTOMER where CustomerId=2
print @mylastname
```



You can also use a variable in the WHERE clause LIKE, e.g., this:

```
declare @find varchar(30)
set @find = 'J%'
select * from CUSTOMER
where LastName LIKE @find
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	2	1001	Jackson	Smith	45	London	22222222
2	3	1002	Johnsen	John	32	London	33333333

Built-in Global Variables

SQL have lots of built-in variables that are very useful to use in queries and scripts.

@@IDENTITY

After an INSERT, SELECT INTO, or bulk copy statement is completed, @@IDENTITY contains the last identity value that is generated by the statement. If the statement did not affect any tables with identity columns, @@IDENTITY returns NULL. If multiple rows are inserted, generating multiple identity values, @@IDENTITY returns the last identity value generated.

Example:

Given to tables; SCHOOL and COURSE:

SCHOOL table:

	SchoolId	SchoolName	Description	Address	Phone	PostCode	PostAddress
1	1	TUC	NULL	NULL	NULL	NULL	NULL
2	2	NTNU	NULL	NULL	NULL	NULL	NULL

COURSE table:

	CourseId	CourseName	SchoolId	Description
1	1	SCE2006	1	NULL
2	2	SCE1106	1	NULL
3	3	SCE4206	1	NULL
4	4	SCE4106	1	NULL

We want to insert a new School into the SCHOOL table and we want to insert 2 new Courses in the COURSE table that belong to the School we insert. To find the "SchoolId" we can use the @@IDENTITY variable:

```
declare @SchoolId int

-- Insert Data into SCHOOL table
insert into SCHOOL(SchoolName) values ('MIT')

select @SchoolId = @@IDENTITY


-- Insert Courses for the specific School above in the COURSE table
insert into COURSE(SchoolId,CourseName) values (@SchoolId, 'MIT-101')
insert into COURSE(SchoolId,CourseName) values (@SchoolId, 'MIT-201')
```

The result becomes:

SCHOOL table:

	SchoolId	SchoolName	Description	Address	Phone	PostCode	PostAddress
1	1	TUC	NULL	NULL	NULL	NULL	NULL
2	2	NTNU	NULL	NULL	NULL	NULL	NULL
3	16	MIT	NULL	NULL	NULL	NULL	NULL

COURSE table:

	CourseId	CourseName	SchoolId	Description
1	1	SCE2006	1	NULL
2	2	SCE1106	1	NULL
3	3	SCE4206	1	NULL
4	4	SCE4106	1	NULL
5	5	MIT-101	16	NULL
6	6	MIT-201	16	NULL

Flow Control

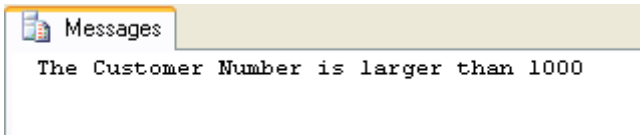
As with other programming languages you can use different kind of flow control, such as IF-ELSE, WHILE, etc, which is very useful.

IF – ELSE

The IF-ELSE is very useful. Below we see an example:

```
declare @customerNumber int

select @customerNumber=CustomerNumber from CUSTOMER
where CustomerId=2
```

```
if @customerNumber > 1000
      print 'The Customer Number is larger than 1000'
else
      print 'The Customer Number is not larger than 1000'
```



BEGIN…END:

If more than one line of code is to be executed within an IF sentence you need to use **BEGIN**…**END**.

Example:

```
select @customerNumber=CustomerNumber from CUSTOMER where CustomerId=2

if @customerNumber > 1000
      begin
            print 'The Customer Number is larger than 1000'
            update CUSTOMER set AreaCode=46 where CustomerId=2
      end
else
      print 'The Customer Number is not larger than 1000'
```

WHILE

We can also use WHILE, which is known from other programming languages.

Example:

We are using the CUSTOMER table:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

and the following query:

```
while (select AreaCode from CUSTOMER where CustomerId=1) < 20
begin
   update CUSTOMER set AreaCode = AreaCode + 1
end

select * from CUSTOMER
```

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	20	California	11111111
2	2	1001	Jackson	Smith	53	London	22222222
3	3	1002	Johnsen	John	40	London	33333333

As you can see the code inside the WHILE loop is executed as long as “AreaCode” for CustomerId=1 is less than 20. For each iteration is the “AreaCode” for that customer incremented with 1.

CASE

The CASE statement evaluates a list of conditions and returns one of multiple possible result expressions.

Example:

We have a “GRADE” table that contains the grades for each student in different courses:

```
select GradeId, StudentId, CourseId, Grade from GRADE
```

	GradeId	StudentId	CourseId	Grade
1	1	1	1	4
2	2	2	1	5
3	3	3	3	0
4	4	4	3	3
5	5	1	3	5

In the “GRADE” table is the grades stored as numbers, but since the students get grades with the letters A..F (A=5, B=4, C=3, D=2, E=1, F=0), we want to convert the values in the table into letters using a CASE statement:

```
select
GradeId,
StudentId,
CourseId,
case Grade
      when 5 then 'A'
      when 4 then 'B'
      when 3 then 'C'
      when 2 then 'D'
      when 1 then 'E'
      when 0 then 'F'
      else '-'
end   as Grade
from
GRADE
```

	GradeId	StudentId	CourseId	Grade
1	1	1	1	B
2	2	2	1	A
3	3	3	3	F
4	4	4	3	C
5	5	1	3	A

CURSOR

In advances scripts, CURSORs may be very useful. A CURSOR works like an advanced WHILE loop which we use to iterate through the records in one or more tables.

CURSORS are used mainly in stored procedures, triggers, and SQL scripts.

Example:

We use the CUSTOMER table as an example:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	20	California	11111111
2	2	1001	Jackson	Smith	53	London	2222
3	3	1002	Johnsen	John	40	London	33333333
4	6	1003	Obama	Barak	51	Nevada	4444

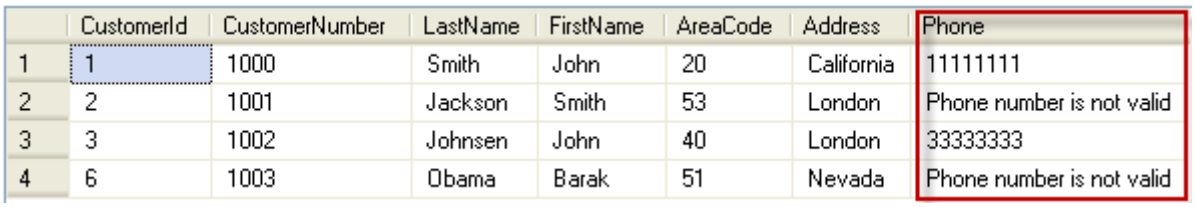
We will create a CURSOR that iterate through all the records in the CUSTOMER table and check if the Phone number consists of 8 digits, if not the script will replace the invalid Phone number with the text “Phone number is not valid”.

Here is the SQL Script using a CURSOR:

```
DECLARE
@CustomerId int,
@phone varchar(50)


DECLARE db_cursor CURSOR
FOR SELECT CustomerId from CUSTOMER

OPEN db_cursor
FETCH NEXT FROM db_cursor INTO @CustomerId

WHILE @@FETCH_STATUS = 0
BEGIN

       select @phone=Phone from CUSTOMER where CustomerId=@CustomerId

       if LEN(@phone) < 8
             update CUSTOMER set Phone='Phone number is not valid' where
CustomerId=@CustomerId
```

```
        FETCH NEXT FROM db_cursor INTO @CustomerId
END

CLOSE db_cursor
DEALLOCATE db_cursor
```

The CUSTOMER table becomes:



	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	20	California	11111111
2	2	1001	Jackson	Smith	53	London	Phone number is not valid
3	3	1002	Johnsen	John	40	London	33333333
4	6	1003	Obama	Barak	51	Nevada	Phone number is not valid

Creating and using a CURSOR includes these steps:

- Declare SQL variables to contain the data returned by the cursor. Declare one variable for each result set column.
- Associate a SQL cursor with a SELECT statement using the DECLARE CURSOR statement. The DECLARE CURSOR statement also defines the characteristics of the cursor, such as the cursor name and whether the cursor is read-only or forward-only.
- Use the OPEN statement to execute the SELECT statement and populate the cursor.
- Use the FETCH INTO statement to fetch individual rows and have the data for each column moved into a specified variable. Other SQL statements can then reference those variables to access the fetched data values.
- When you are finished with the cursor, use the CLOSE statement. Closing a cursor frees some resources, such as the cursor's result set and its locks on the current row. The DEALLOCATE statement completely frees all resources allocated to the cursor, including the cursor name.

Views

Views are virtual table for easier access to data stored in multiple tables.

Create View:

```
IF EXISTS (SELECT name
             FROM   sysobjects
             WHERE  name = 'CourseData'
             AND      type = 'V')
          DROP VIEW CourseData
GO

CREATE VIEW CourseData
AS

SELECT
SCHOOL.SchoolId,
SCHOOL.SchoolName,
COURSE.CourseId,
COURSE.CourseName,
COURSE.Description

FROM
SCHOOL
INNER JOIN COURSE ON SCHOOL.SchoolId = COURSE.SchoolId
GO
```

A View is a "virtual" table that can contain data from multiple tables

The Name of the View

Inside the View you join the different tables together using the **JOIN** operator

You can Use the View as an ordinary table in Queries :

Using the View:

```
select * from CourseData
```

	SchoolId	SchoolName	CourseId	CourseName	Description
1	1	TUC	1	Industrial IT	The best course ever
2	1	TUC	2	Control with Implementation	Control Theory
3	1	TUC	3	Systems and Control Laboratory	Practical Lav course

Syntax for creating a View:

```
CREATE VIEW <ViewName>
AS
…
```

... but it might be easier to do it in the graphical view designer that are built into SQL Management Studio.

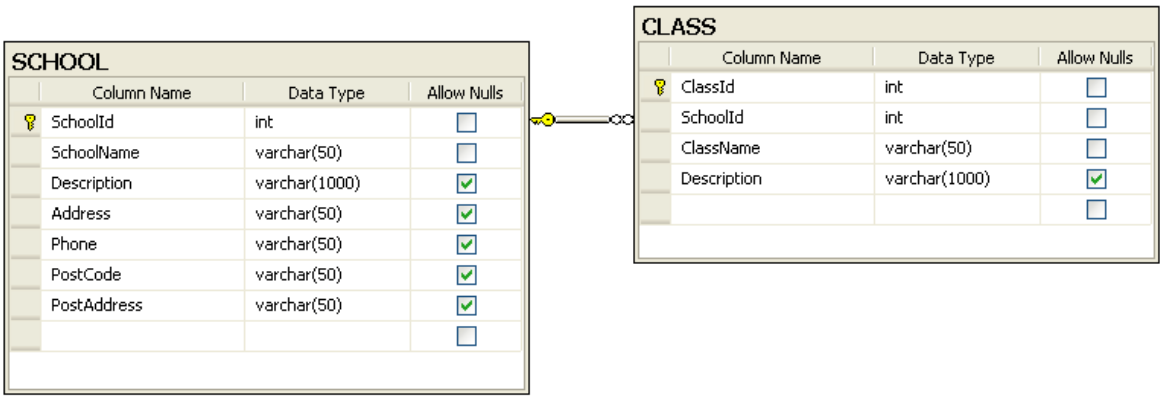
Syntax for using a View:

```
select * from <MyView> where …
```

As shown above, we use a VIEW just like we use an ordinary table.

Example:

We use the SCHOOL and CLASS tables as an example for our View. We want to create a View that lists all the existing schools and the belonging classes.



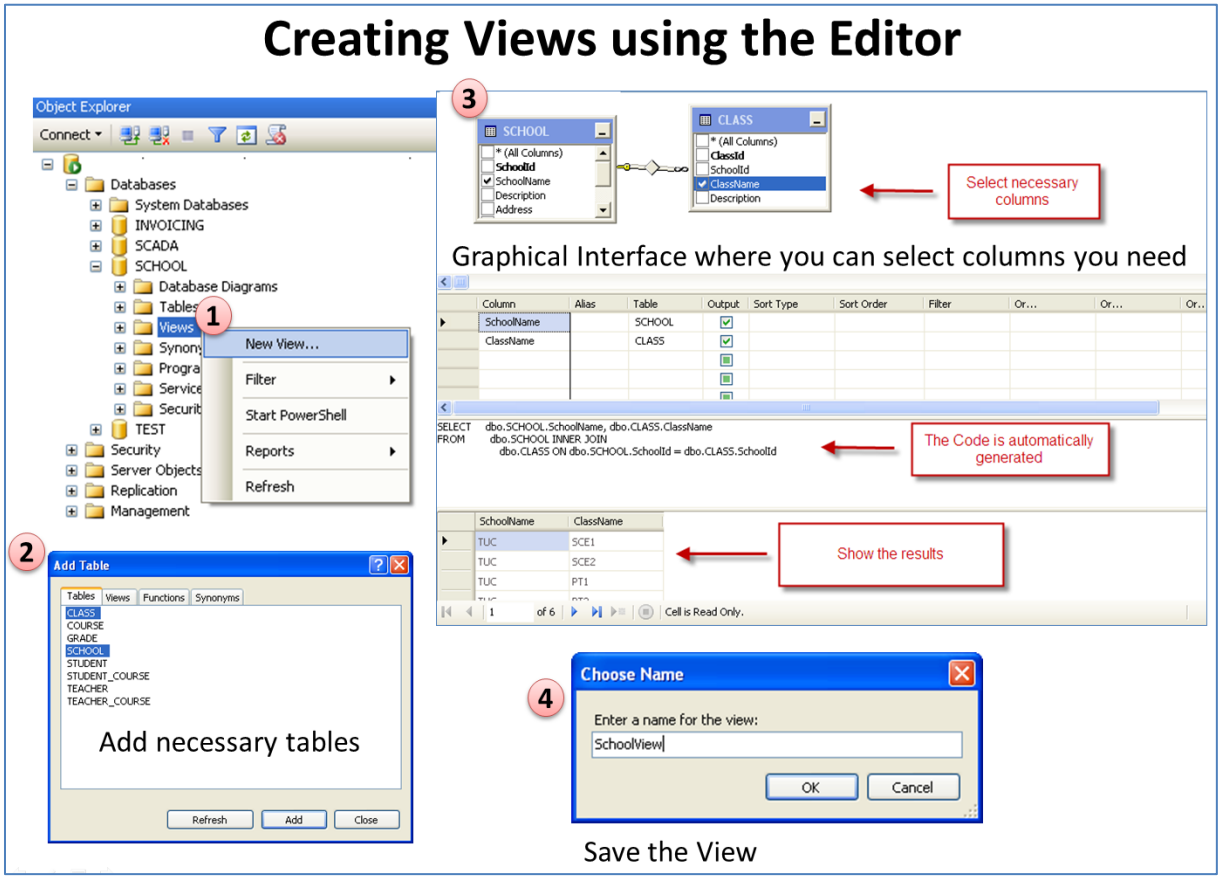
We create the VIEW using the CREATE VIEW command:

```
CREATE VIEW SchoolView
AS

SELECT
SCHOOL.SchoolName,
CLASS.ClassName
FROM
SCHOOL
INNER JOIN CLASS ON SCHOOL.SchoolId = CLASS.SchoolId
```

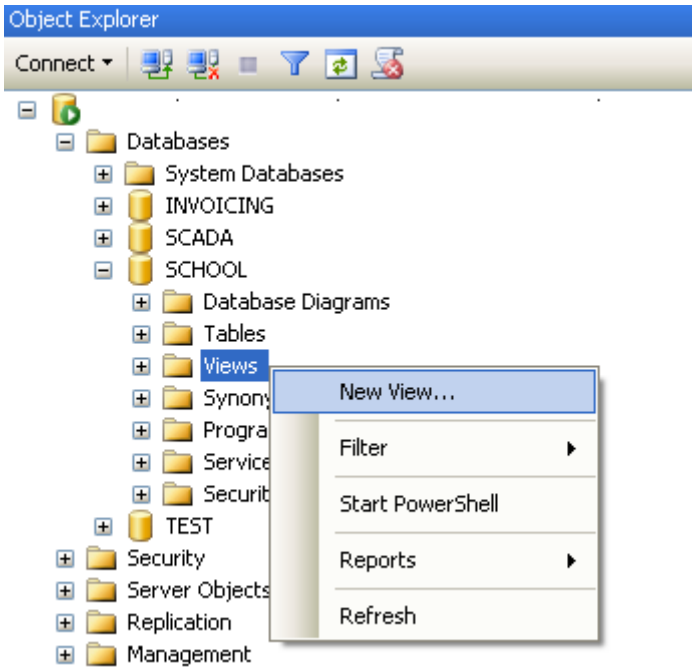
Note! In order to get information from more than one table, we need to link the tables together using a JOIN.

Using the Graphical Designer

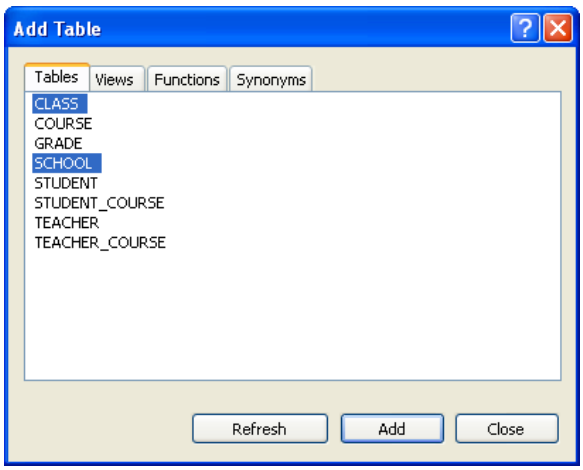
We create the same View using the graphical designer in SQL Server Management Studio:



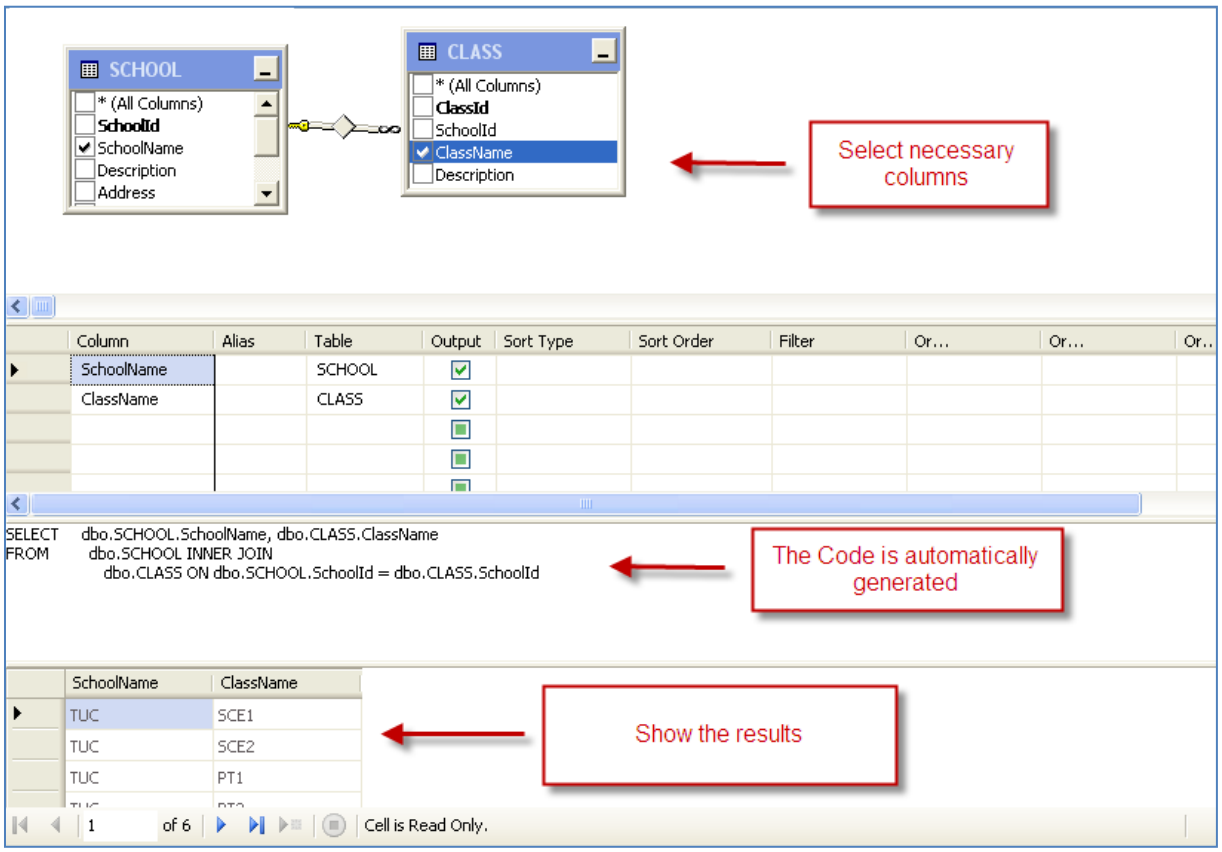
Step 1: Right-click on the View node and select “New View…”:



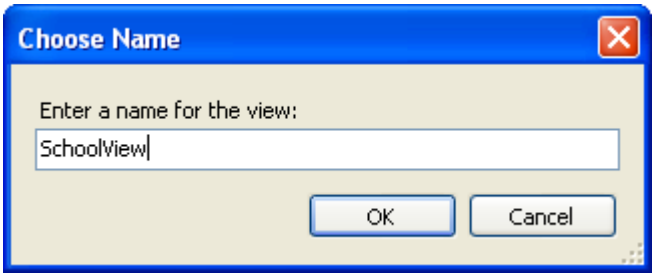
Step 2: Add necessary tables:



Step 3: Add Columns, etc.



Step 4: Save the VIEW:



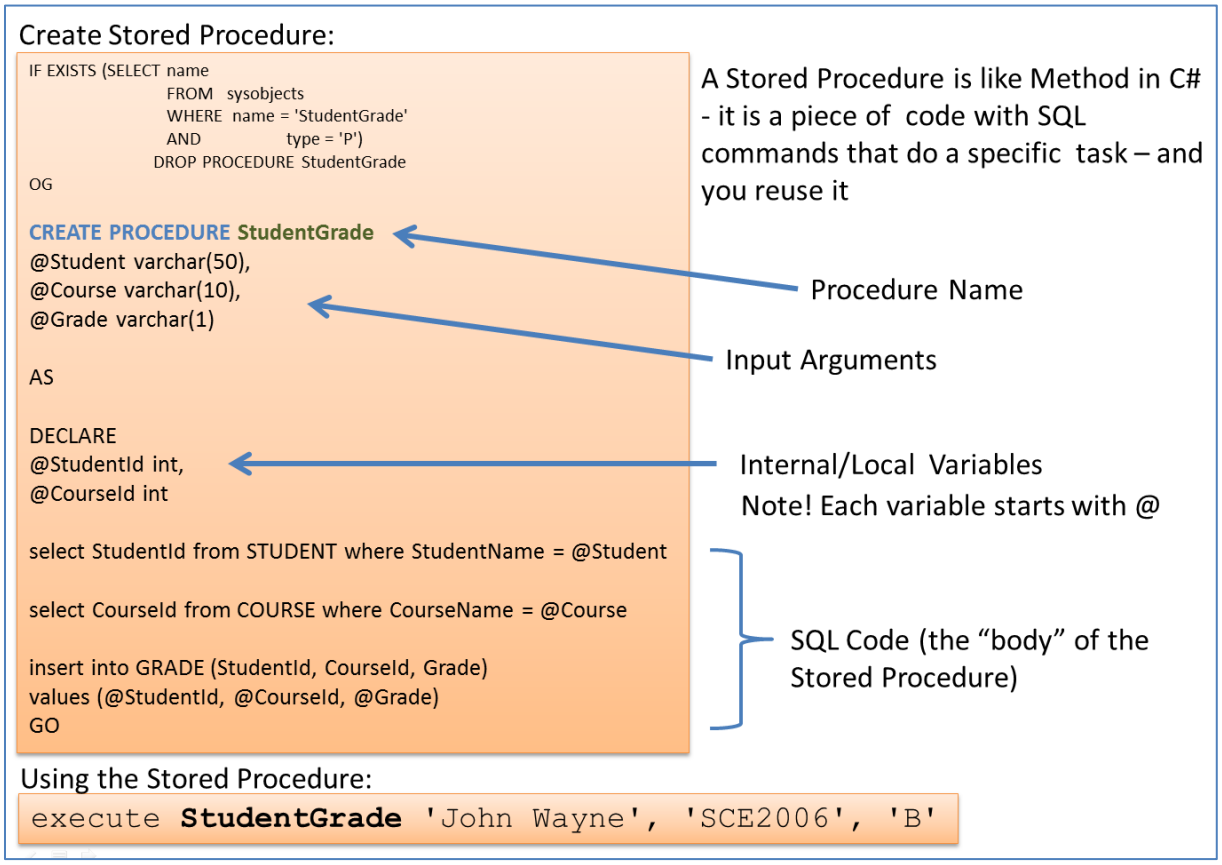
Step 5: Use the VIEW in a query:

```
select * from SchoolView
```

	SchoolName	ClassName
1	TUC	SCE1
2	TUC	SCE2
3	TUC	PT1
4	TUC	PT2
5	NTNU	A1
6	NTNU	A2

Stored Procedures

A Stored Procedure is a precompiled collection of SQL statements. In a stored procedure you can use if sentence, declare variables, etc.



Syntax for creating a Stored Procedure:

```
CREATE PROCEDURE <ProcedureName>
@<Parameter1> <datatype>
…
declare
@myVariable <datatype>
… Create your Code here
```

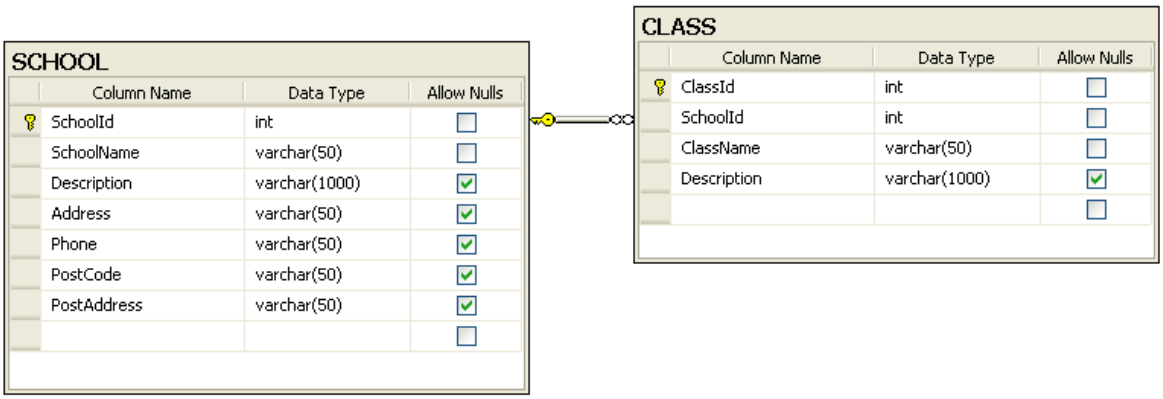
Note! You need to use the symbol "@" before variable names.

Syntax for using a Stored Procedure:

```
EXECUTE <ProcedureName(…)>
```

Example:

We use the SCHOOL and CLASS tables as an example for our Stored Procedure. We want to create a Stored Procedure that lists all the existing schools and the belonging classes.



We create the Stored Procedure as follows:

```
CREATE PROCEDURE GetAllSchoolClasses
AS

select
SCHOOL.SchoolName,
CLASS.ClassName
from
SCHOOL
inner join CLASS on SCHOOL.SchoolId = CLASS.SchoolId
order by SchoolName, ClassName
```

When we have created the Stored Procedure we can run (or execute) the Stored procedure using the execute command like this:

```
execute GetAllSchoolClasses
```

	SchoolName	ClassName
1	NTNU	A1
2	NTNU	A2
3	TUC	PT1
4	TUC	PT2
5	TUC	SCE1
6	TUC	SCE2

We can also create a Store Procedure with input parameters.

Example:

We use the same tables in this example (SCHOOL and CLASS) but now we want to list all classes for a specific school.

The Stored Procedure becomes:

```
CREATE PROCEDURE GetSpecificSchoolClasses
@SchoolName varchar(50)
AS

select
SCHOOL.SchoolName,
CLASS.ClassName
from
SCHOOL
inner join CLASS on SCHOOL.SchoolId = CLASS.SchoolId
where SchoolName=@SchoolName
order by ClassName
```

We run (or execute) the Stored Procedure:

```
execute GetSpecificSchoolClasses 'TUC'
```

	SchoolName	ClassName
1	TUC	PT1
2	TUC	PT2
3	TUC	SCE1
4	TUC	SCE2

or:

```
execute GetSpecificSchoolClasses 'NTNU'
```

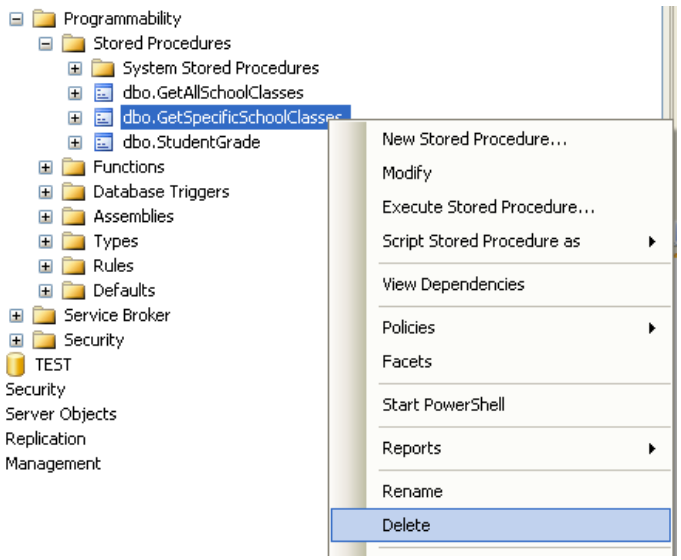
	SchoolName	ClassName
1	NTNU	A1
2	NTNU	A2

When we try to create a Stored Procedure that already exists we get the following error message:

```
There is already an object named 'GetSpecificSchoolClasses' in the database.
```

Then we first need to delete (or DROP) the old Stored Procedure before we can recreate it again.

We can do this manually in the Management Studio in SQL like this:



A better solution is to add code for this in our script, like this:

```
IF EXISTS (SELECT name
        FROM   sysobjects
        WHERE  name = GetSpecificSchoolClasses '
        AND       type = 'P')
     DROP PROCEDURE GetSpecificSchoolClasses
GO

CREATE PROCEDURE GetSpecificSchoolClasses
@SchoolName varchar(50)
AS

select
SCHOOL.SchoolName,
CLASS.ClassName
from
SCHOOL
inner join CLASS on SCHOOL.SchoolId = CLASS.SchoolId
where SchoolName=@SchoolName
order by ClassName
```

So we use CREATE PROCEDURE to create a Stored Procedure and we use DROP PROCEDURE to delete a Stored Procedure.

NOCOUNT ON/NOCOUNT OFF

In advanced Stored Procedures and Script, performance is very important. Using SET NOCOUNT ON and SET NOCOUNT OFF makes the Stored Procedure run faster.

SET NOCOUNT ON stops the message that shows the count of the number of rows affected by a Transact-SQL statement or stored procedure from being returned as part of the result set.

SET NOCOUNT ON prevents the sending of DONE_IN_PROC messages to the client for each statement in a stored procedure. For stored procedures that contain several statements that do not return much actual data, or for procedures that contain Transact-SQL loops, setting SET NOCOUNT to ON can provide a significant performance boost, because network traffic is greatly reduced.

Example:

```
IF EXISTS (SELECT name
         FROM   sysobjects
         WHERE  name = 'sp_LIMS_IMPORT_REAGENT'
         AND       type = 'P')
      DROP PROCEDURE sp_LIMS_IMPORT_REAGENT
GO

CREATE PROCEDURE sp_LIMS_IMPORT_REAGENT
@Name varchar(100),
@LotNumber varchar(100),
@ProductNumber varchar(100),
@Manufacturer varchar(100)

AS
SET NOCOUNT ON

if not exists (SELECT ReagentId FROM LIMS_REAGENTS WHERE [Name]=@Name)
      INSERT INTO LIMS_REAGENTS ([Name], ProductNumber, Manufacturer)
      VALUES (@Name, @ProductNumber, @Manufacturer)
else
UPDATE LIMS_REAGENTS SET
      [Name] = @Name,
      ProductNumber = @ProductNumber,
      Manufacturer = @Manufacturer,
      WHERE [Name] = @Name

SET NOCOUNT OFF
GO
```

This Stored Procedure updates a table in the database and in this case you don’t normally need feedback, sp setting SET NOCOUNT ON at the top in the stored procedure is a good idea. it is also good practice to SET NOCOUNT OFF at the bottom of the stored procedure.

Functions

With SQL and SQL Server you can use lots of built-in functions or you may create your own functions. Here we will learn to use some of the most used built-in functions and in addition we will create our own function.

Built-in Functions

SQL has many built-in functions for performing calculations on data.

We have 2 categories of functions, namely **aggregate** functions and **scalar** functions. Aggregate functions return a single value, calculated from values in a column, while scalar functions return a single value, based on the input value.

Aggregate functions - examples:

- **AVG()** - Returns the average value
- **STDEV()** - Returns the standard deviation value
- **COUNT()** - Returns the number of rows
- **MAX()** - Returns the largest value
- **MIN()** - Returns the smallest value
- **SUM()** - Returns the sum
- etc.

Scalar functions - examples:

- **UPPER()** - Converts a field to upper case
- **LOWER()** - Converts a field to lower case
- **LEN()** - Returns the length of a text field
- **ROUND()** - Rounds a numeric field to the number of decimals specified
- **GETDATE()** - Returns the current system date and time
- etc.

String Functions

Here are some useful functions used to manipulate with strings in SQL Server:

- CHAR
- CHARINDEX
- REPLACE
- SUBSTRING
- LEN
- REVERSE
- LEFT
- RIGHT
- LOWER
- UPPER
- LTRIM
- RTRIM

Read more about these functions in the SQL Server Help.

Date and Time Functions

Here are some useful Date and Time functions in SQL Server:

- DATEPART
- GETDATE
- DATEADD
- DATEDIFF
- DAY
- MONTH
- YEAR
- ISDATE

Read more about these functions in the SQL Server Help.

Mathematics and Statistics Functions

Here are some useful functions for mathematics and statistics in SQL Server:

- COUNT
- MIN, MAX
- COS, SIN, TAN
- SQRT
- STDEV
- MEAN
- AVG

Read more about these functions in the SQL Server Help.

AVG()

The AVG() function returns the average value of a numeric column.

Syntax:

```
SELECT AVG(column_name) FROM table_name
```

Example:

Given a GRADE table:

	Column Name	Data Type	Allow Nulls
🔑	GradeId	int	☐
	StudentId	int	☐
	CourseId	int	☐
	Grade	float	☐
	Comment	varchar(1000)	☑

We want to find the average grade for a specific student:

```
select AVG(Grade) as AvgGrade from GRADE where StudentId=1
```

	AvgGrade
1	4,5

COUNT()

The COUNT() function returns the number of rows that matches a specified criteria.

The COUNT(column_name) function returns the number of values (NULL values will not be counted) of the specified column:

```
SELECT COUNT(column_name) FROM table_name
```

The COUNT(*) function returns the number of records in a table:

```
SELECT COUNT(*) FROM table_name
```

We use the CUSTOMER table as an example:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

```
select COUNT(*) as NumbersofCustomers from CUSTOMER
```

	NumberofCustomers
1	3

The GROUP BY Statement

Aggregate functions often need an added GROUP BY statement.

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns.

Syntax

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
```

Example:

We use the CUSTOMER table as an example:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	12	California	11111111
2	2	1001	Jackson	Smith	45	London	22222222
3	3	1002	Johnsen	John	32	London	33333333

If we try the following:

```
select FirstName, MAX(AreaCode) from CUSTOMER
```

We get the following error message:

```
Column 'CUSTOMER.FirstName' is invalid in the select list because it is not contained in either
an aggregate function or the GROUP BY clause.
```

The solution is to use the GROUP BY:

```
select FirstName, MAX(AreaCode) from CUSTOMER
group by FirstName
```

	FirstName	(No column name)
1	John	32
2	Smith	45

The HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

Syntax:

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value
```

We use the GRADE table as an example:

```
select * from GRADE
```

	GradeId	StudentId	CourseId	Grade	Comment
1	1	1	1	4	NULL
2	2	2	1	5	NULL
3	3	3	3	0	NULL
4	4	4	3	3	NULL
5	5	1	3	5	NULL

First we use the GROUP BY statement:

```
select CourseId, AVG(Grade) from GRADE
group by CourseId
```

	CourseId	(No column name)
1	1	4,5
2	3	2,66666666666667

While the following query:

```
select CourseId, AVG(Grade) from GRADE
group by CourseId
having AVG(Grade)>3
```

	CourseId	(No column name)
1	1	4,5

User-defined Functions

IN SQL we may also create our own functions, so-called user-defined functions.

A user-defined function is a routine that accepts parameters, performs an action, such as a complex calculation, and returns the result of that action as a value. The return value can either be a scalar (single) value or a table. Use this statement to create a reusable routine that can be used in other queries.

In SQL databases, a user-defined function provides a mechanism for extending the functionality of the database server by adding a function that can be evaluated in SQL statements. The SQL standard distinguishes between scalar and table functions. A scalar function returns only a single value (or NULL), whereas a table function returns a (relational) table comprising zero or more rows, each row with one or more columns.

Stored Procedures vs. Functions:

- Only functions can return a value (using the RETURN keyword).
- Stored procedures can use RETURN keyword but without any value being passed[1]
- Functions could be used in SELECT statements, provided they don't do any data manipulation and also should not have any OUT or IN OUT parameters.
- Functions must return a value, but for stored procedures this is not compulsory.
- A function can have only IN parameters, while stored procedures may have OUT or IN OUT parameters.
- A function is a subprogram written to perform certain computations and return a single value.
- A stored procedure is a subprogram written to perform a set of actions, and can return multiple values using the OUT parameter or return no value at all.

User-defined functions in SQL are declared using the **CREATE FUNCTION** statement.

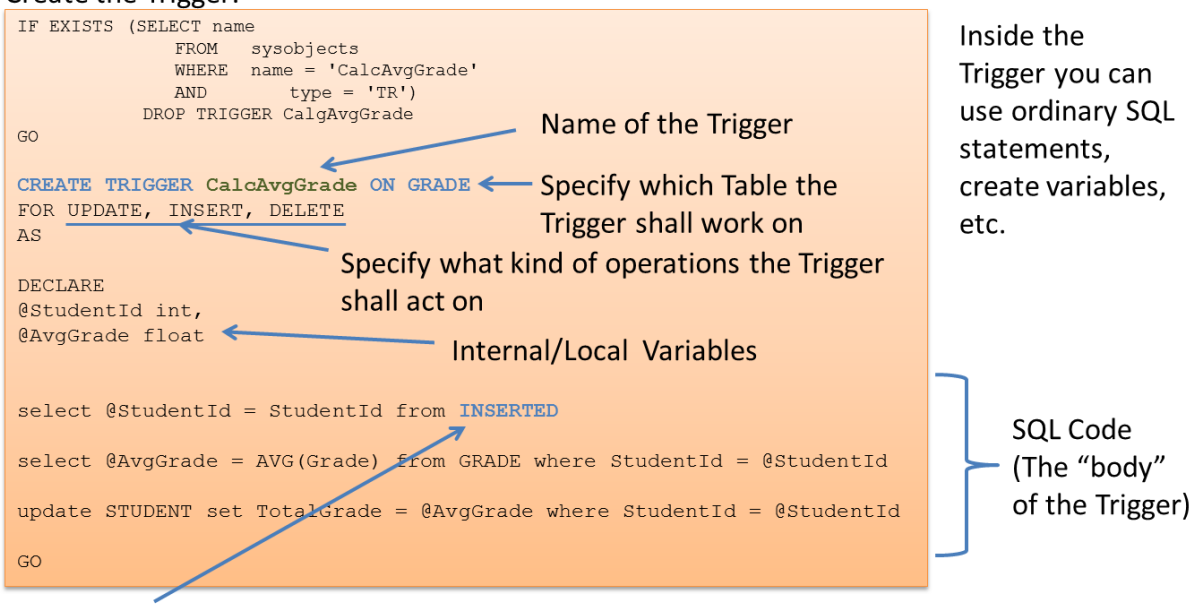
When we have created the function, we can use the function the same way we use built-in functions.

Triggers

A database trigger is code that is automatically executed in response to certain events on a particular table in a database.

A Trigger is executed when you insert, update or delete data in a Table specified in the Trigger.

Create the Trigger:



Note! "INSERTED" is a temporarily table containing the latest inserted data, and it is very handy to use inside a trigger

Syntax for creating a Trigger:

```
CREATE TRIGGER <TriggerName> on <TableName>
FOR INSERT, UPDATE, DELETE
AS
… Create your Code here
GO
```

The Trigger will automatically be executed when data is inserted, updated or deleted in the table as specified in the Trigger header.

INSERTED and DELETED:

Inside triggers we can use two special tables: the DELETED table and the INSERTED tables. SQL Server automatically creates and manages these tables. You can use these temporary,

memory-resident tables to test the effects of certain data modifications. You cannot modify the data in these tables.

The DELETED table stores copies of the affected rows during DELETE and UPDATE statements. During the execution of a DELETE or UPDATE statement, rows are deleted from the trigger table and transferred to the DELETED table.

The INSERTED table stores copies of the affected rows during INSERT and UPDATE statements. During an insert or update transaction, new rows are added to both the INSERTED table and the trigger table. The rows in the INSERTED table are copies of the new rows in the trigger table.

Example:

We will use the CUSTOMER table as an example:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	20	California	11111111
2	2	1001	Jackson	Smith	53	London	22222222
3	3	1002	Johnsen	John	40	London	33333333

We will create a TRIGGER that will check if the Phone number is valid when we insert or update data in the CUSTOMER table. The validation check will be very simple, i.e., we will check if the Phone number is less than 8 digits (which is normal length in Norway). If the Phone number is less than 8 digits, the following message “Phone Number is not valid” be written in place of the wrong number in the Phone column.

The TRIGGER becomes something like this:

```
IF EXISTS (SELECT name
        FROM   sysobjects
        WHERE  name = 'CheckPhoneNumber'
        AND       type = 'TR')
     DROP TRIGGER CheckPhoneNumber
GO

CREATE TRIGGER CheckPhoneNumber ON CUSTOMER
FOR UPDATE, INSERT
AS

DECLARE
@CustomerId int,
@Phone varchar(50),
@Message varchar(50)

set nocount on

select @CustomerId = CustomerId from INSERTED

select @Phone = Phone from INSERTED
```

```
set @Message = 'Phone Number ' + @Phone + ' is not valid'


if len(@Phone) < 8 --Check if Phone Number have less than 8 digits
      update CUSTOMER set Phone = @Message where CustomerId = @CustomerId

set nocount off

GO
```

We test the TRIGGER with the following INSERT INTO statement:

```
INSERT INTO CUSTOMER
(CustomerNumber, LastName, FirstName, AreaCode, Address, Phone)

VALUES
('1003', 'Obama', 'Barak', 51, 'Nevada', '4444')
```

The results become:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	20	California	11111111
2	2	1001	Jackson	Smith	53	London	22222222
3	3	1002	Johnsen	John	40	London	33333333
4	6	1003	Obama	Barak	51	Nevada	Phone Number 4444 is not valid

As you can see, the TRIGGER works as expected.

We try to update the Phone number to a valid number:

```
update CUSTOMER set Phone = '44444444' where CustomerNumber = '1003'
```

The results become:

	CustomerId	CustomerNumber	LastName	FirstName	AreaCode	Address	Phone
1	1	1000	Smith	John	20	California	11111111
2	2	1001	Jackson	Smith	53	London	22222222
3	3	1002	Johnsen	John	40	London	33333333
4	6	1003	Obama	Barak	51	Nevada	44444444

Communicate from other Applications

A Database is a structured way to store lots of information. The information is stored in different tables. “Everything” today is stored in databases.

Examples:

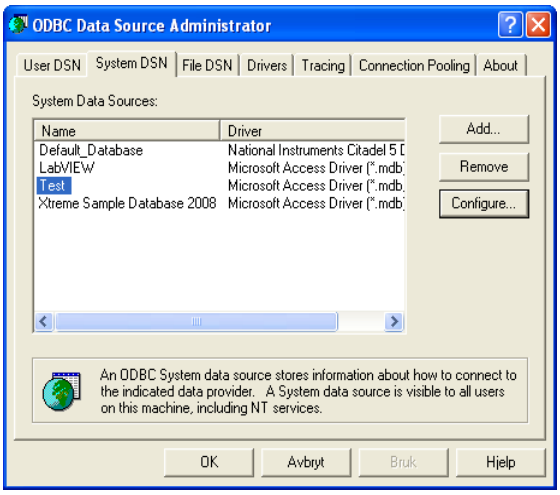
- Bank/Account systems
- Information in Web pages such as Facebook, Wikipedia, YouTube
- … lots of other examples

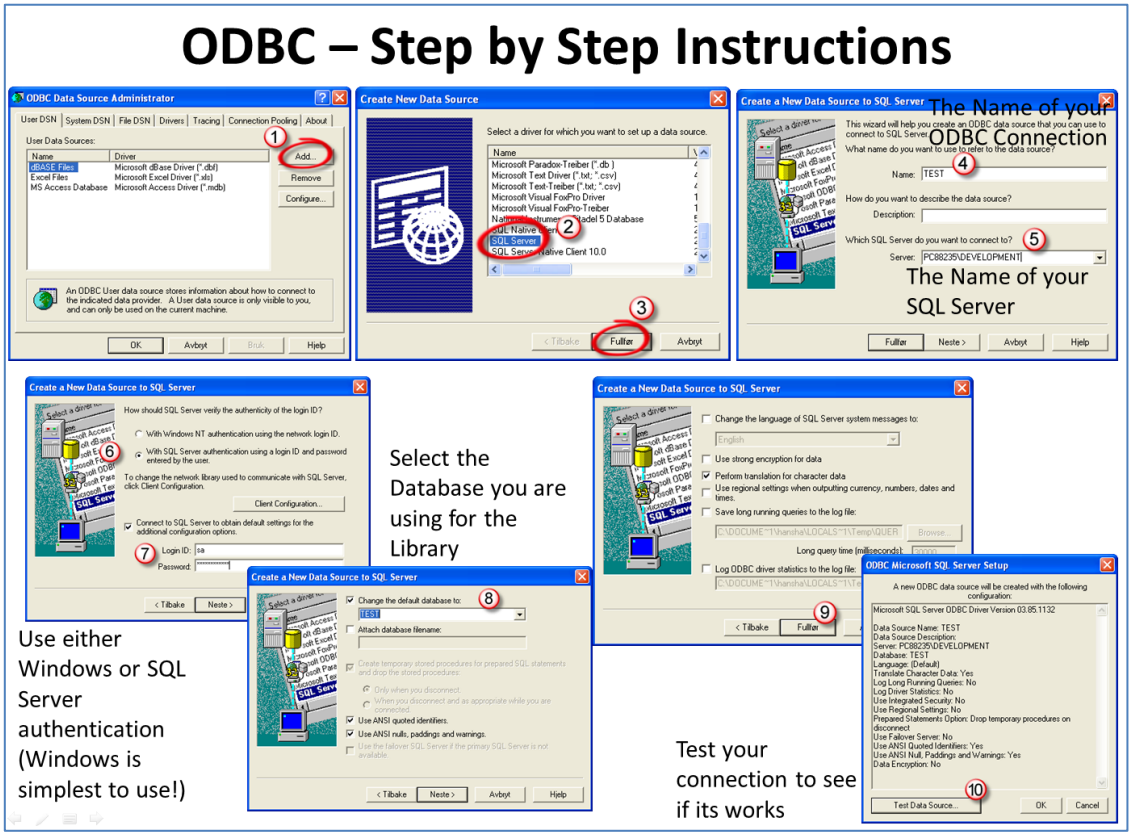
This means we need to be able to communicate with the database from other applications and programming languages in order to insert, update or retrieve data from the database.

ODBC

ODBC (Open Database Connectivity) is a standardized interface (API) for accessing the database from a client. You can use this standard to communicate with databases from different vendors, such as Oracle, SQL Server, etc. The designers of ODBC aimed to make it independent of programming languages, database systems, and operating systems.

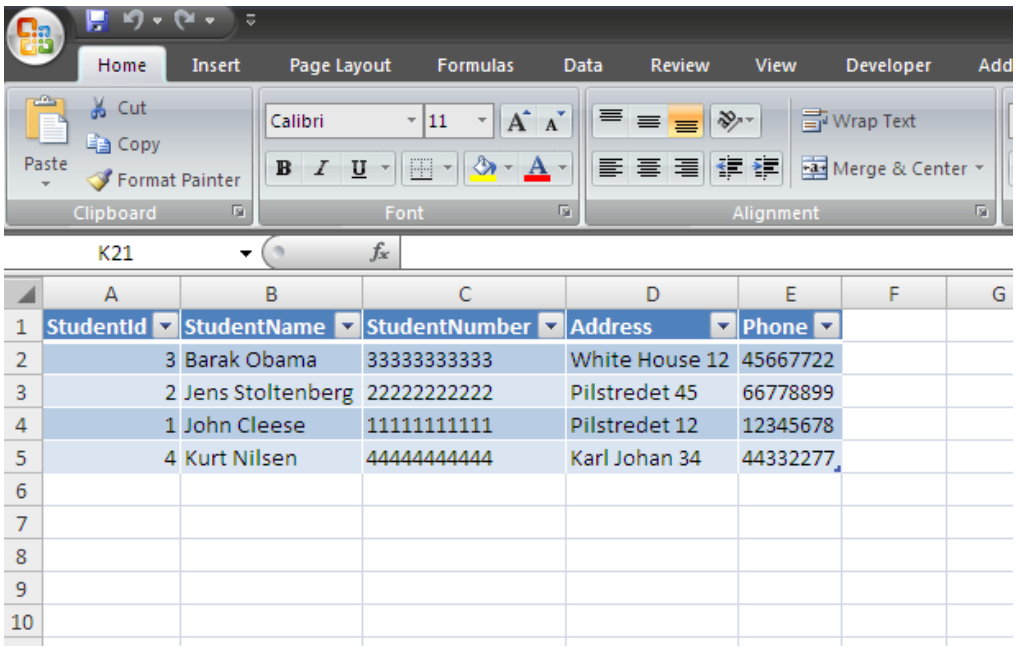
We will use the ODBC Data Source Administrator:



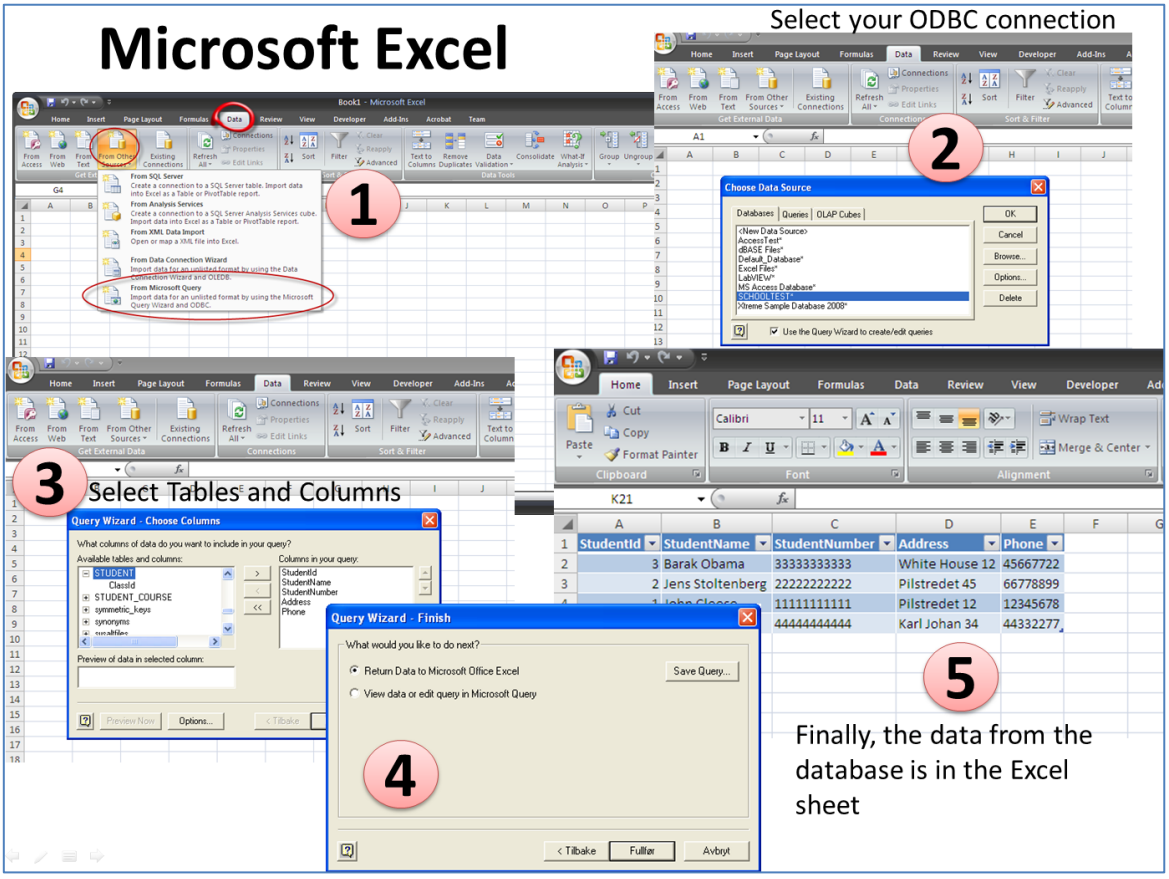


Microsoft Excel

Microsoft Excel has the ability to retrieve data from different data sources, including different database systems. It is very simple to retrieve data from SQL Server into Excel since Excel and SQL Server has the same vendor (Microsoft).



	A	B	C	D	E
1	StudentId	StudentName	StudentNumber	Address	Phone
2	3	Barak Obama	33333333333	White House 12	45667722
3	2	Jens Stoltenberg	22222222222	Pilstredet 45	66778899
4	1	John Cleese	11111111111	Pilstredet 12	12345678
5	4	Kurt Nilsen	44444444444	Karl Johan 34	44332277


Microsoft Excel
1
Select your ODBC connection
2
Choose Data Source
3
Select Tables and Columns
Query Wizard - Choose Columns
4
Query Wizard - Finish
What would you like to do next?
Return Data to Microsoft Office Excel
View data or edit query in Microsoft Query
5
StudentId StudentName StudentNumber Address Phone
3 Barak Obama 33333333333 White House 12 45667722
2 Jens Stoltenberg 22222222222 Pilstredet 45 66778899
11111111111 Pilstredet 12 12345678
44444444444 Karl Johan 34 44332277
Finally, the data from the database is in the Excel sheet

References

Microsoft official SQL Server Web site - http://www.microsoft.com/sqlserver

SQL Server Books Online - http://msdn.microsoft.com/en-us/library/ms166020.aspx

SQL Server Help

w3shools.com - http://www.w3schools.com/sql

Wikipedia – Microsoft SQL Server - http://en.wikipedia.org/wiki/Microsoft_SQL_Server

Wikipedia - SQL - http://en.wikipedia.org/wiki/SQL

Wikipedia – Transact SQL - http://en.wikipedia.org/wiki/T-SQL

Høgskolen i Telemark

Telemark University College

Faculty of Technology

Kjølnes Ring 56

N-3914 Porsgrunn, Norway

www.hit.no

Hans-Petter Halvorsen, M.Sc.

Telemark University College

Faculty of Technology

Department of Electrical Engineering, Information Technology and Cybernetics

E-mail: hans.p.halvorsen@hit.no

Blog: http://home.hit.no/~hansha/