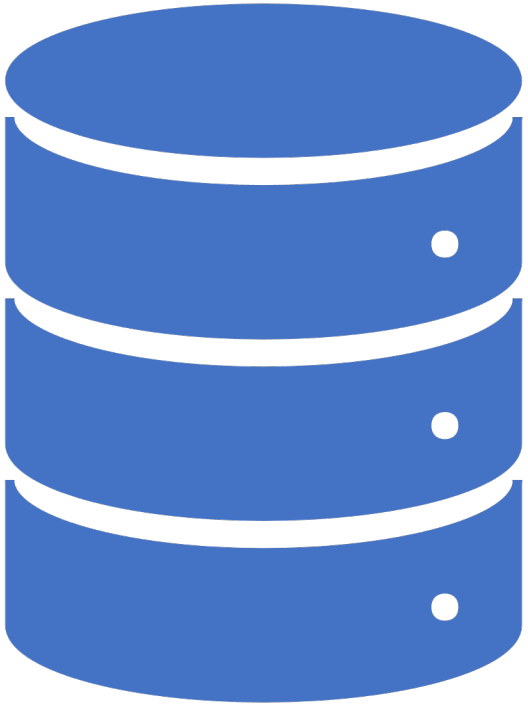


Seminar 3

Stored procedures.
Global Variables.
Dynamic Execution.
The Output Clause.
Flow Control Language.
- SQL Server -

Stored Procedures

- ***Stored procedure*** = a group of Transact-SQL statements compiled in a single execution plan
- Stored procedure:
  - Can have input parameters and output parameters with multiple values (e.g. tables)
  - Contain programming statements that operate on the database, and also procedure calls
  - Return a value that indicate if the stored procedure was executed with success or not
- Advantages:
  - Reduce the network traffic
  - Better control on the security
  - Code reuse
  - Simplified maintenance
  - Increased performance
  - Include code to ensure the handling of the errors

Stored Procedures

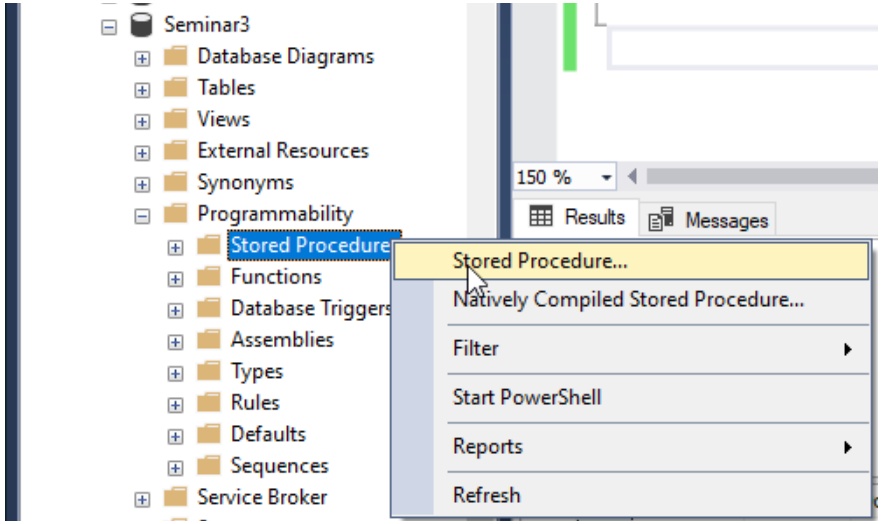
- Stored procedure **syntax**:

```
CREATE PROCEDURE sp_name
(@parameter1 parameter_datatype,
@parameter2 parameter_datatype, …)
AS
BEGIN
        -- sequence of SQL statements
END
GO
```

- Stored procedure **execution**:
  - EXEC sp_name or EXEC sp_name parameter1, parameter2, …
  - sp_name or sp_name parameter1, parameter2, …
  - EXECUTE sp_name or EXECUTE sp_name parameter1, parameter2, …

Stored Procedures

- Stored procedure SQL Server:




```sql
-- Use the Specify Values for Template Parameters
-- command (Ctrl-Shift-M) to fill in the parameter
-- values below.
--
-- This block of comments will not be included in
-- the definition of the procedure.
-- ============================================
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Author:      <Author,,Name>
-- Create date: <Create Date,,>
-- Description: <Description,,>
-- =============================================
CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>
    -- Add the parameters for the stored procedure here
    <@Param1, sysname, @p1> <Datatype_For_Param1, , int> = <Default_Value_For_Param1, , 0>,
    <@Param2, sysname, @p2> <Datatype_For_Param2, , int> = <Default_Value_For_Param2, , 0>
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    -- Insert statements for procedure here
    SELECT <@Param1, sysname, @p1>, <@Param2, sysname, @p2>
END
GO
```

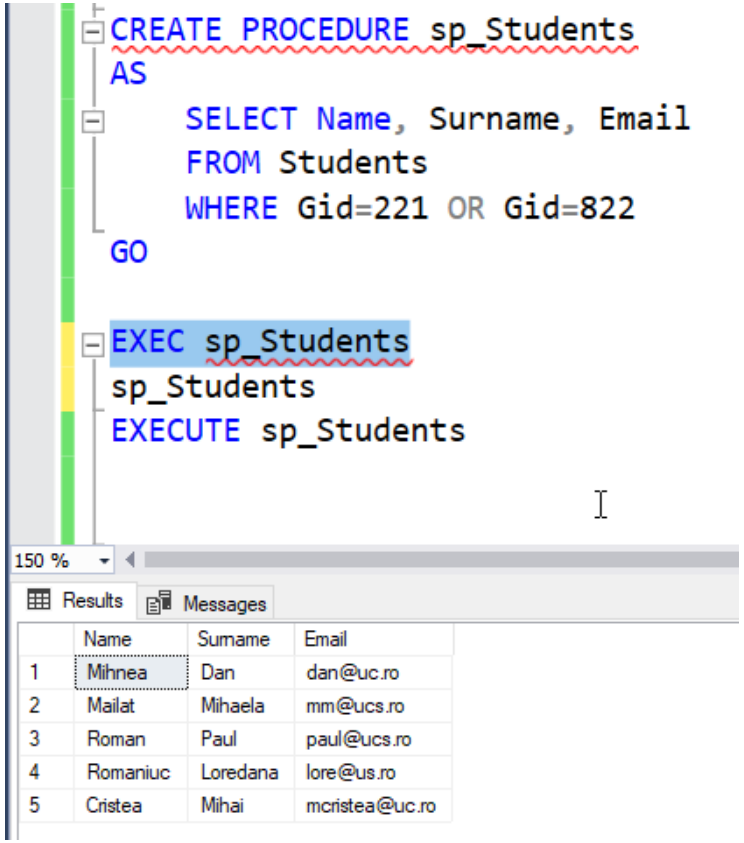
Stored Procedures

- Stored procedure example:

Stored procedure that display the name, surname and e-mail address for the students that are in groups 221 and 822.

```
CREATE PROCEDURE sp_Students
AS
    SELECT Name, Surname, Email
    FROM Students
    WHERE Gid=221 OR Gid=822
GO
```

```
EXEC sp_Students
sp_Students
EXECUTE sp_Students
```



Stored Procedures

- Stored procedure examples:

Stored procedure that add a new column

```
CREATE PROCEDURE sp_Students_add_column
AS
    ALTER TABLE Students
    ADD Age INT
GO

EXEC sp_Students_add_column

SELECT * FROM Students
```

```
CREATE PROCEDURE sp_Students_add_column
AS
    ALTER TABLE Students
    ADD Age INT
GO

EXEC sp_Students_add_column

SELECT * FROM Students
```

150 %

Results | Messages

	Sid	Name	Surname	Dob	Email	Gid	Age
1	1	Mihnea	Dan	2000-03-09	dan@uc.ro	822	NULL
2	2	Mailat	Mihaela	2001-11-08	mm@ucs.ro	822	NULL
3	3	Roman	Paul	2001-01-10	paul@ucs.ro	221	NULL
4	4	Romaniuc	Loredana	2000-05-11	lore@us.ro	221	NULL
5	5	Cristea	Mihai	2001-11-12	mcristea@uc.ro	221	NULL
6	6	Pitic	Mirela	2000-07-05	pmirela@ucs.ro	924	NULL

Stored Procedures

- Stored procedure with parameters examples:

Stored procedure with one parameter: Display the students from a given group.

```
CREATE PROCEDURE sp_Students_Group (@gid INT)
AS
        SELECT Name, Surname, Email
        FROM Students
        WHERE Gid=@gid
GO
```

```
EXEC sp_Students_Group 221
sp_Students_Group 822 or
sp_Students_Groups @gid=822
```

	Name	Surname	Email
1	Roman	Paul	paul@ucs.ro
2	Romaniuc	Loredana	lore@us.ro
3	Cristea	Mihai	mcristea@uc.ro

	Name	Surname	Email
1	Mihnea	Dan	dan@uc.ro
2	Mailat	Mihaela	mm@ucs.ro

Stored procedure with 2 parameters - by modifying the previous procedure: Display the students with a given name and from a given group.

```
ALTER PROCEDURE sp_Students_Group (@name VARCHAR(50), @gid INT)
AS
        SELECT Name, Surname, Email
        FROM Students
        WHERE Name=@name AND Gid=@gid
GO
```

```
EXEC sp_Students_Group
'Roman', 221
```

Results | Messages

	Name	Surname	Email
1	Roman	Paul	paul@ucs.ro

Stored Procedures

- Stored procedure with OUTPUT parameter examples:

  Display the number of the students from a given group.

```
CREATE PROCEDURE sp_Students_Group_output (@gid INT, @number INT OUTPUT)
AS
    SELECT @number = COUNT(*)
    FROM Students
    WHERE Gid=@gid
GO

    DECLARE @number INT
    SET @number=0
    EXEC sp_Students_Group_output 221, @number=@number OUTPUT
    PRINT @number
```

```
DECLARE @number INT
SET @number=0
EXEC sp_Students_Group_output 221, @number=@number OUTPUT
PRINT @number
```

150 %

Messages

3

Completion time: 2021-10-12T00:06:29.1211548+03:00

Stored Procedures

- Stored procedure – ***RAISERROR***

RAISERROR statement generates an error message and initiates error processing for the session

RAISERROR can refer a message defined by the user and stored in *sys.messages* catalog view, or, can build a message in dynamical mode.

Syntax: RAISERROR ({ msg_id | msg_str |. @local_variable } {, severity, state})

- *msg_id* – a user-defined error message number; should be greater than 50000 (the default value)
- *msg_str* – a user-defined message with formatting similar as in C programming language (printf); maximum 2047 characters; if it is not specified, an error message with the error number of 50000 is displayed
- *@local_variable* – a variable of any valid character data type that contains a string formatted as *msg_str*; should be ***char*** or ***varchar***
- *severity* – the user-defined severity level defined by the user and associated to the message (the users can specify a severity level between 0 and 18; (19-25 can be specified by sysadmins', WITH LOG option is required)
- *state* – an integer from 0 to 255; the negative values are default to 1; values greater than 255 should not be used

Stored Procedures

- Stored procedure – ***RAISERROR*** example:

  Display the number of the students from a given group.

```
ALTER PROCEDURE sp_Students_Group_output (@gid INT, @number INT OUTPUT)
AS
	SELECT @number = COUNT(*)
	FROM Students
	WHERE Gid=@gid

	IF @number=0
		RAISERROR('No student in the given group', 10, 1)
GO

DECLARE @number INT
SET @number=0
EXEC sp_Students_Group_output 222, @number=@number OUTPUT
PRINT @number
```

Messages

```
No student in the given group
0

Completion time: 2021-10-12T00:30:46.5516992+03:00
```

Stored Procedures

- Stored procedure – ***RAISERROR***

```
-- specify -1 to return the severity value associated with the error
RAISERROR(15600, -1, -1, 'sp_Students')
```

```
Messages
Msg 15600, Level 15, State 1, Line 224
An invalid parameter or option was specified for procedure 'sp_Students'.

Completion time: 2021-10-12T01:03:32.3835459+03:00
```

- specify -1 to return the severity value associated with the error

  RAISERROR(15600, -1, -1, 'sp_Students')

- the first argument of N'number' replaces the first conversion specification of %s (string)
- the second argument of 5 replaces the second conversion specification of %d (integer)

  RAISERROR (N'This is the message %s %d', -- message text
  10, -- severity
  1, -- state
  N'number', -- first argument
  3) -- second argument

  Result: This is the message number

```
-- the first argument of N'number' replaces the first conversion specification of
-- the second argument of 5 replaces the second conversion specification of %d
RAISERROR (N'This is the message %s %d', -- message text
    10, -- severity
    1, -- state
    N'number', -- first argument
    3) -- second argument
-- This is the message number 3
```

```
Messages
This is the message number 3

Completion time: 2021-10-12T01:09:52.9849947+03:00
```

Stored Procedures

- Stored procedure – **DROP PROCEDURE**

Syntax:

```
DROP PROCEDURE [schema_name.]sp_name
```

Example:

```
DROP PROCEDURE sp_Students

DROP PROCEDURE sp_Students_add_column
```

OR

```
DROP PROCEDURE dbo. sp_Students_Group

DROP PROCEDURE dbo.sp_Students_Group_output
```

Global Variables

Global Variables = special type of variables; the server maintain their values

- Provide information about the server and the current user session
- Their name starts with @@
- They don’t need to be declared because the server is constantly maintain them (practically are system functions)
- ***@@ERROR*** – the error number for the last executed Transact-SQL statement; 0– no error ocurred
- ***@@IDENTITY*** – the last inserted IDENTITY value
- ***@@ROWCOUNT*** – the number of rows affected in the last statement
- ***@@SERVERNAME*** – the name of the local server on which SQL Server is runing
- ***@@SPID*** – the session ID for the current user process
- ***@@VERSION*** – system and current build of the installed server information

Global Variables

Global Variables examples:

- IF **@@ERROR** <> 0 or SELECT **@@ERROR**
  PRINT 'Your error message’;

- create table test(tid int primary key **identity**, tname varchar(50), description varchar(50), code int)
  INSERT INTO test(tname, description, code) Values ('test1', 'decription 1', 8)
  select **@@IDENTITY** As 'Identity’

- IF **@@ROWCOUNT** = 0 or SELECT **@@ROWCOUNT**
  PRINT 'Warning: No rows were updated’;

- SELECT **@@SERVERNAME** AS 'Server Name’

- SELECT **@@SPID** AS 'ID', SYSTEM_USER AS 'Login Name', USER AS 'User Name’

- SELECT **@@VERSION** AS 'SQL Server Version'

SET NOCOUNT

- ***SET NOCOUNT ON*** – stops the returning of the message that involves the number of rows affected in the last Transact-SQL statement / stored procedure

- ***SET NOCOUNT OFF*** – returns the message that involves the number of rows affected in the last Transact-SQL statement / stored procedure, near to the result set
  - (1 row affected)

- The global variable @@ROWCOUNT will always be modified

- With SET NOCOUNT ON, the performance of the stored procedures with Transact-SQL loops / statements that return only a few information, is increased

Dynamic Execution

- ***EXEC (<command>)*** - used to execute dynamically Transact-SQL statements
- Allows as a parameter a string and execute the inside Transact-SQL code
- Drawback: may arise performance and possible security problems

- Instead of **EXEC** can be used the stored procedure **sp_executesql**

- ***sp_executesql*** – avoid a large part of the problems generated by the **SQL injection** and also can be faster than **EXEC**
- ***sp_executesql*** – allows only Unicode strings
- ***sp_executesql*** – allows input and output parameters

Dynamic Execution - example

Results | Messages

	Sid	CoId	Grade
1	1	11	9
2	1	23	5

- Classic:

  SELECT Sid, CoId, Grade FROM Exams WHERE Sid = 1
- With EXEC

  EXEC('SELECT Sid, CoId, Grade FROM Exams WHERE Sid = 1’**)**
- With EXEC - parameter

  DECLARE @s VARCHAR(MAX)

  SET @s = 'SELECT Sid, CoId, Grade FROM Exams WHERE Sid = 1'

  EXEC(@s)
- Alternative with sp_executesql

  EXECUTE sp_executesql N'SELECT Sid, CoId, Grade FROM Exams WHERE Sid = @Sid', N'@Sid INT', @Sid = 1;

The SELECT INTO Clause

- Save a result set in a table

```
IF OBJECT_ID ('dbo.StudentsCopy', 'U') IS NOT NULL
DROP TABLE dbo.StudentsCopy;
GO

select Name, Surname, Email
INTO dbo.StudentsCopy
from Students
where email like'%@uc.ro’
GO

SELECT * FROM StudentsCopy
```

Results | Messages

	Name	Surname	Email
1	Mihnea	Dan	dan@uc.ro
2	Cristea	Mihai	mcristea@uc.ro

The Output Clause

OUTPUT – give access to the added / modified / deleted rows

Select * from Courses
create table ModifiedCourses(CoId INT primary key, Tile VARCHAR(50) not null, NoOfCredits INT, ModificationDate DATETIME, Suser NVARCHAR(128)) -- Is the optional login security identification number

Select * from Courses
select * from ModifiedCourses

Results | Messages

	CoId	Title	NoOfCredits
1	11	Databases	6
2	12	Operation Systems	5
3	22	Algebra	4
4	23	Geometry	4

CoId	Tile	NoOfCredits	ModificationDate	Suser

UPDATE Courses
SET Title = 'Database Management Systems' -- 'Databases' was before this UPDATE
OUTPUT inserted.CoId, deleted.Title, inserted.NoOfCredits, GETDATE(), SUSER_SNAME() -- Is the optional login security identification number
INTO ModifiedCourses
WHERE CoId = 11

select * from Courses
select * from ModifiedCourses

Results | Messages

	CoId	Title	NoOfCredits
1	11	Database Management Systems	6
2	12	Operation Systems	5
3	22	Algebra	4
4	23	Geometry	4

	CoId	Tile	NoOfCredits	ModificationDate	Suser
1	11	Databases	6	2021-10-14 12:03:53.180	DESKTOP-VB4PGBM\A

Flow Control Language

- Transact-SQL offers a set of keywords used to control the flow of the executions of the Transact-SQL language, blocks of statements, used-defined functions, stored procedures, called ***flow control language***
- Without the flow control language, the Transact-SQL statements are executed sequential
- **BEGIN … END** – all the Transact-SQL statements inside are executed together; can be embedded

```
BEGIN
        {sql_statement | sql_block}
END
```

- **RETURN** – goes out from a query / stored procedure without any additional condition
- RETURN – can be used anywhere to get out from a stored procedure, batch, block of statements
- RETURN – used to return status codes: stored procedures return 0 (success) or an integer value not 0 (faillure)

```
RETURN [integer_expression]
```

Flow Control Language

Example: stored procedure that returns status code

```
CREATE PROCEDURE usp_check_status (@groupid INT)
AS
    BEGIN
        IF (SELECT TOP 1 Name FROM Students WHERE Gid=@groupid) = 'Roman'
            RETURN 1;
        ELSE
            RETURN 2;
    END

-- return 1
DECLARE @status INT;
EXEC @status=usp_check_status 221;
SELECT 'Status'=@status;
    -- return 2
    DECLARE @status INT;
    EXEC @status=usp_check_status 822;
    SELECT 'Status'=@status;
```

Flow Control Language

IF … ELSE – allows to put a condition on a Transact-SQL statement / block of statements

```
IF boolean_expression
        {sql_statement |  statement_block}
[ELSE
        {sql_statement |  statement_block}]
```

Example:

```
CREATE PROCEDURE uspCheckCountry @country varchar(50)
AS
IF @country = 'Romania'
RETURN 1
ELSE
RETURN 2;
GO
```

```
-- 1
DECLARE @ret_status_code int
EXEC @ret_status_code= uspCheckCountry 'Romania'
SELECT @ret_status_code
```

```
-- 2
DECLARE @ret_status_code int
EXEC @ret_status_code= uspCheckCountry 'England'
SELECT @ret_status_code
```

Flow Control Language

WHILE – set a condition to repeatably execute a Transact-SQL statement / block of statements

WHILE boolean_expression

{sql_statement | statement_block | BREAK | CONTINUE}

BREAK – goes out from the most inside WHILE loop or from any IF … ELSE statement that is inside of a WHILE loop

CONTINUE – start again the WHILE loop and ignore all the statements after it

Example:

- with Break

```
DECLARE @i INT = 1;
WHILE @i <= 5
BEGIN
   SET @i = @i + 1;
   IF @i = 4
      BREAK;
   PRINT @i;
END
```

- with break & continue

```
DECLARE @i INT = 1;
WHILE @i <= 8
BEGIN
   SET @i = @i + 1;
   IF @i = 4
      CONTINUE;
      IF @i = 6
      BREAK;
   PRINT @i;
END
```

Flow Control Language

GOTO label – execute a jump in execution of a part of code marked in a label

label: -- some Transact-SQL statements
GOTO label

Example:

```
DECLARE @i int = 1;
WHILE @i <=8
BEGIN
   SET @i = @i + 1
   IF @i = 4 GOTO Label1 -- goes to Label1
   IF @i = 6 GOTO Label2  -- goes to Label2
END
Label1:
   SELECT 'First Label'
Label2:
   PRINT 'Secund Label';
```

Flow Control Language

WAITFOR- block the execution of a batch, transaction or stored procedure until a specified interval of time or a specified time is reached or a specified statement modify or return at least one record

```
WAITFOR
{
DELAY ‘time_to_pass’ | TIME ‘time_to_execute’ | [(receive_statement) |
(get_conversation_group_statement)] [, TIMEOUT timeout]
}
```

Due to the level of activity on the server, the waiting time may vary, so it can be higher than the one specified in **WAITFOR**

Example:

```
-- execution continues at 07:15
WAITFOR TIME '07:15'
-- execution continues after 3 hours
WAITFOR DELAY '03:00'
```

```
-- between the first SELECT and the second SELECT will be
-- an extra delay of 4 secunds
select Top 1 * from Students
waitfor delay '00:00:04'
select Top 1 * from Students
```

Flow Control Language

THROW – throw an exception and transfer the execution of a CATCH block from a TRY … CATCH block

```
THROW [ {error_number | @local_variable},
        {message | @local_variable},
        {state | @local_variable} ] [;]
```

The severity of the exception is always set on the value 16

Example: THROW 51000,'50 rows have been modified’,1;

TRY … CATCH – handling the errors from Transact-SQL

```
BEGIN TRY
{sql_statement | statement_block}
END TRY
BEGIN CATCH
[{sql_statement | statement_block}]
END CATCH [;]
```

Catch all the execution errors that have the severity level higher than 10 and does not close the connection to the database

Flow Control Language

TRY … CATCH – handling the errors from Transact-SQL

- Inside of a CATCH block can be used the following system functions to get information about the error that caused the execution of the block CATCH
  - **ERROR_NUMBER()** – return the number of the error
  - **ERROR_SEVERITY()** – return the severity of the error
  - **ERROR_STATE()** – return the error state number
  - **ERROR_PROCEDURE()** – return the name of the stored procedure / trigger in which the error occurred
  - **ERROR_LINE()** – return the number of line in which the error occurred
  - **ERROR_MESSAGE()** – return the error message

Flow Control Language

TRY … CATCH – handling the errors from Transact-SQL

- **Error messages**
  - *Error number* (number of the error)
    - A value between 1 and 49999
    - For the custom errors (defined by the user), the value is between 50000 and 2147483647
  - *Error severity* (the severity of the error)
    - 26 severity levels
    - The errors with the severity level >=16 are recorded in the error log automatically
    - The errors with the severity level between 20 and 25 are fatal and close the connection instantly
  - *Error message* (the message of the error) – NVARCHAR(2048)

Flow Control Language

TRY … CATCH examples:

```
-- Divide by zero error encountered.
BEGIN TRY
   -- Generate a divide-by-zero error.
   SELECT 1/0;
END TRY
BEGIN CATCH
   SELECT
      ERROR_NUMBER() AS ErrorNumber,
                        ERROR_SEVERITY() AS ErrorSeverity,
      ERROR_STATE() AS ErrorState,
      ERROR_PROCEDURE() AS ErrorProcedure,
      ERROR_LINE() AS ErrorLine,
      ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO
```

```
-- the error is not catched
      BEGIN TRY
         -- Table does not exist; object name resolution
         -- error not caught
         SELECT * FROM NonexistentTable;
      END TRY
      BEGIN CATCH
         SELECT
           ERROR_NUMBER() AS ErrorNumber,
           ERROR_MESSAGE() AS ErrorMessage;
      END CATCH
```