	<b>UNIVERSIDAD DON BOSCO</b> <b>FACULTAD DE ESTUDIOS TECNOLOGICOS</b> <b>ESCUELA DE COMPUTACION</b>
<b>CICLO 01-2016</b>	<b>GUIA DE LABORATORIO N°10</b>
	<b>Nombre de la práctica:</b> Implementación de Procedimientos almacenados y desencadenadores <b>Lugar de ejecución:</b> Laboratorio de Informática <b>Tiempo estimado:</b> 2 horas y 30 minutos <b>Materia:</b> Base de datos

## I. Objetivos

Qué el estudiante sea capaz de:

- Validar la información que se utilizará en la Base de datos por medio de procedimientos almacenados.
- Utilizar desencadenadores para activar mensajes de advertencia
- Implementar los diferentes tipos de desencadenadores

## II. Introducción Teórica

### Desencadenadores (TRIGGERS)

Los disparadores de procedimiento, más comúnmente conocidos como TRIGGERS, son una especie de procedimientos almacenados, a diferencia que se ejecutan cuando ocurre un evento sobre alguna tabla. Entendemos por evento, cualquier acción del tipo:

- Inserción
- Borrado
- Actualización

La sintaxis de la sentencia de creación de TRIGGERS es la siguiente:

```
CREATE TRIGGER nombre
ON tabla
FOR [DELETE | INSERT | UPDATE]
AS
Sentencias
```

Las palabras reservadas DELETE, INSERT y UPDATE corresponden a cada una de las acciones para las cuales se puede definir un desencadenador dentro de la tabla especificada.

El bloque de sentencias permite prácticamente cualquier tipo de ellas dentro del lenguaje T-SQL, pero con ciertas limitaciones. Por ejemplo, no se podrá utilizar la sentencia SELECT, ya que un TRIGGER no puede devolver datos al usuario, sino que simplemente se ejecuta para cambiar o comprobar los datos que se van a insertar, actualizar o borrar.

Por ejemplo, si queremos crear un TRIGGER llamado modificacion\_Clientes, sobre la tabla Customers, que muestre un mensaje cada vez que se actualiza una fila de la tabla, deberemos escribir el Código:

```
CREATE TRIGGER modificacion_Clientes
ON Customers FOR UPDATE AS
PRINT 'Han actualizado la tabla de Customers'
```

Para comprobar el funcionamiento de este TRIGGER, podemos actualizar cualquier fila de la tabla de Customers, por ejemplo con la sentencia:

```
UPDATE Customers SET ContactName='Maria Walters'
WHERE CustomerID='AROUT'
```

Con esto conseguimos dos cosas, actualizar el nombre del Cliente cuyo código es AROUT y obtener el mensaje que se muestra como ejecución del TRIGGER de actualización.

Sin embargo, los TRIGGERS en SQL Server tienen una serie de limitaciones:

1. No se puede disparar un TRIGGER dentro de otro TRIGGER, ya que daría lugar a un bucle infinito
2. Por esta razón, un TRIGGER no puede ejecutar instrucciones DDL (lenguaje de definición de datos)
3. No se pueden ejecutar sentencias como SELECT INTO o de creación de dispositivos dentro de un TRIGGER

Del mismo modo, para borrar un TRIGGER, deberemos ejecutar la sentencia DROP TRIGGER TRIGGER. Por ejemplo, si queremos borrar el TRIGGER anteriormente creado, ejecutaremos  
DROP TRIGGER modificacion\_Clientes

### Tablas DELETED e INSERTED

Dentro de la definición de un TRIGGER, podemos hacer referencia a un par de tablas lógicas, cuya estructura es similar a la tabla donde se está ejecutando el TRIGGER; es decir, es una copia de la tabla en la cual se van a insertar o borrar los datos, y que contiene, precisamente, los datos que van a ser añadidos o borrados. La utilidad de estas dos tablas es la de realizar comprobaciones entre los datos antiguos y los nuevos. Así, por ejemplo, si queremos recuperar los datos de la tabla que estamos borrando, dentro del TRIGGER, se deberá ejecutar el siguiente código:

```
SELECT *
FROM deleted
```

### Tipos de desencadenadores

SQL-Server permite la definición de varios tipos de TRIGGERS, entre los cuales cabe destacar los siguientes:

- **Desencadenadores múltiples:** para una misma tabla, se pueden definir distintos TRIGGERS para la misma acción, es decir, si definimos un TRIGGER para insert, y resulta que dicha tabla ya tenía definido un TRIGGER para esa misma acción, se ejecutarán ambos TRIGGERS cuando ocurra dicho evento sobre la tabla.
- **Desencadenadores recursivos:** se permite la recursividad entre las llamadas a los TRIGGERS, es decir, un TRIGGER puede llamar a su vez a otro, bien de forma directa, bien de forma indirecta.
- **Desencadenadores anidados:** si un TRIGGER cambia una tabla en la que se encuentra definido otro TRIGGER, se provoca la llamada de este último que, si a su vez vuelve a modificar otra tabla, puede

provocar la ejecución de otro TRIGGER, y así sucesivamente. Si se supera el nivel de anidamiento permitido, se cancelará la ejecución de los TRIGGERS.

### Limitaciones de los TRIGGERS

Aunque ya se han comentado algunas de las limitaciones a la hora de programar TRIGGERS, veamos en detalle las restricciones que implica la definición de TRIGGERS:

- Un TRIGGER sólo se puede aplicar a una tabla
- Aunque un TRIGGER se defina dentro una sola base de datos, puede hacer referencia a objetos que se encuentran fuera de la misma
- La misma acción del desencadenador puede utilizarse para definir más de un TRIGGER sobre la misma tabla.
- La opción SET elegida dentro de la ejecución de un desencadenador, volverá a su estado previamente definido una vez concluya la ejecución del mismo.
- Así mismo, no se permite la utilización de las sentencias del DDL dentro de la definición de un TRIGGER.
- En una vista no se puede utilizar un desencadenador.

### III. Requerimientos

Nº	Cantidad	Descripción
1	1	Guía de Laboratorio #11 de BD
2	1	Memoria USB
3	1	Maquina con SQL Server 2012

### IV. Procedimiento

#### Parte 1: Iniciando sesion desde SQL Server Managment Studio

1. Hacer clic en el botón **Inicio**
2. Hacer clic en la opción **Todos los programas** y hacer clic en **Microsoft SQL Server 2012**

Para conectarse con el servidor de base de datos elija los siguientes parámetros de autenticación:

- **Tipo de servidor:** Database Engine
- **Nombre del servidor:** Colocar el nombre del servidor local, por ejemplo PCNumMaquina-SALA#  
Nota: NumMaquina es el número de la maquina local
- **Autenticación:** SQL Server Authentication
- **Login:** sa
- **Password:** 123456

3. Luego seleccionara del menú estándar la opción (Nueva Consulta/New Query) para empezar a trabajar con las sentencia de T-SQL.
4. Localice el icono de guardar, con el fin de guardar el nuevo archivo de sentencias T-SQL (**Guia11\_Procedimiento\_SuCarnet.sql**).

## Parte 2: Implementación de estructuras repetitivas en un procedimiento almacenado

1. Hacer uso de la base de datos **Northwind**
2. En los ejercicios de este punto cambiar en todos los ejercicios la palabra SuCarnet por su número de carnet
3. Crear el siguiente procedimiento almacenado:

```
CREATE PROCEDURE Mostrar_10_pedidos_SuCarnet
AS
DECLARE @contador int
DECLARE @num int

SET @contador=0
--Obteniendo el primer valor del campo OrderID de la tabla Orders
SET @num=(SELECT TOP 1 OrderID FROM Orders ORDER BY OrderID)

--Evalua si el contador es menor que 10, si la condicion se cumple
--realiza la instruccion SELECT
WHILE @contador<10
BEGIN
    SELECT OrderID, OrderDate FROM Orders WHERE OrderID=@num+@contador
    --Se incrementa el contador
    SET @contador=@contador+1
END
```

4. Ejecutamos el procedimiento almacenado

```
--Ejecutamos el procedimiento almacenado
EXEC Mostrar_10_pedidos_SuCarnet
```

5. En el ejemplo siguiente, si el precio de venta promedio de un producto es inferior a \$300, se realiza dentro del bucle WHILE la acción de duplicar los precios y, a continuación, selecciona el precio máximo. Si el precio máximo es menor o igual que \$500, el bucle WHILE se reinicia y se vuelve a duplicar los precios. El bucle realiza la operación hasta que el precio máximo sea mayor a \$500, después de lo cual finaliza el bucle WHILE.
6. Antes de ejecutar el procedimiento se debe verificar la información de la tabla Products, por ejemplo al ejecutar la consulta:

```
SELECT UnitPrice FROM Products
ORDER BY UnitPrice DESC
```

Y obtenemos los siguientes resultados:

	UnitPrice
1	263.50
2	123.79
3	97.00
4	81.00
5	62.50
6	55.00
7	53.00
8	49.30
9	46.00
10	45.60

Creamos el siguiente procedimiento:

```
CREATE PROCEDURE Actualizar_precio_SuCarnet
AS
WHILE (SELECT AVG (UnitPrice) FROM Products) < 30
BEGIN
    UPDATE Products SET UnitPrice = UnitPrice * 2
    SELECT MAX (UnitPrice) AS [Precio Máximo] FROM Products
    IF (SELECT MAX (UnitPrice) FROM Products) < 50
        BREAK
        --La instrucción BREAK se utiliza para salir de una instrucción WHILE
        --o una instrucción IF...ELSE
    ELSE
        CONTINUE
        --Reinicia un bucle WHILE y continúa con la siguiente iteración del lazo
END
--Aquí terminar el código del procedimiento
```

7. Ejecutamos el procedimiento

```
--Ejecutamos el procedimiento
EXECUTE Actualizar_precio_SuCarnet
```

8. Ejecutamos de nuevo la consulta SELECT

```
SELECT UnitPrice FROM Products
ORDER BY UnitPrice DESC
```

9. Se obtienen los siguientes resultados (estos pueden variar, tomar en cuenta los resultados obtenidos en la consulta)

	UnitPrice
1	527.00
2	247.58
3	194.00
4	162.00
5	125.00
6	110.00
7	106.00
8	98.60
9	92.00
10	91.20
11	87.80

Observe que se han duplicado los precios de cada producto

- En el siguiente ejemplo se desea mostrar los nombres de los clientes que se encuentran en la tabla Customers, haciendo uso de cursores

```
CREATE PROC Mostrar_Clientes_SuCarnet
AS
    DECLARE @Nombre NVARCHAR(40)
    --Se declara el cursor @cursor, el cual se utilizara para recorrer
    --cada resultado de la consulta SELECT
    DECLARE @cursor CURSOR

    --Se asigna el primer dato al cursor
    SET @cursor = CURSOR FOR
    SELECT CompanyName FROM Customers
    --Abrir el cursor
    OPEN @cursor
    --Recupera las filas del cursor
    FETCH NEXT
    FROM @cursor INTO @Nombre
    WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT 'El nombre del cliente es: ' + @Nombre
        --Se mueve al siguiente registro
        FETCH NEXT FROM @cursor INTO @Nombre
    END
    --Este comando hace desaparecer el puntero sobre el registro actual
    CLOSE @cursor
    DEALLOCATE @cursor
GO
```

- Ejecutar el procedimiento almacenado

```
--Ejecutamos el procedimiento almacenado
EXEC Mostrar_Clientes_SuCarnet
```

### Parte 3: Uso de desencadenadores

- En la base de datos Norhtwind, se crearan los siguientes TRIGGERS o desencadenadores

```

CREATE TRIGGER Disp_SEGURIDAD_SuCarnet
ON DATABASE FOR DROP_TABLE, ALTER_TABLE
AS
    BEGIN
        --RAISERROR se usa para devolver mensajes a las aplicaciones con el
        --mismo formato que un error del sistema
        RAISERROR ('No está permitido borrar ni modificar tablas!' , 16, 1)
        --16 Severidad
        --1 Estado
        ROLLBACK TRANSACTION
    END
GO

```

**Nota:**

ROLLBACK TRANSATION: Revierte una transacción explícita o implícita hasta el inicio de la transacción o hasta un punto de retorno dentro de la transacción.

2. Crear la siguiente tabla:

```

--Crear la tabla
CREATE TABLE prueba_SuCarnet
(campo1 int
)

```

3. Ejecutar la siguiente instrucción:

```

--Eliminar la tablas
DROP TABLE prueba_SuCarnet

```

4. Verifique los resultados, debe de mostrar el mensaje indicando de que no se puede eliminar ni modificar una tabla en la base de datos, el disparador se activa en el momento que un usuario desea: eliminar o modificar una tabla. Por eso es que no se puede eliminar la tabla prueba\_SuCarnet
5. Crear el siguiente TRIGGER a la tabla prueba\_SuCarnet

```

CREATE TRIGGER Mensaje_Insercion_SuCarnet
ON prueba_SuCarnet
AFTER insert
AS
BEGIN
    IF (SELECT COUNT (campo1) FROM prueba_SuCarnet WHERE campo1= (SELECT campo1 FROM inserted))>1
        --Si ya hay un dato con el mismo valor que se quiere agregar, se debe realizar un
        --ROLLBACK TRANSACTION y así no se puede agregar de nuevo ese dato
        ROLLBACK TRANSACTION
    ELSE
        PRINT 'Se agregó un nuevo registro a la tabla'
END

```

6. Agregar un dato a la tabla

```

--Verificando el TRIGGER
INSERT INTO prueba_SuCarnet VALUES (1)

```

Cada vez que se agregue un nuevo dato a la tabla se mostrara el mensaje indicando que se ha agregado un nuevo dato a la tabla, de lo contrario no agrega el dato a la tabla

7. Agregar un nuevo dato a la tabla Employees (el dato a ingresar son su apellido y nombre)

```
--Agregando un nuevo dato a la tabla Employees
INSERT INTO Employees(Lastname,Firstname) VALUES ('Cañas','Blanca')
```

8. Crear el siguiente TRIGGER

```
CREATE TRIGGER actualizar_emple_SuCarnet
ON Employees
AFTER UPDATE
AS
    IF UPDATE(Lastname)
    BEGIN
        PRINT 'Se realizo un cambio en el apellido de los empleados'
    END
```

9. Antes de verificar el TRIGGER, realice la consulta para conocer el código del empleado (EmployeeID) que se agregó en el punto 7

```
--Verificando los datos de los empleados
SELECT * FROM Employees
```

	EmployeeID	LastName	FirstName	Title
11	11	Cañas	Blanca	NULL

10. Comprobar el TRIGGER realizando una actualización en el apellido del empleado donde el código del empleado sea igual al que obtuvo en la consulta anterior

```
--Actualizando el apellido del empleado
UPDATE Employees SET LastName='Abarca'
WHERE EmployeeID=11
```

11. Verificar los resultados de la consulta y del TRIGGER
12. Guardar los cambios
13. Agregar un nuevo empleado a la tabla Employees
14. Crear el siguiente procedimiento almacenado



```

CREATE TRIGGER eliminar_emple_SuCarnet
ON employees
FOR DELETE
AS
IF(SELECT COUNT(*) FROM deleted)>1
BEGIN
    PRINT 'No se puede borrar mas de un empleado al mismo tiempo'
ROLLBACK TRANSACTION
END

```

La tabla DELETED almacena copias de las filas afectadas por las instrucciones DELETE y UPDATE. Durante la ejecución de una instrucción DELETE o UPDATE, las filas se eliminan de la tabla del desencadenador (TRIGGER) y se transfieren a la tabla DELETED.

15. Ejecutar la siguiente consulta para eliminar más de un registro de la tabla Employees

```

--La instrucción DELETE activa el desencadenador y evita la transacción.
DELETE FROM Employees WHERE EmployeeID > 10

```

16. Ahora ejecutar la siguiente consulta

```

--La instrucción DELETE activa el desencadenador y permite la transacción.
DELETE FROM Employees WHERE EmployeeID = 11

```

17. Verificar los resultados de la consulta

Tomar un dato de la columna Employees, por ejemplo uno de los códigos de empleado que acaba de agregar

18. Guardar los cambios

## V. Ejercicio complementario

1. En un nuevo Script, crear el ejercicio complementario, debe colocar como nombre: **Guia11\_EjercicioComplementario\_SuCarnet.sql**
2. Crear la base de datos, copiar el siguiente código:

```

--Crear la base de datos
CREATE DATABASE Control_Examen_SuCarnet
GO

--Hacer uso de la base de datos
USE Control_Examen_SuCarnet
GO

```

3. Crear las siguientes tablas, puede copiar el siguiente código:

```

CREATE TABLE Alumno
(
    Carnet          VARCHAR(10) NOT NULL PRIMARY KEY,
    Nombre          VARCHAR(30) NOT NULL,
    Apellido1       VARCHAR(30) NOT NULL,
    Apellido2       VARCHAR(30) NOT NULL,
    Sexo            VARCHAR(2)  NOT NULL,

```

```

)
GO

CREATE TABLE Examenes
(
    cod_examen          INT          NOT NULL          PRIMARY KEY,
    titulo              VARCHAR(100) NOT NULL,
    n_preguntas         INT          NOT NULL,
)
GO

CREATE TABLE Notas
(
    cod_examen          INT          NOT NULL          FOREIGN KEY REFERENCES
Examenes(cod_examen),
    Carnet              VARCHAR(10)  NOT NULL          FOREIGN KEY REFERENCES
Alumno(Carnet),
    nota_examen         DECIMAL(4,2) NOT NULL,
    fecha              SMALLDATETIME NOT NULL,
)
GO

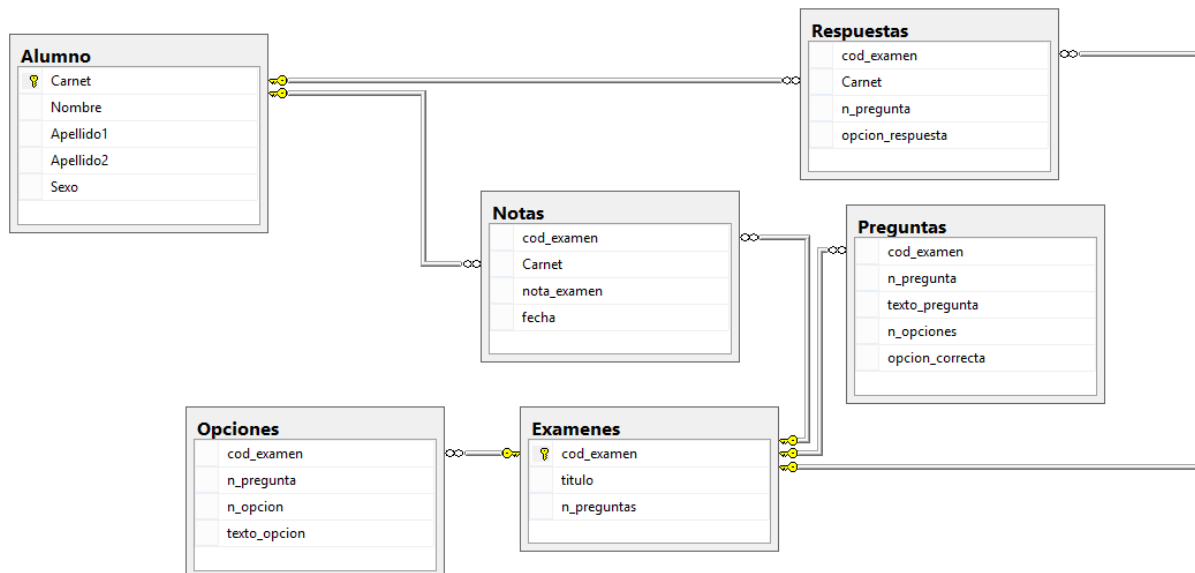
CREATE TABLE Respuestas
(
    cod_examen          INT          NOT NULL          FOREIGN KEY
REFERENCES Examenes(cod_examen),
    Carnet              VARCHAR(10)  NOT NULL          FOREIGN KEY
REFERENCES Alumno(Carnet),
    n_pregunta          INT          NOT NULL,
    opcion_respuesta    INT          NOT NULL,
)
GO

CREATE TABLE Preguntas
(
    cod_examen          INT          NOT NULL          FOREIGN KEY REFERENCES
Examenes(cod_examen),
    n_pregunta          INT          NOT NULL,
    texto_pregunta      VARCHAR(100) NOT NULL,
    n_opciones          INT          NOT NULL,
    opcion_correcta     INT          NOT NULL,
)
GO

CREATE TABLE Opciones
(
    cod_examen          INT          NOT NULL          FOREIGN KEY REFERENCES
Examenes(cod_examen),
    n_pregunta          INT          NOT NULL,
    n_opcion            INT          NOT NULL,
    texto_opcion        VARCHAR(50)  NULL,
)
GO

```

#### 4. Crear el diagrama de la base de datos



5. Insertando Datos a partir de Procedimientos Almacenados
6. Para insertar datos en la tabla Alumno, crear un procedimiento llamado Insert\_Alumno, tal como se muestra a continuación:

```

CREATE PROCEDURE Insert_Alumno
@ carnet CHAR(8), @ nombre CHAR(15), @apell1 CHAR(15), @apell2 CHAR(15), @sexo CHAR(1)
AS
    INSERT INTO Alumno VALUES ( @carnet, @nombre, @apell1, @apell2, @sexo)
    return (0)
GO
  
```

7. Utilizar el procedimiento almacenado creado anteriormente para insertar los siguientes datos:

```

EXEC Insert_Alumno 'SB970326', 'Alicia', 'Salinas', 'Benitez', 'F'
EXEC Insert_Alumno 'SC970245', 'Pedro', 'Salazar', 'Calderon', 'M'
EXEC Insert_Alumno 'RC970201', 'Karla', 'Ramirez', 'Chicas', 'F'
  
```

8. Se insertaran datos en las tablas Exámenes, Preguntas y Opciones en base al siguiente formato de examen:

**Código del Examen: 1**

**Título: "Conceptos de Procedimientos Almacenados"**

**No. de Preguntas: 2**

1. Con que sentencia se crea un procedimiento almacenado:
  1. CREATE PROCEDURE
  2. CREATE PROC
  3. Ambas

**(# de opcion correcta: 3)**
2. Con que comando se ejecuta un procedimiento almacenado:
  1. EXECUTE
  2. sp\_executesql

**(# de opcion correcta: 1)**

9. Crear el siguiente procedimiento almacenado para llenar los datos en la tabla Exámenes:

```
CREATE PROC Insert_Exa
@cod INT,
@title VARCHAR(100),
@NPreg INT
AS
    INSERT Exámenes VALUES(@cod,@title,@NPreg)
    return(0)
```

10. Agregar los siguientes registros a la tabla Exámenes por medio del procedimiento almacenado

```
--Agregando Datos del Examen
EXEC Insert_Exa 1, 'Conceptos de Procedimientos Almacenados', 2
EXEC Insert_Exa 2, 'Conceptos sobre Redes de Area Local', 3
```

11. Por medio de instrucción INSERT agregar los siguientes registros a la tabla Preguntas

```
--Agregando datos a preguntas
INSERT INTO Preguntas VALUES (1,1, 'Con que sentencia se crea un procedimiento almacenado:', 3, 3)
INSERT INTO Preguntas VALUES (1,2, 'Con que comando se ejecuta un procedimiento almacenado:', 2, 1)
```

12. Agregando registros a la tabla Opciones por medio de la instrucción INSERT

```
--Agregando datos a opciones
INSERT INTO Opciones VALUES (1,1,1, 'Create Procedure')
INSERT INTO Opciones VALUES (1,1,2, 'Create Proc')
INSERT INTO Opciones VALUES (1,1,3, 'Ambas opciones')
INSERT INTO Opciones VALUES (1,2,1, 'EXECUTE')
INSERT INTO Opciones VALUES (1,2,2, 'sp_executesql')
```

## Mensajes de Error definidos por el usuario

Antes de insertar datos en la tabla de Respuestas, se creará un TRIGGER que despliegue un mensaje de advertencia cuando el alumno este contestando la última pregunta del examen (en este caso sería la pregunta 2).

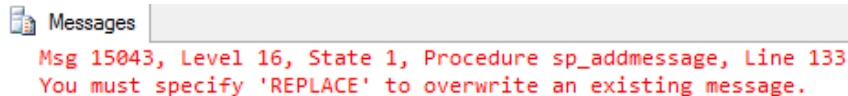
Para ello crear primero un mensaje definido por el usuario, este mensaje es añadido a la tabla **sysmessages** (sysmessages se encuentra en la **base de datos master**) utilizando el procedimiento almacenado del sistema **sp\_addmessage**.

Para crear el mensaje debe especificarse el número del mensaje, el tipo o nivel de severidad y el texto del mensaje. (Los mensajes de error deben tener un número arriba de 50000, pues los otros números son reservados por SQL Server).

13. Crear el siguiente mensaje:

```
--Creando los mensajes o verificar si ya existe el mensaje
USE master
GO
EXEC sp_addmessage 50001, 16, N'Usted se encuentra en la última pregunta del examen'
```

Si al ejecutar la consulta anterior le aparece el siguiente mensaje de error:



Messages  
Msg 15043, Level 16, State 1, Procedure sp\_addmessage, Line 133  
You must specify 'REPLACE' to overwrite an existing message.

Significa que el número del mensaje 50001 ya existe en la base de datos master, así que cambie el número por uno consecutivo (por ejemplo 50002)

14. Hacer uso de nuevo de la base de datos **Control\_examen\_SuCarnet** y creamos un procedimiento almacenado el cual va a realizar la operación de calcular la nota del alumno después de haber contestado el examen
15. Antes de crear el procedimiento almacenado, se verifica la existencia de este en la base de datos, si el procedimiento existe se elimina y sino no se realiza ninguna acción

```
--Verificar si existe el procedimiento y si este existe se elimina
IF EXISTS (SELECT name FROM sysobjects
WHERE name = 'Calculo_de_Nota' AND type = 'P')
    DROP PROCEDURE Calculo_de_Nota
GO
```

16. Crear el siguiente procedimiento almacenado

```
--Creacion del procedimiento
CREATE PROC Calculo_de_Nota
@carnetx VARCHAR(10), @codexamen INT
AS
BEGIN
    DECLARE @totcorrecta INT, --contador para respuestas correctas
            @porcentaje FLOAT, --porcentaje de cada pregunta
            @npreguntas INT, --numero de preguntas del examen
            @correctaresp INT, --captura la opcion correcta del examen(opc_correcta)
            @respuesta INT, --captura la opcion que selecciono el alumno (opcion_respuesta)
            @perfect FLOAT, --constante = 10
            @NOTA FLOAT--nota del examen

    --Verifica la cantidad de preguntas que tiene el examen
    SELECT @npreguntas= CONVERT(FLOAT,COUNT(n_pregunta))FROM Preguntas
    WHERE cod_examen=@codexamen
    SELECT @perfect = 10.00 --Asignar un dato a una variable
    SELECT @porcentaje= @perfect/@npreguntas --calcula de porcentaje para c/pregunta
    SELECT @totcorrecta=0 --se inicializa un contador de respuestas correctas a 0

    WHILE(@npreguntas > 0)
    BEGIN
        --Asignar la opcion correcta del examen
        SELECT @correctaresp = opcion_correcta FROM Preguntas
        WHERE (cod_examen = @codexamen and n_pregunta = @npreguntas)
        --Asignar la respuesta del alumno del examen
        SELECT @respuesta = opcion_respuesta FROM Respuestas
        WHERE (cod_examen = @codexamen and Carnet = @carnetx and n_pregunta = @npreguntas)
```

```

--Compara la opcion correcta del examen con la respuesta del alumno
IF(@correctaresp = @respuesta)
BEGIN
    --Si coinciden los datos se incrementa un contador, el cual controla el total de respuestas
    --correctas por parte del alumno
    SELECT @totcorrecta = @totcorrecta + 1
END -- fin de IF
SELECT @npreguntas=@npreguntas-1 --se decrementa el total de preguntas
END --fin de WHILE

--Calcula la nota del examen
SELECT @NOTA = @totcorrecta * @porcentaje
--Agrega la nota a la tabla Notas
INSERT INTO Notas VALUES (@codexamen,@carnetx,@NOTA,GETDATE())
END -- fin del cuerpo del procedimiento

--Para ver la nota en la ventana de resultados
SELECT "NOTA_EXAMEN"=CONVERT(FLOAT,@NOTA)

```

17. Crear el siguiente TRIGGER el cual se activa después de cada inserción que se realiza en la tabla respuesta verifica que pregunta a contestado el alumno de un examen específico y se llama al procedimiento Calculo\_de\_Nota para calcular la nota del examen

18. Verificar la existencia del TRIGGER, ejecutar la siguiente consulta:

```

--Verificamos si existe el trigger y si este existe se elimina
IF EXISTS (SELECT name FROM sysobjects
    WHERE name = 'Warning' AND type = 'TR')
    DROP TRIGGER Warning
GO

```

19. Digitar el siguiente código:

```

--Creacion del trigger
CREATE TRIGGER Warning
ON Respuestas
FOR INSERT --indica que se dispara al insertar datos en la tabla respuestas
AS
    DECLARE @tot_preguntas INT,
            @Npregunta INT,
            @codExa INT,
            @Nota FLOAT,
            @carnetx VARCHAR(10)

    SELECT @codExa = cod_examen FROM inserted --La tabla inserted almacena copias de las filas
    --afectadas durante las instrucciones INSERT y UPDATE
    SELECT @tot_preguntas = n_preguntas FROM Examenes
    WHERE cod_examen = @codExa
    --Asigna a la variable @Npregunta la pregunta del examen que se esta contestando
    SELECT @Npregunta = n_pregunta FROM inserted
    --Evalua el numero de la pregunta con el total que tiene el examen

```

```

IF @Npregunta = @tot_preguntas
BEGIN
    --Colocar el numero de mensaje de error que creo en el punto 13 del ejercicio
    RAISERROR (50001, 16, 1)
    SELECT @carnetx = carnet FROM inserted
    --Notar que un Trigger puede llamar a un Store Procedure
    EXEC Calculo_de_Nota @carnetx,@codExa
    --Almacenda en la variable @Nota la nota del alumno con respecto al examen que a contestado
    SELECT @Nota = nota_examen FROM Notas WHERE cod_examen=@codExa
    AND carnet=@carnetx
    PRINT 'Su nota fue: ' + convert(varchar(4),@Nota)
END --fin de IF
GO

```

20. Realice las siguientes pruebas, ejecutar la siguiente consulta:

```

--Realizando pruebas
--A la pregunta 1 contesto la opcion 3, es la correcta
INSERT INTO Respuestas VALUES (1,'RC970201',1,3)

```

21. Ejecutar la siguiente consulta

```

--A la pregunta 2 contesto la opcion 2, contesto mal
INSERT INTO Respuestas VALUES (1,'RC970201',2,2)

```

Al contestar esta última pregunta se activa el TRIGGER y cuando el alumno contesta la última pregunta se ejecuta el procedimiento almacenado para calcular la nota del examen

22. Ejecutar las siguientes consultas

```

--Verificar los registros de la tabla Respuestas
SELECT * FROM Respuestas
--Verificar los registros de la tabla Notas
SELECT * FROM Notas

```

23. Guardar los cambios en el archivo

## VI. Análisis de resultados

Crear el siguiente ejercicio:

Utilizando la siguiente base de datos realizar lo siguiente.

```

CREATE DATABASE Control_Pedidos
GO

USE Control_Pedidos
GO

CREATE TABLE producto
(idproducto CHAR(8) not null,
nombreproducto VARCHAR(25),
existencia INT not null,
precio DECIMAL(10,2) not null, --precio costo
preciov DECIMAL(10,2) not null, --precio venta
CONSTRAINT pk_idproducto PRIMARY KEY (idproducto)
)
GO

CREATE TABLE pedidos
(
idpedido INT IDENTITY,
idproducto CHAR(8) not null,
cantidad_pedido INT,
CONSTRAINT fk_idbod FOREIGN KEY (idproducto) REFERENCES producto(idproducto)
)
GO

```

Agregar los siguientes registros:

```

INSERT INTO producto VALUES('prod001','Filtros pantalla',5,10,12.5)
INSERT INTO producto VALUES('prod002','parlantes',7,10,11.5)
INSERT INTO producto VALUES('prod003','mouse',8,4.5,6)
INSERT INTO producto VALUES('prod004','monitor',10,60.2,80.0)
INSERT INTO producto VALUES('prod005','lapiz',5,1.2,2.0)

```

### Ejercicios:

1. Crear un desencadenador que se active cada vez que se inserte un registro en la tabla pedidos y otro para la tabla producto.
2. Crear un desencadenador para la tabla producto, que se active cada vez que se inserte un registro o se actualice la columna precio, la condición para aceptar al inserción o la actualización es que el precio costo no debe ser mayor que el precio venta.
3. Crear un desencadenador para la tabla pedidos que cada vez que se realice un pedido descuenta la existencia de la tabla productos, en caso que la cantidad del pedido supere a la existencia debe deshacer la transacción y mostrar un mensaje de error.

## VII. Referencia Bibliográfica

Microsoft SQL Server 2008, Guía Práctica, Francisco Charle Ojeda; Anaya Multimedia.