**Stored Procedure Basics**

1. **What is a Stored Procedure?**
   * A stored procedure is a precompiled collection of one or more SQL statements stored under a name and processed as a unit.
2. **Creating a Simple Stored Procedure**

CREATE PROCEDURE ProcedureName

AS

BEGIN

-- SQL statements

SELECT \* FROM TableName

END

1. **Executing a Stored Procedure**

EXEC ProcedureName

1. **Dropping a Stored Procedure**

DROP PROCEDURE IF EXISTS ProcedureName

**Parameters in Stored Procedures**

1. **Input Parameters**

CREATE PROCEDURE ProcedureName

@ParameterName DataType

AS

BEGIN

-- SQL statements using @ParameterName

SELECT \* FROM TableName WHERE ColumnName = @ParameterName

END

* + **Execute with Input Parameters**

EXEC ProcedureName @ParameterName = Value

1. **Output Parameters**

CREATE PROCEDURE ProcedureName

@InputParam DataType,

@OutputParam DataType OUTPUT

AS

BEGIN

SET @OutputParam = (SELECT COUNT(\*) FROM TableName WHERE ColumnName = @InputParam);

END

* + **Execute with Output Parameters**

DECLARE @Result DataType

EXEC ProcedureName @InputParam = Value, @OutputParam = @Result OUTPUT

1. **Default Parameters**

CREATE PROCEDURE ProcedureName

@Param1 DataType = DefaultValue

AS

BEGIN

-- SQL statements

END

**Control Flow in Stored Procedures**

1. **IF...ELSE**

IF Condition

BEGIN

-- SQL statements if true

END

ELSE

BEGIN

-- SQL statements if false

END

1. **WHILE Loop**

WHILE Condition

BEGIN

-- SQL statements

END

1. **RETURN Statement**

RETURN IntegerValue

**Transactions in Stored Procedures**

1. **BEGIN, COMMIT, ROLLBACK TRANSACTION**

CREATE PROCEDURE ProcedureName

AS

BEGIN

BEGIN TRANSACTION

BEGIN TRY

-- SQL statements

COMMIT TRANSACTION

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION

-- Error handling

END CATCH

END

**Error Handling**

1. **TRY...CATCH Block**

BEGIN TRY

-- SQL statements that might throw an error

END TRY

BEGIN CATCH

-- Error handling code

SELECT ERROR\_MESSAGE() AS ErrorMessage

END CATCH

1. **RAISEERROR**

RAISERROR('Error message', SeverityLevel, State)

**Advanced Concepts**

1. **Dynamic SQL in Stored Procedures**
   * **Using sp\_executesql**

CREATE PROCEDURE ProcedureName

@DynamicSQL NVARCHAR(MAX)

AS

BEGIN

EXEC sp\_executesql @DynamicSQL

END

* + **Building Dynamic SQL with Parameters**

CREATE PROCEDURE ProcedureName

@TableName NVARCHAR(128)

AS

BEGIN

DECLARE @SQL NVARCHAR(MAX)

SET @SQL = 'SELECT \* FROM ' + @TableName

EXEC sp\_executesql @SQL

END

1. **Recursive Stored Procedures**

CREATE PROCEDURE ProcedureName

@Param DataType

AS

BEGIN

-- Base case

IF @Param <= 0

RETURN

-- Recursive case

EXEC ProcedureName @Param - 1

END

1. **Using Temporary Tables in Stored Procedures**

CREATE PROCEDURE ProcedureName

AS

BEGIN

CREATE TABLE #TempTable (Column1 DataType, Column2 DataType);

INSERT INTO #TempTable

SELECT \* FROM SomeTable

SELECT \* FROM #TempTable

END

1. **Working with Cursors**

CREATE PROCEDURE ProcedureName

AS

BEGIN

DECLARE @Variable DataType

DECLARE CursorName CURSOR FOR

SELECT Column FROM TableName

OPEN CursorName;

FETCH NEXT FROM CursorName INTO @Variable

WHILE @@FETCH\_STATUS = 0

BEGIN

-- SQL statements using @Variable

FETCH NEXT FROM CursorName INTO @Variable

END

CLOSE CursorName

DEALLOCATE CursorName

END

**Stored Procedure Best Practices**

1. **Use SET NOCOUNT ON**
   * To prevent the sending of DONE\_IN\_PROC messages to the client for each statement.

CREATE PROCEDURE ProcedureName

AS

BEGIN

SET NOCOUNT ON

-- SQL statements

END

1. **Avoid SELECT \***
   * Always specify column names to avoid issues with changes in the table schema.
2. **Minimize the Use of Cursors**
   * Cursors can be slow; consider using set-based operations instead.
3. **Use Transactions Carefully**
   * Ensure transactions are committed or rolled back to avoid locking issues.