```
/****************/
/* SQL SERVER   */
/*    INDEX     */
/****************/

--2-- CREATE, ALTER, AND DROP A DATABASE
--3-- CREATE TABLES AND ENFORCE CONSTRAINTS
--4-- ADDING DEFAULT CONSTRAINTS
--5-- CASCADING REFERENTIAL INTEGRITY CONSTRAINTS
--6-- ADDING A CHECK CONSTRAINT
--7-- IDENTITY COLUMN
--8-- RETRIEVE THE LAST GENERATED IDENTITY COLUMN VALUE
--9-- UNIQUE KEY CONSTRAINT
--10-- ALL ABOUT SELECT
--11-- GROUP BY, HAVING
--12-- JOIN STATEMENTS
--13-- INTELLIGENT JOINS
--14-- SELF JOINS
--15-- REPLACING NULLS
--16-- COALESCE FUNCTION
--17-- UNION AND UNION ALL
--18-- STORED PROCEDURES
--19-- STORED PROCEDURES WITH OUTPUT PARAMETERS
--20-- STORED PROCEDURES RETURN VALUES AND OUTPUT PARAMETERS
--21-- ADVANTAGES OF STORED PROCEDURES
--22-- BUILT IN STRING FUNCTIONS (SYSTEM FUNCTIONS)
--23-- BUILT IN STRING FUNCTIONS (SYSTEM FUNCTIONS) CONT.


/***************/
/* SQL SERVER */
/***************/


-------------------------------------------------
-- 2 -- CREATE, ALTER AND DROP A DATABASE  -- 2 --
-------------------------------------------------

CREATE DATABASE Sample2
-- MDF FILE IS THE DATA FILE, LDF IS THE TRANSACTION LOG FILE

-- TO CHANGE THE NAME OF THE DATABASE:
ALTER DATABASE Sample2 MODIFY NAME = Sample3

-- CHANGE A DATABASE NAME VIA SYSTEM DEFINED STORED PROCEURE
sp_renameDB 'Sample3'.'Sample4'

-- DROP A DATABASE, THAT IS DELETE IT COMPLETELY
DROP DATABASE Sample4
```

```
-- You cannot drop a database if it is being used
-- Bring it to single user mode if people are connected to db
-- you do that using the ALTER command
ALTER DATABASE Sample4 SET SINGLE_USER WITH ROLLBACK IMMEDIATE
-- which will rollback any incomplete transactions and closes the connection
-- to the database

-------------------------------------------------------
-- 3 -- CREATE TABLES AND ENFORCE CONSTRAINTS  -- 3 --
-------------------------------------------------------

-- Primary key used to identify uniquely each record in the table
-- Primary key is an integer data type that does not allow nulls

USE [Sample]
GO
-- here we are saying use the database named sample
CREATE TABLE tblGender
( id INT NOT NULL PRIMARY KEY,
  gender NVARCHAR(50) NOT NULL
);
-- not null does not allow nulls in that column

-- let's mark a constraint on a table... that is admit only certain values
-- a foreign key constraint
ALTER TABLE tblPerson ADD CONSTRAINT tblPerson_Gender_FK
FOREIGN KEY (GenderId) REFERENCES tblGender(ID)
-- the foreign key statement says, make GenderId in tblPerson table have
-- a foreign key, that references the primary key of the id column of tblGender
-- the primary key in tblGender does not necessarily need to coincide with the
-- tblPerson(GenderId) column values. It just means that we are referring to
-- that record

-- a foreign key in one table references a primary key in another table
-- and a foreign key constraint prevents invalid data to be inserted, updated, etc

-----------------------------------------
-- 4 -- ADDING DEFAULT CONSTRAINTS  -- 4 --
-----------------------------------------

SELECT * FROM tblGender
SELECT * FROM tblPerson

INSERT INTO tblPerson (ID, Name, Email) VALUES (7, 'Rich', 'r@r')
-- if we don't know the gender of the person, I don't want NULL to be inserted
-- i want a value, a default value to be inserted, for this, we add default constraints

ALTER TABLE tblperson
ADD CONSTRAINT DF_tblPerson_GenderId -- just by looking at the constraint name
```

```
-- we can tell what table and field it refers to
DEFAULT 3 FOR GenderId

-- now when we insert a new record, genderId will be populated with 3 if nothing
-- else is specified
INSERT INTO tblPerson (ID, Name, Email) VALUES (8, 'Mike', 'm@m')

-- but if we supply a value for genderId
INSERT INTO tblPerson (ID, Name, Email, GenderId) VALUES (9, 'Sarah', 's@s', 1)
-- then 3 will not be inserted into genderId

-- if you pass NULL it will not take the default 3, it will take NULL
INSERT INTO tblPerson (ID, Name, Email, GenderId) VALUES (10, 'Juan', 'j@j', NULL)

-- Adding a new column with default value to an existing table
ALTER TABLE (TABLE_NAME)
ADD (COLUMN_NAME) (DATA_TYPE) (NULL | NOT NULL)
CONSTRAINT (CONSTRAINT_NAME) DEFAULT (DEFAULT_VALUE)

-- Dropping a constraint
ALTER TABLE tblPerson
DROP CONSTRAINT DF_tblPerson_GenderId

------------------------------------------------------------
-- 5 -- CASCADINIG REFERENTIAL INTEGRITY CONSTRAINTS  -- 5 --
------------------------------------------------------------

-- you can't delete records that will compromise the integrity of the database
-- you can't delete rows for which there are references

-- Options when setting up cascading referential integrity constraints

-- If an attempt is made to delete or update a row with a key
-- referenced by foreign keys in existing rows in other tables,

--1) NO ACTION: an error is raised and the DELETE AND UPDATE is rolled back
--2) CASCADE: ... all rows containing those foreign keys are also deleted or updated
--3) SET NULL: ... all rows containing those foreign keys are set to NULL
--4) SET DEFAULT: ... all rows containing those foreign keys are set to default values

-----------------------------------------
-- 6 -- ADDING A CHECK CONSTRAINT   -- 6 --
-----------------------------------------

-- a checked constraint limits the range of values that can be entered in a column
-- for example if we want age to take values between 0 and 150

INSERT INTO tblperson VALUES(4, 'Sarah', 's@s.com', 2, -970)
-- this will accept an age of -970...
```

```
-- it is possible for us to define a range of numbers that are allowed in age

-- a constraint is a boolean expression that returns true or false
-- CK is a prefix fo checked constraint we use when naming a checked constraint
DELETE FROM tblPerson WHERE Id = 4

-- now we implement a CK, a checked constraint
ALTER TABLE tblPerson
ADD CONSTRAINT CK_tblperson_Age CHECK (Age > 0 AND Age < 150)

-- now if we try to insert the new row, we'll fail
-- because the boolean expression in parenthesis will return false
INSERT INTO tblperson VALUES(4, 'Sarah', 's@s.com', 2, -970)

-- if I pass NULL instead of -970, NULL is inserted and the constraint is ignored

-- how to drop a constraint
ALTER TABLE tblPerson
DROP CONSTRAINT CK_tableperson_Age

-----------------------------------------
-- 7 --       IDENTITY COLUMN       -- 7 --
-----------------------------------------

-- If a column is not an identity column, in sql server you will need to supply
-- a value for it. The column id is something internal, so we make use of the
-- identity column. If you mark a column as identity column, you do not need
-- to supply a value for it. You want the value to be automatically computed
-- when you insert a row. The identity seed is what the starting value for that
-- column should be, and identity increment is by how many units you want to
-- increment each new record every time you enter a new record.
-- You specify these in column properties.

INSERT INTO dbo.tblperson1 values ('John')
-- this will put the Personid at 1 for record of Name John, this is because
-- we marked personid as an identity column

DELETE FROM tblperson1 WHERE PersonId = 1
-- once deleted, if you insert a new (fourth) record, it will give it an id of 4
-- not 1, even if you deleted PersonId = 1.
-- However, if you want to reuse the PersonId = 1, you can't just insert and pass
-- Personid = 1.

-- First you need to turn INDENTITY_INSERT on
SET IDENTITY INSERT tblPerson1 on
-- This let's the db know that we can insert identity insert feature
-- which forces you to need to explicitly specify the id
-- This allows you to fill the gaps of deleted records PersonId
```

```
INSERT INTO dbo.tblperson1 (PersonId, Name) VALUES (1, 'Jane')
-- Now yes, you can insert the record of Jane with a value of 1 in PersonId

DELETE FROM dbo.tblPerson1
-- deletes all rows from tblPerson1 table
-- if you insert a new row, the identity value will NOT go back to zero

-- this will start the PersonId seed starting at 0
DBCC CHECKIDENT('tblPerson1', RESEED, 0)

INSERT INTO dbo.tblperson1 values ('Martin')
-- Martin will have a PersonId of 1 now.

------------------------------------------------------------------------
-- 8 -- RETRIEVE THE LAST GENERATED IDENTITY COLUMN VALUE       -- 8 --
------------------------------------------------------------------------

-- lets create tables
CREATE TABLE Test1
(
ID INT IDENTITY(1,1), -- seed 1, increment 1
Value NVARCHAR(20)
)

CREATE TABLE Test2
(
ID INT IDENTITY(1,1),
Value NVARCHAR(20)
)

-- insert values
INSERT INTO Test1 VALUES ('X')

SELECT * FROM Test1

-- use the SCOPE_IDENTITY function

SELECT SCOPE_IDENTITY()
-- returns the last generated identity column value
-- returns the last identity value in the same session/connection
-- and within the same scope

SELECT @@IDENTITY
-- returns the last generated identity column value
-- returns the last identity value in the same session/connection
-- but across any scope

-- create a trigger, whenever someone inserts a row into Table1, we
-- want a row to be inserted into Table2
```

```
CREATE TRIGGER trForInsert ON Test1 FOR INSERT
AS
BEGIN
	INSERT INTO Test2 VALUES('YYYY')
END

-- insert values
INSERT INTO Test1 VALUES ('X')
-- this will now insert a row also into Test2 table

SELECT SCOPE_IDENTITY()
-- will give you identity value 3 (from Test1 table)
-- so within the same scope

SELECT @@IDENTITY
-- will give you identity value 1 (from Test2 table)
-- but across any scope


-- In one Session (user 1 session):
INSERT INTO Test2 VALUES('xxx')

-- In another Session (user 2 session):
INSERT into Test2 VALUES('yyy')

SELECT SCOPE_IDENTITY() -- returns 2
SELECT @@IDENTITY -- returns 2
SELECT IDENT_CURRENT('Test2') -- retunrs 3
-- will give you the last identity across any session and any scope
-- so this will return 3, while the first two will return lesser values

-------------------------------------
-- 9 -- UNIQUE KEY CONSTRAINT -- 9 --
-------------------------------------

-- A table can havre only one primary key, but more than one UNIQUE key.
-- Primary keys do not allow NULLS, whereas UNIQUE keys allow one NULL

-- Both primary key and unique key are used to enforce the uniqueness of a
-- column. So when do you choose one over the other?
-- A table can have only one primary key. If you want to enforce uniqueness
-- on 2 or more columns then we use unique key constraint

ALTER TABLE tblPerson
ADD CONSTRAINT UQ_tblPerson_Email UNIQUE(Email)
-- this just added a UNIQUE constraint to the column Email in tblPerson table.

INSERT INTO tblPerson VALUES (2, 'XYZ', 'a@a.com', 1, 20)
```

```
INSERT INTO tblPerson VALUES (2, 'XYZ', 'a@a.com', 1, 20)
-- will yield error message

ALTER TABLE tblPerson
DROP CONSTRAINT UQ_tblPerson_Email
-- this will drop the constraint

INSERT INTO tblPerson VALUES (2, 'XYZ', 'a@a.com', 1, 20)
INSERT INTO tblPerson VALUES (2, 'XYZ', 'a@a.com', 1, 20)
-- will NOT yield an error

---------------------------------
-- 10 -- ALL ABOUT SELECT -- 10 --
---------------------------------

SELECT * FROM tblPerson

SELECT [id], [Name], [Email], [GenderId], [Age], [City]
FROM [Sample].[dbo].[tblPerson]

SELECT DISTINCT City FROM tblPerson -- returns only distinct cities, not repeated

SELECT DISTINCT Name, City FROM tblPerson -- returns values should be distinct across
-- two columns, the name of row one can't be the same as the city in that row

SELECT * FROM tblPerson
WHERE City = 'London' -- retunrs the two records of people living in London

SELECT * FROM tblPerson
WHERE City <> 'London'

SELECT * FROM tblPerson
WHERE City != 'London' -- same as previous statement

SELECT * FROM tblPerson
WHERE Age = 20 OR Age = 23 OR Age = 29

SELECT * FROM tblPerson
WHERE Age IN (20,23,29)

SELECT * FROM tblPerson
WHERE Age BETWEEN 20 AND 25 -- boundaries or range are inclusive

SELECT * FROM tblPerson
WHERE City LIKE 'l%' -- starts with l, after l it can have any characters

SELECT * FROM tblPerson
WHERE Email LIKE '%@%' -- field should have an at somewher
```

```
SELECT * FROM tblPerson
WHERE Email NOT LIKE '%@%' -- field should have an at somewher

-- the underscore is also a wildcard but replaces only one character

SELECT * FROM tblPerson
WHERE Email LIKE '-@-' -- field should have one and only one character
-- after and before the at symbol

-- square brackets specify a list of characters
SELECT * FROM tblPerson
WHERE Name LIKE '[MST]%' -- name value should start with m, s or t and then
-- followed by any number of characters

SELECT * FROM tblPerson
WHERE Name LIKE '[^MST]%' -- name value should NOT start with m, s or t, and
-- the followed by any number of characters

SELECT * FROM tblPerson
WHERE (City = 'London' OR City = 'Mumbai') AND Age > 25
-- will chose people who live in London or Mumbai AND whose age is
-- greater than 25

SELECT * FROM tblPerson
ORDER BY Name
-- will sort results by Name in Ascending order (by default)

SELECT * FROM tblPerson
ORDER BY Name DESC

SELECT * FROM tblPerson
ORDER BY Name DESC, Age ASC
-- sort by name descending order and then by age ascending

SELECT TOP 10 * FROM tblPerson
-- returns first 10 records from tblPerson table

SELECT TOP 1% * FROM tblPerson
-- also can specify the percentage of records you want to return

-- How do you find the top salary of an employee, or the olders person

SELECT TOP 1 * FROM tblPerson
ORDER BY Age by DESC
-- will give me the oldest person in the table

---------------------------------
-- 11 -- GROUP BY, HAVING -- 11 --
---------------------------------
```

```
SELECT SUM(Salary) FROM tblEmployee
-- will give you the total salary from the records in tblEmployee

SELECT MIN(Salary) FROM tblEmployee
-- will give you the lowest salary in the table

SELECT City, SUM(Salary) AS TotalSalary FROM tblEmployee
-- AS gives an alias to the sum of salaries by city
GROUP BY City
-- this will return records of grouped cities with aggregate salaries
-- for that city

-- if you ommit GROUP BY you will get an error because the table
-- will not make sense

SELECT City, Gender, SUM(Salary) AS TotalSalary FROM tblEmployee
-- AS gives an alias to the sum of salaries by city
GROUP BY City, Gender
ORDER BY City

SELECT Gender, City, SUM(Salary) AS TotalSalary FROM tblEmployee
GROUP BY Gender, City
ORDER BY Gender, City

SELECT COUNT(ID) from tblEmployee

SELECT Gender, City, SUM(Salary) AS TotalSalary, COUNT(ID) AS [Total Employees]
-- you use the square brackets to add spaces to aliases
FROM tblEmployee
GROUP BY Gender, City

-- to filter rows we use the where clause
SELECT Gender, City, SUM(Salary) AS TotalSalary, COUNT(ID) AS [Total Employees]
-- you use the square brackets to add spaces to aliases
FROM tblEmployee
WHERE Gender = 'Male'
GROUP BY Gender, City
-- will show only male employees total salaries and total number of employees

SELECT Gender, City, SUM(Salary) AS TotalSalary, COUNT(ID) AS [Total Employees]
-- you use the square brackets to add spaces to aliases
FROM tblEmployee
GROUP BY Gender, City
HAVING Gender = 'Male'
-- HAVING goes after GROUP BY...
-- using where instead of having, filters non males
-- using having instead of where, aggregates male records, then filters
-- the output is the same in the two scenarios
```

```
-- DIFFERENCE BETWEEN WHERE AND HAVING

-- you can use HAVING clause with SELECT statement, where can be used outside of SELECT
-- aggregate functions can only be used with HAVING clause and not WHERE

SELECT Gender, City, SUM(Salary) AS TotalSalary, COUNT(ID) AS [Total Employees]
FROM tblEmployee
GROUP BY Gender, City
HAVING SUM(Salary) > 5000
-- this we could not do with where

---------------------------------
-- 12 -- JOIN STATEMENTS -- 12 --
---------------------------------

-- joins are used to retrieve data from 2 or more related tables
-- inner join will return only the rows that have a match in the second table
-- inner join will only give you matching rows

SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
JOIN tblDepartment ON tblEmployee.DepartmentId = tblDepartment.Id
-- OR
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
INNER JOIN tblDepartment ON tblEmployee.DepartmentId = tblDepartment.Id
-- only returns matching records in two tables

-- i want all the employees, everything on the left table - matching and non matching
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
LEFT JOIN tblDepartment ON tblEmployee.DepartmentId = tblDepartment.Id
-- will return two employees with non matching department ids as well
-- OR
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
LEFT OUTTER JOIN tblDepartment ON tblEmployee.DepartmentId = tblDepartment.Id

SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
RIGHT JOIN tblDepartment ON tblEmployee.DepartmentId = tblDepartment.Id
-- will return a row for a department for which there are no employees with
-- that department assigned on table 1.
-- OR
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
RIGHT OUTTER JOIN tblDepartment ON tblEmployee.DepartmentId = tblDepartment.Id
-- the Name, Gender and Salary will be assigned NULL for the departemnt
```

```
-- for which there are no employees in table 1

SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
FULL JOIN tblDepartment ON tblEmployee.DepartmentId = tblDepartment.Id
-- OR
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
FULL OUTTER JOIN tblDepartment ON tblEmployee.DepartmentId = tblDepartment.Id
-- will give me all the records from table1 and table2
-- the columns for which there is no data will be assigned null

-- A CROSS JOIN
-- it will take each record from the employee table and associate it
-- with every record in the department table
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
CROSS JOIN tblDepartment
-- a cross join will not have an ON clause
-- the number of rows in the employees table is being multiplied by the
-- department tables... it is the cartesian product of the two tables
-- involved in the join


-----------------------------------
-- 13 -- INTELLIGENT JOINS -- 13 --
-----------------------------------

-- suppose you want only non matching rows from left or right table, or both

SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
LEFT JOIN tblDepartment ON tblEmployee.DepartmentID = tblDepartment.Id
WHERE tblEmployee.DepartmentID IS NULL -- never use = NULL in SQL SERVER
-- OR
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
LEFT JOIN tblDepartment ON tblEmployee.DepartmentID = tblDepartment.Id
WHERE tblDepartment.ID IS NULL
-- will return employees for which there is no department assigned in table2
-- (tblDepartment)

SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
RIGHT JOIN tblDepartment ON tblEmployee.DepartmentID = tblDepartment.Id
WHERE tblEmployee.DepartmentID IS NULL
-- OR
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
```

```
RIGHT JOIN tblDepartment ON tblEmployee.DepartmentID = tblDepartment.Id
WHERE tblDepartment.ID IS NULL
-- will return the department row for which there are no corresponding
-- employees

SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
FULL JOIN tblDepartment ON tblEmployee.DepartmentID = tblDepartment.Id
WHERE tblEmployee.DepartmentID IS NULL OR tblDepartment.ID IS NULL
-- will return the employees for which there is no department assigned and
-- the departments for which there are no employees assigned

--------------------------
-- 14 -- SELF JOIN -- 14 --
--------------------------

-- Self join is a join with itself
-- it is not a different type of join fromt the previous
-- if you have a table where a column value "Manager" (int) refers to the
-- id column of the same table, we are effectively referring the same table.
-- we have a need to join employee table with itself

-- Inner Self Join
SELECT E.Name AS EmployeeName, M.Name AS ManagerName
FROM tblEmployee E
INNER JOIN tblEmployee M ON E.tblEmloyeeID = M.tblEmployeeID

-- Left Outter Self Join
SELECT E.Name AS EmployeeName, M.Name AS ManagerName
FROM tblEmployee E
LEFT JOIN tblEmployee M ON E.tblEmloyeeID = M.tblEmployeeID
-- well get employees that have no manager assigned

-- Right Outter Join
SELECT E.Name AS EmployeeName, M.Name AS ManagerName
FROM tblEmployee E
RIGHT JOIN tblEmployee M ON E.tblEmloyeeID = M.tblEmployeeID
-- well get managers that have no employees assigned

-- Cross Self Join
SELECT E.Name AS EmployeeName, M.Name AS ManagerName
FROM tblEmployee E
CROSS JOIN tblEmployee M ON E.tblEmloyeeID = M.tblEmployeeID
-- will get all combinations of employees and managers


-------------------------------
-- 15 -- REPLACING NULL -- 15 --
-------------------------------
```

```
-- instead of NULL, i want output to say "No manager"
-- SELECT ISNULL(NULL, "No Manager") AS ManagerName
-- if the first parameter evaluates to null, the second parameter will
-- be used, so we could do

SELECT E.Name EmployeeName, ISNULL(M.Name, "No Manager") AS ManagerName
FROM tblEmployee E
LEFT JOIN tblEmployee M ON E.tblEmloyeeID = M.tblEmployeeID

-- COALESCE FUNCTION
-- another way to replace null
-- COALESCE returns the first non null value
-- SELECT COALESCE(NULL, "No Manger")
-- you can pass multiple expression and it returns the first non null value.

SELECT E.Name EmployeeName, COALESCE(M.Name, "No Manager") AS ManagerName
FROM tblEmployee E
LEFT JOIN tblEmployee M ON E.tblEmloyeeID = M.tblEmployeeID

-- CASE STATEMENT
-- CASE WHEN Expression THEN " ELSE "
-- If expression returns true return what follows then, if not return else

SELECT E.Name EmployeeName,
        CASE WHEN M.Name IS NULL
                THEN 'No Manager'
                ELSE M.Name
                END
        AS ManagerName
FROM tblEmployee E
LEFT JOIN tblEmployee M ON E.tblEmloyeeID = M.tblEmployeeID

-----------------------------------
-- 16 -- COALESCE FUNCTION -- 16 --
-----------------------------------

-- COALESCE returns the first non null value
SELECT Id, COALESCE(FirstName,MiddleName,LastName) AS Name
FROM tblEmployee
-- will return FirstName if it is not null, if it is it will return
-- MiddleName if it is not null, if it is, it will return the LastName
-- if all arguments evaluate to NULL, then coalesce returns NULL


-------------------------------------
-- 17 -- UNION AND UNION ALL -- 17 --
-------------------------------------
```

```
-- Union and union all are used to combine the result sets of
-- two select queries

SELECT * FROM tblIndiaCustomers
UNION ALL
SELECT * FROM tblUKCustomers
-- returns unified results from two select statements
-- includes duplicate records
-- does not sort the ouput

SELECT * FROM tblIndiaCustomers
UNION
SELECT * FROM tblUKCustomers
-- returns unified results from two select statements
-- but removes duplicate records
-- sorts the output

-- Differences between UNION and UNION ALL
-- UNION is a bit slower, because it performs more operations
-- it uses a DISTINCT SORT

-- for UNION statements to work the data types, number of columns
-- has to be the same on both tables, the order of columns has to
-- be the same also
-- if you dont use the exact same order, it will try to do an implicit
-- conversion as long as the data type of the unordered columns are
-- the same

-- Sorting the results of UNION and UNION ALL
SELECT * FROM tblIndiaCustomers
UNION ALL
SELECT * FROM tblUKCustomers
ORDER BY Name
-- order by should be AFTER the UNION statements

-- Difference between UNIONS and JOINS
-- UNION combines select queries results including all rows
-- JOIN retrieves data from two or more tables related by foreign keys


-----------------------------------
-- 18 -- STORED PROCEDURES  -- 18 --
-----------------------------------

-- we wrap queries in stored procedures so we don't have to type
-- code repeated times, if we expect to use the procedure many
-- times

-- Create a Stored Procedure without Parameters:
```

```
CREATE PROCEDURE spGetEmployees -- also can write CREATE PROC
AS
BEGIN
	SELECT Name, Gender from tblEmployee
END

-- execute the procedure:
EXECUTE spGetEmployees -- also can write EXEC or FULL EXECUTE

-- Create a Stored Procedure with Parameters:
CREATE PROCEDURE spGetEmployeesByGenderAndDepartment
@Gender NVARCHAR(20),
@DepartmentId INT
AS
BEGIN
	SELECT Name, Gender, DepartmentId FROM tblEmployee
	WHERE Gender = @Gender AND DepartmentId = @DepartmentId
END

-- Execute the procedure:
EXECUTE spGetEmployeesByGenderAndDepartment
'Male', 1
-- here the parameters are passed in the right order, however:

-- will attempt implicit conversion if you pass the parameters
-- in the wrong order, if it tried to convert string to int
-- it will throw an exception

-- look at the definition of a stored procedure
sp_helptext dbo.spGetEmployees

-- We want to change the implementation of a Stored Procedures
ALTER PROCEDURE spGetEmployees -- also can write CREATE PROC
AS
BEGIN
	SELECT Name, Gender from tblEmployee
	ORDER by Name
END

-- To Drop a Procedure
DROP PROCEDURE spGetEmployees

-- To Encrypt the Text of a Stored Procedure

ALTER PROCEDURE spGetEmployeesByGenderAndDepartment
@Gender NVARCHAR(20),
@DepartmentId INT
WITH ENCRYPTION
AS
```

```
BEGIN
	SELECT Name, Gender, DepartmentId FROM tblEmployee
	WHERE Gender = @Gender AND DepartmentId = @DepartmentId
END
-- now if you want to view the definition, you will get an error

-- You can then delete the stored procedure

-------------------------------------------------------------
-- 19 -- STORED PROCEDURES WITH OUTPUT PARAMETERS  -- 19 --
-------------------------------------------------------------

CREATE PROCEDURE spGetEmployeeGetCountByGender
@Gender NVARCHAR(20),
@EmployeeCount INT OUTPUT
AS
BEGIN
	SELECT @EmployeeCount = COUNT(Id) FROM tblEmployee
	WHERE Gender = @Gender
END

-- excute the stored procedure:
DECLARE @TotalCount INT
EXECUTE spGetEmployeeGetCountByGender
'Male', @TotalCount OUPUT -- this line passes the male input parameter
-- and tells the execute statement to store @EmployeeCount value into
-- the @TotalCout output variable
IF (@TotalCount) IS NULL
		PRINT "@TotalCount IS NULL"
ELSE
		PRINT "@TotalCount IS NOT NULL"


-- if you specify which parameters you are passing you may alter
-- the order when you execute and set the input parameter values
DECLARE @TotalCount2 INT
EXECUTE spGetEmployeeGetCountByGender2
@EmployeeCount = @TotalCount OUT, -- you can use OUT for OUTPUT
@Gender = 'Male'
PRINT @TotalCount


-- Useful System Stored Procedures:

-- view information about stored procedures
-- including parameters expected and data types
sp_help spGetEmployeeGetCountByGender
-- you can use this also with TABLES, VIEWS, TRIGGERS, etc...
-- you will get information about the different objects
```

```
-- view the definition of a stored procedure
-- that is all the text of the query
sp_helptext spGetEmployeeGetCountByGender

-- view the dependencies of stored procedure
-- if it depends on a TABLE for example
-- where the stored procedure is getting the info from
sp_depends spGetEmployeeGetCountByGender
-- you can use this also with TABLES, VIEWS, TRIGGERS, etc...
-- you will get information about the different objects

-----------------------------------------------------------------------
-- 20 -- STORED PROCEDURES RETURN VALUES AND OUTPUT PARAMETERS  -- 20 --
-----------------------------------------------------------------------

-- A return value of 0 indicates success and non zero indicates failure
CREATE PROCEDURE spGetTotalCount1
@TotalCount INT OUT
AS
BEGIN
	SELECT @TotalCount = COUNT(Id) FROM tblEmployee
END

DELCARE @TotalEmployees INT
EXEC spGetTotalCount1
@TotalEmployees OUT
PRINT @TotalEmployees
-- this is one way to do it with output parameters

CREATE PROCEDURE spGetTotalCount2
@TotalCount INT OUT
AS
BEGIN
	RETURN (SELECT @TotalCount = COUNT(Id) FROM tblEmployee)
END

DECLARE @TotalEmployees INT
EXECUTE @TotalEmployees = spGetTotalCount2
PRINT @TotalEmployees
-- this procedure does the same as the previous, with return values




CREATE PROCEDURE spGetNameById1
@Id INT,
@Name NVARCHAR(20) OUTPUT
```

```
AS
BEGIN
	SELECT @Name = Name FROM tblEmployeee
	WHERE Id = @Id
END

-- execute the procedure

DECLARE @Name NVARCHAR(20)
EXECUTE spGetNameById1
1,
@Name OUT
PRINT "Name = " + @Name


-- we do the same using return values
-- but the next example won't work because
-- return values are integers and in the following example
-- we are trying to return a name
-- so an error message will be displayed
CREATE PROCEDURE spGetNameById2
@Id INT,
AS
BEGIN
	RETURN (SELECT Name FROM tblEmployeee
	WHERE Id = @Id)
END

-- execute the procedure
DECLARE @Name NVARCHAR(20)
EXECUTE spGetNameById2
@Name = spGetNameById2
1
PRINT "Name = " + @Name

-- Return values only return integer data types
-- Ouput parameter return any data type


-----------------------------------------------
-- 21 -- ADVANTAGES OF STORED PROCEDURES  -- 21 --
-----------------------------------------------
-- PERFORMANCE ADVANTAGES:
-- Execution plan retention and reusability
	-- Stored procedures cache the execution plan and reuse it
	-- if you execute queries without using parameters like in stored procedures
	-- the execution plan cannot be reused

-- Reduce network traffic
```

```
-- Code reusability and better maintainability
        -- You only have one place to change a stored procedure instead
        -- if it is inline sql every application where the statements
        -- are made have all to be changed. Maintenance becomes a nightmare

-- SECURITY ADVANTAGES:
-- Better Security
        -- You can grant access to stored procedure without granting access to
        -- a table itself. The same goes for views.

-- Avoid SQL Injection Attacks
        -- People can type in SQL and inject, not so with stored procedures

---------------------------------------------
-- 22 -- BUILT IN STRING FUNCTIONS  -- 22 --
---------------------------------------------

-- Commonly string functions
SELECT ASCII("A")
-- returns 65

SELECT ASCII("ABC")
-- returns the ascii value of the first character, 65

SELECT CHAR(65)
-- returns A

DECLARE @Start INT
SET @Start = 65
WHILE(@Start <= 90)
BEGIN
        PRINT CHAR(@Start)
        SET @Start = @Start + 1
END
-- will print A through Z

DECLARE @Start INT
SET @Start = 97
WHILE(@Start <= 122)
BEGIN
        PRINT CHAR(@Start)
        SET @Start = @Start + 1
END
-- will print a through z

DECLARE @Start INT
SET @Start = 48
```

```
WHILE(@Start <= 57)
BEGIN
	PRINT CHAR(@Start)
	SET @Start = @Start + 1
END
-- will print 0 to 9


SELECT LTRIM('    Hello')
-- will remove left white space

SELECT LTRIM(FirstName) AS FirstName, MiddleName, LastName FROM tblEmployee
-- will remove left white space from FirstName column values

SELECT RTRIM('Hello    ')
-- will remove right white space

SELECT RTRIM(LTRIM(FirstName)) + ' ' + MiddleName + ' ' + LastName
AS FullName
FROM tblEmployee
-- will remove left and right white space from FirstName column values

SELECT UPPER(RTRIM(LTRIM(FirstName))) + ' ' + MiddleName + ' ' + LOWER(LastName)
AS FullName
FROM tblEmployee

-- say we want Sam to be Mas for first name, that is the names is reversed


SELECT REVERSE(FirstName) + ' ' + MiddleName + ' ' + LastName
AS FullName
FROM tblEmployee


-- find the number of characters in the first namme

SELECT FirstName, LEN(FirstName) AS [Total Characters] FROM tblEmployee
-- will return a table with two columns "FirstName" and "Total Characters"
-- the second column will contain how many characters the first name has

--------------------------------------------------
-- 23 -- BUILT IN STRING FUNCTIONS (CONT.) -- 23 --
--------------------------------------------------

SELECT LEFT('ABCDEF', 3) -- you want ABC, the first three characters from
-- the left

SELECT RIGHT('ABCDEF', 3) -- you want DEF, the first three character from
-- the right
```

'sarah@aaa.com'
-- suppose we want to find the index of the @ symbol
SELECT CHARINDEX('@', 'sarah@aaa.com', 1)
-- will return the index of the @ in the expression sarah@aaa.com, starting
-- at the left of the s, position 1

SELECT SUBSTRING('sarah@aaa.com', 6, 7)
-- this will return aaa.com, searching sarah@aaacom starting in position 6,
-- 7 positions to the right of that

SELECT SUBSTRING('pam@bbb.com', CHARINDEX('@', 'pam@bbb.com') + 1,
	LEN(pam@bbb.com) - CHARINDEX('@', 'pam@bbb.com'))
-- will return bbb.com, searching pam@bbb.com, starting in whatever
-- position is after the @ symbol, however many positions to the right
-- of that are needed.

-- complex example:
SELECT SUBSTRING(Email, CHARINDEX('@', Email) + 1,
	LEN(Email) - CHARINDEX('@', Email)) AS EmailDomain,
	COUNT(Email) AS Total
FROM tblEmployee
GROPU BY SUBSTRING(Email, CHARINDEX('@', Email) + 1,
	LEN(Email) - CHARINDEX('@', Email))
-- will return the email domains on one column with the total number
-- of times that email domain appears