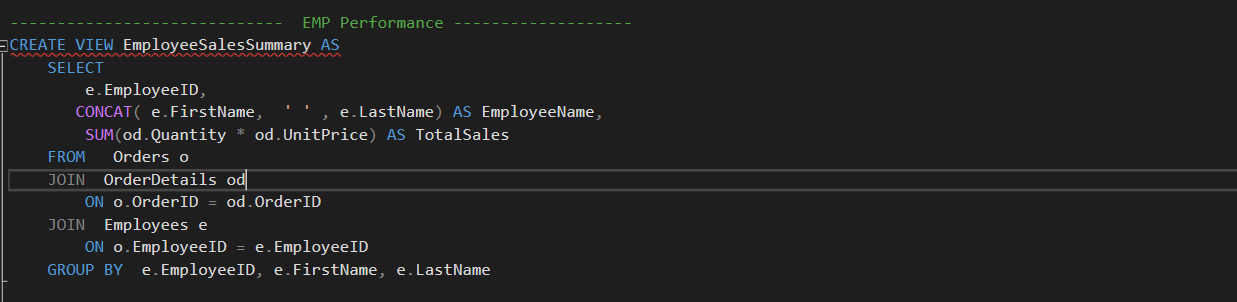
* **Views:**

1. **EMPLOYEE PERFORMANCE :**

**- Description :**

This SQL code creates a **view** called EmployeeSalesSummary, which summarizes total sales made by each employee. It joins the Orders, OrderDetails, and Employees tables to calculate total sales (Quantity \* UnitPrice) for each employee. The results are grouped by EmployeeID and employee name. This view helps in tracking employee sales performance efficiently.

**-Code:**

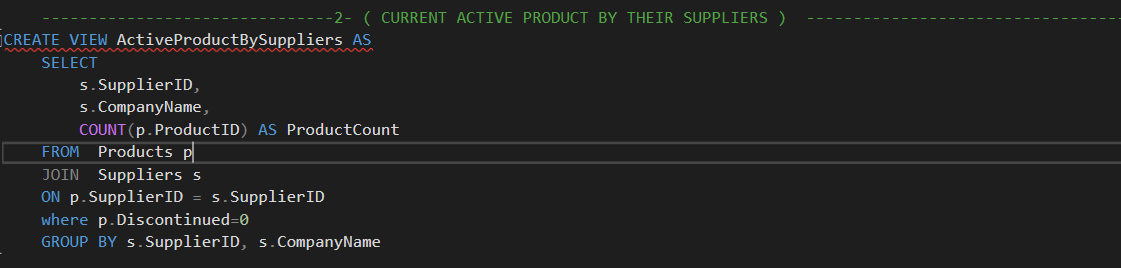
****

1. **Current Active Product By Suppliers :**

**- Description :**

This SQL code creates a **view** called ActiveProductBySuppliers, which displays the number of active (non-discontinued) products supplied by each supplier. It joins the Products and Suppliers tables based on SupplierID, filters out discontinued products (Discontinued = 0), and counts the active products for each supplier. The results are grouped by SupplierID and CompanyName. This view helps in analyzing suppliers with active product offerings.

**-Code:**

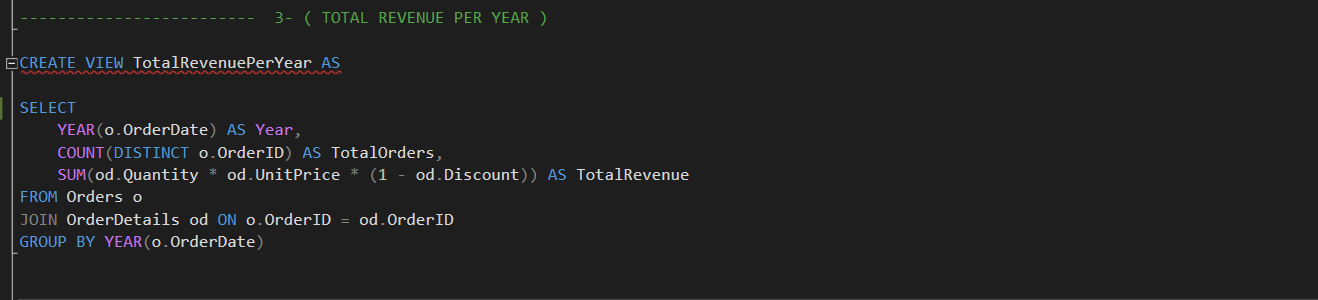
****

1. **Total Revenue Per Year :**

**- Description :**

This SQL code creates a view called TotalRevenuePerYear, which displays the total number of orders and total revenue for each year. It joins the Orders and OrderDetails tables based on OrderID, extracts the year from OrderDate, counts the distinct orders as TotalOrders, and calculates TotalRevenue by considering quantity, unit price, and discounts. The results are grouped by year, helping analyze yearly sales performance.

**-Code:**

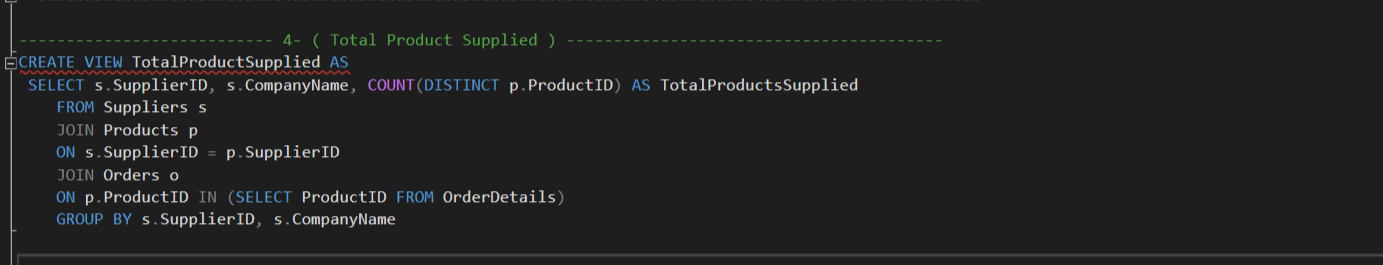
****

1. **Total Product supplied by Every Suppliers :**

**- Description :**

**This SQL code creates a view called TotalProductSupplied, which displays the total number of unique products supplied by each supplier. It joins the Suppliers and Products tables based on SupplierID, then links to the Orders table by checking if the product appears in OrderDetails. The query counts distinct ProductID values to determine the total products supplied by each supplier. The results are grouped by SupplierID and CompanyName, helping analyze supplier contributions.**

**-Code:**

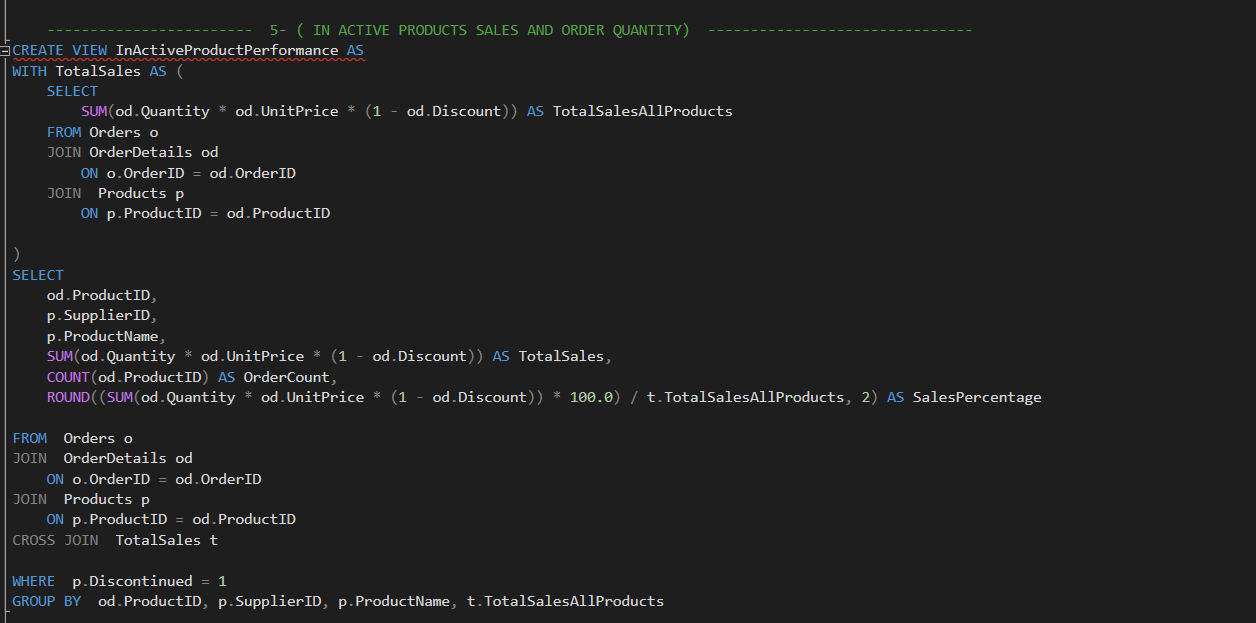
****

1. **Inactive Products & Quantity Ordered And percentage from total Sales :**

**- Description :**

This SQL code creates a view called **InActiveProductPerformance**, which analyzes the sales performance of discontinued products. It uses a Common Table Expression (CTE) called **TotalSales** to calculate the total revenue from all products. The main query retrieves the **ProductID, SupplierID**, and **ProductName** for discontinued products (Discontinued = 1), along with their total sales, order count, and sales percentage relative to total sales. The CROSS JOIN with **TotalSales** allows calculating the sales contribution of each discontinued product. The results help assess the impact of inactive products on overall sales.

**-Code:**

****

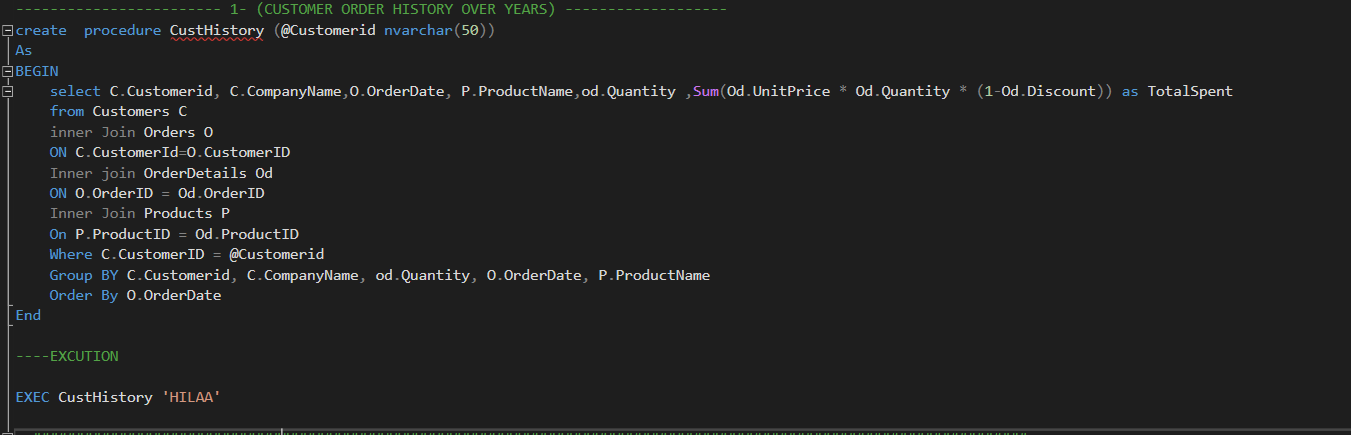
* **Stored Procedures :**

1. **Customers Order History Over Years :**

**- Description :**

**This SQL code creates a stored procedure named CustHistory, which retrieves the order history of a specific customer. It takes @Customerid as a parameter and returns details including the CustomerID, CompanyName, OrderDate, ProductName, Quantity, and the total amount spent on each product. The procedure joins the Customers, Orders, OrderDetails, and Products tables to fetch relevant data, filters by the given CustomerID, groups the results to calculate the total spent per product, and sorts them by OrderDate. This procedure helps analyze a customer's purchasing history.**

**-Code:**

****

1. **Customers Performance with most spend Product & Most Ordered Product :**

**- Description :**

This SQL code creates a stored procedure named CustPerformance, which analyzes customer spending behavior and retrieves the top N customers based on total spending (defaulting to 10).

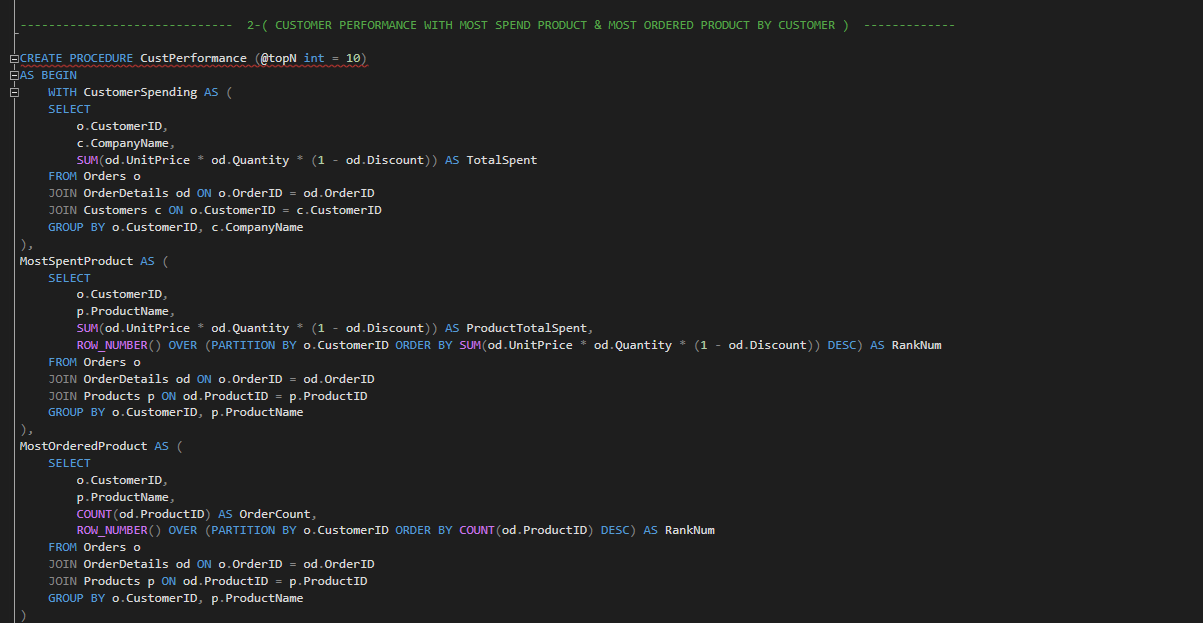
**Breakdown of Functionality:**

1. **CustomerSpending CTE**: Calculates the total amount spent by each customer.
2. **MostSpentProduct CTE**: Identifies the most expensive product each customer has spent the most money on.
3. **MostOrderedProduct CTE**: Determines the most frequently ordered product for each customer.
4. **Final Query**: Retrieves the top N customers (default 10) with their total spending, most expensive product, and most frequently ordered product, sorted by total spending in descending order.

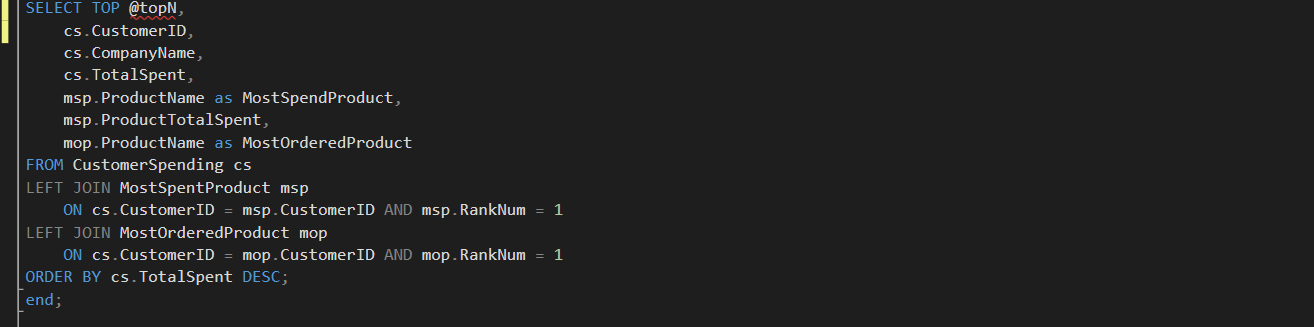
This procedure helps in analyzing customer purchasing behavior and identifying key products contributing to their spending.

**-Code:**

**-CTE’S**

****

**-Final Query**

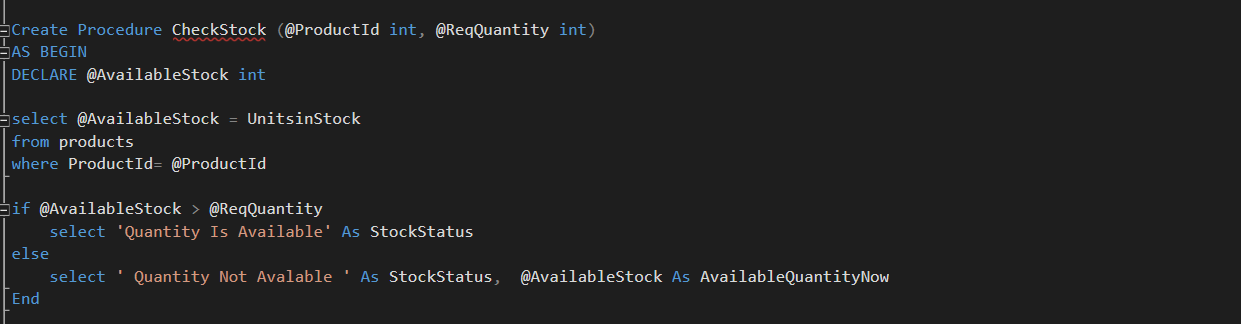
****

1. **Checking Stock Before Ordering :**

**- Description :**

**This SQL code creates a stored procedure named CheckStock, which checks the availability of a specific product in stock. It takes @ProductId and @ReqQuantity as parameters and retrieves the current stock (UnitsInStock) from the Products table. If the available stock is greater than the requested quantity, it returns 'Quantity Is Available'. Otherwise, it returns 'Quantity Not Available' along with the current available stock. This procedure helps ensure sufficient stock before processing an order.**

**- Code :**

****

1. **Automating Reorder Low Stock Products :**

**- Description :**

This SQL code creates a **table** and a **stored procedure** to manage product reorders automatically when stock levels are low.

**Breakdown of Functionality:**

**1. PurchaseOrders Table**

* Stores purchase orders with columns for ProductID, SupplierID, ReorderQuantity, OrderDate, and Status (defaulting to 'Pending').

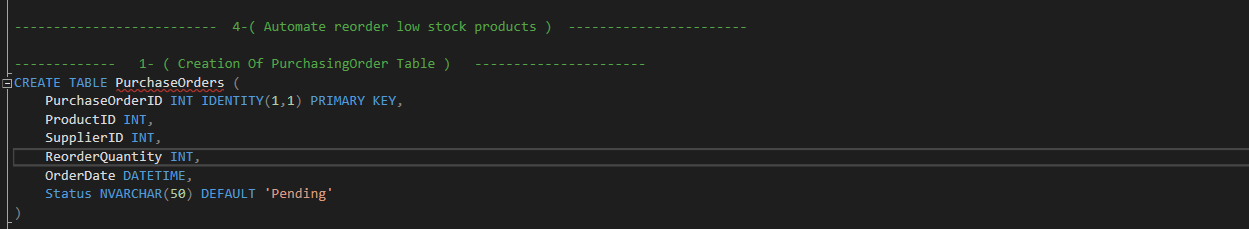
**2. AutoReorderAndInsertPurchaseOrder Stored Procedure**

* Accepts @ProductID and @OrderQuantity as input parameters.
* Retrieves stock details (UnitsInStock, ReorderLevel, SupplierID, and Discontinued status) from the **Products** table.
* If the product doesn’t exist, it prints an error message.
* Checks if **stock is below the reorder level** and if the product is **not discontinued**:
  + Calculates reorder quantity.
  + Inserts a new **purchase order** into the PurchaseOrders table.
  + Returns reorder details.
* If stock is sufficient, it displays a message indicating no reorder is needed.

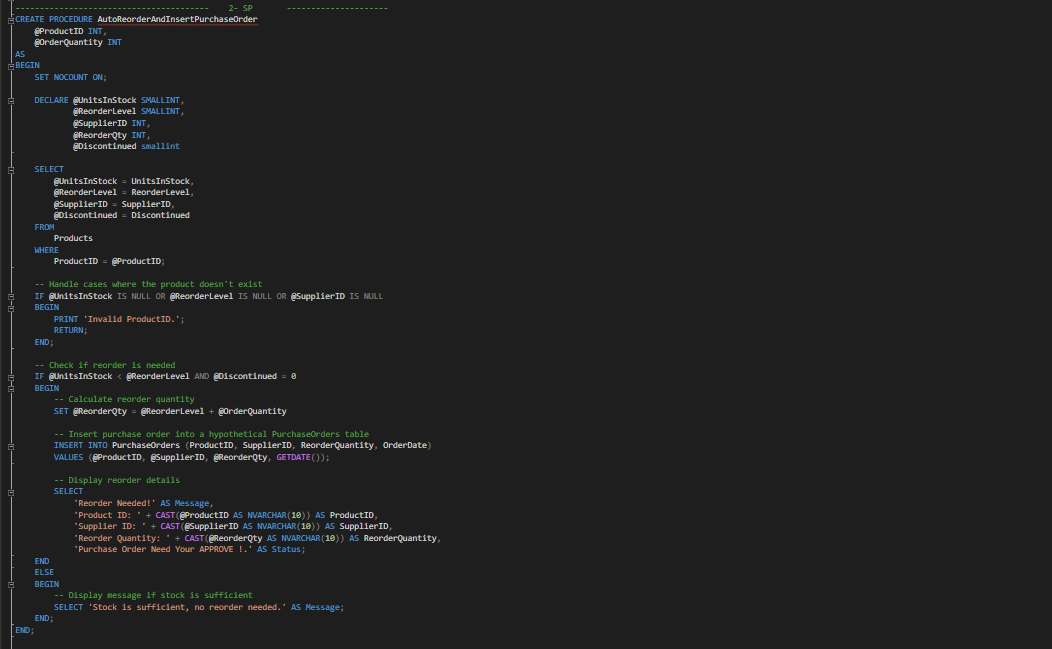
This procedure **automates reordering** and helps maintain optimal inventory levels.

**- Code :**

**- Creation of Purchasing Table**

****

**- Stored Procedure**

****

1. **Automating Reorder Low Stock Products :**

**- Description :**

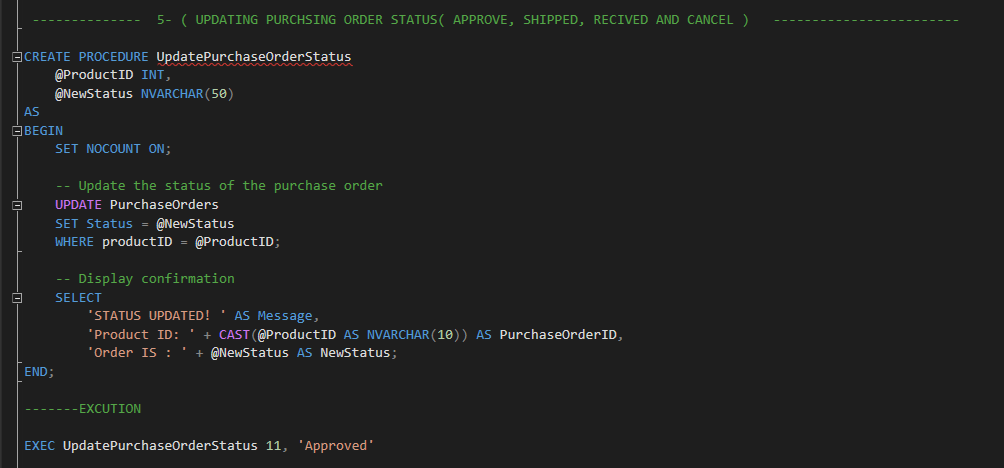
This SQL code creates a **stored procedure** named UpdatePurchaseOrderStatus, which updates the status of a purchase order based on the given ProductID.

### ****Breakdown of Functionality****:

* Accepts @ProductID and @NewStatus as input parameters.
* Updates the **Status** column in the PurchaseOrders table for the specified ProductID.
* Displays a confirmation message showing the updated status.

This procedure helps **manage and track purchase order statuses** efficiently.

**- Code :**

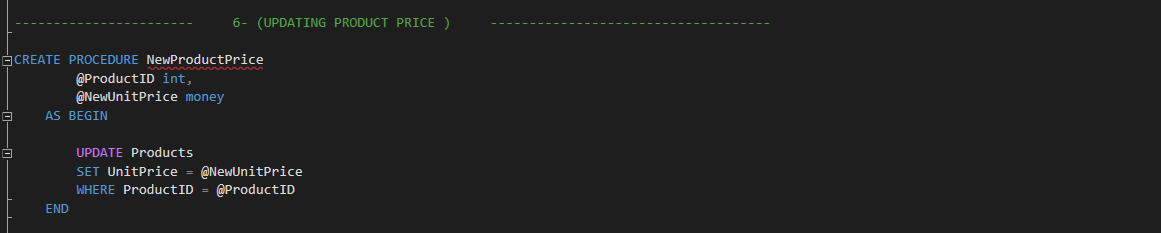
****

1. **Update Product Price :**

**- Description :**

This SQL code creates a stored procedure named NewProductPrice, which updates the unit price of a specific product. It takes @ProductID and @NewUnitPrice as parameters and updates the **UnitPrice** in the **Products** table where the ProductID matches the given value. This procedure allows for efficient and controlled price updates.

**- Code :**

****

1. **Employee Performance By Year & Month ( EMP Of The Month according to Sales) :**

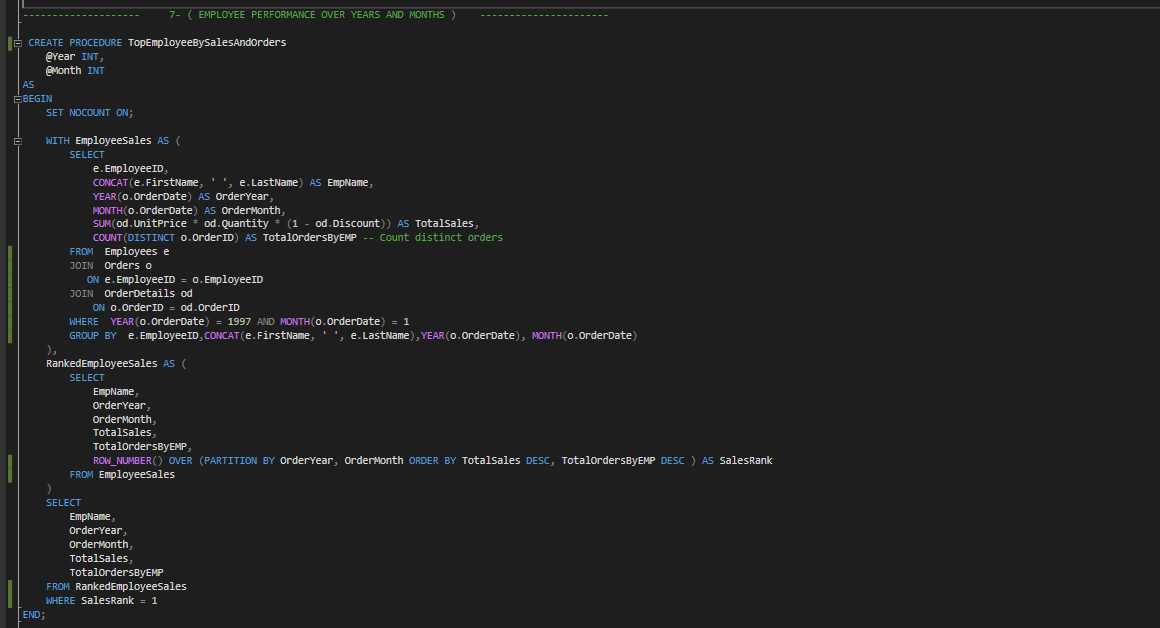
**- Description :**

This SQL code creates a stored procedure named TopEmployeeBySalesAndOrders, which retrieves the top-performing employee based on sales and order count for a given year and month.

**Breakdown of Functionality:**

* Takes @Year and @Month as input parameters.
* The EmployeeSales CTE calculates total sales and distinct orders for each employee within the specified month and year.
* The RankedEmployeeSales CTE assigns a rank to employees based on their total sales and order count.
* The final query selects the top-ranked employee with the highest sales and orders for the given period.

**- Code :**

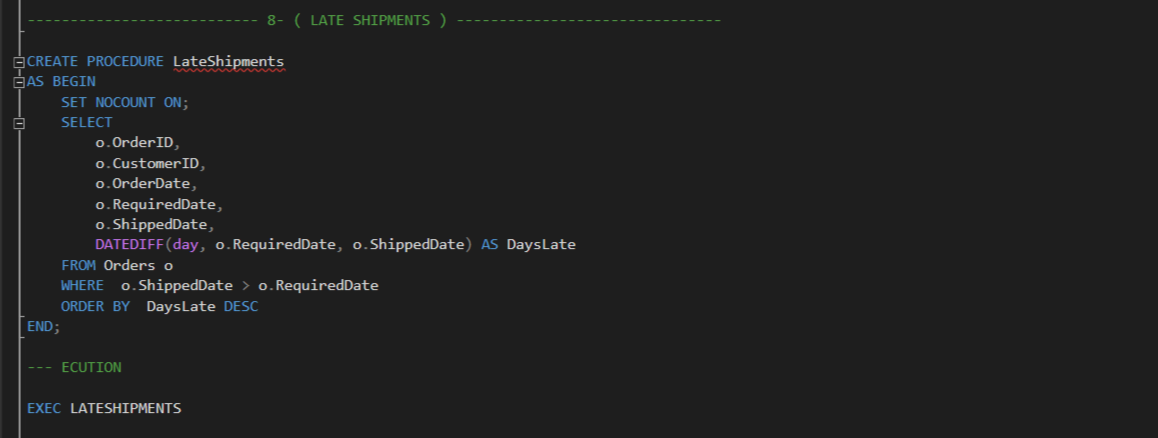
****

1. **Late Shipment :**

**- Description :**

This SQL code creates a stored procedure named LateShipments, which retrieves orders that were shipped later than their required delivery date. It selects order details, calculates the delay in days, filters for late shipments, and sorts them in descending order of delay duration. This procedure helps in tracking and analyzing late shipments for better logistics management.

**- Code :**

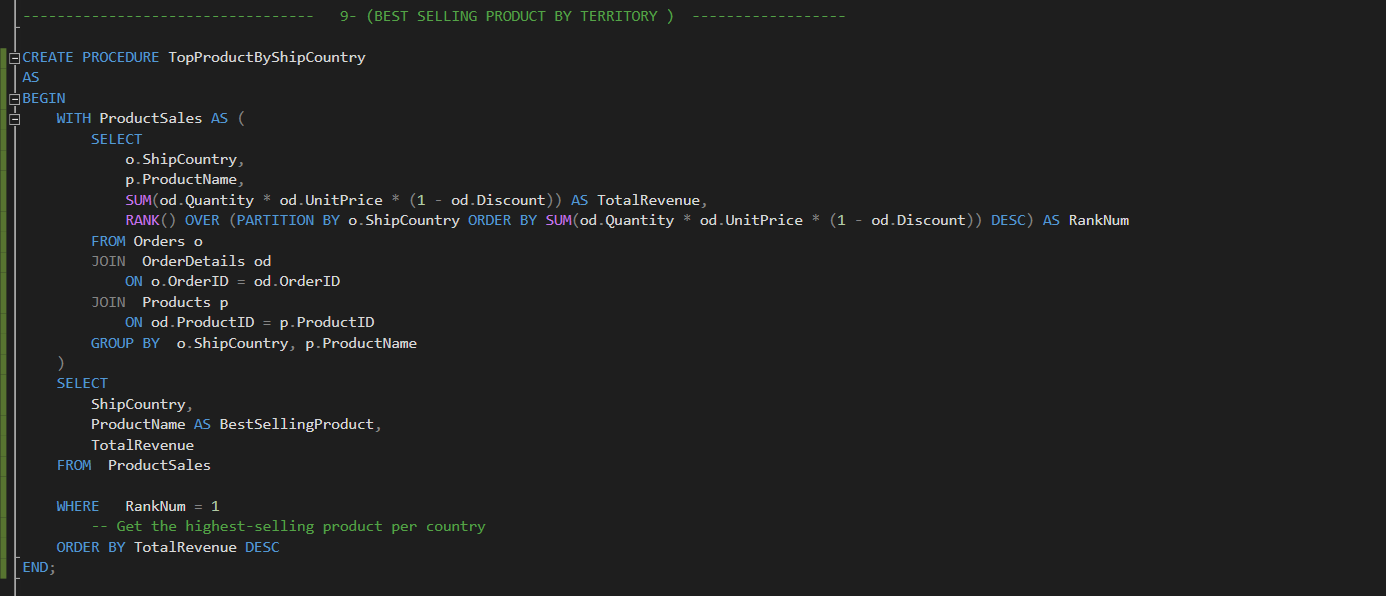
****

1. **Best Selling Products By Country :**

**- Description :**

This SQL code creates a stored procedure named TopProductByShipCountry, which retrieves the best-selling product for each shipping country based on total revenue. It calculates total sales for each product per country, ranks them using the RANK() function, and selects the top-ranked product in each country. The results are sorted by total revenue in descending order, helping to identify the most profitable products in different regions.

**- Code :**



1. **Numbers Of Orders By Customers & their Country in a Period Of Time:**

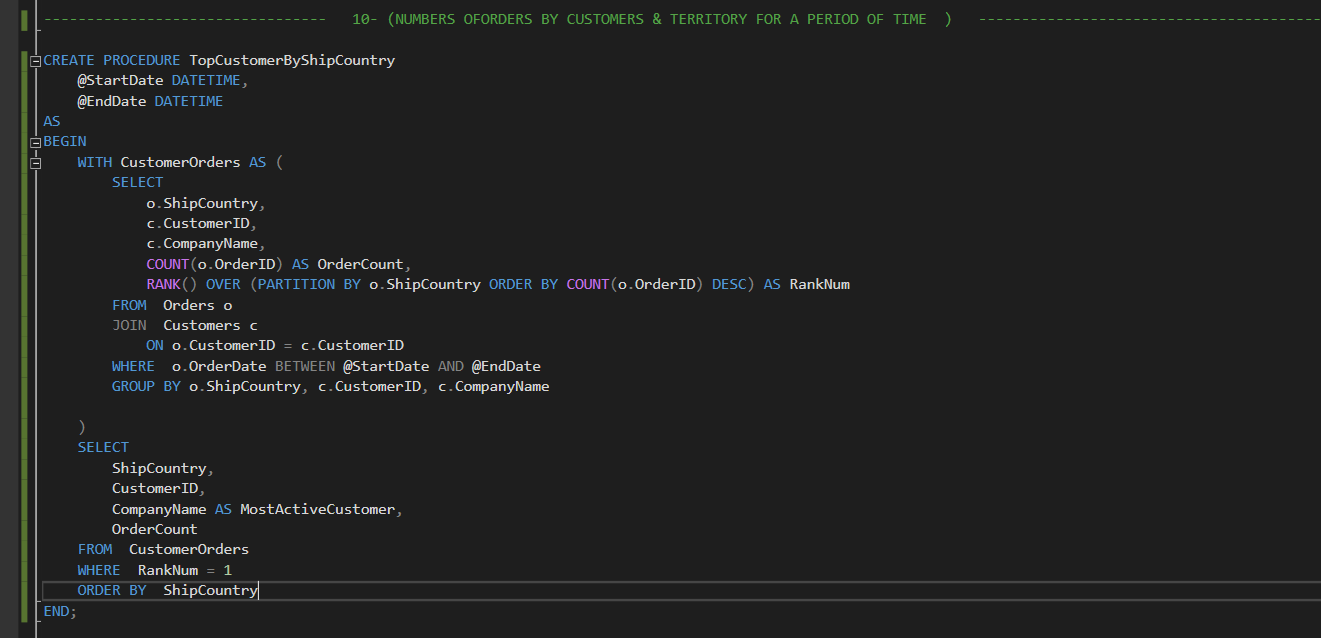
**- Description :**

This SQL code creates a stored procedure named TopCustomerByShipCountry, which retrieves the most active customer in each shipping country within a specified date range.

**Breakdown of Functionality:**

* Takes @StartDate and @EndDate as input parameters to filter orders within the given period.
* Calculates the total number of orders placed by each customer per shipping country.
* Uses the RANK() function to determine the customer with the highest order count in each country.
* Returns the shipping country, customer ID, company name, and total order count, sorted by country.

**- Code :**

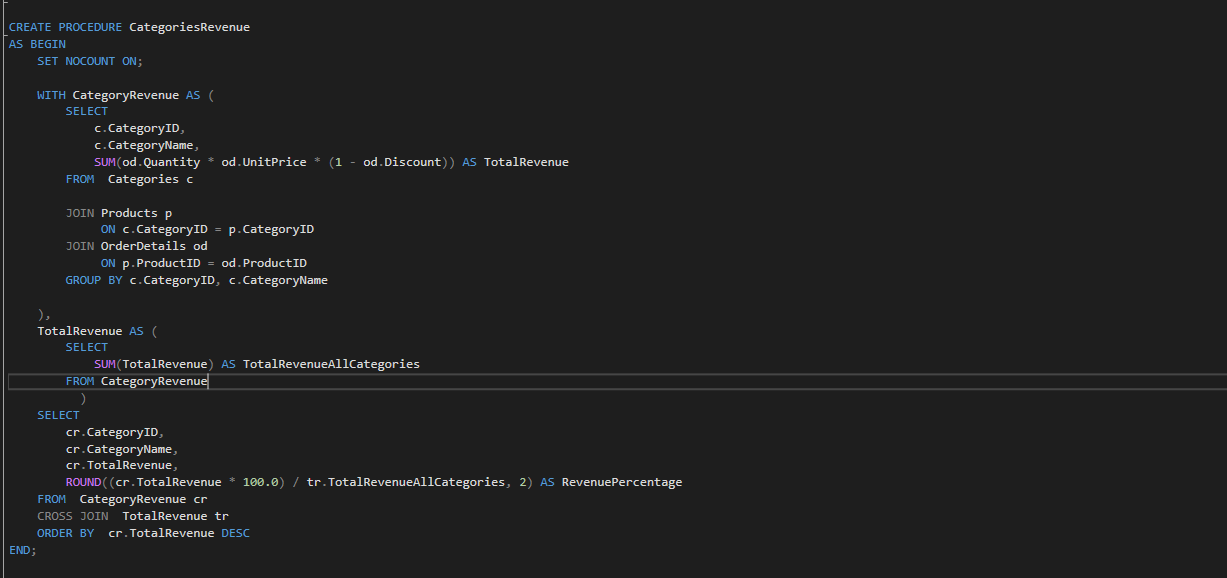
****

1. **Revenue By Category:**

**- Description :**

This SQL code creates a stored procedure named CategoriesRevenue, which calculates the total revenue generated by each product category. It retrieves category details and computes total sales revenue by summing product sales, factoring in quantity, unit price, and discounts. The procedure also calculates the percentage contribution of each category's revenue to the overall total revenue. Finally, it returns the category ID, name, total revenue, and revenue percentage, sorted in descending order of revenue.

**- Code :**

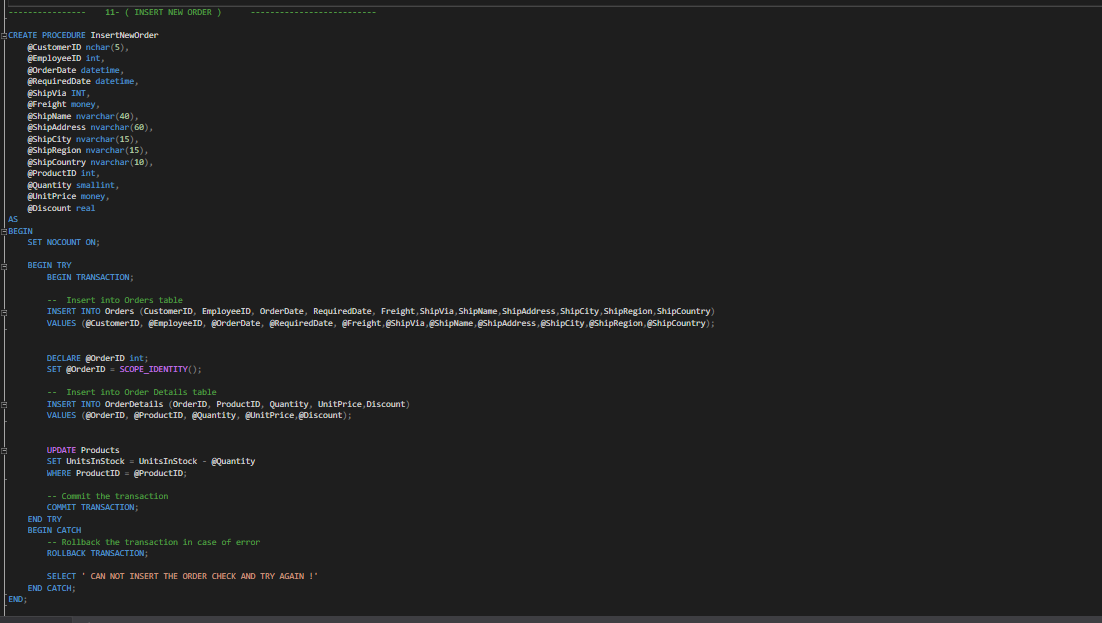
****

1. **Inserting New Order:**

**- Description :**

This SQL code creates a stored procedure named InsertNewOrder, which inserts a new order into the Orders and OrderDetails tables while updating product stock levels. It takes multiple parameters, including customer and shipping details, product ID, quantity, unit price, and discount. The procedure ensures data consistency by using a transaction—committing the changes if successful or rolling back if an error occurs. If an issue arises, it returns an error message indicating that the order could not be inserted.

**- Code:**

****

* **Triggers :**
  1. **Validate Purchasing Order Status :**

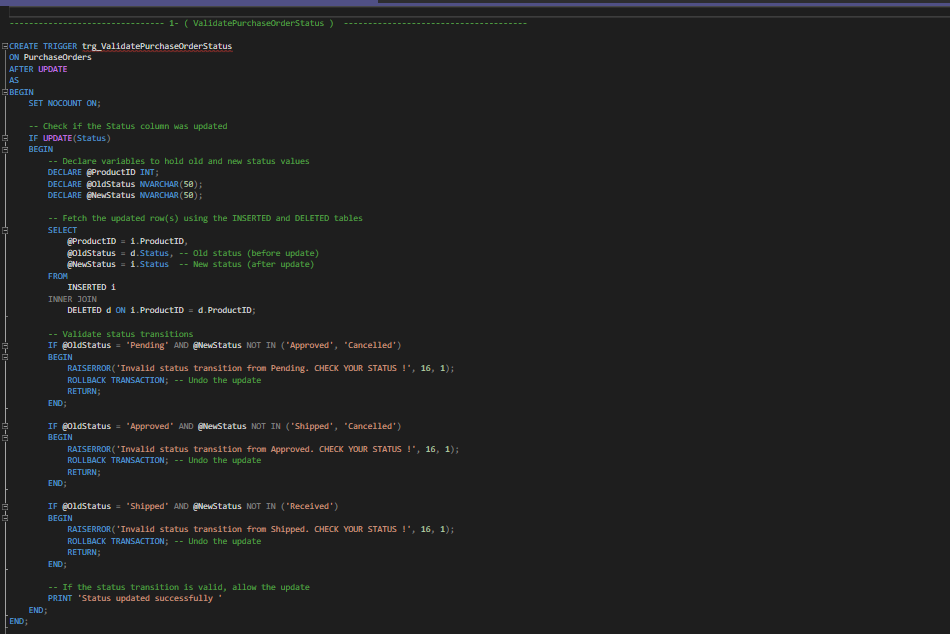
**- Description :**

This SQL code creates a trigger named trg\_ValidatePurchaseOrderStatus, which enforces valid status transitions in the PurchaseOrders table. The trigger activates **after an update** on the table and checks if the Status column was modified. It retrieves the **old and new status values** using the INSERTED and DELETED tables and ensures that transitions follow predefined rules:

* **"Pending"** can only change to **"Approved"** or **"Cancelled"**
* **"Approved"** can only change to **"Shipped"** or **"Cancelled"**
* **"Shipped"** can only change to **"Received"**

If an invalid transition is detected, the trigger **raises an error and rolls back the update** to maintain data integrity. Otherwise, the update is allowed, and a success message is printed.

**- Code :**

****

* 1. **Update ShipDate ( Order Status ) :**

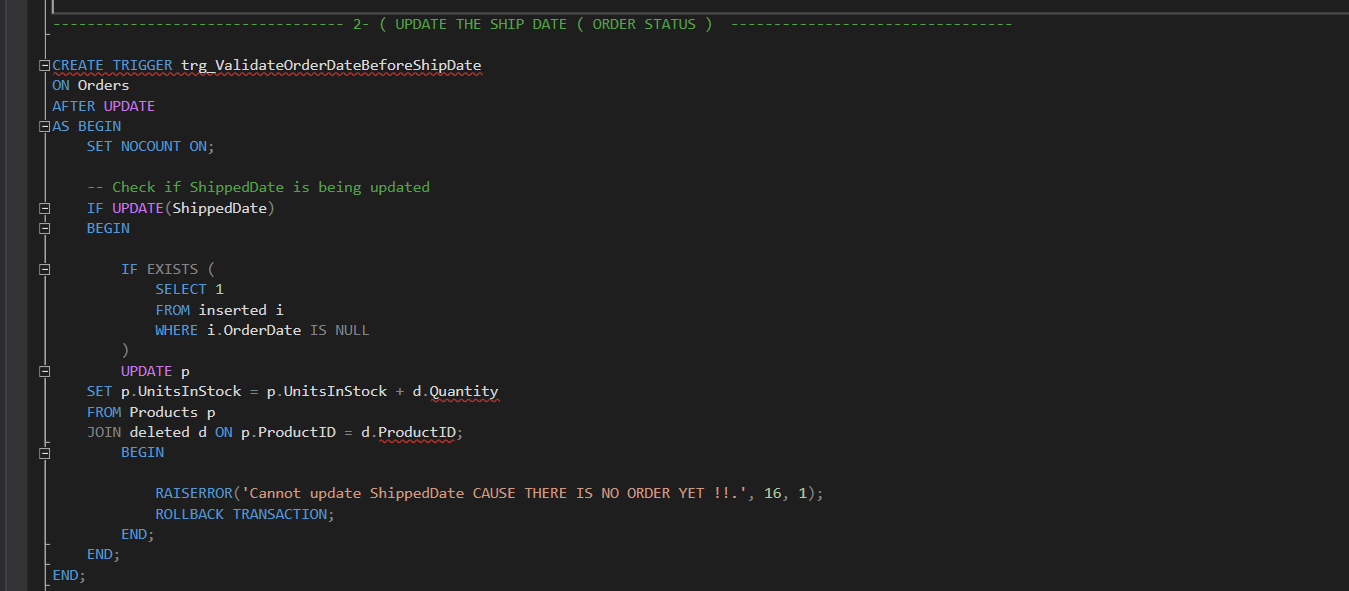
**- Description :**

This SQL code creates a **trigger** named trg\_ValidateOrderDateBeforeShipDate, which enforces a rule in the Orders table to ensure that the ShippedDate is not updated when the OrderDate is missing. The trigger executes **AFTER an UPDATE** on the table and performs the following checks:

* It verifies if the ShippedDate column is being updated.
* If any updated row has a **NULL OrderDate**, it means the order does not exist yet.
* In this case, the trigger **raises an error** and **rolls back the transaction** to prevent the update.

Additionally, there is an **incorrectly placed UPDATE statement** in the trigger, which attempts to restore product stock by increasing UnitsInStock in the Products table using the deleted table. However, it is **misplaced inside the IF EXISTS condition** and will not execute correctly. The logic should be revised to function properly.

**- Code :**

****

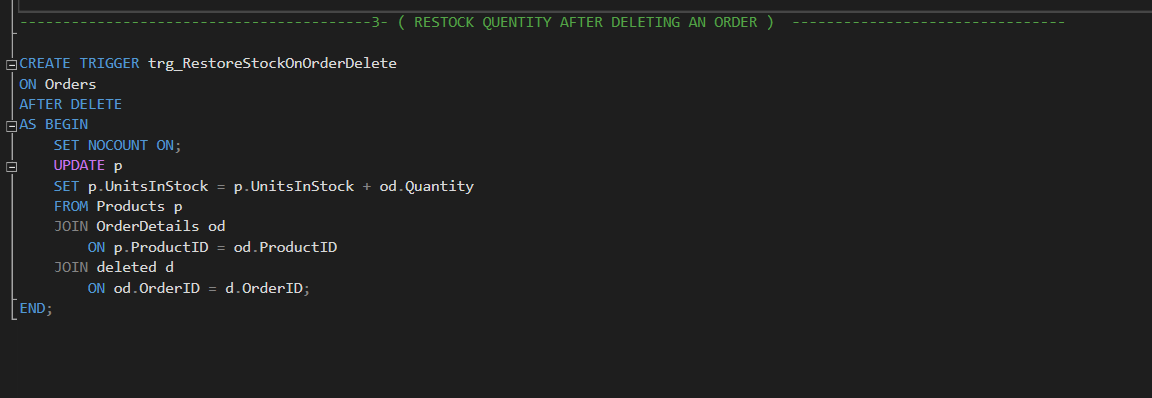
* 1. **Restock Quantity After Deleting an Order :**

**- Description :**

This SQL code creates a **trigger** named trg\_RestoreStockOnOrderDelete, which automatically restores stock levels when an order is deleted from the Orders table.

* The trigger executes **AFTER a DELETE** operation on Orders.
* It updates the UnitsInStock in the Products table by adding back the quantity of products from the deleted order.
* This ensures that when an order is removed, the associated products are returned to stock.

**- Code :**

****