Stored Procedures, Views, Triggers, Constraint and Indexes

Database objects are the fundamental structures used to **store, manage, and manipulate** data in a database. They help organize data efficiently and ensure data integrity.

Without database objects, managing data would be **chaotic, slow, and error prone.**

Each of these objects plays a crucial role in maintaining the **efficiency, security, and integrity** of the database.

The most common database objects include:

1. Stored Procedures
2. Views
3. Triggers
4. Constraints
5. Indexes

1. Stored Procedures

A **Stored Procedure** is a **precompiled collection of SQL statements** that can be executed as a single unit.

It allows you to **store reusable SQL code** in the database, improving performance and maintainability.

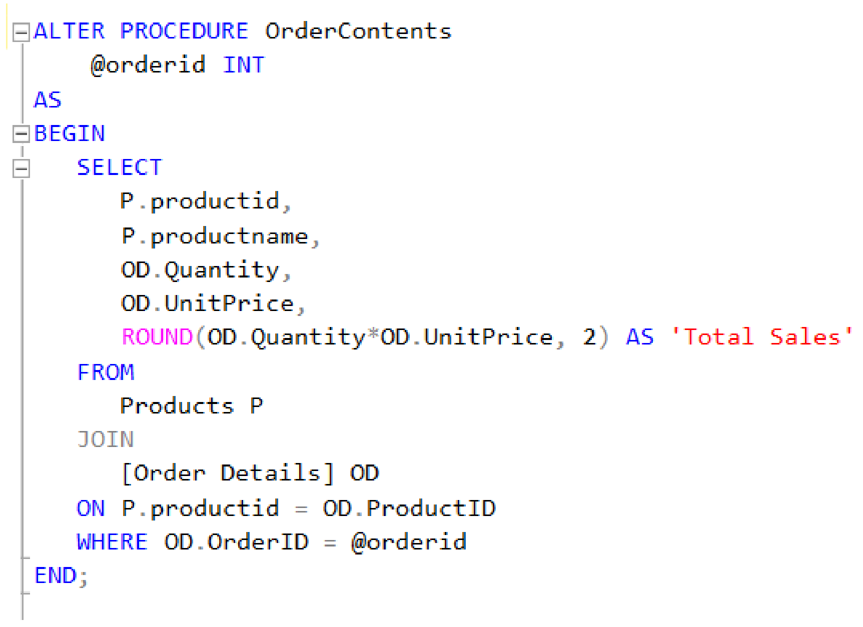
It could help with:

1. **Reusability** – Write once, use multiple times without rewriting SQL queries.
2. **Performance Optimization** – Precompiled, reducing execution time.
3. **Security** – Restrict access to tables by allowing users to execute procedures instead.
4. **Maintainability** – Centralized logic makes updating business rules easier.
5. **Reduced Network Traffic** – Instead of sending multiple queries, call one procedure.

49 Stored Procedures have been created:

1. **OrderContents**

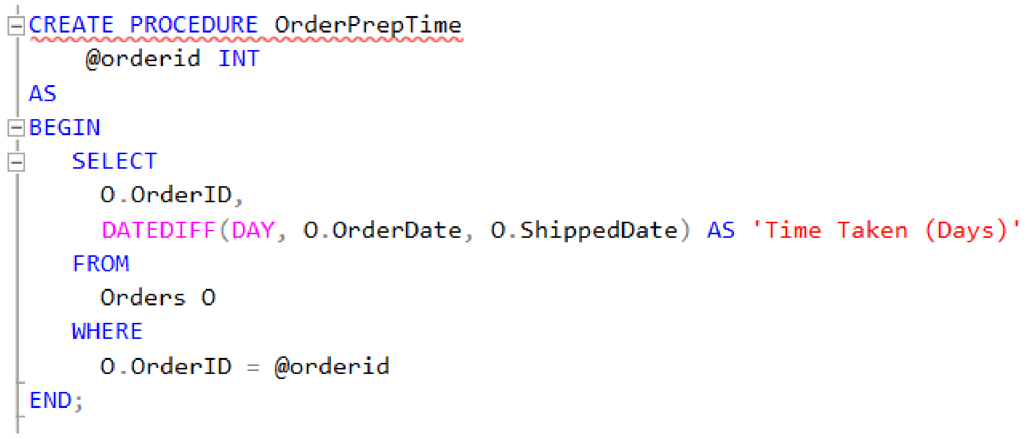
The OrderContents procedure takes an **@orderid** as input and returns the **product details**, **quantity**, **unit price**, and **total sales** for each item in the specified order. It helps analyze the contents and value of individual orders.



```sql
ALTER PROCEDURE OrderContents
    @orderid INT
AS
BEGIN
    SELECT
        P.productid,
        P.productname,
        OD.Quantity,
        OD.UnitPrice,
        ROUND(OD.Quantity*OD.UnitPrice, 2) AS 'Total Sales'
    FROM
        Products P
    JOIN
        [Order Details] OD
    ON P.productid = OD.ProductID
    WHERE OD.OrderID = @orderid
END;
```

2. **OrderPrepTime**

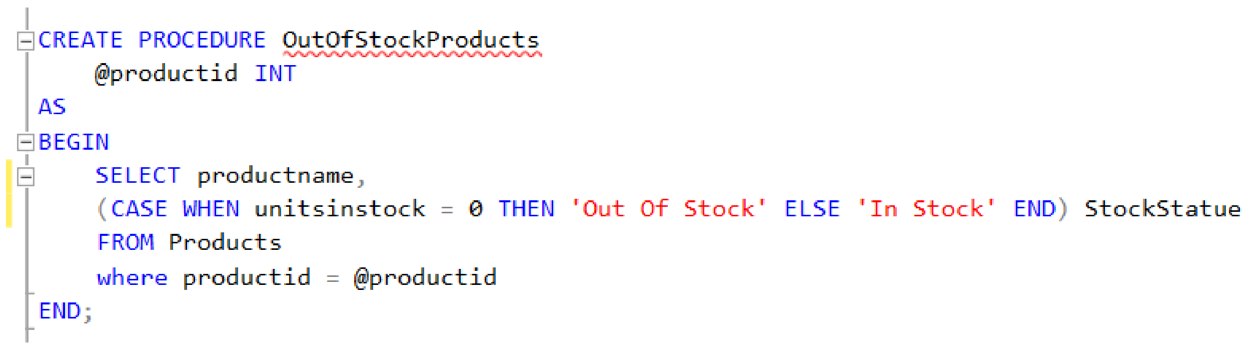
The OrderPrepTime procedure takes an **@orderid** as input and calculates the **number of days** between the OrderDate and ShippedDate for the specified order. It helps monitor order processing efficiency and delivery timelines.



```sql
CREATE PROCEDURE OrderPrepTime
    @orderid INT
AS
BEGIN
    SELECT
      O.OrderID,
      DATEDIFF(DAY, O.OrderDate, O.ShippedDate) AS 'Time Taken (Days)'
    FROM
      Orders O
    WHERE
      O.OrderID = @orderid
END;
```

3. **OutOfStockProducts**

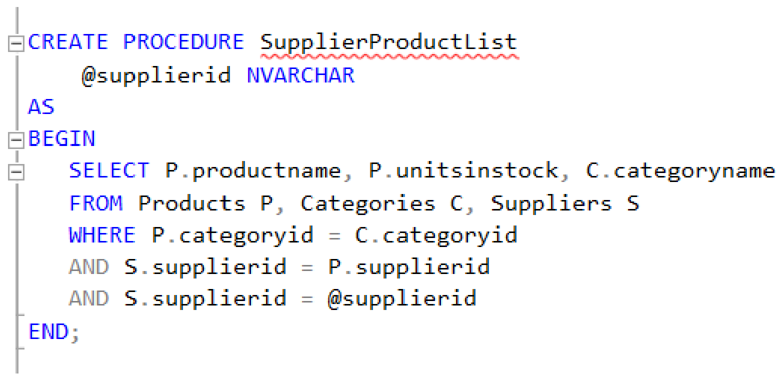
The OutOfStockProducts procedure takes a **@productid** as input and returns the **product name** and its **stock status** (either **'Out Of Stock'** or **'In Stock'**) based on
the unitsinstock value. It helps quickly identify the availability of a specific product



```
CREATE PROCEDURE OutOfStockProducts
    @productid INT
AS
BEGIN
    SELECT productname,
    (CASE WHEN unitsinstock = 0 THEN 'Out Of Stock' ELSE 'In Stock' END) StockStatue
    FROM Products
    where productid = @productid
END;
```

4. SupplierProductList

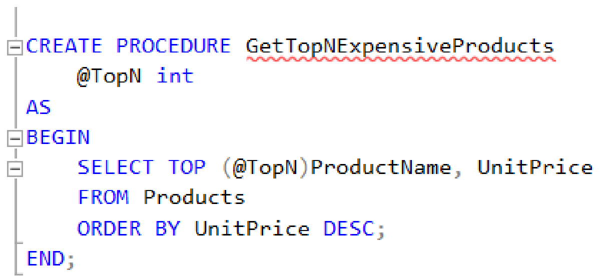
The SupplierProductList procedure takes a **@supplierid** as input and returns the **product name**, **units in stock**, and **category name** for all products associated with that supplier. It helps track and manage inventory supplied by specific vendors.



```
CREATE PROCEDURE SupplierProductList
    @supplierid NVARCHAR
AS
BEGIN
   SELECT P.productname, P.unitsinstock, C.categoryname
   FROM Products P, Categories C, Suppliers S
   WHERE P.categoryid = C.categoryid
   AND S.supplierid = P.supplierid
   AND S.supplierid = @supplierid
END;
```

5. GetTopNExpensiveProducts

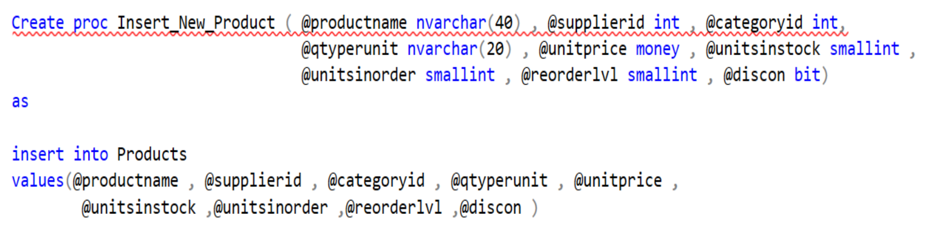
The GetTopNExpensiveProducts procedure takes an integer **@TopN** as input and returns the **top N most expensive products** based on their UnitPrice. It helps identify high-value products for pricing or promotional strategies.



```
CREATE PROCEDURE GetTopNExpensiveProducts
    @TopN int
AS
BEGIN
    SELECT TOP (@TopN)ProductName, UnitPrice
    FROM Products
    ORDER BY UnitPrice DESC;
END;
```

6. InsertNewProduct

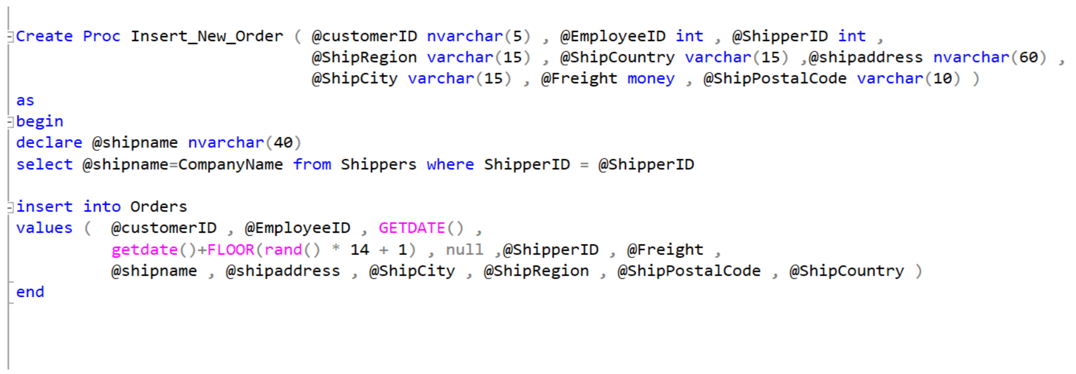
Inserts a new product into the **Products** table, including product name, supplier ID, category ID, quantity per unit, unit price, stock levels, reorder level, and whether the product is discontinued.



```sql
Create proc Insert_New_Product ( @productname nvarchar(40) , @supplierid int , @categoryid int,
                                  @qtyperunit nvarchar(20) , @unitprice money , @unitsinstock smallint ,
                                  @unitsinorder smallint , @reorderlvl smallint , @discon bit)
as

insert into Products
values(@productname , @supplierid , @categoryid , @qtyperunit , @unitprice ,
        @unitsinstock ,@unitsinorder ,@reorderlvl ,@discon )
```

7. InsertNewOrder

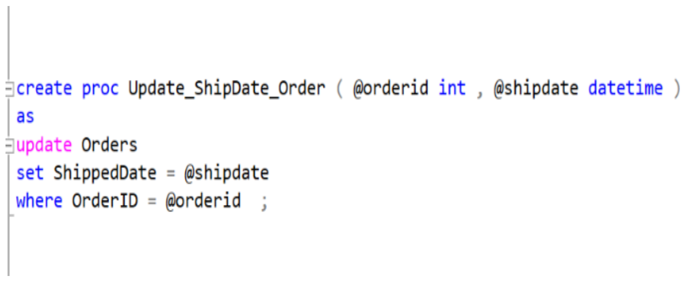
Creates a new order in the **Orders** table with customer ID, employee ID, shipper ID, shipping details, and freight cost. It also retrieves the shipper's name from the **Shippers** table and assigns an estimated shipping date.



```sql
Create Proc Insert_New_Order ( @customerID nvarchar(5) , @EmployeeID int , @ShipperID int ,
                               @ShipRegion varchar(15) , @ShipCountry varchar(15) ,@shipaddress nvarchar(60) ,
                               @ShipCity varchar(15) , @Freight money , @ShipPostalCode varchar(10) )
as
begin
declare @shipname nvarchar(40)
select @shipname=CompanyName from Shippers where ShipperID = @ShipperID

insert into Orders
values (  @customerID , @EmployeeID , GETDATE() ,
          getdate()+FLOOR(rand() * 14 + 1) , null ,@ShipperID , @Freight ,
          @shipname , @shipaddress , @ShipCity , @ShipRegion , @ShipPostalCode , @ShipCountry )
end
```

8. UpdateShipDateOrder

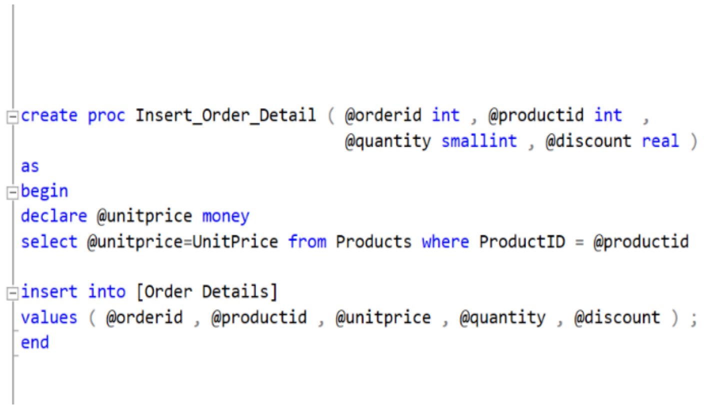
Updates the shipping date of an existing order in the **Orders** table based on the provided order ID and ship date.



```sql
create proc Update_ShipDate_Order ( @orderid int , @shipdate datetime )
as
update Orders
set ShippedDate = @shipdate
where OrderID = @orderid  ;
```

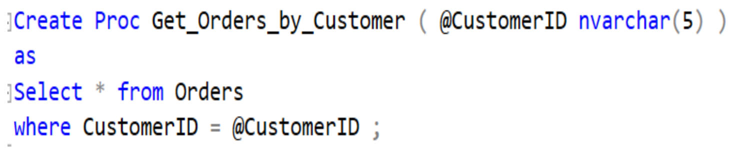
9. InsertOrderDetail

Adds a new order detail entry into the **Order Details** table, including product ID, quantity, discount, and unit price, which is fetched from the **Products** table.



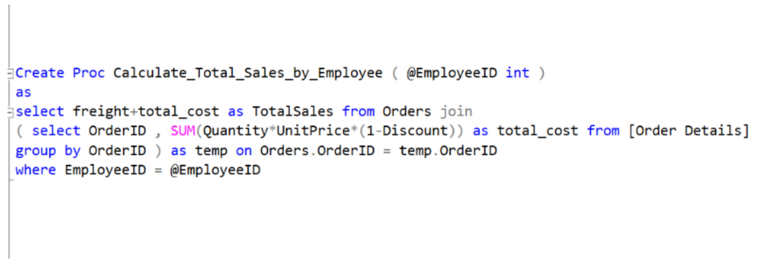
```
create proc Insert_Order_Detail ( @orderid int , @productid int  ,
                                  @quantity smallint , @discount real )
as
begin
declare @unitprice money
select @unitprice=UnitPrice from Products where ProductID = @productid

insert into [Order Details]
values ( @orderid , @productid , @unitprice , @quantity , @discount ) ;
end
```

10. **GetOrdersbyCustomer**
    Retrieves all orders placed by a specific customer, identified by their customer ID, from the **Orders** table.



```
Create Proc Get_Orders_by_Customer ( @CustomerID nvarchar(5) )
as
Select * from Orders
where CustomerID = @CustomerID ;
```

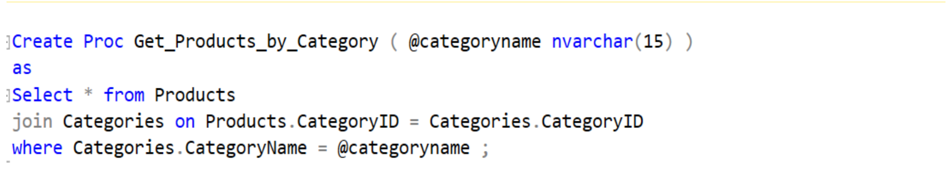
11. **CalculateTotalSalesByEmployee**
    Computes the total sales made by a specific employee by summing up order freight costs and total product costs from the **Orders** and **Order Details** tables.



```
Create Proc Calculate_Total_Sales_by_Employee ( @EmployeeID int )
as
select freight+total_cost as TotalSales from Orders join
( select OrderID , SUM(Quantity*UnitPrice*(1-Discount)) as total_cost from [Order Details]
group by OrderID ) as temp on Orders.OrderID = temp.OrderID
where EmployeeID = @EmployeeID
```

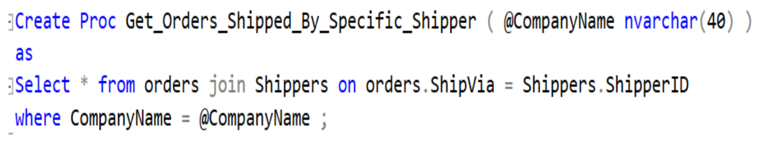
12. **GetProductsByCategory**

Returns a list of all products that belong to a specific category by joining the **Products** and **Categories** tables based on the category name.



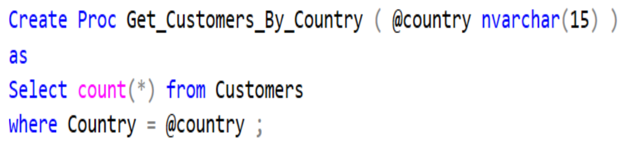
```sql
Create Proc Get_Products_by_Category ( @categoryname nvarchar(15) )
as
Select * from Products
join Categories on Products.CategoryID = Categories.CategoryID
where Categories.CategoryName = @categoryname ;
```

13. **GetOrdersShippedBySpecificShipper**
    Retrieves all orders that were shipped using a specific shipping company by matching the shipper's company name in the **Shippers** table.



```sql
Create Proc Get_Orders_Shipped_By_Specific_Shipper ( @CompanyName nvarchar(40) )
as
Select * from orders join Shippers on orders.ShipVia = Shippers.ShipperID
where CompanyName = @CompanyName ;
```

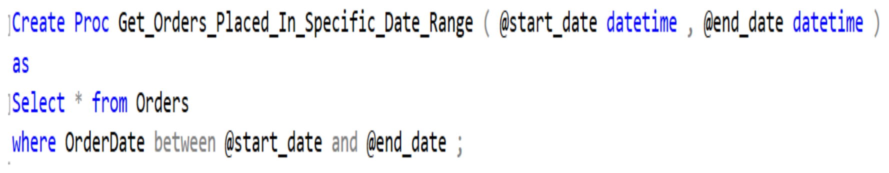
14. **GetCustomersByCountry**
    Counts and returns the total number of customers from a specific country by filtering the **Customers** table based on the country name.



```sql
Create Proc Get_Customers_By_Country ( @country nvarchar(15) )
as
Select count(*) from Customers
where Country = @country ;
```

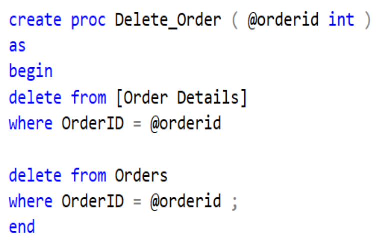
15. **GetOrdersPlacedInSpecificDateRange**

Returns all orders that were placed within a specified date range by filtering records from the **Orders** table based on order date.


```
Create Proc Get_Orders_Placed_In_Specific_Date_Range ( @start_date datetime , @end_date datetime )
as
Select * from Orders
where OrderDate between @start_date and @end_date ;
```

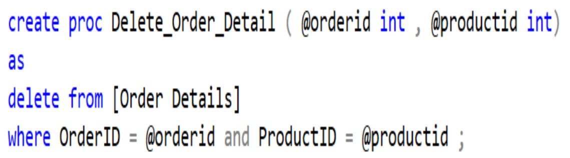
16. DeleteOrder

Deletes an order and all its associated details from the **Orders** and **OrderDetails** tables based on the order ID.


```
create proc Delete_Order ( @orderid int )
as
begin
delete from [Order Details]
where OrderID = @orderid

delete from Orders
where OrderID = @orderid ;
end
```

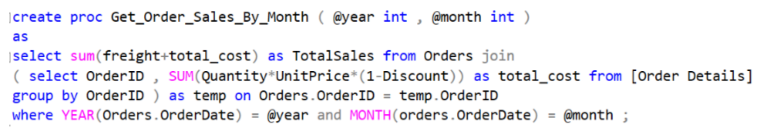
17. DeleteOrderDetail

Removes a specific product from an order in the **Order Details** table by using the order ID and product ID.


```
create proc Delete_Order_Detail ( @orderid int , @productid int)
as
delete from [Order Details]
where OrderID = @orderid and ProductID = @productid ;
```

18. GetOrderSalesByMonth

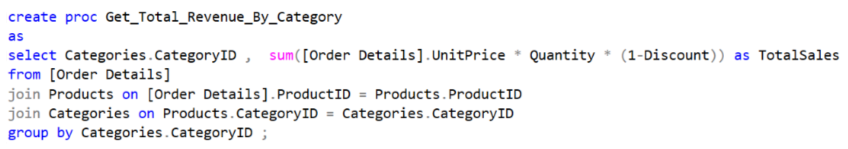
Calculates and returns the total sales for a given year and month by summing up freight costs and product sales from the **Orders** and **Order Details** tables.



```
create proc Get_Order_Sales_By_Month ( @year int , @month int )
as
select sum(freight+total_cost) as TotalSales from Orders join
( select OrderID , SUM(Quantity*UnitPrice*(1-Discount)) as total_cost from [Order Details]
group by OrderID ) as temp on Orders.OrderID = temp.OrderID
where YEAR(Orders.OrderDate) = @year and MONTH(orders.OrderDate) = @month ;
```

19. GetTotalRevenueByCategory

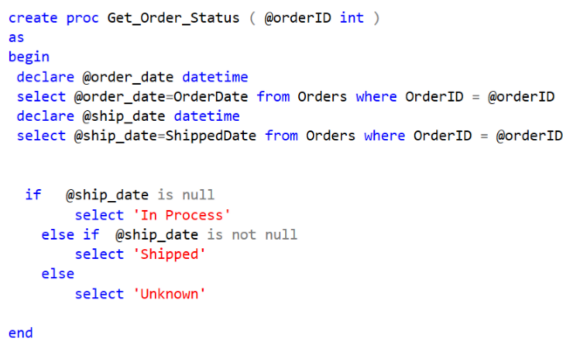
Computes the total revenue for each product category by summing up the sales of all products in the **Order Details** table and grouping them by category.



```
create proc Get_Total_Revenue_By_Category
as
select Categories.CategoryID ,  sum([Order Details].UnitPrice * Quantity * (1-Discount)) as TotalSales
from [Order Details]
join Products on [Order Details].ProductID = Products.ProductID
join Categories on Products.CategoryID = Categories.CategoryID
group by Categories.CategoryID ;
```

20. GetOrderStatus

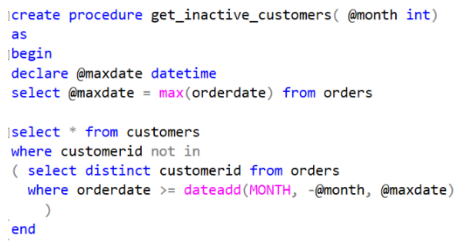
Determines the current status of an order. If the order has not been shipped, it returns **"In Process"**; if it has been shipped, it returns **"Shipped"**; otherwise, it returns **"Unknown"**.



```
create proc Get_Order_Status ( @orderID int )
as
begin
 declare @order_date datetime
 select @order_date=OrderDate from Orders where OrderID = @orderID
 declare @ship_date datetime
 select @ship_date=ShippedDate from Orders where OrderID = @orderID


  if   @ship_date is null
        select 'In Process'
    else if  @ship_date is not null
        select 'Shipped'
    else
        select 'Unknown'

end
```

21. GetInactiveCustomers

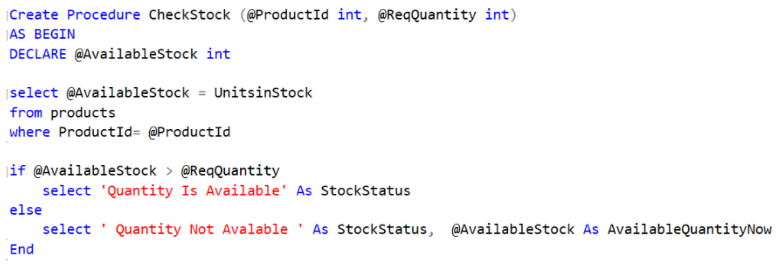
Identifies customers who have not placed any orders within the last given number of months by comparing the most recent order dates.



```sql
create procedure get_inactive_customers( @month int)
as
begin
declare @maxdate datetime
select @maxdate = max(orderdate) from orders

select * from customers
where customerid not in
( select distinct customerid from orders
  where orderdate >= dateadd(MONTH, -@month, @maxdate)
    )
end
```

22. CheckStock

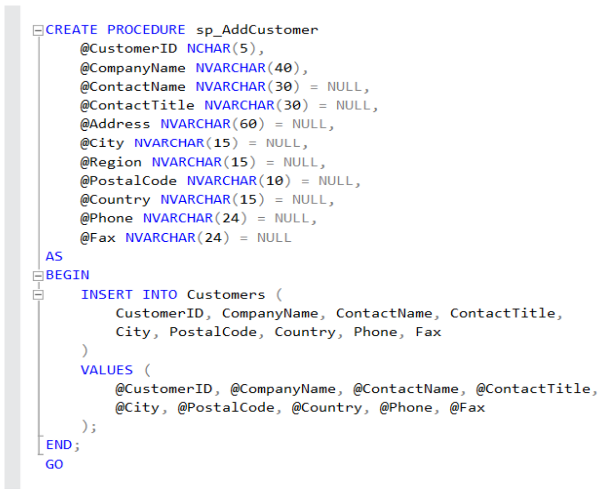
Checks if a product has enough available stock for a requested quantity. If sufficient stock is available, it confirms availability; otherwise, it returns a message indicating the current available quantity.



```sql
Create Procedure CheckStock (@ProductId int, @ReqQuantity int)
AS BEGIN
DECLARE @AvailableStock int

select @AvailableStock = UnitsinStock
from products
where ProductId= @ProductId

if @AvailableStock > @ReqQuantity
    select 'Quantity Is Available' As StockStatus
else
    select ' Quantity Not Avalable ' As StockStatus,  @AvailableStock As AvailableQuantityNow
End
```

23. AddCustomer

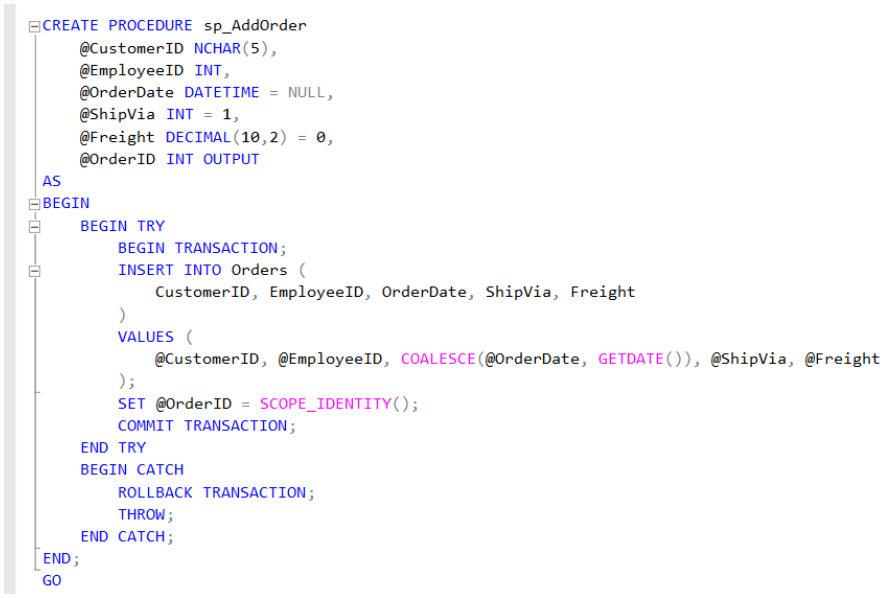
Inserts a new customer into the **Customers** table with required and optional contact and address details.



```sql
CREATE PROCEDURE sp_AddCustomer
    @CustomerID NCHAR(5),
    @CompanyName NVARCHAR(40),
    @ContactName NVARCHAR(30) = NULL,
    @ContactTitle NVARCHAR(30) = NULL,
    @Address NVARCHAR(60) = NULL,
    @City NVARCHAR(15) = NULL,
    @Region NVARCHAR(15) = NULL,
    @PostalCode NVARCHAR(10) = NULL,
    @Country NVARCHAR(15) = NULL,
    @Phone NVARCHAR(24) = NULL,
    @Fax NVARCHAR(24) = NULL
AS
BEGIN
    INSERT INTO Customers (
        CustomerID, CompanyName, ContactName, ContactTitle,
        City, PostalCode, Country, Phone, Fax
    )
    VALUES (
        @CustomerID, @CompanyName, @ContactName, @ContactTitle,
        @City, @PostalCode, @Country, @Phone, @Fax
    );
END;
GO
```

24. AddOrder

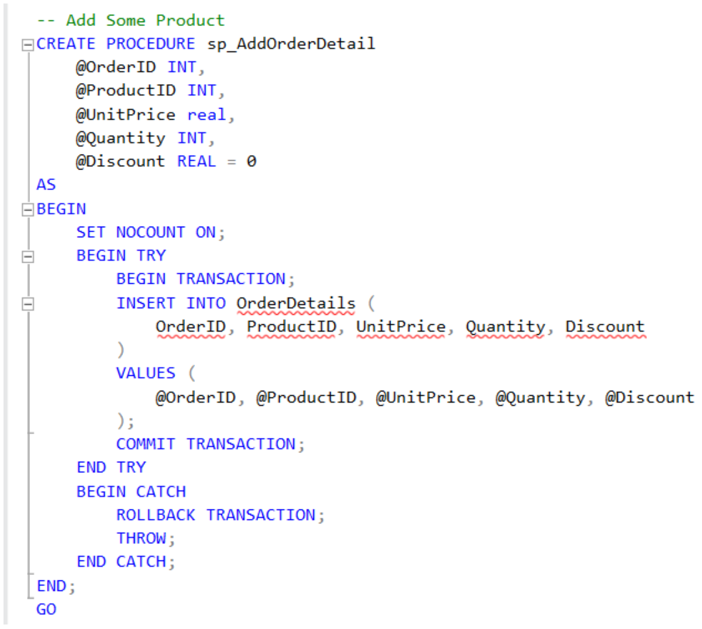
Create a new order in the **Orders t**able with a transaction to ensure data integrity, returning the generated **OrderID.** Use the current date if **OrderDate** is not provided.



```sql
CREATE PROCEDURE sp_AddOrder
    @CustomerID NCHAR(5),
    @EmployeeID INT,
    @OrderDate DATETIME = NULL,
    @ShipVia INT = 1,
    @Freight DECIMAL(10,2) = 0,
    @OrderID INT OUTPUT
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;
        INSERT INTO Orders (
            CustomerID, EmployeeID, OrderDate, ShipVia, Freight
        )
        VALUES (
            @CustomerID, @EmployeeID, COALESCE(@OrderDate, GETDATE()), @ShipVia, @Freight
        );
        SET @OrderID = SCOPE_IDENTITY();
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        THROW;
    END CATCH;
END;
GO
```

25. AddOrderDetail

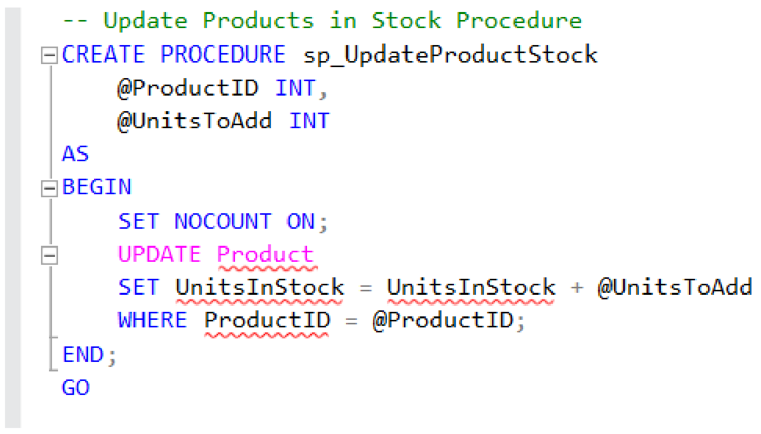
Adds a new order detail entry to the **[Order Details]** table within a transaction, ensuring atomicity for order line items.



```sql
-- Add Some Product
CREATE PROCEDURE sp_AddOrderDetail
    @OrderID INT,
    @ProductID INT,
    @UnitPrice real,
    @Quantity INT,
    @Discount REAL = 0
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRANSACTION;
        INSERT INTO OrderDetails (
            OrderID, ProductID, UnitPrice, Quantity, Discount
        )
        VALUES (
            @OrderID, @ProductID, @UnitPrice, @Quantity, @Discount
        );
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        THROW;
    END CATCH;
END;
GO
```

26. UpdateProductStock

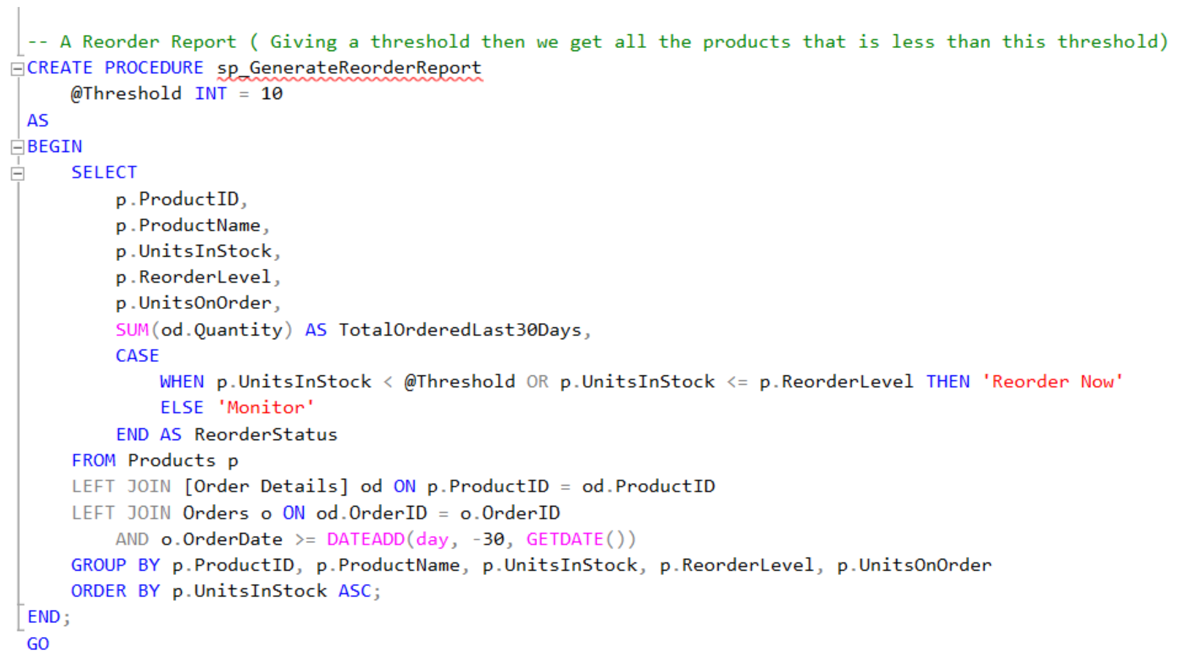
Updates the **UnitsInStock** for a specified product by adding a given number of units, useful for restocking operations.



```sql
-- Update Products in Stock Procedure
CREATE PROCEDURE sp_UpdateProductStock
    @ProductID INT,
    @UnitsToAdd INT
AS
BEGIN
    SET NOCOUNT ON;
    UPDATE Product
    SET UnitsInStock = UnitsInStock + @UnitsToAdd
    WHERE ProductID = @ProductID;
END;
GO
```

27. GenerateReorderReport

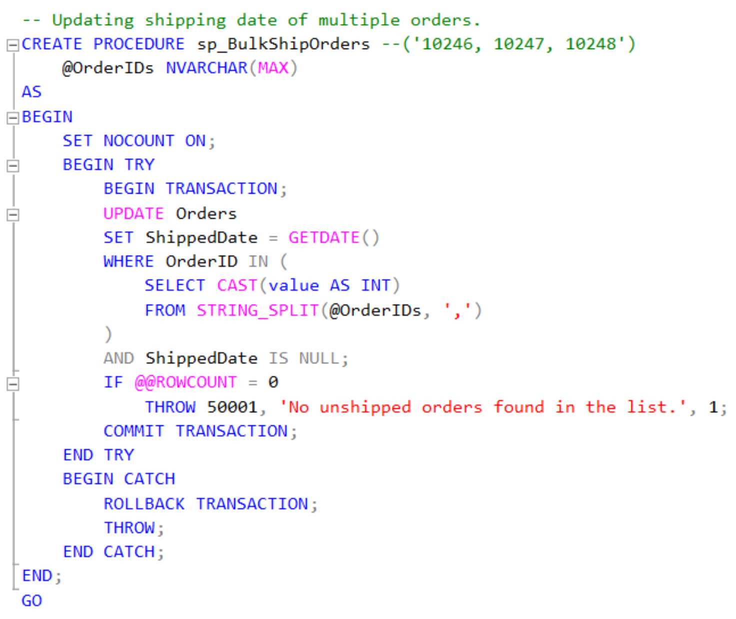
Generates a report of products with stock levels below a specified threshold or reorder level, including order history from the **last 30 days** and reorder status.



```sql
-- A Reorder Report ( Giving a threshold then we get all the products that is less than this threshold)
CREATE PROCEDURE sp_GenerateReorderReport
    @Threshold INT = 10
AS
BEGIN
    SELECT
        p.ProductID,
        p.ProductName,
        p.UnitsInStock,
        p.ReorderLevel,
        p.UnitsOnOrder,
        SUM(od.Quantity) AS TotalOrderedLast30Days,
        CASE
            WHEN p.UnitsInStock < @Threshold OR p.UnitsInStock <= p.ReorderLevel THEN 'Reorder Now'
            ELSE 'Monitor'
        END AS ReorderStatus
    FROM Products p
    LEFT JOIN [Order Details] od ON p.ProductID = od.ProductID
    LEFT JOIN Orders o ON od.OrderID = o.OrderID
        AND o.OrderDate >= DATEADD(day, -30, GETDATE())
    GROUP BY p.ProductID, p.ProductName, p.UnitsInStock, p.ReorderLevel, p.UnitsOnOrder
    ORDER BY p.UnitsInStock ASC;
END;
GO
```

28. BulkShipOrders

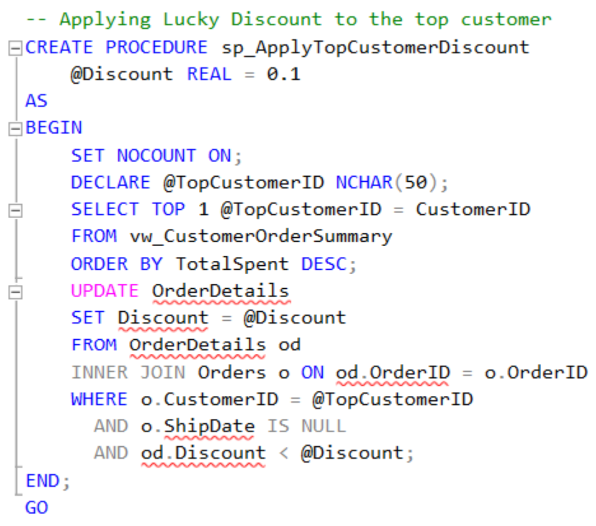
Updates the **ShippedDate** to the current date for a comma-separated list of unshipped order IDs, with error handling if no valid orders are found.



```sql
-- Updating shipping date of multiple orders.
CREATE PROCEDURE sp_BulkShipOrders --('10246, 10247, 10248')
    @OrderIDs NVARCHAR(MAX)
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRANSACTION;
        UPDATE Orders
        SET ShippedDate = GETDATE()
        WHERE OrderID IN (
            SELECT CAST(value AS INT)
            FROM STRING_SPLIT(@OrderIDs, ',')
        )
        AND ShippedDate IS NULL;
        IF @@ROWCOUNT = 0
            THROW 50001, 'No unshipped orders found in the list.', 1;
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        THROW;
    END CATCH;
END;
GO
```

29. ApplyTopCustomerDiscount

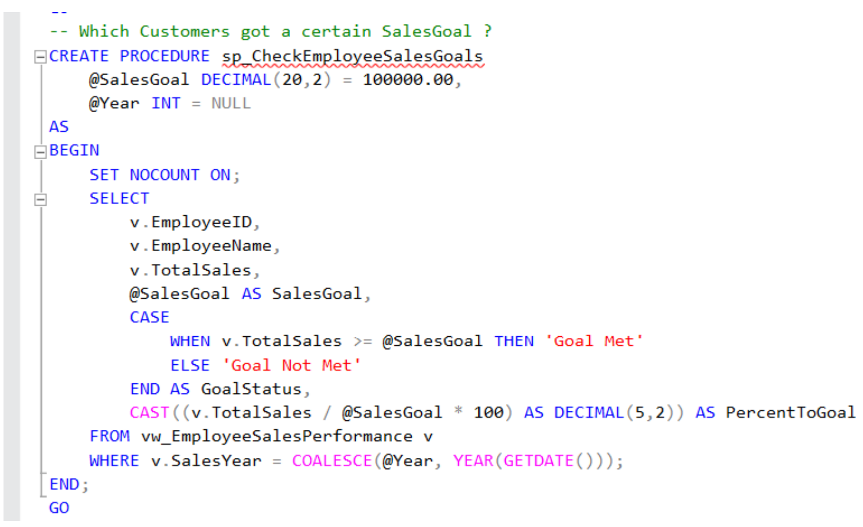
Applies a discount to unshipped orders of the top-spending customer (based on **vw_CustomerOrderSummary),** updating existing lower discounts.



```sql
-- Applying Lucky Discount to the top customer
CREATE PROCEDURE sp_ApplyTopCustomerDiscount
    @Discount REAL = 0.1
AS
BEGIN
    SET NOCOUNT ON;
    DECLARE @TopCustomerID NCHAR(50);
    SELECT TOP 1 @TopCustomerID = CustomerID
    FROM vw_CustomerOrderSummary
    ORDER BY TotalSpent DESC;
    UPDATE OrderDetails
    SET Discount = @Discount
    FROM OrderDetails od
    INNER JOIN Orders o ON od.OrderID = o.OrderID
    WHERE o.CustomerID = @TopCustomerID
      AND o.ShipDate IS NULL
      AND od.Discount < @Discount;
END;
GO
```

30. CheckEmployeeSalesGoals

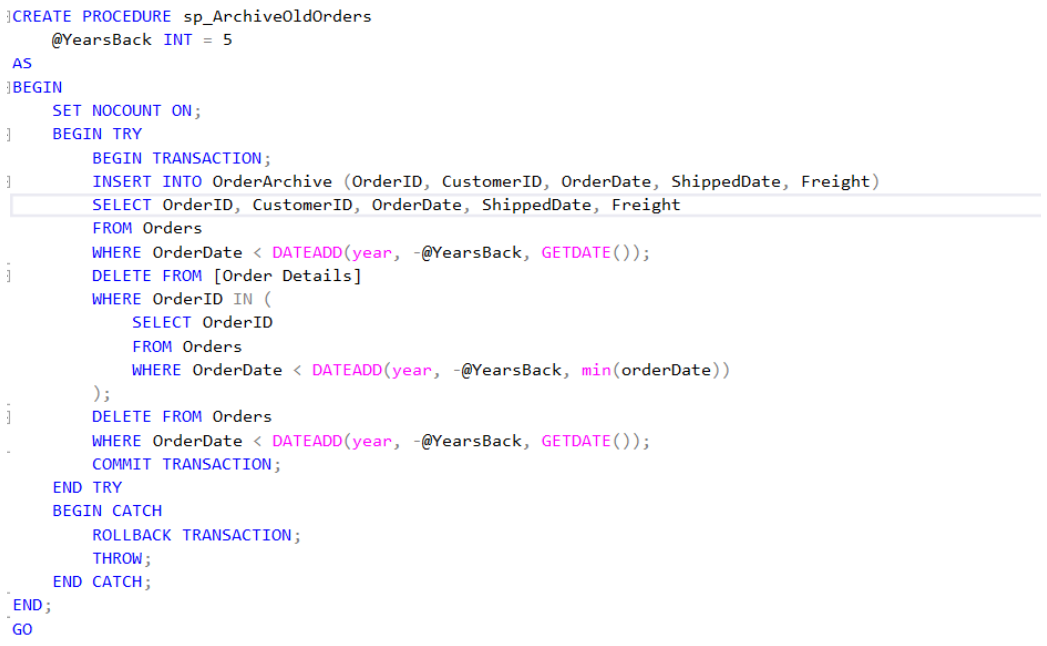
Evaluates employee sales performance against a specified sales goal for a given year, returning status and percentage to goal based on **vw_EmployeeSalesPerformance.**



```sql
--
-- Which Customers got a certain SalesGoal ?
CREATE PROCEDURE sp_CheckEmployeeSalesGoals
    @SalesGoal DECIMAL(20,2) = 100000.00,
    @Year INT = NULL
AS
BEGIN
    SET NOCOUNT ON;
    SELECT
        v.EmployeeID,
        v.EmployeeName,
        v.TotalSales,
        @SalesGoal AS SalesGoal,
        CASE
            WHEN v.TotalSales >= @SalesGoal THEN 'Goal Met'
            ELSE 'Goal Not Met'
        END AS GoalStatus,
        CAST((v.TotalSales / @SalesGoal * 100) AS DECIMAL(5,2)) AS PercentToGoal
    FROM vw_EmployeeSalesPerformance v
    WHERE v.SalesYear = COALESCE(@Year, YEAR(GETDATE()));
END;
GO
```

31. ArchiveOldOrders

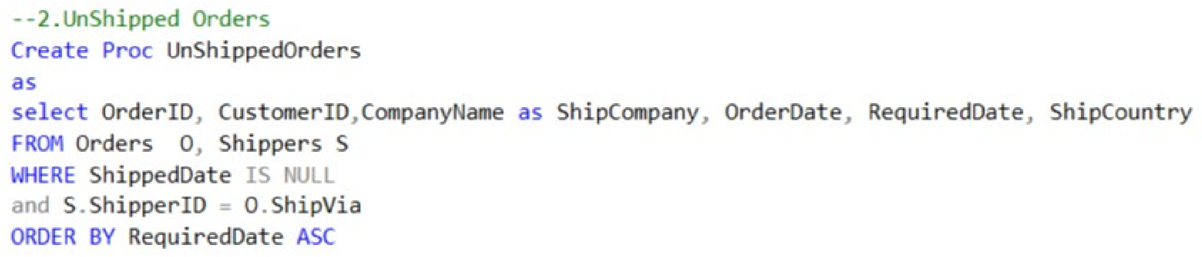
Archives orders older than a specified number of years to the **OrderArchive** table and deletes them from **Orders** and **[Order Details]**, using a transaction for consistency.



```sql
CREATE PROCEDURE sp_ArchiveOldOrders
    @YearsBack INT = 5
AS
BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        BEGIN TRANSACTION;
        INSERT INTO OrderArchive (OrderID, CustomerID, OrderDate, ShippedDate, Freight)
        SELECT OrderID, CustomerID, OrderDate, ShippedDate, Freight
        FROM Orders
        WHERE OrderDate < DATEADD(year, -@YearsBack, GETDATE());
        DELETE FROM [Order Details]
        WHERE OrderID IN (
            SELECT OrderID
            FROM Orders
            WHERE OrderDate < DATEADD(year, -@YearsBack, min(orderDate))
        );
        DELETE FROM Orders
        WHERE OrderDate < DATEADD(year, -@YearsBack, GETDATE());
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        THROW;
    END CATCH;
END;
GO
```

32. UnShippedOrders

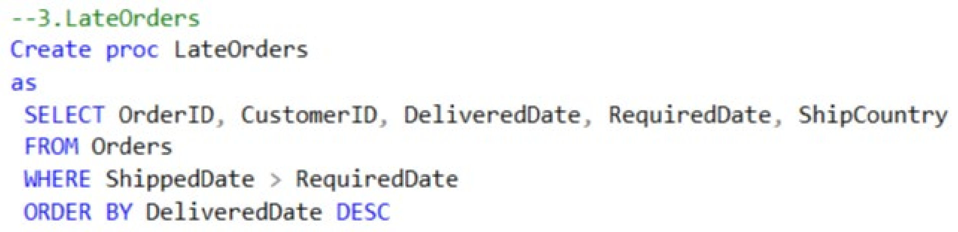
The UnShippedOrders retrieves orders that have **not been shipped** yet, using the ShippedDate Column



```sql
--2.UnShipped Orders
Create Proc UnShippedOrders
as
select OrderID, CustomerID,CompanyName as ShipCompany, OrderDate, RequiredDate, ShipCountry
FROM Orders  O, Shippers S
WHERE ShippedDate IS NULL
and S.ShipperID = O.ShipVia
ORDER BY RequiredDate ASC
```

33. LateOrders

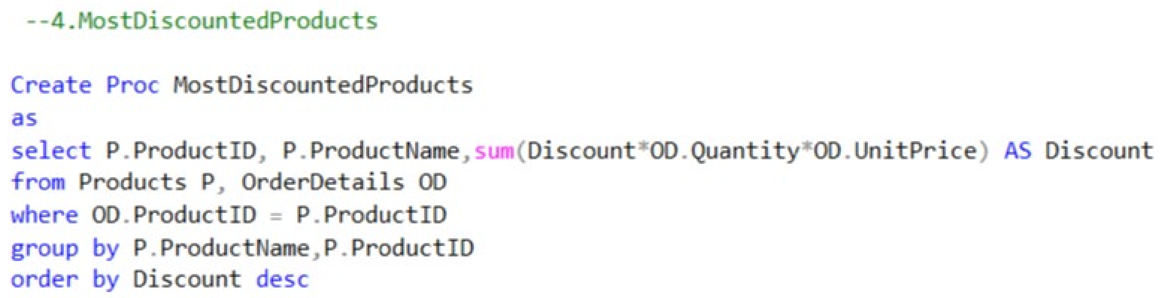
This procedure retrieves **orders that are overdue,** meaning they **should have been shipped by now but haven't been**.



```sql
--3.LateOrders
Create proc LateOrders
as
 SELECT OrderID, CustomerID, DeliveredDate, RequiredDate, ShipCountry
 FROM Orders
 WHERE ShippedDate > RequiredDate
 ORDER BY DeliveredDate DESC
```

34. MostDiscountedProducts

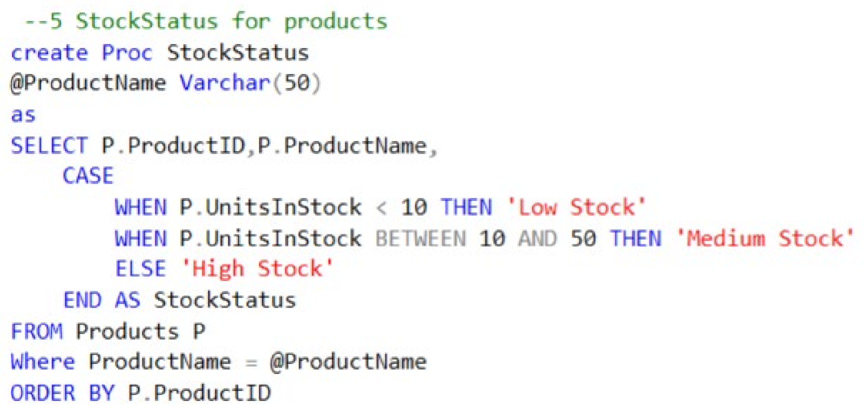
This stored procedure retrieves **the products with the highest total discount value** by calculating the total discount amount for each product.



```sql
 --4.MostDiscountedProducts

Create Proc MostDiscountedProducts
as
select P.ProductID, P.ProductName,sum(Discount*OD.Quantity*OD.UnitPrice) AS Discount
from Products P, OrderDetails OD
where OD.ProductID = P.ProductID
group by P.ProductName,P.ProductID
order by Discount desc
```

35. **Stock Status for products**

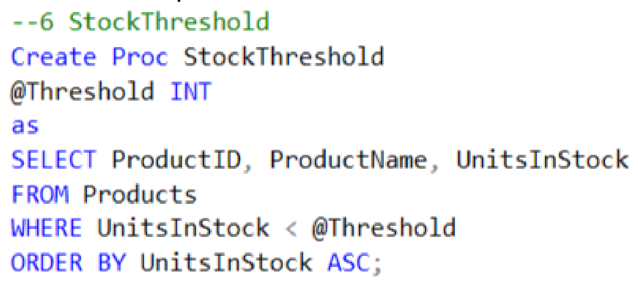
This stored procedure checks the stock status of a specific product and categorizes it as Low, Medium, or High Stock based on **UnitsInStock**



```
 --5 StockStatus for products
create Proc StockStatus
@ProductName Varchar(50)
as
SELECT P.ProductID,P.ProductName,
    CASE
        WHEN P.UnitsInStock < 10 THEN 'Low Stock'
        WHEN P.UnitsInStock BETWEEN 10 AND 50 THEN 'Medium Stock'
        ELSE 'High Stock'
    END AS StockStatus
FROM Products P
Where ProductName = @ProductName
ORDER BY P.ProductID
```

36. **StockThreshold**

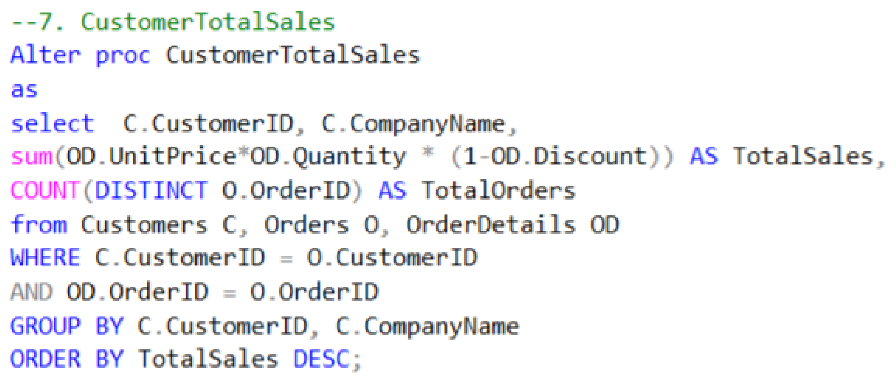
This procedure is likely designed to identify products that are running low on stock based on a predefined threshold. It helps in inventory management by notifying when stock levels are below a critical point



```
--6 StockThreshold
Create Proc StockThreshold
@Threshold INT
as
SELECT ProductID, ProductName, UnitsInStock
FROM Products
WHERE UnitsInStock < @Threshold
ORDER BY UnitsInStock ASC;
```

37. **CustomerTotalSales**

This stored procedure is likely designed to calculate the total sales per customer by summing the order values from the Orders and OrderDetails tables.

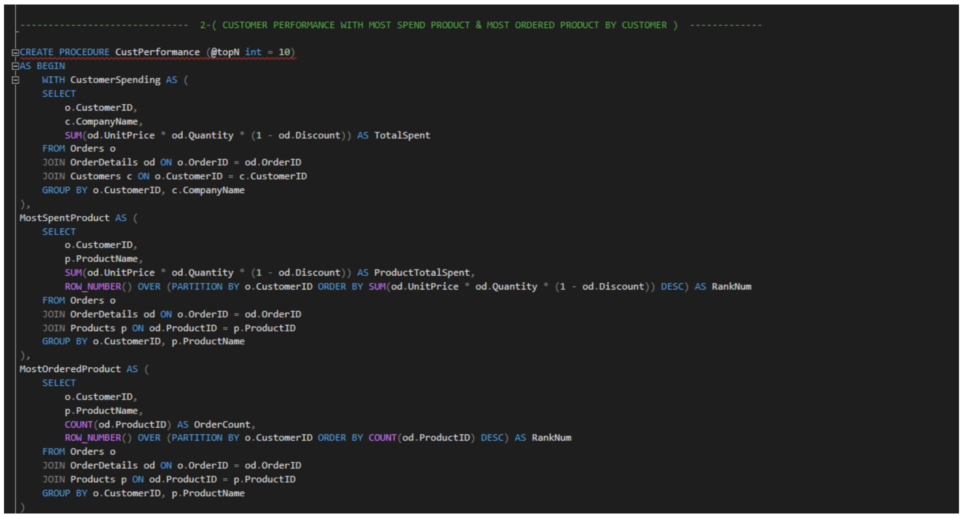


```
--7. CustomerTotalSales
Alter proc CustomerTotalSales
as
select  C.CustomerID, C.CompanyName,
sum(OD.UnitPrice*OD.Quantity * (1-OD.Discount)) AS TotalSales,
COUNT(DISTINCT O.OrderID) AS TotalOrders
from Customers C, Orders O, OrderDetails OD
WHERE C.CustomerID = O.CustomerID
AND OD.OrderID = O.OrderID
GROUP BY C.CustomerID, C.CompanyName
ORDER BY TotalSales DESC;
```

38. Customers Performance with most spend Product & Most Ordered Product

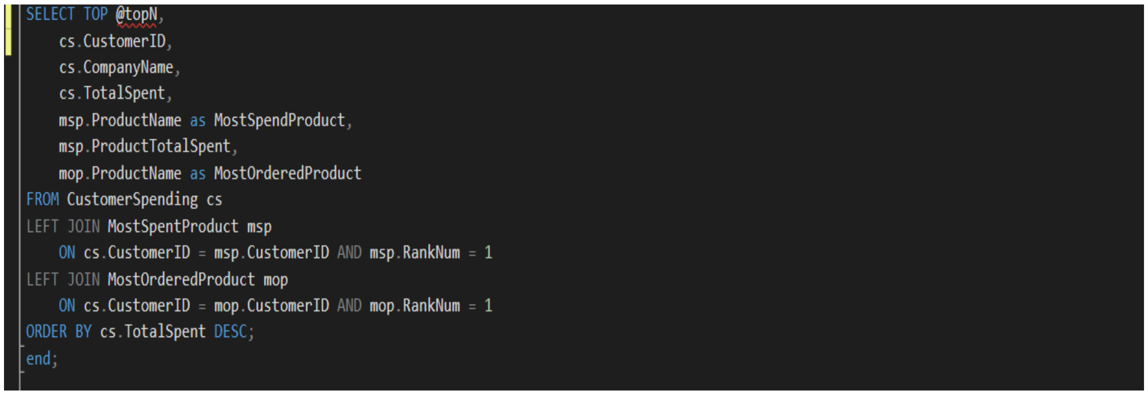
- **Breakdown of Functionality**

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



```sql
-----------------------------  2-( CUSTOMER PERFORMANCE WITH MOST SPEND PRODUCT & MOST ORDERED PRODUCT BY CUSTOMER )  ------------

CREATE PROCEDURE CustPerformance (@topN int = 10)
AS BEGIN
    WITH CustomerSpending AS (
    SELECT
        o.CustomerID,
        c.CompanyName,
        SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) AS TotalSpent
    FROM Orders o
    JOIN OrderDetails od ON o.OrderID = od.OrderID
    JOIN Customers c ON o.CustomerID = c.CustomerID
    GROUP BY o.CustomerID, c.CompanyName
),
MostSpentProduct AS (
    SELECT
        o.CustomerID,
        p.ProductName,
        SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) AS ProductTotalSpent,
        ROW_NUMBER() OVER (PARTITION BY o.CustomerID ORDER BY SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) DESC) AS RankNum
    FROM Orders o
    JOIN OrderDetails od ON o.OrderID = od.OrderID
    JOIN Products p ON od.ProductID = p.ProductID
    GROUP BY o.CustomerID, p.ProductName
),
MostOrderedProduct AS (
    SELECT
        o.CustomerID,
        p.ProductName,
        COUNT(od.ProductID) AS OrderCount,
        ROW_NUMBER() OVER (PARTITION BY o.CustomerID ORDER BY COUNT(od.ProductID) DESC) AS RankNum
    FROM Orders o
    JOIN OrderDetails od ON o.OrderID = od.OrderID
    JOIN Products p ON od.ProductID = p.ProductID
    GROUP BY o.CustomerID, p.ProductName
)
```

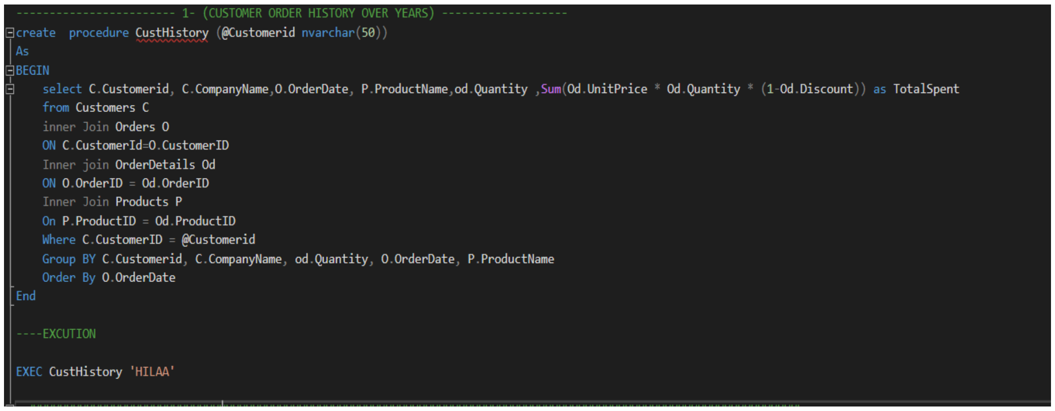
Final Query



```sql
SELECT TOP @topN,
    cs.CustomerID,
    cs.CompanyName,
    cs.TotalSpent,
    msp.ProductName as MostSpendProduct,
    msp.ProductTotalSpent,
    mop.ProductName as MostOrderedProduct
FROM CustomerSpending cs
LEFT JOIN MostSpentProduct msp
    ON cs.CustomerID = msp.CustomerID AND msp.RankNum = 1
LEFT JOIN MostOrderedProduct mop
    ON cs.CustomerID = mop.CustomerID AND mop.RankNum = 1
ORDER BY cs.TotalSpent DESC;
end;
```

39. Customers Order History Over Years

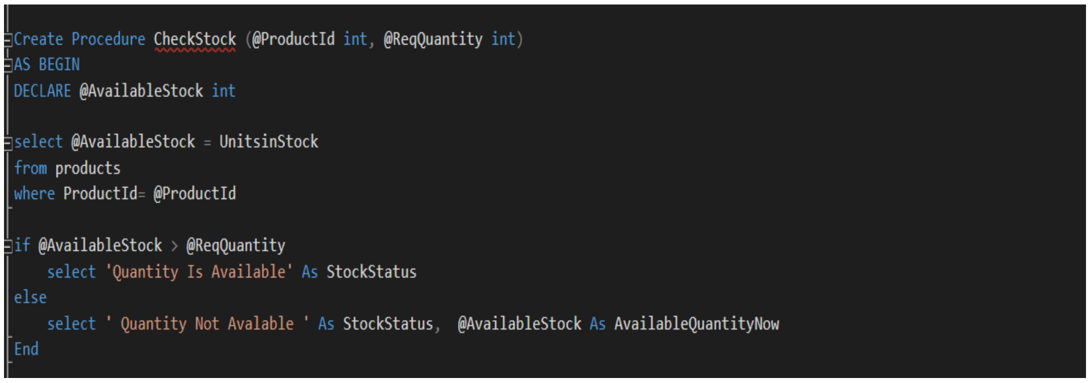
This SQL code creates a stored procedure named CustHistory, which retrieves the order history of a specific customer. It takes @Customerid as a parameter and returns details. This procedure helps analyze a customer's purchasing history.



```sql
----------------------- 1- (CUSTOMER ORDER HISTORY OVER YEARS) -------------------
create  procedure CustHistory (@Customerid nvarchar(50))
As
BEGIN
    select C.Customerid, C.CompanyName,O.OrderDate, P.ProductName,od.Quantity ,Sum(Od.UnitPrice * Od.Quantity * (1-Od.Discount)) as TotalSpent
    from Customers C
    inner Join Orders O
    ON C.CustomerId=O.CustomerID
    Inner join OrderDetails Od
    ON O.OrderID = Od.OrderID
    Inner Join Products P
    On P.ProductID = Od.ProductID
    Where C.CustomerID = @Customerid
    Group BY C.Customerid, C.CompanyName, od.Quantity, O.OrderDate, P.ProductName
    Order By O.OrderDate
End

----EXCUTION

EXEC CustHistory 'HILAA'
```

40. Checking Stock Before Ordering :

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.



```sql
Create Procedure CheckStock (@ProductId int, @ReqQuantity int)
AS BEGIN
DECLARE @AvailableStock int

select @AvailableStock = UnitsinStock
from products
where ProductId= @ProductId

if @AvailableStock > @ReqQuantity
    select 'Quantity Is Available' As StockStatus
else
    select ' Quantity Not Avalable ' As StockStatus,  @AvailableStock As AvailableQuantityNow
End
```

41. Automating Reorder Low Stock Products :

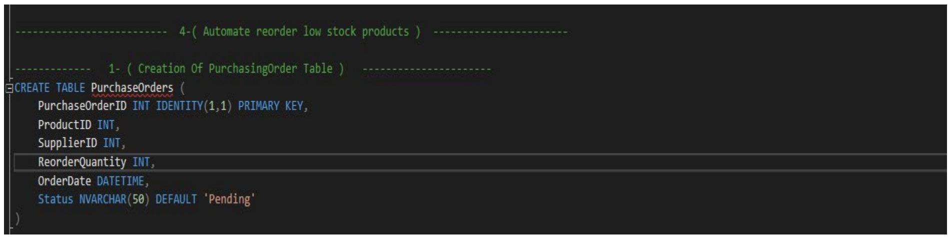
This SQL code creates a **table** and a **stored procedure** to manage product reorders automatically when stock levels are low.

Breakdown of Functionality:

1. PurchaseOrders Table

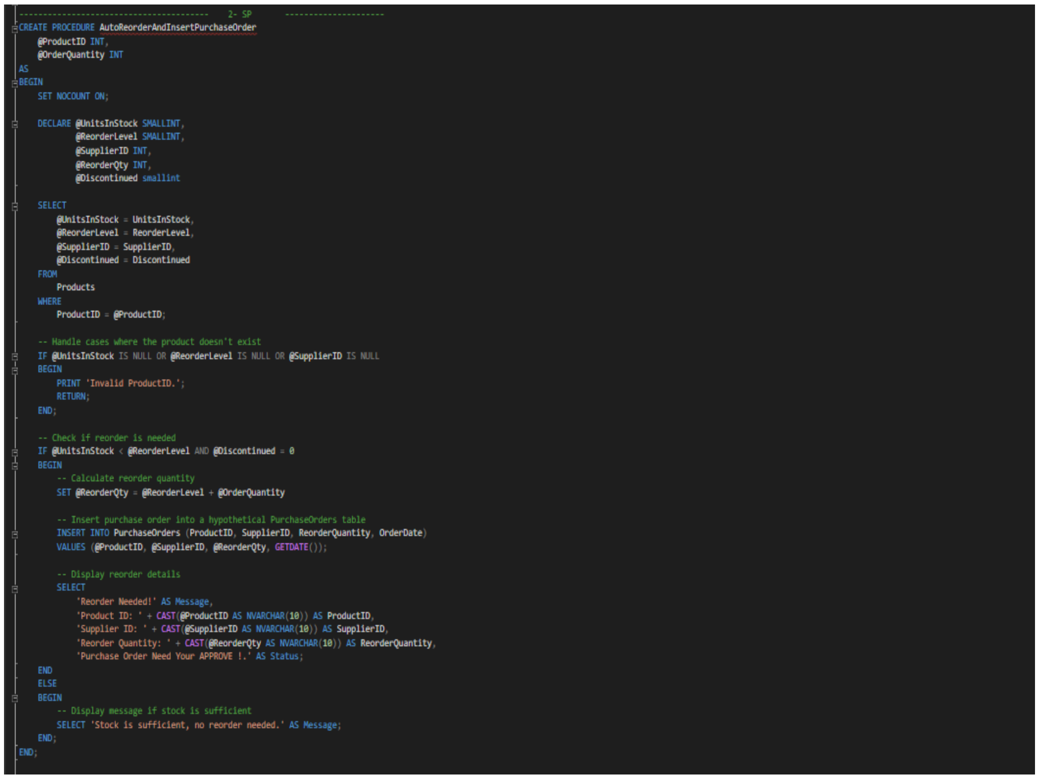
2. AutoReorderAndInsertPurchaseOrder Stored Procedure

- Creation of Purchasing Table



```sql
-------------------------  4-( Automate reorder low stock products )  ----------------------

------------   1- ( Creation Of PurchasingOrder Table )   ---------------------
CREATE TABLE PurchaseOrders (
    PurchaseOrderID INT IDENTITY(1,1) PRIMARY KEY,
    ProductID INT,
    SupplierID INT,
    ReorderQuantity INT,
    OrderDate DATETIME,
    Status NVARCHAR(50) DEFAULT 'Pending'
)
```

- Stored Procedure



```sql
----------------------------------------    2- SP      --------------------
CREATE PROCEDURE AutoReorderAndInsertPurchaseOrder
    @ProductID INT,
    @OrderQuantity INT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @UnitsInStock SMALLINT,
            @ReorderLevel SMALLINT,
            @SupplierID INT,
            @ReorderQty INT,
            @Discontinued smallint

    SELECT
        @UnitsInStock = UnitsInStock,
        @ReorderLevel = ReorderLevel,
        @SupplierID = SupplierID,
        @Discontinued = Discontinued
    FROM
        Products
    WHERE
        ProductID = @ProductID;

    -- Handle cases where the product doesn't exist
    IF @UnitsInStock IS NULL OR @ReorderLevel IS NULL OR @SupplierID IS NULL
    BEGIN
        PRINT 'Invalid ProductID.';
        RETURN;
    END;

    -- Check if reorder is needed
    IF @UnitsInStock < @ReorderLevel AND @Discontinued = 0
    BEGIN
        -- Calculate reorder quantity
        SET @ReorderQty = @ReorderLevel + @OrderQuantity

        -- Insert purchase order into a hypothetical PurchaseOrders table
        INSERT INTO PurchaseOrders (ProductID, SupplierID, ReorderQuantity, OrderDate)
        VALUES (@ProductID, @SupplierID, @ReorderQty, GETDATE());

        -- Display reorder details
        SELECT
            'Reorder Needed!' AS Message,
            'Product ID: ' + CAST(@ProductID AS NVARCHAR(10)) AS ProductID,
            'Supplier ID: ' + CAST(@SupplierID AS NVARCHAR(10)) AS SupplierID,
            'Reorder Quantity: ' + CAST(@ReorderQty AS NVARCHAR(10)) AS ReorderQuantity,
            'Purchase Order Need Your APPROVE !.' AS Status;
    END
    ELSE
    BEGIN
        -- Display message if stock is sufficient
        SELECT 'Stock is sufficient, no reorder needed.' AS Message;
    END;
END;
```

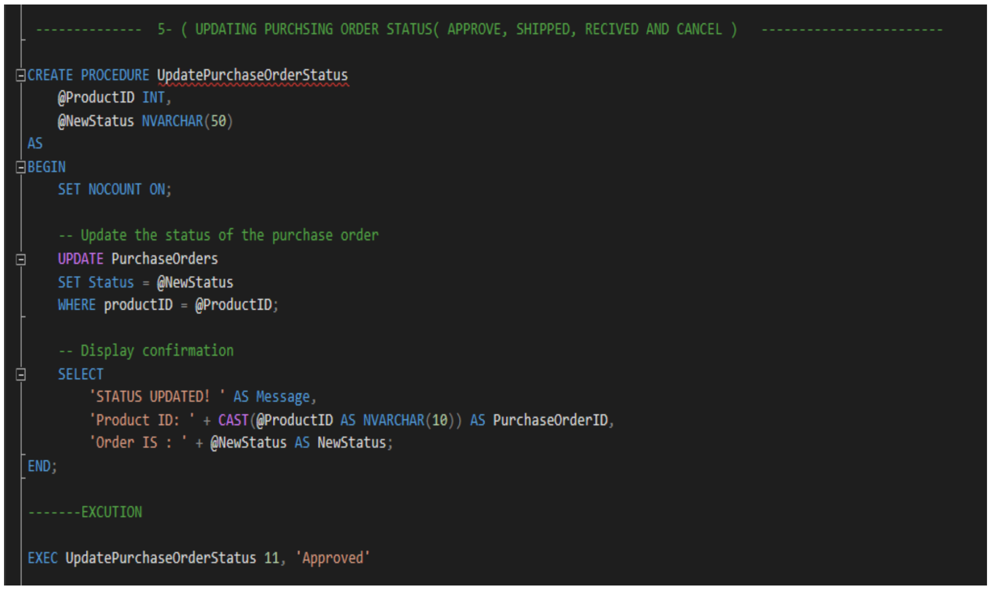
42. Automating Reorder Low Stock Products :

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.

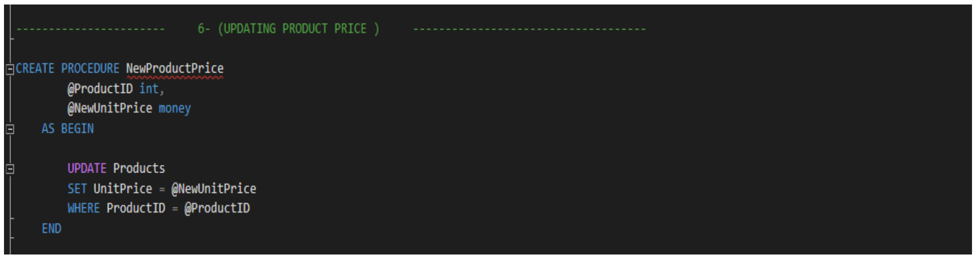
This procedure helps **manage and track purchase order statuses** efficiently.



```sql
 -------------  5- ( UPDATING PURCHSING ORDER STATUS( APPROVE, SHIPPED, RECIVED AND CANCEL )   -----------------------

CREATE PROCEDURE UpdatePurchaseOrderStatus
    @ProductID INT,
    @NewStatus NVARCHAR(50)
AS
BEGIN
    SET NOCOUNT ON;

    -- Update the status of the purchase order
    UPDATE PurchaseOrders
    SET Status = @NewStatus
    WHERE productID = @ProductID;

    -- Display confirmation
    SELECT
        'STATUS UPDATED! ' AS Message,
        'Product ID: ' + CAST(@ProductID AS NVARCHAR(10)) AS PurchaseOrderID,
        'Order IS : ' + @NewStatus AS NewStatus;
END;

-------EXCUTION

EXEC UpdatePurchaseOrderStatus 11, 'Approved'
```

43. Update Product Price :

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.



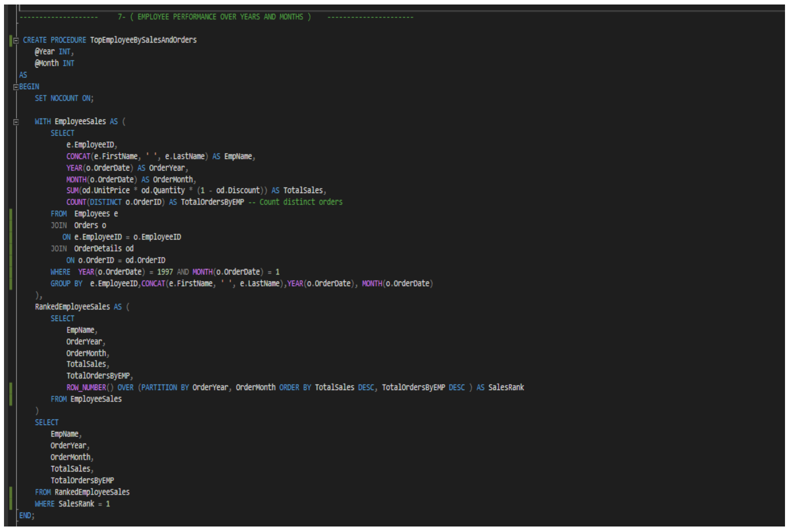
```sql
----------------------     6- (UPDATING PRODUCT PRICE )     -----------------------------------

CREATE PROCEDURE NewProductPrice
        @ProductID int,
        @NewUnitPrice money
    AS BEGIN

        UPDATE Products
        SET UnitPrice = @NewUnitPrice
        WHERE ProductID = @ProductID
    END
```

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

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:

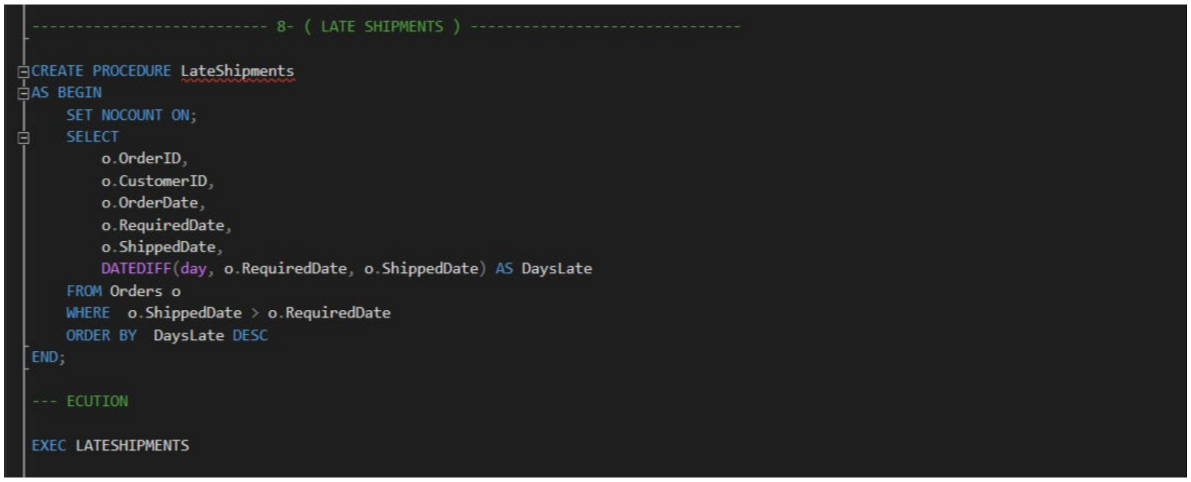
- Takes @Year and @Month as input parameters.
- The EmployeeSales CTE calculates total sales and distinct orders for each employee within the specified month and year.
- The RankedEmployeeSales CTE assigns a rank to employees based on their total sales and order count.
- The final query selects the top-ranked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



```sql
--------------------     7- ( EMPLOYEE PERFORMANCE OVER YEARS AND MONTHS )     ---------------------

CREATE PROCEDURE TopEmployeeBySalesAndOrders
    @Year INT,
    @Month INT
AS
BEGIN
    SET NOCOUNT ON;

    WITH EmployeeSales AS (
        SELECT
            e.EmployeeID,
            CONCAT(e.FirstName, ' ', e.LastName) AS EmpName,
            YEAR(o.OrderDate) AS OrderYear,
            MONTH(o.OrderDate) AS OrderMonth,
            SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) AS TotalSales,
            COUNT(DISTINCT o.OrderID) AS TotalOrdersByEMP -- Count distinct orders
        FROM  Employees e
        JOIN  Orders o
          ON e.EmployeeID = o.EmployeeID
        JOIN  OrderDetails od
            ON o.OrderID = od.OrderID
        WHERE  YEAR(o.OrderDate) = 1997 AND MONTH(o.OrderDate) = 1
        GROUP BY  e.EmployeeID,CONCAT(e.FirstName, ' ', e.LastName),YEAR(o.OrderDate), MONTH(o.OrderDate)
    ),
    RankedEmployeeSales AS (
        SELECT
            EmpName,
            OrderYear,
            OrderMonth,
            TotalSales,
            TotalOrdersByEMP,
            ROW_NUMBER() OVER (PARTITION BY OrderYear, OrderMonth ORDER BY TotalSales DESC, TotalOrdersByEMP DESC ) AS SalesRank
        FROM EmployeeSales
    )
    SELECT
        EmpName,
        OrderYear,
        OrderMonth,
        TotalSales,
        TotalOrdersByEMP
    FROM RankedEmployeeSales
    WHERE SalesRank = 1
END;
```

45. Late Shipment

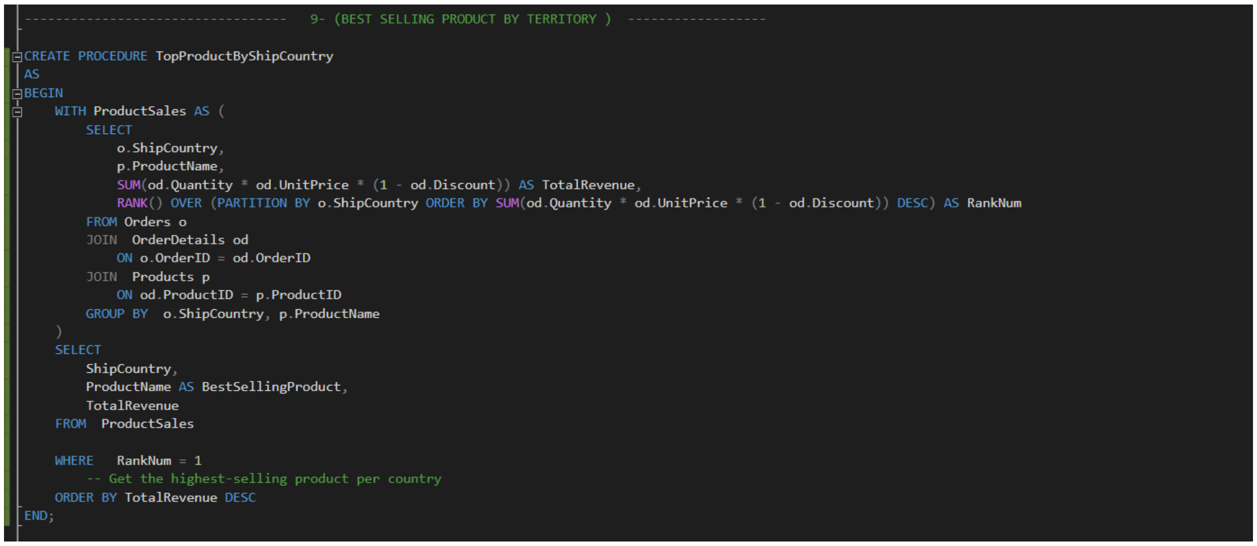
This SQL code creates a stored procedure named LateShipments, which retrieves orders that were shipped later than their required delivery date. It selects order details, calculates the delay in days, filters for late shipments, and sorts them in descending order of delay duration. This procedure helps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



```sql
-------------------------- 8- ( LATE SHIPMENTS ) ------------------------------

CREATE PROCEDURE LateShipments
AS BEGIN
    SET NOCOUNT ON;
    SELECT
        o.OrderID,
        o.CustomerID,
        o.OrderDate,
        o.RequiredDate,
        o.ShippedDate,
        DATEDIFF(day, o.RequiredDate, o.ShippedDate) AS DaysLate
    FROM Orders o
    WHERE  o.ShippedDate > o.RequiredDate
    ORDER BY  DaysLate DESC
END;

--- ECUTION

EXEC LATESHIPMENTS
```

46. Best Selling Products By Country

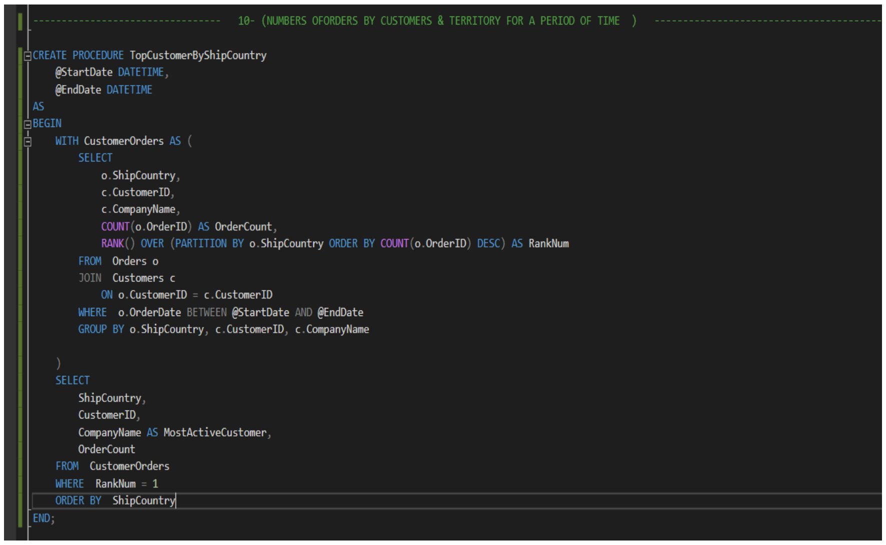
This SQL code creates a stored procedure named TopProductByShipCountry, which retrieves the best-selling product for each shipping country based on total revenue. It calculates total sales for each product per country, ranks them using the RANK() function, and selects the top-ranked product in each country.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



```sql
---------------------------------   9- (BEST SELLING PRODUCT BY TERRITORY )  -----------------

CREATE PROCEDURE TopProductByShipCountry
AS
BEGIN
    WITH ProductSales AS (
        SELECT
            o.ShipCountry,
            p.ProductName,
            SUM(od.Quantity * od.UnitPrice * (1 - od.Discount)) AS TotalRevenue,
            RANK() OVER (PARTITION BY o.ShipCountry ORDER BY SUM(od.Quantity * od.UnitPrice * (1 - od.Discount)) DESC) AS RankNum
        FROM Orders o
        JOIN  OrderDetails od
            ON o.OrderID = od.OrderID
        JOIN  Products p
            ON od.ProductID = p.ProductID
        GROUP BY  o.ShipCountry, p.ProductName
    )
    SELECT
        ShipCountry,
        ProductName AS BestSellingProduct,
        TotalRevenue
    FROM  ProductSales

    WHERE   RankNum = 1
        -- Get the highest-selling product per country
    ORDER BY TotalRevenue DESC
END;
```

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

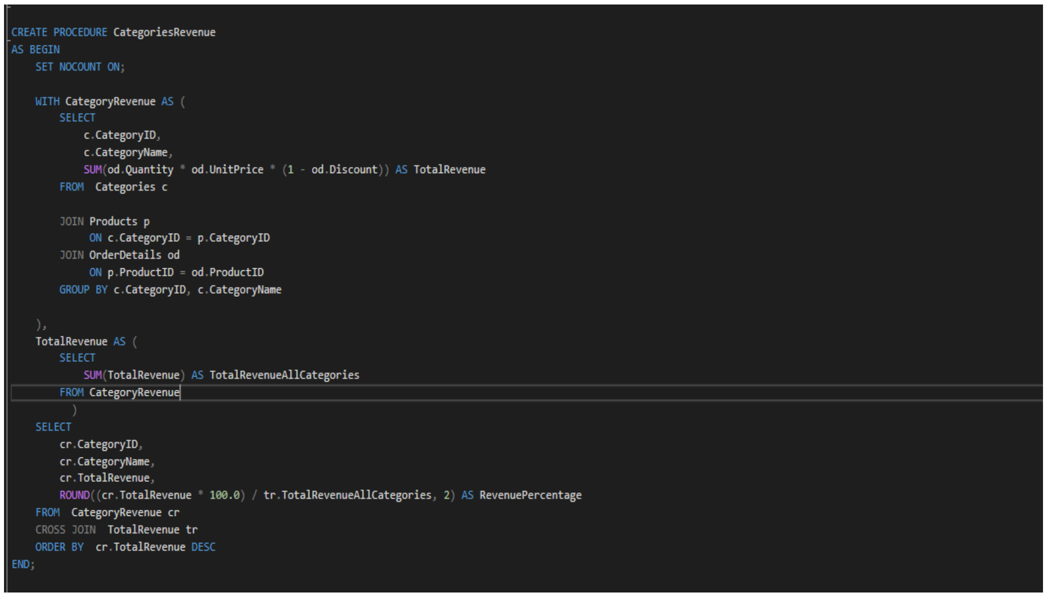
This SQL code creates a stored procedure named TopCustomerByShipCountry, which retrieves the most active customer in each shipping country within a specified date range.



```sql
--------------------------------   10- (NUMBERS OFORDERS BY CUSTOMERS & TERRITORY FOR A PERIOD OF TIME  )   ----------------------------------------

CREATE PROCEDURE TopCustomerByShipCountry
    @StartDate DATETIME,
    @EndDate DATETIME
AS
BEGIN
    WITH CustomerOrders AS (
        SELECT
            o.ShipCountry,
            c.CustomerID,
            c.CompanyName,
            COUNT(o.OrderID) AS OrderCount,
            RANK() OVER (PARTITION BY o.ShipCountry ORDER BY COUNT(o.OrderID) DESC) AS RankNum
        FROM  Orders o
        JOIN  Customers c
            ON o.CustomerID = c.CustomerID
        WHERE  o.OrderDate BETWEEN @StartDate AND @EndDate
        GROUP BY o.ShipCountry, c.CustomerID, c.CompanyName

    )
    SELECT
        ShipCountry,
        CustomerID,
        CompanyName AS MostActiveCustomer,
        OrderCount
    FROM  CustomerOrders
    WHERE  RankNum = 1
    ORDER BY  ShipCountry
END;
```

48. Revenue By Category

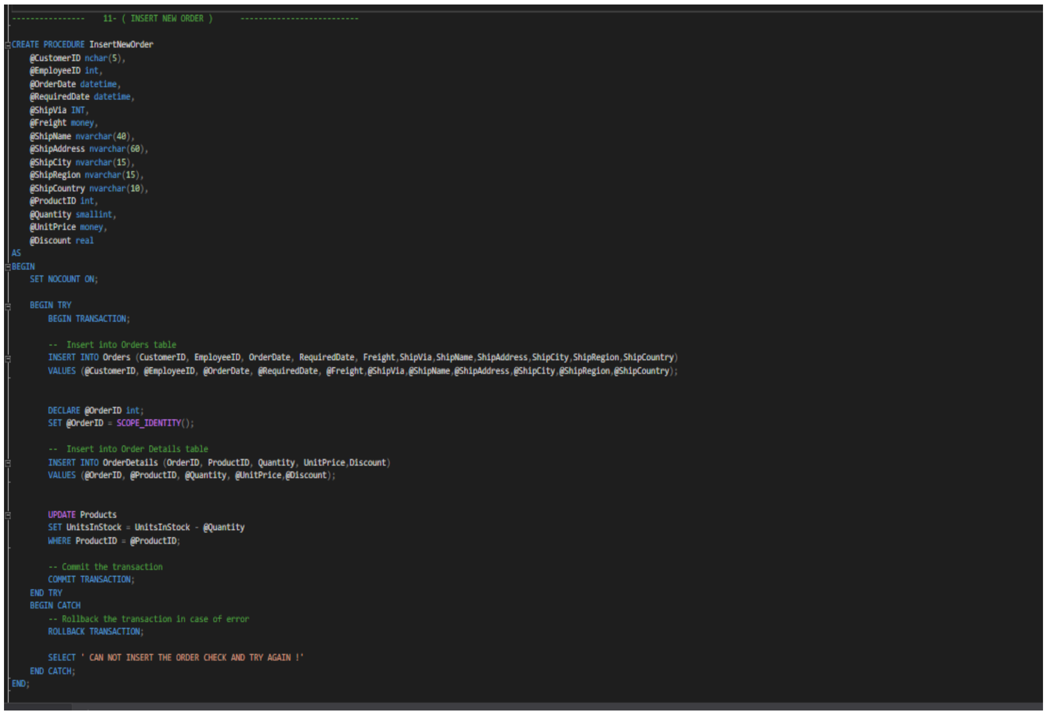
This SQL code creates a stored procedure named CategoriesRevenue, which calculates the total revenue generated by each product category. It retrieves category details and computes total sales revenue by summing product sales, factoring in quantity, unit price, and discounts. The procedure also calculates 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.



```sql
CREATE PROCEDURE CategoriesRevenue
AS BEGIN
    SET NOCOUNT ON;

    WITH CategoryRevenue AS (
        SELECT
            c.CategoryID,
            c.CategoryName,
            SUM(od.Quantity * od.UnitPrice * (1 - od.Discount)) AS TotalRevenue
        FROM  Categories c

        JOIN Products p
             ON c.CategoryID = p.CategoryID
        JOIN OrderDetails od
             ON p.ProductID = od.ProductID
        GROUP BY c.CategoryID, c.CategoryName

    ),
    TotalRevenue AS (
        SELECT
            SUM(TotalRevenue) AS TotalRevenueAllCategories
        FROM CategoryRevenue
          )
    SELECT
        cr.CategoryID,
        cr.CategoryName,
        cr.TotalRevenue,
        ROUND((cr.TotalRevenue * 100.0) / tr.TotalRevenueAllCategories, 2) AS RevenuePercentage
    FROM  CategoryRevenue cr
    CROSS JOIN  TotalRevenue tr
    ORDER BY  cr.TotalRevenue DESC
END;
```

49. Inserting New Order:

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.



```sql
---------------    11- ( INSERT NEW ORDER )      --------------------------

CREATE PROCEDURE InsertNewOrder
    @CustomerID nchar(5),
    @EmployeeID int,
    @OrderDate datetime,
    @RequiredDate datetime,
    @ShipVia INT,
    @Freight money,
    @ShipName nvarchar(40),
    @ShipAddress nvarchar(60),
    @ShipCity nvarchar(15),
    @ShipRegion nvarchar(15),
    @ShipCountry nvarchar(10),
    @ProductID int,
    @Quantity smallint,
    @UnitPrice money,
    @Discount real
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        BEGIN TRANSACTION;

        --  Insert into Orders table
        INSERT INTO Orders (CustomerID, EmployeeID, OrderDate, RequiredDate, Freight,ShipVia,ShipName,ShipAddress,ShipCity,ShipRegion,ShipCountry)
        VALUES (@CustomerID, @EmployeeID, @OrderDate, @RequiredDate, @Freight,@ShipVia,@ShipName,@ShipAddress,@ShipCity,@ShipRegion,@ShipCountry);


        DECLARE @OrderID int;
        SET @OrderID = SCOPE_IDENTITY();

        --  Insert into Order Details table
        INSERT INTO OrderDetails (OrderID, ProductID, Quantity, UnitPrice,Discount)
        VALUES (@OrderID, @ProductID, @Quantity, @UnitPrice,@Discount);


        UPDATE Products
        SET UnitsInStock = UnitsInStock - @Quantity
        WHERE ProductID = @ProductID;

        -- Commit the transaction
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        -- Rollback the transaction in case of error
        ROLLBACK TRANSACTION;

        SELECT ' CAN NOT INSERT THE ORDER CHECK AND TRY AGAIN !'
    END CATCH;
END;
```

2. Views

A view in SQL is a **virtual table** that is based on the result of a SELECT query. It does not store data physically but provides a way to simplify complex queries, enhance security, and improve data management.

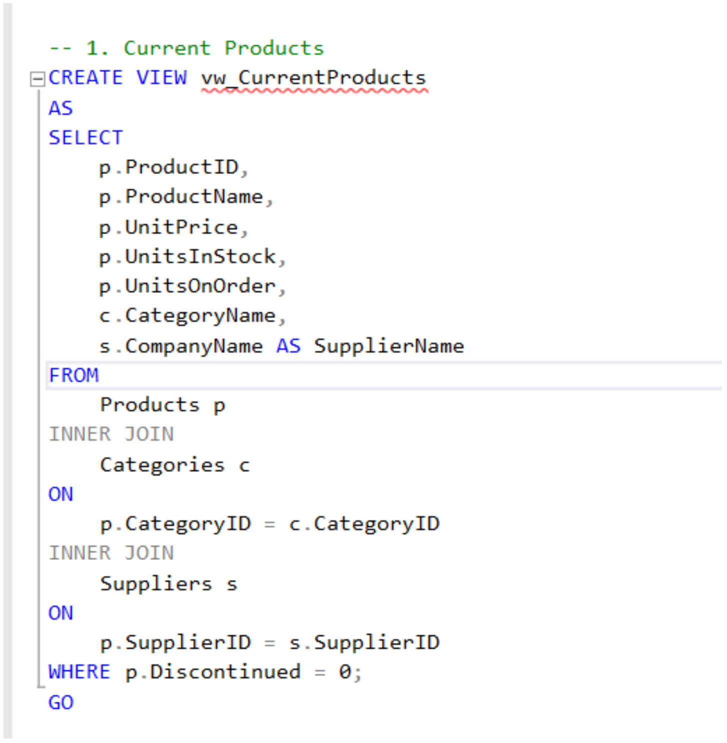
Functions & Uses of Views

- **Data Abstraction** – Hides complex queries and table joins behind a simple interface.
- **Security & Access Control** – Restricts access to specific columns or rows by controlling what data users can see.
- **Simplifies Queries** – Stores reusable, frequently used queries, making reporting easier.
- **Improves Maintainability** – Reduces the need to rewrite complex queries in applications.
- **Aggregated or Filtered Data** – Shows only relevant records

28 Views have been created:

1. CurrentProducts

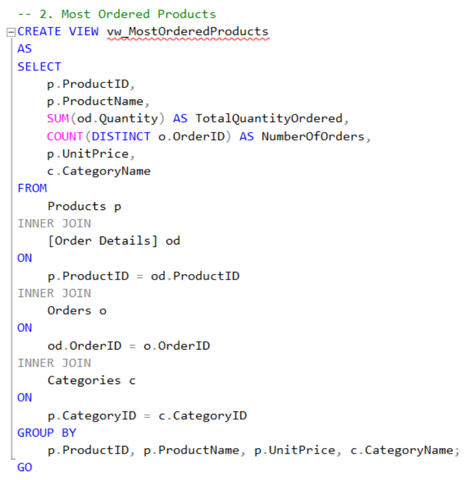
Displays **active products** (not discontinued) with details including category and supplier names for inventory management.



```
-- 1. Current Products
CREATE VIEW vw_CurrentProducts
AS
SELECT
    p.ProductID,
    p.ProductName,
    p.UnitPrice,
    p.UnitsInStock,
    p.UnitsOnOrder,
    c.CategoryName,
    s.CompanyName AS SupplierName
FROM
    Products p
INNER JOIN
    Categories c
ON
    p.CategoryID = c.CategoryID
INNER JOIN
    Suppliers s
ON
    p.SupplierID = s.SupplierID
WHERE p.Discontinued = 0;
GO
```

2. MostOrderedProducts

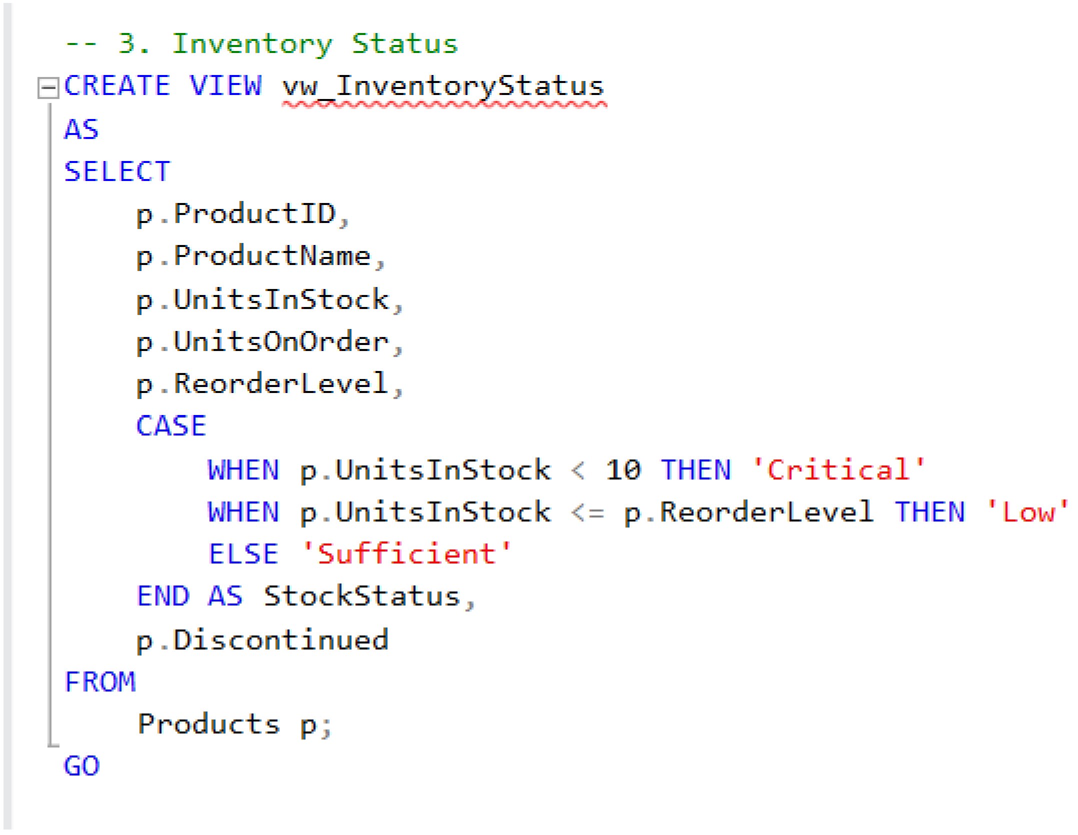
Summarizes the **most ordered products** with total quantities, order counts, and category details for sales analysis.



```sql
-- 2. Most Ordered Products
CREATE VIEW vw_MostOrderedProducts
AS
SELECT
    p.ProductID,
    p.ProductName,
    SUM(od.Quantity) AS TotalQuantityOrdered,
    COUNT(DISTINCT o.OrderID) AS NumberOfOrders,
    p.UnitPrice,
    c.CategoryName
FROM
    Products p
INNER JOIN
    [Order Details] od
ON
    p.ProductID = od.ProductID
INNER JOIN
    Orders o
ON
    od.OrderID = o.OrderID
INNER JOIN
    Categories c
ON
    p.CategoryID = c.CategoryID
GROUP BY
    p.ProductID, p.ProductName, p.UnitPrice, c.CategoryName;
GO
```

3. InventoryStatus

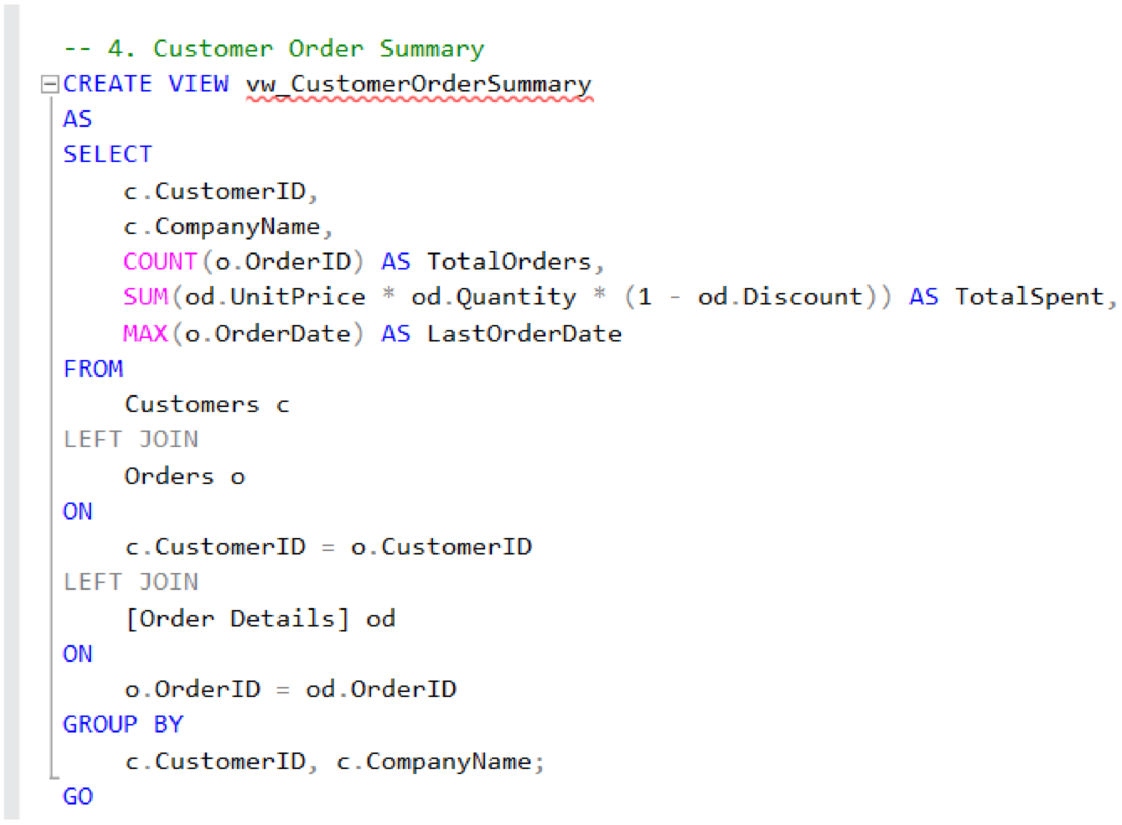
Provides an overview of product inventory with stock status (Critical, Low, Sufficient) based on **UnitsInStock** and **ReorderLevel.**



```sql
-- 3. Inventory Status
CREATE VIEW vw_InventoryStatus
AS
SELECT
    p.ProductID,
    p.ProductName,
    p.UnitsInStock,
    p.UnitsOnOrder,
    p.ReorderLevel,
    CASE
        WHEN p.UnitsInStock < 10 THEN 'Critical'
        WHEN p.UnitsInStock <= p.ReorderLevel THEN 'Low'
        ELSE 'Sufficient'
    END AS StockStatus,
    p.Discontinued
FROM
    Products p;
GO
```

4. CustomerOrderSummary

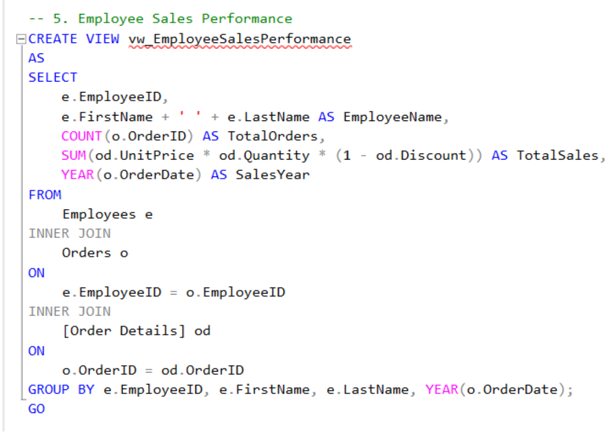
Aggregates customer order data, including total orders, spending, and last order date for customer performance tracking.



```sql
-- 4. Customer Order Summary
CREATE VIEW vw_CustomerOrderSummary
AS
SELECT
    c.CustomerID,
    c.CompanyName,
    COUNT(o.OrderID) AS TotalOrders,
    SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) AS TotalSpent,
    MAX(o.OrderDate) AS LastOrderDate
FROM
    Customers c
LEFT JOIN
    Orders o
ON
    c.CustomerID = o.CustomerID
LEFT JOIN
    [Order Details] od
ON
    o.OrderID = od.OrderID
GROUP BY
    c.CustomerID, c.CompanyName;
GO
```

5. EmployeeSalesPerformance

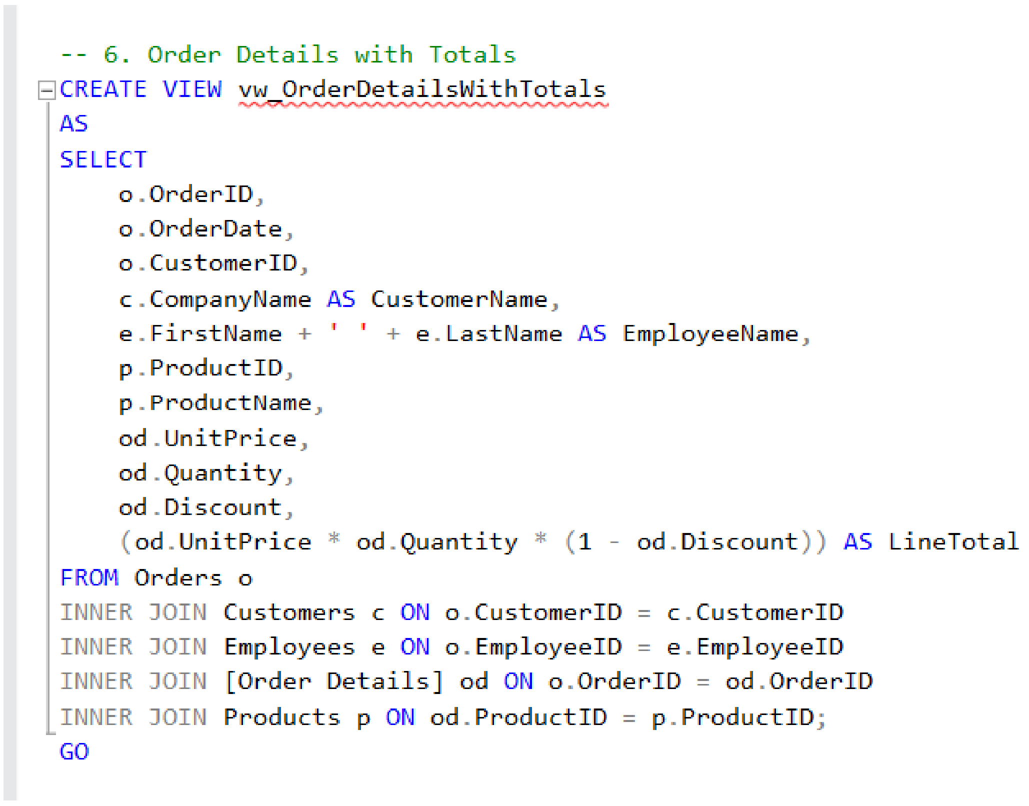
Summarizes employee sales by year, including total orders and sales amounts for performance evaluation.



```sql
-- 5. Employee Sales Performance
CREATE VIEW vw_EmployeeSalesPerformance
AS
SELECT
    e.EmployeeID,
    e.FirstName + ' ' + e.LastName AS EmployeeName,
    COUNT(o.OrderID) AS TotalOrders,
    SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) AS TotalSales,
    YEAR(o.OrderDate) AS SalesYear
FROM
    Employees e
INNER JOIN
    Orders o
ON
    e.EmployeeID = o.EmployeeID
INNER JOIN
    [Order Details] od
ON
    o.OrderID = od.OrderID
GROUP BY e.EmployeeID, e.FirstName, e.LastName, YEAR(o.OrderDate);
GO
```

6. OrderDetailsWithTotals

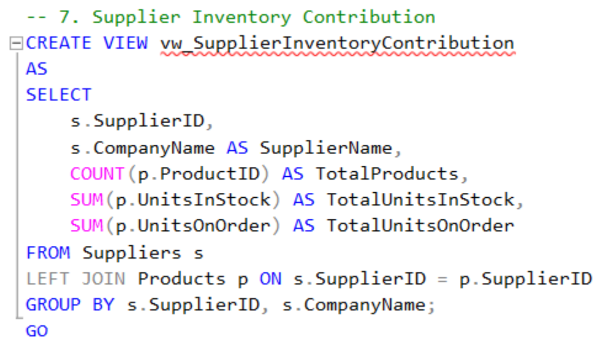
Combines **order details with customer and employee information,** including line totals for detailed order analysis.



```sql
-- 6. Order Details with Totals
CREATE VIEW vw_OrderDetailsWithTotals
AS
SELECT
    o.OrderID,
    o.OrderDate,
    o.CustomerID,
    c.CompanyName AS CustomerName,
    e.FirstName + ' ' + e.LastName AS EmployeeName,
    p.ProductID,
    p.ProductName,
    od.UnitPrice,
    od.Quantity,
    od.Discount,
    (od.UnitPrice * od.Quantity * (1 - od.Discount)) AS LineTotal
FROM Orders o
INNER JOIN Customers c ON o.CustomerID = c.CustomerID
INNER JOIN Employees e ON o.EmployeeID = e.EmployeeID
INNER JOIN [Order Details] od ON o.OrderID = od.OrderID
INNER JOIN Products p ON od.ProductID = p.ProductID;
GO
```

7. SupplierInventoryContribution

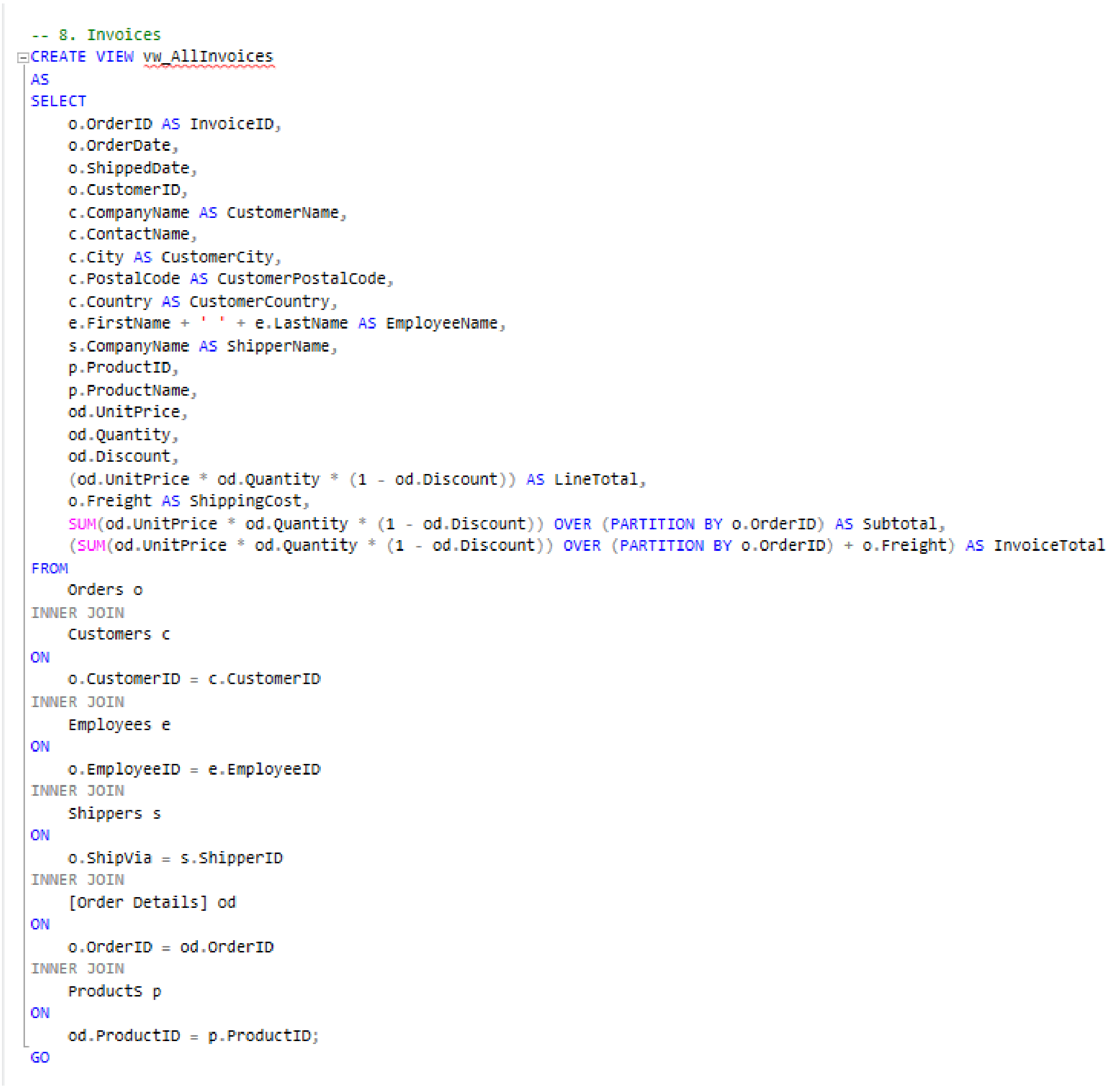
Shows **supplier contributions** to inventory with product counts and stock levels for supplier performance insights.



```sql
-- 7. Supplier Inventory Contribution
CREATE VIEW vw_SupplierInventoryContribution
AS
SELECT
    s.SupplierID,
    s.CompanyName AS SupplierName,
    COUNT(p.ProductID) AS TotalProducts,
    SUM(p.UnitsInStock) AS TotalUnitsInStock,
    SUM(p.UnitsOnOrder) AS TotalUnitsOnOrder
FROM Suppliers s
LEFT JOIN Products p ON s.SupplierID = p.SupplierID
GROUP BY s.SupplierID, s.CompanyName;
GO
```

8. AllInvoices

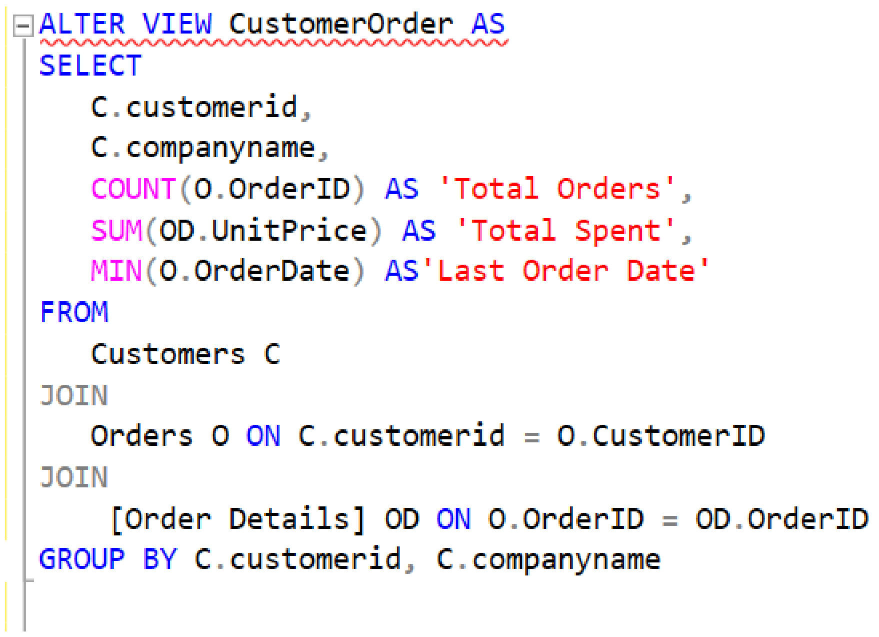
Provides a comprehensive invoice view with order, customer, employee, shipper, and product details, including subtotals and totals.



```
-- 8. Invoices
CREATE VIEW vw_AllInvoices
AS
SELECT
    o.OrderID AS InvoiceID,
    o.OrderDate,
    o.ShippedDate,
    o.CustomerID,
    c.CompanyName AS CustomerName,
    c.ContactName,
    c.City AS CustomerCity,
    c.PostalCode AS CustomerPostalCode,
    c.Country AS CustomerCountry,
    e.FirstName + ' ' + e.LastName AS EmployeeName,
    s.CompanyName AS ShipperName,
    p.ProductID,
    p.ProductName,
    od.UnitPrice,
    od.Quantity,
    od.Discount,
    (od.UnitPrice * od.Quantity * (1 - od.Discount)) AS LineTotal,
    o.Freight AS ShippingCost,
    SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) OVER (PARTITION BY o.OrderID) AS Subtotal,
    (SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) OVER (PARTITION BY o.OrderID) + o.Freight) AS InvoiceTotal
FROM
    Orders o
INNER JOIN
    Customers c
ON
    o.CustomerID = c.CustomerID
INNER JOIN
    Employees e
ON
    o.EmployeeID = e.EmployeeID
INNER JOIN
    Shippers s
ON
    o.ShipVia = s.ShipperID
INNER JOIN
    [Order Details] od
ON
    o.OrderID = od.OrderID
INNER JOIN
    Products p
ON
    od.ProductID = p.ProductID;
GO
```

9. CustomerOrder:

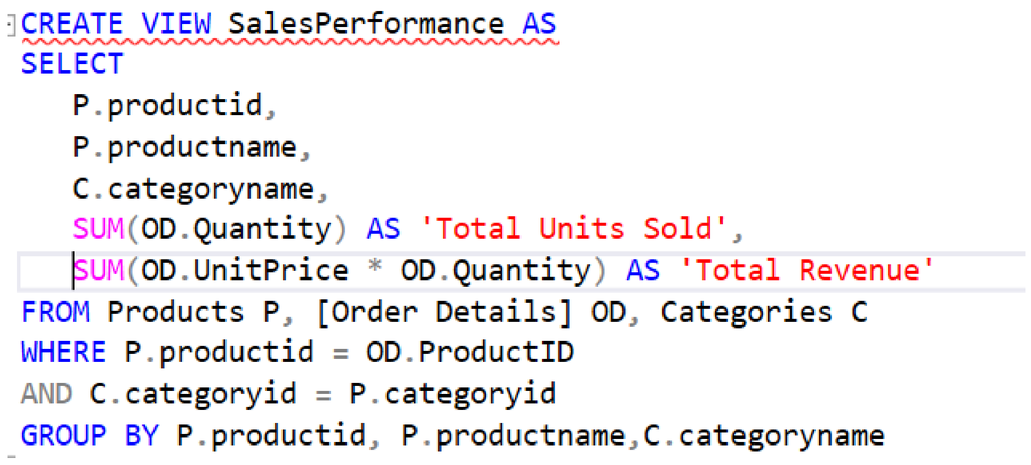
The CustomerOrder view displays **customer details**, **total orders placed**, **total amount spent**, and the **last order date** for each customer. It helps analyze customer purchasing behavior and loyalty.



```
ALTER VIEW CustomerOrder AS
SELECT
   C.customerid,
   C.companyname,
   COUNT(O.OrderID) AS 'Total Orders',
   SUM(OD.UnitPrice) AS 'Total Spent',
   MIN(O.OrderDate) AS'Last Order Date'
FROM
   Customers C
JOIN
   Orders O ON C.customerid = O.CustomerID
JOIN
    [Order Details] OD ON O.OrderID = OD.OrderID
GROUP BY C.customerid, C.companyname
```

10. SalesPerformance:

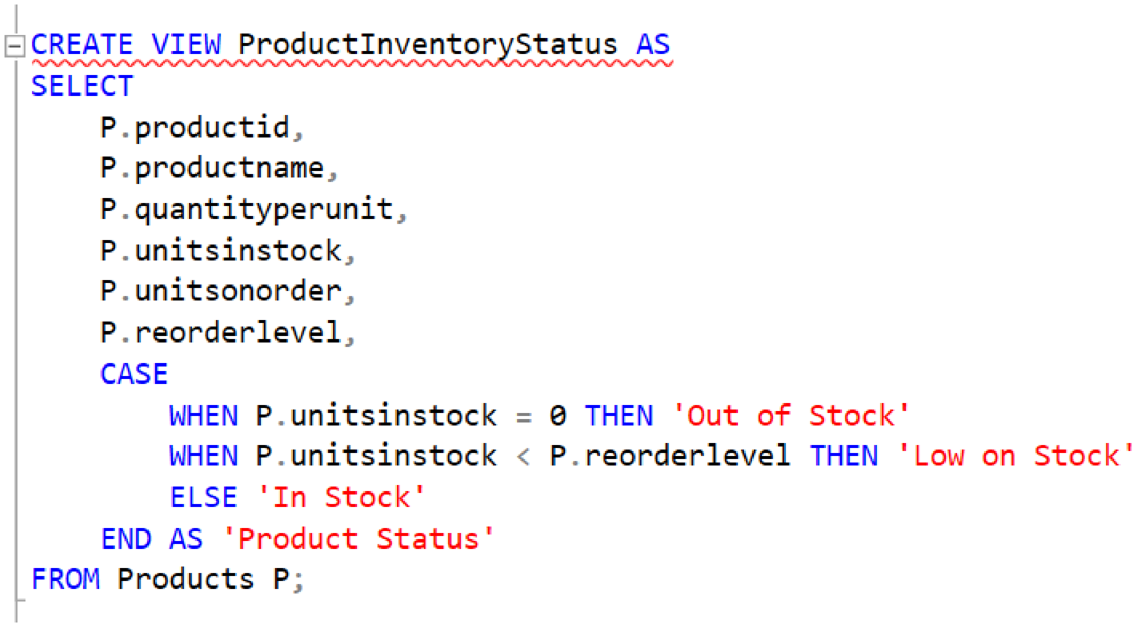
The SalesPerformance view summarizes sales data by product, showing the **total units sold** and **total revenue** for each product, grouped by product and category. It helps analyze product performance and revenue generation.



```
CREATE VIEW SalesPerformance AS
SELECT
   P.productid,
   P.productname,
   C.categoryname,
   SUM(OD.Quantity) AS 'Total Units Sold',
   SUM(OD.UnitPrice * OD.Quantity) AS 'Total Revenue'
FROM Products P, [Order Details] OD, Categories C
WHERE P.productid = OD.ProductID
AND C.categoryid = P.categoryid
GROUP BY P.productid, P.productname,C.categoryname
```

11. ProductInventoryStatus:

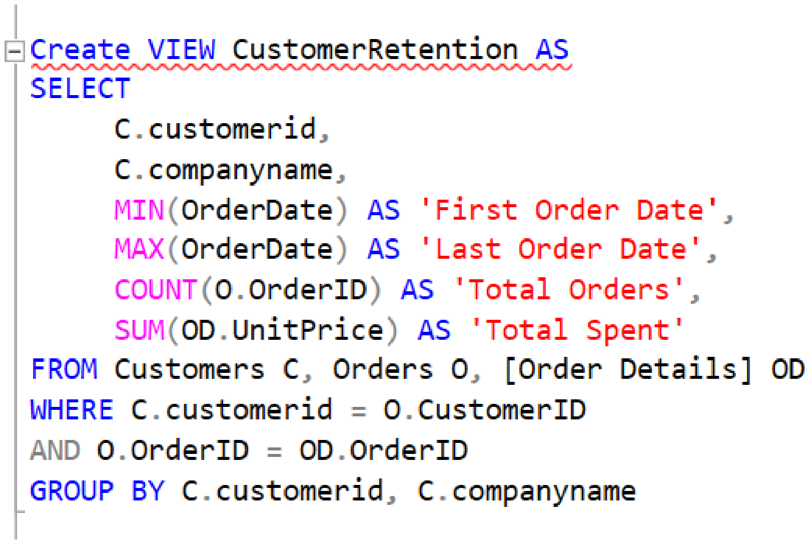
The ProductInventoryStatus view displays product details like unitsinstock, unitsonorder, and reorderlevel, along with a **Product Status** column that categorizes products as **'Out of Stock'**, **'Low on Stock'**, or **'In Stock'** based on inventory levels. It helps monitor and manage product inventory effectively.



```sql
CREATE VIEW ProductInventoryStatus AS
SELECT
    P.productid,
    P.productname,
    P.quantityperunit,
    P.unitsinstock,
    P.unitsonorder,
    P.reorderlevel,
    CASE
        WHEN P.unitsinstock = 0 THEN 'Out of Stock'
        WHEN P.unitsinstock < P.reorderlevel THEN 'Low on Stock'
        ELSE 'In Stock'
    END AS 'Product Status'
FROM Products P;
```

12. CustomerRetention:

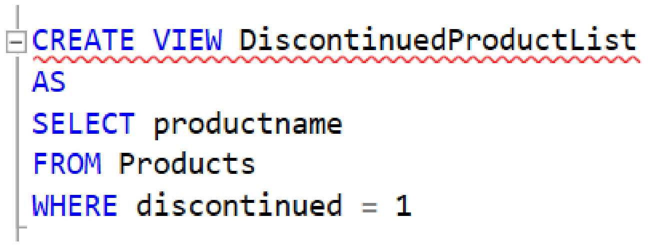
The CustomerRetention view provides insights into customer loyalty by showing the **first and last order dates**, **total orders placed**, and **total amount spent** for each customer. It helps businesses identify loyal customers and evaluate retention strategies.



```sql
Create VIEW CustomerRetention AS
SELECT
     C.customerid,
     C.companyname,
     MIN(OrderDate) AS 'First Order Date',
     MAX(OrderDate) AS 'Last Order Date',
     COUNT(O.OrderID) AS 'Total Orders',
     SUM(OD.UnitPrice) AS 'Total Spent'
FROM Customers C, Orders O, [Order Details] OD
WHERE C.customerid = O.CustomerID
AND O.OrderID = OD.OrderID
GROUP BY C.customerid, C.companyname
```

13. DiscontinuedProductList:

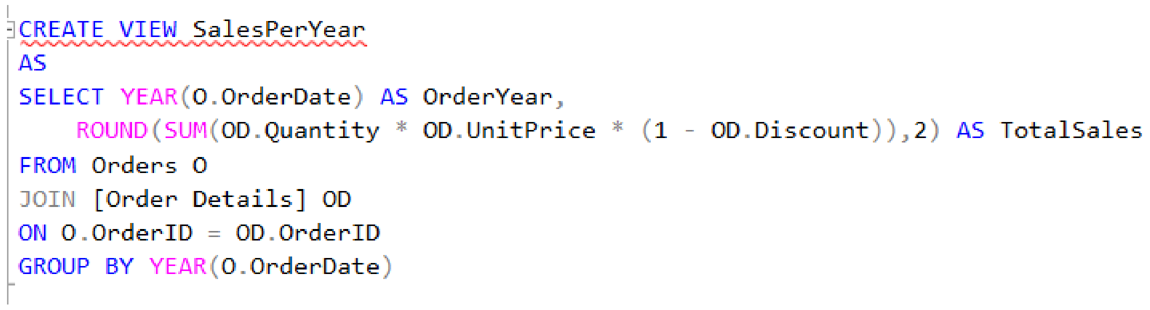
The DiscontinuedProductList view lists all products marked as discontinued (discontinued = 1) by retrieving their productname. It helps businesses track and manage discontinued inventory efficiently.



```
CREATE VIEW DiscontinuedProductList
AS
SELECT productname
FROM Products
WHERE discontinued = 1
```

14. SalesPerYear:

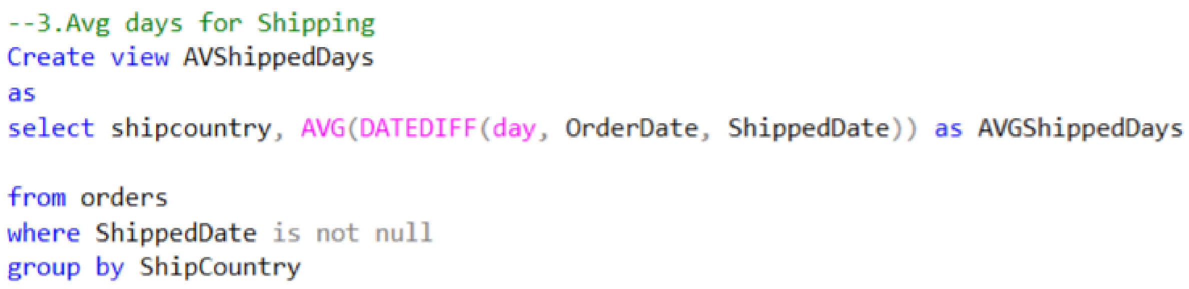
The SalesPerYear view calculates **total sales** for each year, factoring in discounts, and groups the results by OrderYear. It provides a clear overview of yearly revenue trends for business analysis.



```
CREATE VIEW SalesPerYear
AS
SELECT YEAR(O.OrderDate) AS OrderYear,
    ROUND(SUM(OD.Quantity * OD.UnitPrice * (1 - OD.Discount)),2) AS TotalSales
FROM Orders O
JOIN [Order Details] OD
ON O.OrderID = OD.OrderID
GROUP BY YEAR(O.OrderDate)
```

15. AvgShippedDays

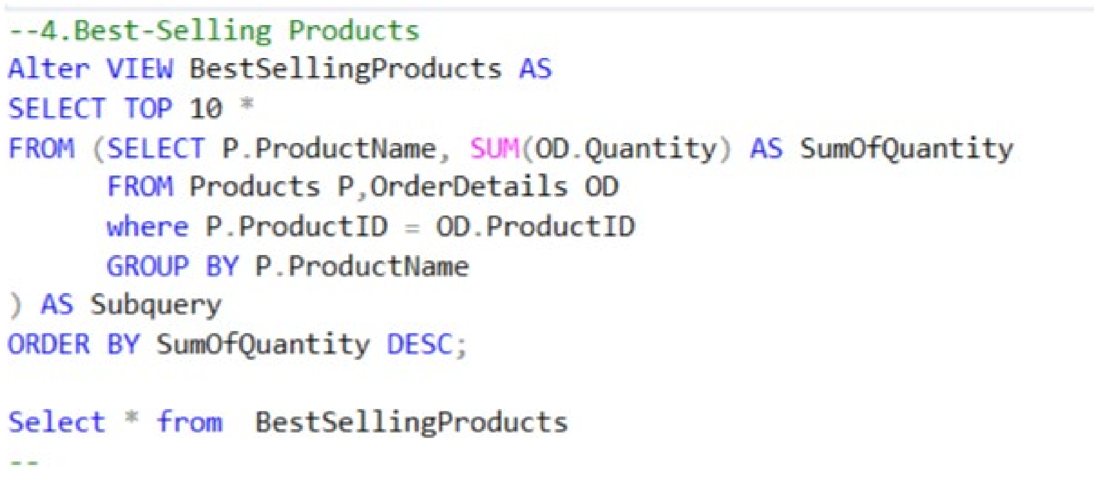
The **AvgShippedDays** view is likely designed to **calculate the average number of days taken to ship orders**. It helps in analyzing shipping efficiency and identifying potential delays in order fulfillment.



```
--3.Avg days for Shipping
Create view AVShippedDays
as
select shipcountry, AVG(DATEDIFF(day, OrderDate, ShippedDate)) as AVGShippedDays

from orders
where ShippedDate is not null
group by ShipCountry
```

16. Best Selling Products

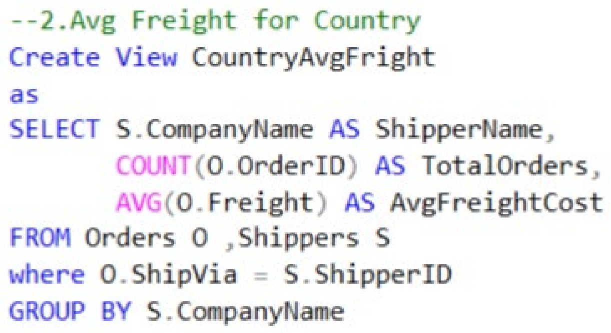
The **BestSellingProducts** view is designed to **identify the top-selling products** based on total sales volume or revenue. This helps businesses understand customer preferences and optimize inventory



```
--4.Best-Selling Products
Alter VIEW BestSellingProducts AS
SELECT TOP 10 *
FROM (SELECT P.ProductName, SUM(OD.Quantity) AS SumOfQuantity
      FROM Products P,OrderDetails OD
      where P.ProductID = OD.ProductID
      GROUP BY P.ProductName
) AS Subquery
ORDER BY SumOfQuantity DESC;

Select * from  BestSellingProducts
--
```

17. CountryAvgFright

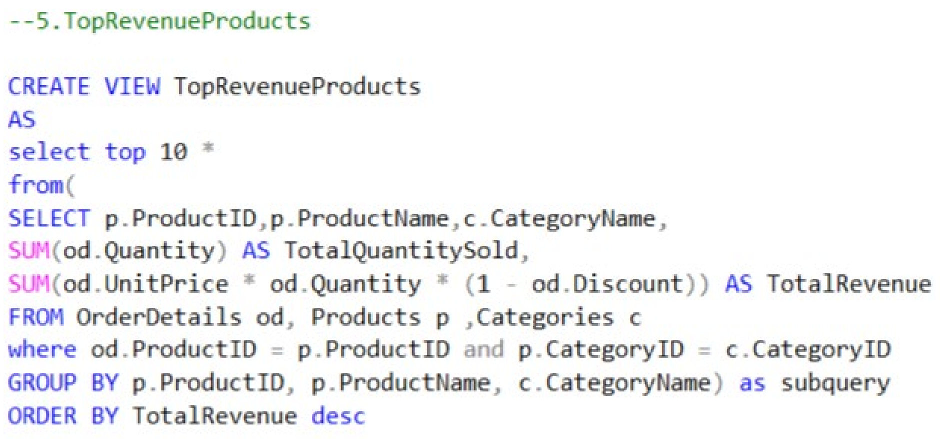
The CountryAvgFreight view calculates the **average freight** cost per country based on orders. It helps businesses analyze shipping expenses and optimize logistics costs.



```
--2.Avg Freight for Country
Create View CountryAvgFright
as
SELECT S.CompanyName AS ShipperName,
       COUNT(O.OrderID) AS TotalOrders,
       AVG(O.Freight) AS AvgFreightCost
FROM Orders O ,Shippers S
where O.ShipVia = S.ShipperID
GROUP BY S.CompanyName
```

18. TopRevenueProducts

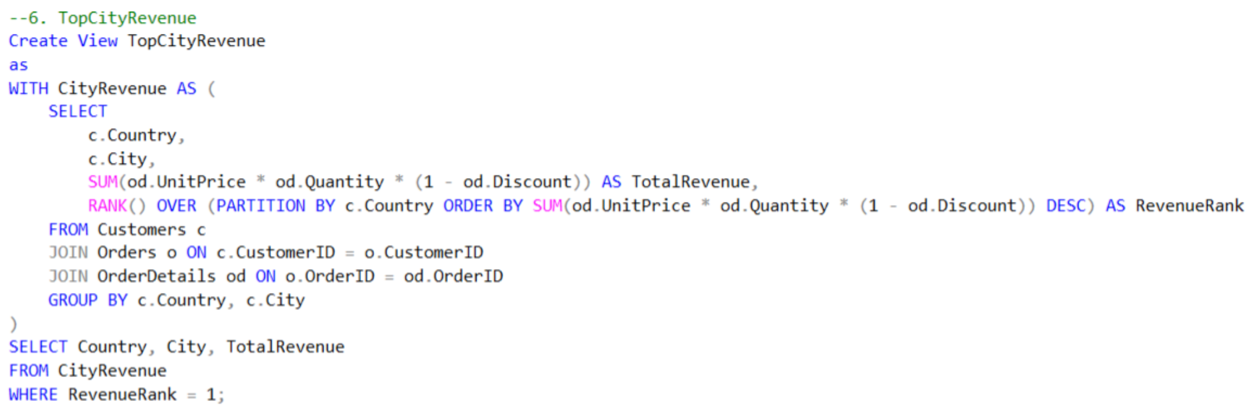
The **TopRevenueProducts** view identifies the **highest revenue-generating products** based on total sales. This helps businesses focus on their most profitable products for marketing, inventory, and pricing strategies



```sql
--5.TopRevenueProducts

CREATE VIEW TopRevenueProducts
AS
select top 10 *
from(
SELECT p.ProductID,p.ProductName,c.CategoryName,
SUM(od.Quantity) AS TotalQuantitySold,
SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) AS TotalRevenue
FROM OrderDetails od, Products p ,Categories c
where od.ProductID = p.ProductID and p.CategoryID = c.CategoryID
GROUP BY p.ProductID, p.ProductName, c.CategoryName) as subquery
ORDER BY TotalRevenue desc
```

19. TopCityRevenue

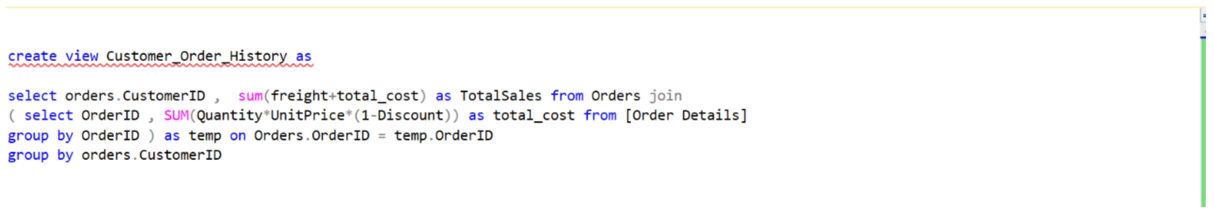
The **TopCityRevenue** view identifies the city with the highest total revenue for each country. It helps businesses understand which cities contribute the most to sales within different regions.



```sql
--6. TopCityRevenue
Create View TopCityRevenue
as
WITH CityRevenue AS (
    SELECT
        c.Country,
        c.City,
        SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) AS TotalRevenue,
        RANK() OVER (PARTITION BY c.Country ORDER BY SUM(od.UnitPrice * od.Quantity * (1 - od.Discount)) DESC) AS RevenueRank
    FROM Customers c
    JOIN Orders o ON c.CustomerID = o.CustomerID
    JOIN OrderDetails od ON o.OrderID = od.OrderID
    GROUP BY c.Country, c.City
)
SELECT Country, City, TotalRevenue
FROM CityRevenue
WHERE RevenueRank = 1;
```

20. CustomerOrderHistory

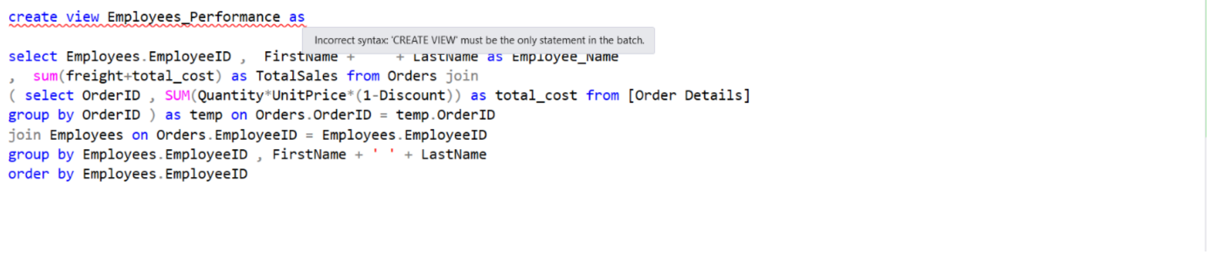
Displays the total sales for each customer by summing up order freight costs and product sales from the Orders and Order Details tables.



```
create view Customer_Order_History as

select orders.CustomerID ,  sum(freight+total_cost) as TotalSales from Orders join
( select OrderID , SUM(Quantity*UnitPrice*(1-Discount)) as total_cost from [Order Details]
group by OrderID ) as temp on Orders.OrderID = temp.OrderID
group by orders.CustomerID
```

21. EmployeesPerformance

Shows employee performance by calculating the total sales handled by each employee, including order freight and product sales. Employees are listed by their full name and sorted by employee ID.

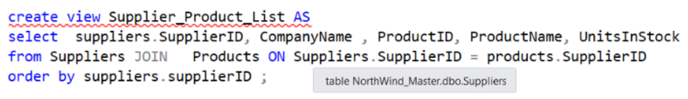


```
create view Employees_Performance as

select Employees.EmployeeID ,  FirstName +      + LastName as Employee_Name
,  sum(freight+total_cost) as TotalSales from Orders join
( select OrderID , SUM(Quantity*UnitPrice*(1-Discount)) as total_cost from [Order Details]
group by OrderID ) as temp on Orders.OrderID = temp.OrderID
join Employees on Orders.EmployeeID = Employees.EmployeeID
group by Employees.EmployeeID , FirstName + ' ' + LastName
order by Employees.EmployeeID
```

Incorrect syntax: 'CREATE VIEW' must be the only statement in the batch.

22. SupplierProductList

Provides a list of suppliers and their respective products, including supplier ID, company name, product ID, product name, and available stock. The results are ordered by supplier ID.

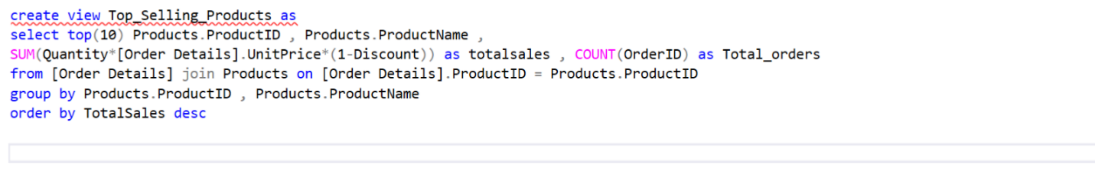


```
create view Supplier_Product_List AS
select  suppliers.SupplierID, CompanyName , ProductID, ProductName, UnitsInStock
from Suppliers JOIN   Products ON Suppliers.SupplierID = products.SupplierID
order by suppliers.supplierID ;
```

table NorthWind_Master.dbo.Suppliers

23. TopSellingProducts

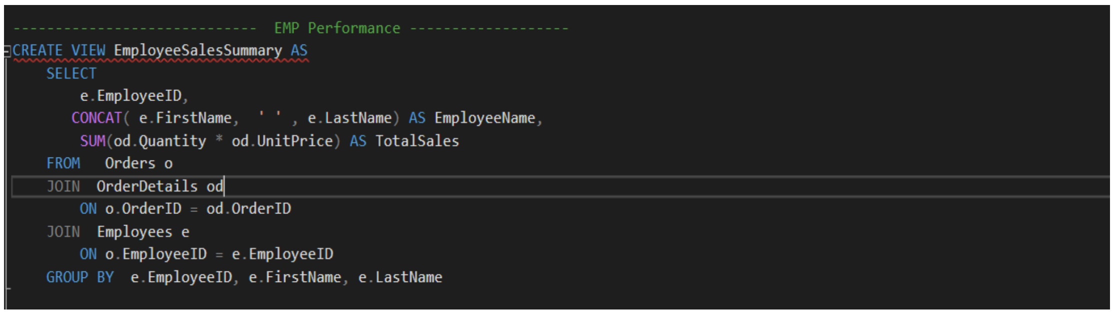
Displays the top 10 best-selling products based on total sales revenue. It includes product ID, product name, total sales amount, and the number of orders each product was included in. The results are sorted in descending order of total sales.



```
create view Top_Selling_Products as
select top(10) Products.ProductID , Products.ProductName ,
SUM(Quantity*[Order Details].UnitPrice*(1-Discount)) as totalsales , COUNT(OrderID) as Total_orders
from [Order Details] join Products on [Order Details].ProductID = Products.ProductID
group by Products.ProductID , Products.ProductName
order by TotalSales desc
```

24. EMPLOYEE PERFORMANCE

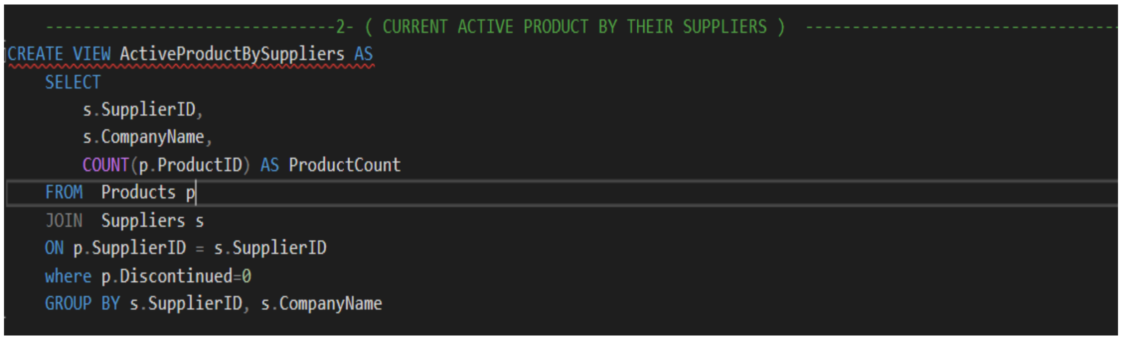
This SQL code creates a **view** called EmployeeSalesSummary, which summarizes total sales made by each employee. It joins the Orders, OrderDetails, and Employees tables to calculate total sales (Quantity * UnitPrice) for each employee. The results are grouped by EmployeeID and employee name. This view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



```
----------------------------  EMP Performance ------------------
CREATE VIEW EmployeeSalesSummary AS
    SELECT
        e.EmployeeID,
       CONCAT( e.FirstName,  ' ' , e.LastName) AS EmployeeName,
        SUM(od.Quantity * od.UnitPrice) AS TotalSales
    FROM   Orders o
    JOIN  OrderDetails od
        ON o.OrderID = od.OrderID
    JOIN  Employees e
        ON o.EmployeeID = e.EmployeeID
    GROUP BY  e.EmployeeID, e.FirstName, e.LastName
```

25. Current Active Product By Suppliers

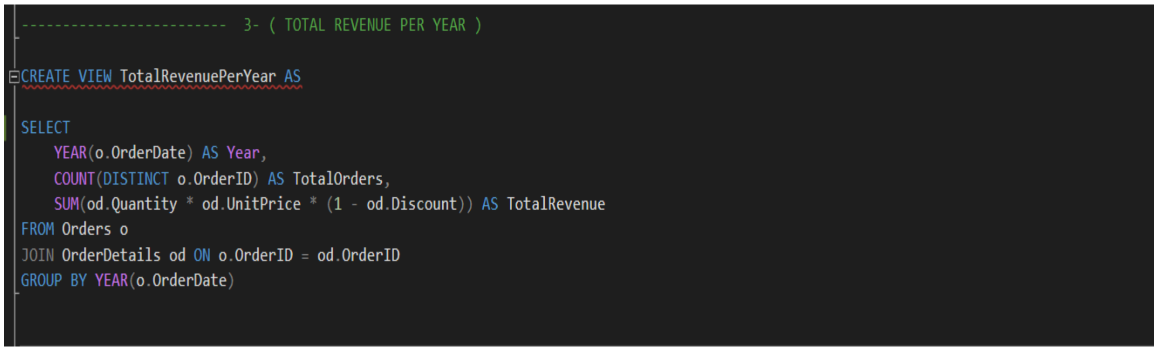
This SQL code creates a **view** called ActiveProductBySuppliers, which displays the number of active (non-discontinued) products supplied by each supplier. It joins the Products and Suppliers tables based on SupplierID, filters out discontinued products (Discontinued = 0), and counts the active products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



```
    -------------------------------2- ( CURRENT ACTIVE PRODUCT BY THEIR SUPPLIERS )  ----------------------------------
CREATE VIEW ActiveProductBySuppliers AS
    SELECT
        s.SupplierID,
        s.CompanyName,
        COUNT(p.ProductID) AS ProductCount
    FROM  Products p
    JOIN  Suppliers s
    ON p.SupplierID = s.SupplierID
    where p.Discontinued=0
    GROUP BY s.SupplierID, s.CompanyName
```

26. Total Revenue Per Year

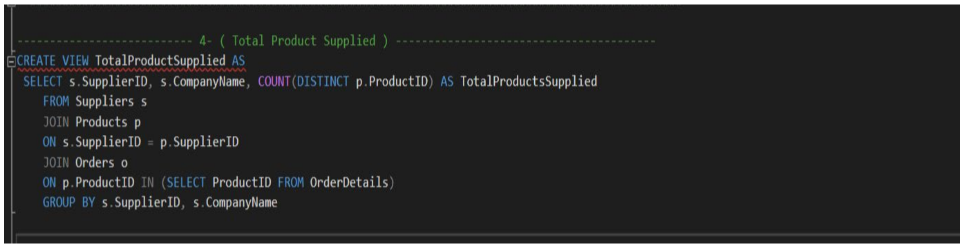
This SQL code creates a view called TotalRevenuePerYear, which displays the total number of orders and total revenue for each year. It joins the Orders and OrderDetails tables based on OrderID, extracts the year from OrderDate, counts the distinct orders as TotalOrders, and calculates TotalRevenue by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



```
------------------------  3- ( TOTAL REVENUE PER YEAR )

CREATE VIEW TotalRevenuePerYear AS

SELECT
    YEAR(o.OrderDate) AS Year,
    COUNT(DISTINCT o.OrderID) AS TotalOrders,
    SUM(od.Quantity * od.UnitPrice * (1 - od.Discount)) AS TotalRevenue
FROM Orders o
JOIN OrderDetails od ON o.OrderID = od.OrderID
GROUP BY YEAR(o.OrderDate)
```

27. Total Product supplied by Every Suppliers

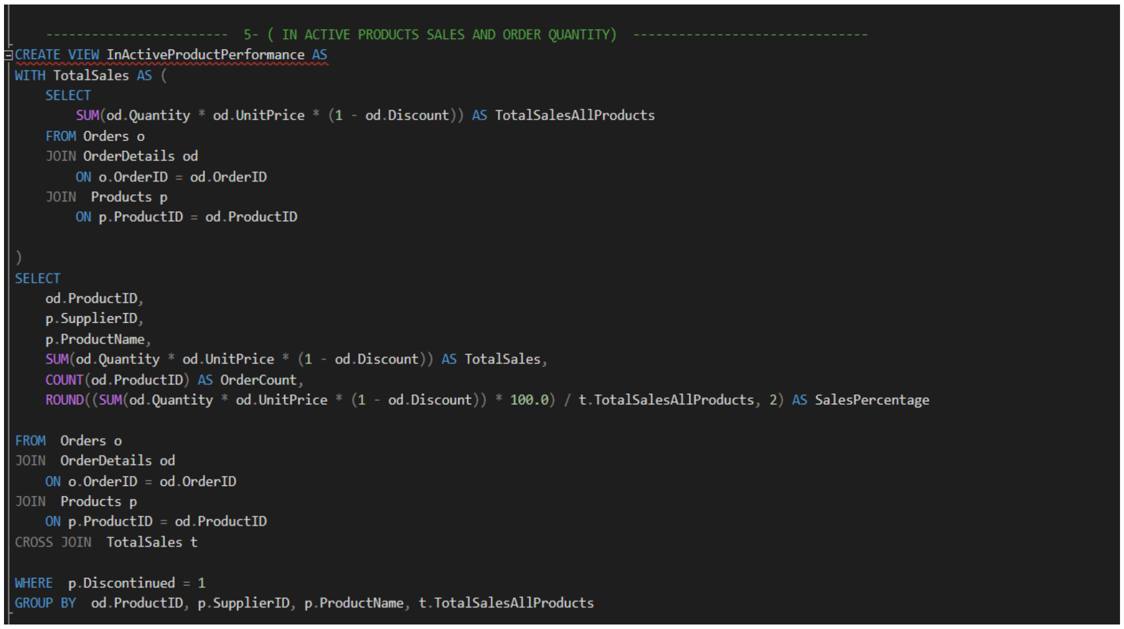
This SQL code creates a view called TotalProductSupplied, which displays the total number of unique products supplied by each supplier. It joins the Suppliers and Products tables based on SupplierID, then links to the Orders table by checking if the product appears in OrderDetails. The query counts distinct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



```
------------------------- 4- ( Total Product Supplied ) --------------------------------------
CREATE VIEW TotalProductSupplied AS
 SELECT s.SupplierID, s.CompanyName, COUNT(DISTINCT p.ProductID) AS TotalProductsSupplied
    FROM Suppliers s
    JOIN Products p
    ON s.SupplierID = p.SupplierID
    JOIN Orders o
    ON p.ProductID IN (SELECT ProductID FROM OrderDetails)
    GROUP BY s.SupplierID, s.CompanyName
```

28. Inactive Products & Quantity Ordered And percentage from total Sales

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.



```
    -----------------------  5- ( IN ACTIVE PRODUCTS SALES AND ORDER QUANTITY)  ------------------------------
CREATE VIEW InActiveProductPerformance AS
WITH TotalSales AS (
    SELECT
        SUM(od.Quantity * od.UnitPrice * (1 - od.Discount)) AS TotalSalesAllProducts
    FROM Orders o
    JOIN OrderDetails od
        ON o.OrderID = od.OrderID
    JOIN  Products p
        ON p.ProductID = od.ProductID

)
SELECT
    od.ProductID,
    p.SupplierID,
    p.ProductName,
    SUM(od.Quantity * od.UnitPrice * (1 - od.Discount)) AS TotalSales,
    COUNT(od.ProductID) AS OrderCount,
    ROUND((SUM(od.Quantity * od.UnitPrice * (1 - od.Discount)) * 100.0) / t.TotalSalesAllProducts, 2) AS SalesPercentage

FROM  Orders o
JOIN  OrderDetails od
    ON o.OrderID = od.OrderID
JOIN  Products p
    ON p.ProductID = od.ProductID
CROSS JOIN  TotalSales t

WHERE  p.Discontinued = 1
GROUP BY  od.ProductID, p.SupplierID, p.ProductName, t.TotalSalesAllProducts
```

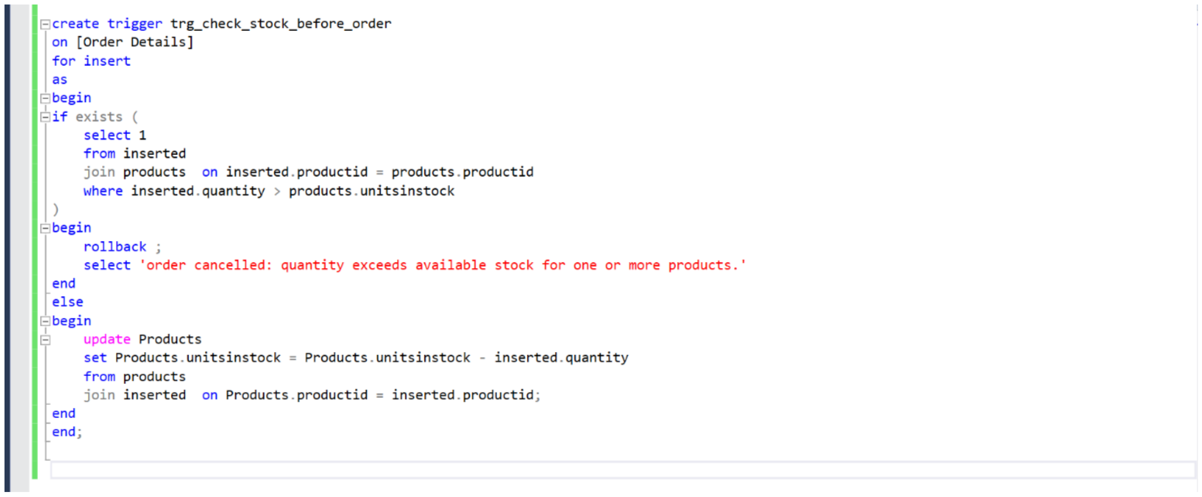
3. Triggers

A **Trigger** is a special type of stored procedure in SQL that **automatically executes** when a specific event (INSERT, UPDATE, DELETE) occurs in a table. Triggers help enforce business rules, maintain data integrity, and automate tasks.

13 Triggers have been created:

1. CheckStockBeforeOrder

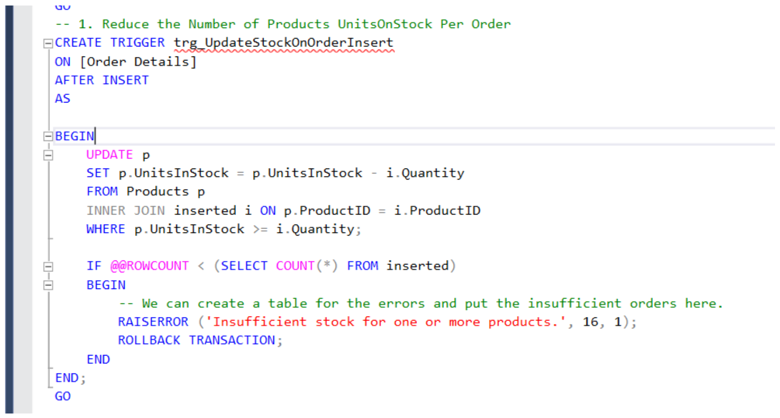
This trigger ensures that an order cannot be placed if the requested quantity of a product exceeds the available stock.



```
create trigger trg_check_stock_before_order
on [Order Details]
for insert
as
begin
if exists (
    select 1
    from inserted
    join products  on inserted.productid = products.productid
    where inserted.quantity > products.unitsinstock
)
begin
    rollback ;
    select 'order cancelled: quantity exceeds available stock for one or more products.'
end
else
begin
    update Products
    set Products.unitsinstock = Products.unitsinstock - inserted.quantity
    from products
    join inserted  on Products.productid = inserted.productid;
end
end;
```

2.UpdateStockOnOrderInsert

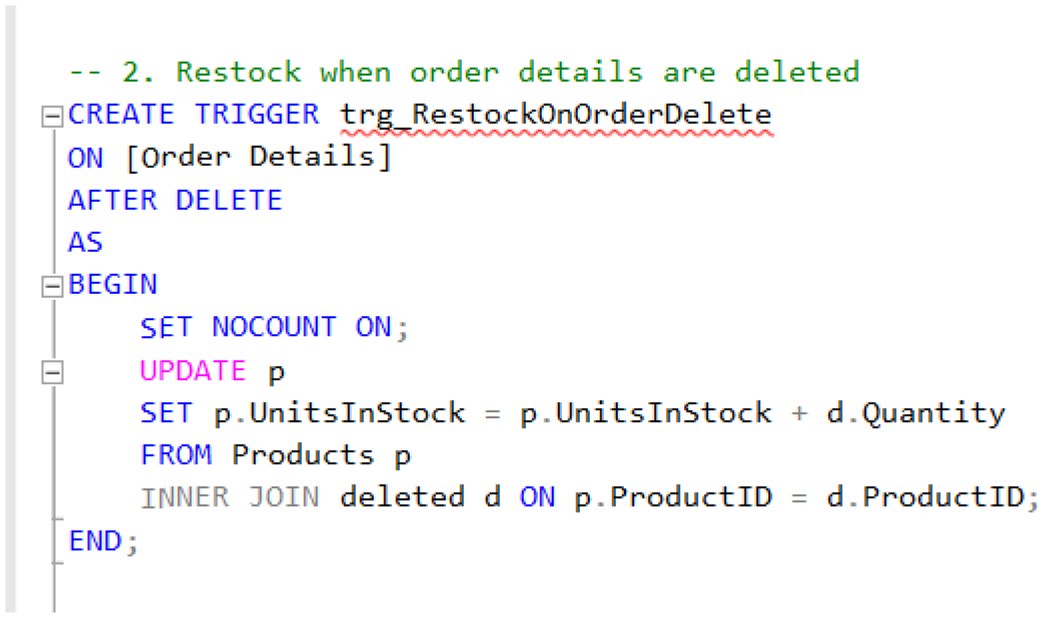
Reduces **UnitsInStock** in the **Products** table when a new order detail is inserted, rolling back if insufficient stock is available.



```
-- 1. Reduce the Number of Products UnitsOnStock Per Order
CREATE TRIGGER trg_UpdateStockOnOrderInsert
ON [Order Details]
AFTER INSERT
AS

BEGIN
    UPDATE p
    SET p.UnitsInStock = p.UnitsInStock - i.Quantity
    FROM Products p
    INNER JOIN inserted i ON p.ProductID = i.ProductID
    WHERE p.UnitsInStock >= i.Quantity;

    IF @@ROWCOUNT < (SELECT COUNT(*) FROM inserted)
    BEGIN
        -- We can create a table for the errors and put the insufficient orders here.
        RAISERROR ('Insufficient stock for one or more products.', 16, 1);
        ROLLBACK TRANSACTION;
    END
END;
GO
```

3.RestockOnOrderDelete

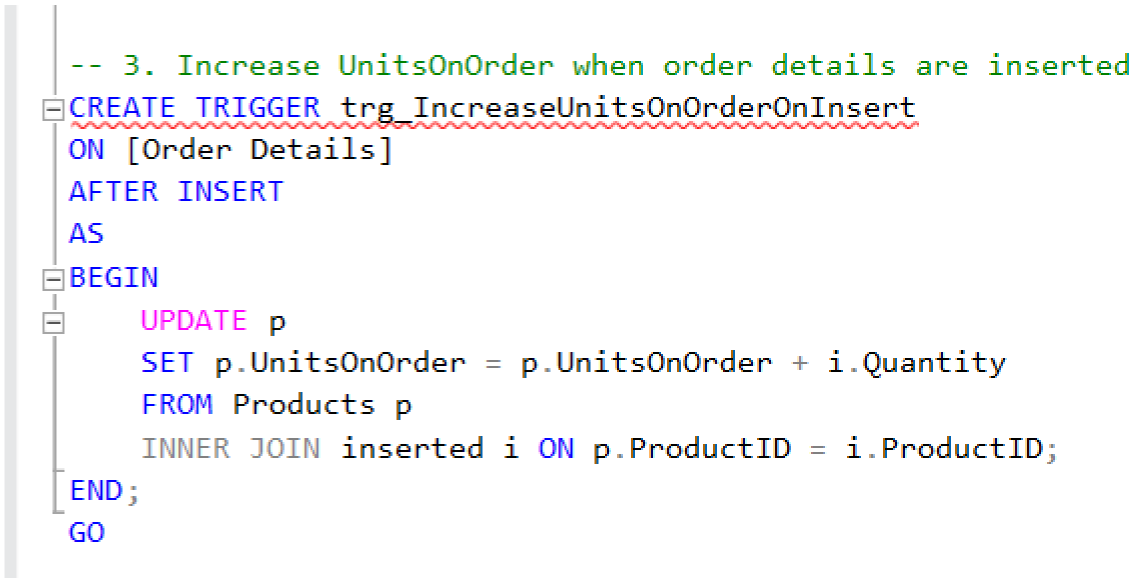
Increases **UnitsInStock** in the **Products** table when an order detail is deleted, effectively restocking the product.



```
-- 2. Restock when order details are deleted
CREATE TRIGGER trg_RestockOnOrderDelete
ON [Order Details]
AFTER DELETE
AS
BEGIN
    SET NOCOUNT ON;
    UPDATE p
    SET p.UnitsInStock = p.UnitsInStock + d.Quantity
    FROM Products p
    INNER JOIN deleted d ON p.ProductID = d.ProductID;
END;
```

4.IncreaseUnitsOnOrderOnInsert

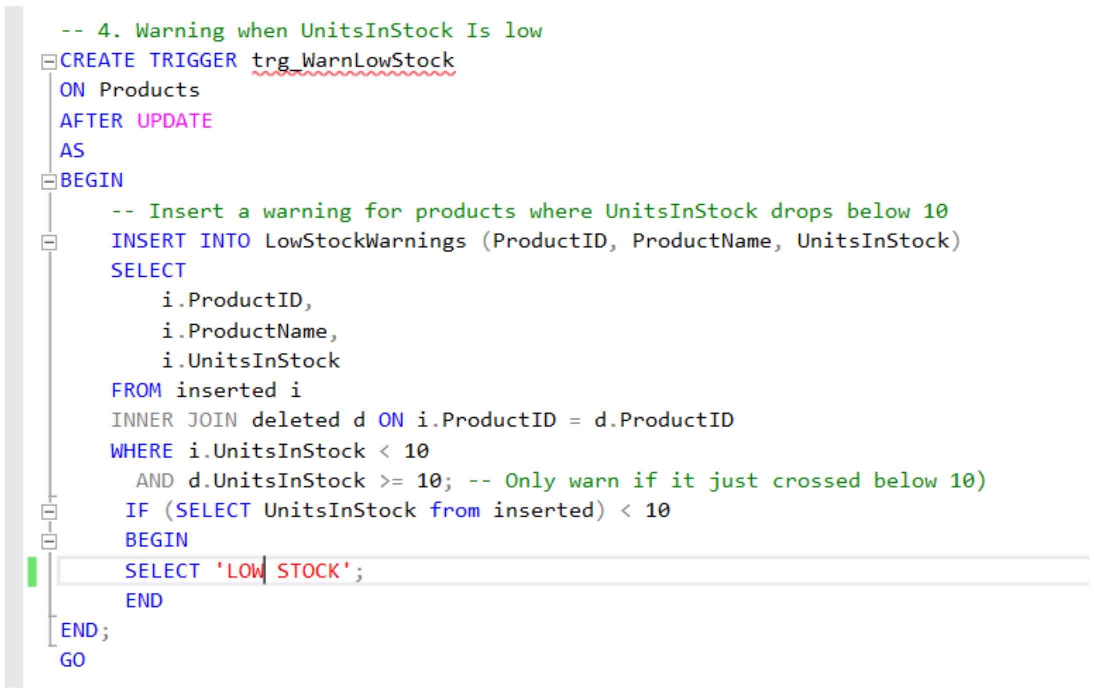
Increases **UnitsOnOrder** in the **Products** table when a new order detail is inserted, tracking pending orders.



```
-- 3. Increase UnitsOnOrder when order details are inserted
CREATE TRIGGER trg_IncreaseUnitsOnOrderOnInsert
ON [Order Details]
AFTER INSERT
AS
BEGIN
    UPDATE p
    SET p.UnitsOnOrder = p.UnitsOnOrder + i.Quantity
    FROM Products p
    INNER JOIN inserted i ON p.ProductID = i.ProductID;
END;
GO
```

5.WarnLowStock

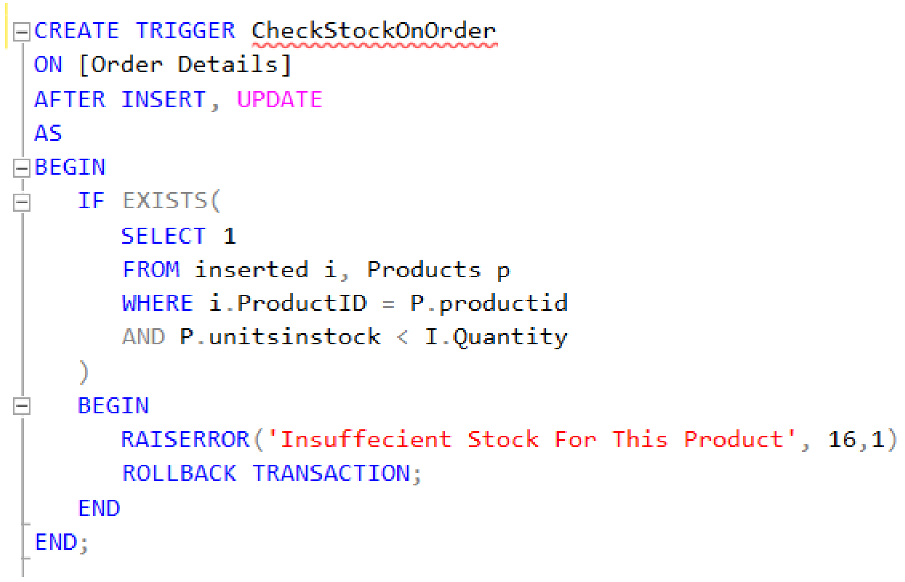
Logs a warning to the **LowStockWarnings** table and outputs a message when **UnitsInStock** falls below 10 after an update to the **Products** table.



```
-- 4. Warning when UnitsInStock Is low
CREATE TRIGGER trg_WarnLowStock
ON Products
AFTER UPDATE
AS
BEGIN
    -- Insert a warning for products where UnitsInStock drops below 10
    INSERT INTO LowStockWarnings (ProductID, ProductName, UnitsInStock)
    SELECT
        i.ProductID,
        i.ProductName,
        i.UnitsInStock
    FROM inserted i
    INNER JOIN deleted d ON i.ProductID = d.ProductID
    WHERE i.UnitsInStock < 10
      AND d.UnitsInStock >= 10; -- Only warn if it just crossed below 10)
     IF (SELECT UnitsInStock from inserted) < 10
     BEGIN
     SELECT 'LOW STOCK';
     END
END;
GO
```

6. CheckStockOnOrder:

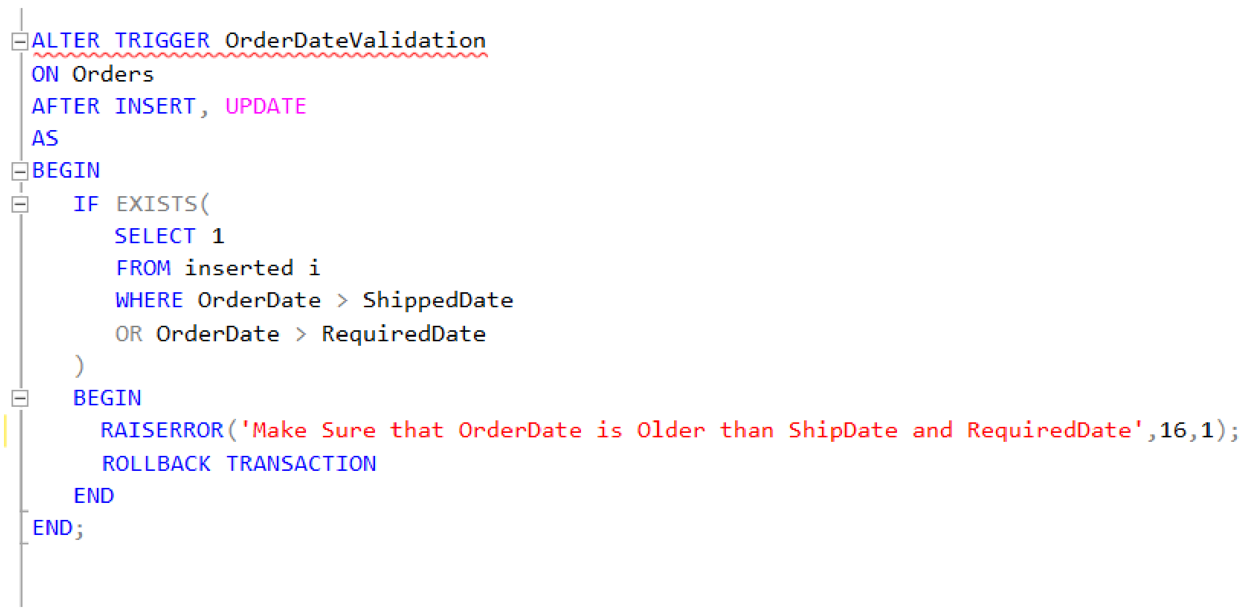
The CheckStockOnOrder trigger checks if the **available stock** (unitsinstock) is sufficient to fulfill the order quantity. If not, it raises an error and rolls back the transaction, preventing orders that exceed available inventory.



```
CREATE TRIGGER CheckStockOnOrder
ON [Order Details]
AFTER INSERT, UPDATE
AS
BEGIN
    IF EXISTS(
        SELECT 1
        FROM inserted i, Products p
        WHERE i.ProductID = P.productid
        AND P.unitsinstock < I.Quantity
    )
    BEGIN
        RAISERROR('Insuffecient Stock For This Product', 16,1)
        ROLLBACK TRANSACTION;
    END
END;
```

7. **OrderDateValidation**:

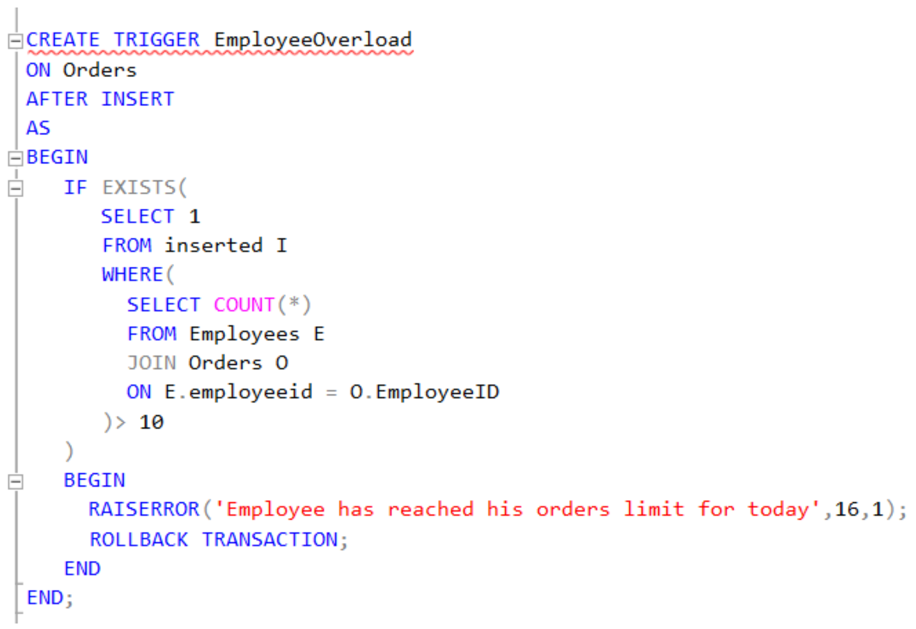
The OrderDateValidation trigger ensures that the **OrderDate** is always earlier than both the **ShippedDate** and **RequiredDate**. If not, it raises an error and rolls back the transaction, maintaining data integrity for order timelines.



```sql
ALTER TRIGGER OrderDateValidation
ON Orders
AFTER INSERT, UPDATE
AS
BEGIN
    IF EXISTS(
        SELECT 1
        FROM inserted i
        WHERE OrderDate > ShippedDate
        OR OrderDate > RequiredDate
    )
    BEGIN
      RAISERROR('Make Sure that OrderDate is Older than ShipDate and RequiredDate',16,1);
      ROLLBACK TRANSACTION
    END
END;
```

8. EmployeeOverload

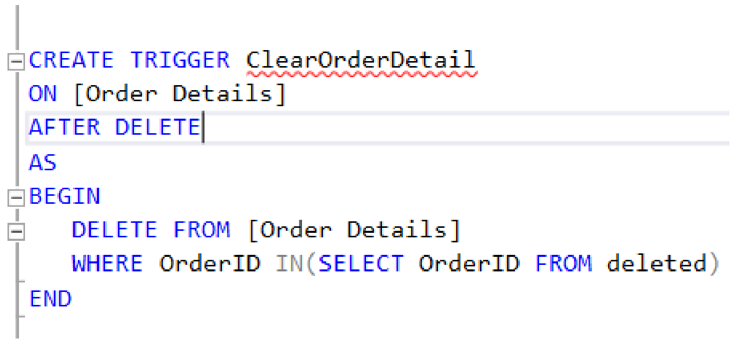
The EmployeeOverload trigger checks if an employee has more than **10 orders** assigned to them in a day. If so, it raises an error and rolls back the transaction, ensuring fair workload distribution among employees.



```sql
CREATE TRIGGER EmployeeOverload
ON Orders
AFTER INSERT
AS
BEGIN
    IF EXISTS(
        SELECT 1
        FROM inserted I
        WHERE(
          SELECT COUNT(*)
          FROM Employees E
          JOIN Orders O
          ON E.employeeid = O.EmployeeID
        )> 10
    )
    BEGIN
      RAISERROR('Employee has reached his orders limit for today',16,1);
      ROLLBACK TRANSACTION;
    END
END;
```

9. ClearOrderDetail

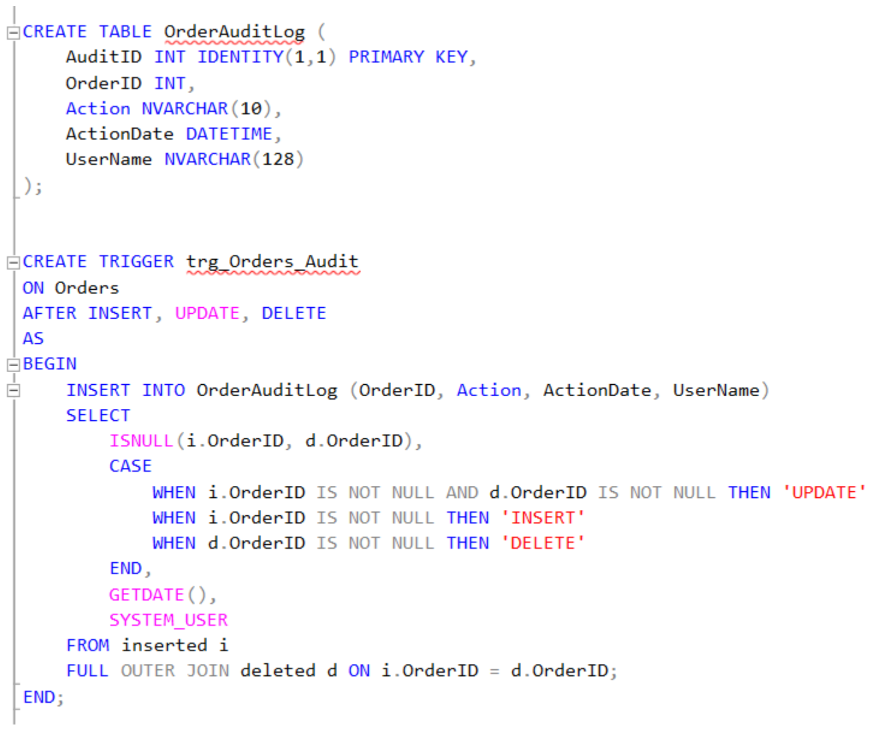
The ClearOrderDetail trigger deletes all rows from the **Order Details** table where the **OrderID** matches the deleted order. It ensures data consistency by removing orphaned order details.



```sql
CREATE TRIGGER ClearOrderDetail
ON [Order Details]
AFTER DELETE
AS
BEGIN
    DELETE FROM [Order Details]
    WHERE OrderID IN(SELECT OrderID FROM deleted)
END
```

10.OrdersAudit

The trg_Orders_Audit trigger logs every change (insert, update, or delete) into an **OrderAuditLog** table, recording the **OrderID**, **action type**, **action date**, and **user** who performed the action. It helps maintain an audit trail for order-related activities.



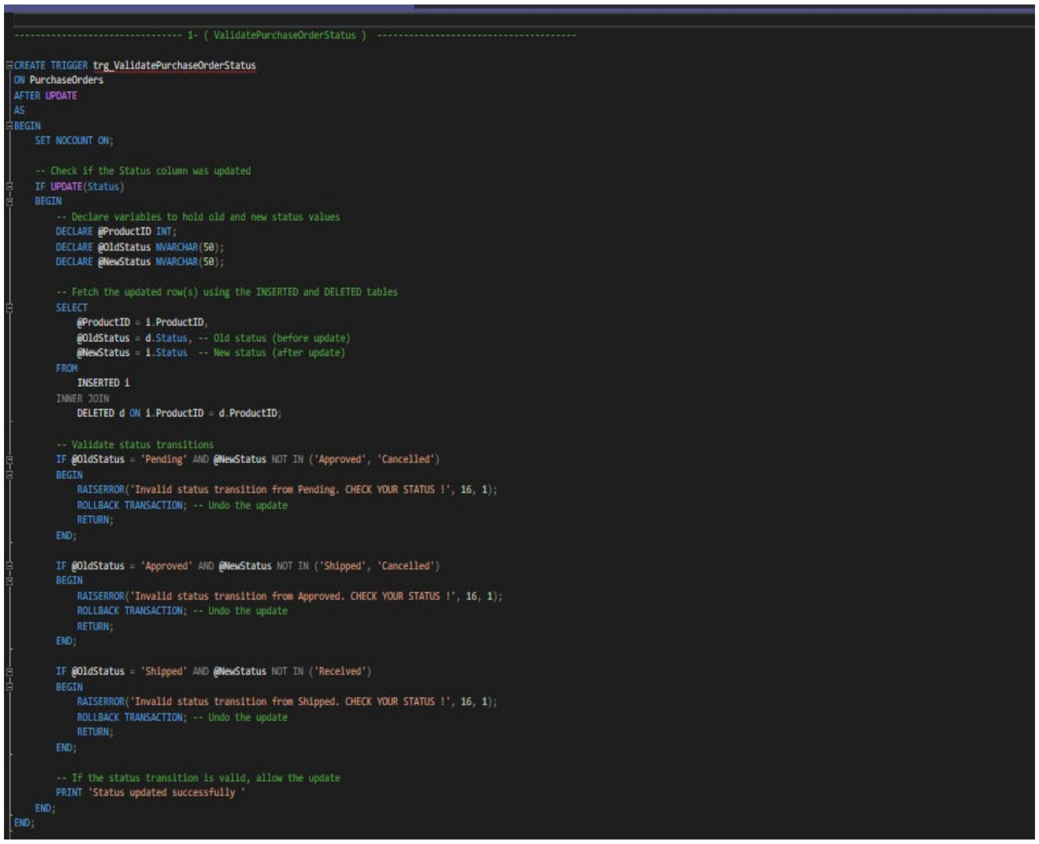
```sql
CREATE TABLE OrderAuditLog (
    AuditID INT IDENTITY(1,1) PRIMARY KEY,
    OrderID INT,
    Action NVARCHAR(10),
    ActionDate DATETIME,
    UserName NVARCHAR(128)
);

CREATE TRIGGER trg_Orders_Audit
ON Orders
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
    INSERT INTO OrderAuditLog (OrderID, Action, ActionDate, UserName)
    SELECT
        ISNULL(i.OrderID, d.OrderID),
        CASE
            WHEN i.OrderID IS NOT NULL AND d.OrderID IS NOT NULL THEN 'UPDATE'
            WHEN i.OrderID IS NOT NULL THEN 'INSERT'
            WHEN d.OrderID IS NOT NULL THEN 'DELETE'
        END,
        GETDATE(),
        SYSTEM_USER
    FROM inserted i
    FULL OUTER JOIN deleted d ON i.OrderID = d.OrderID;
END;
```

11.Validate Purchasing Order Status

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.



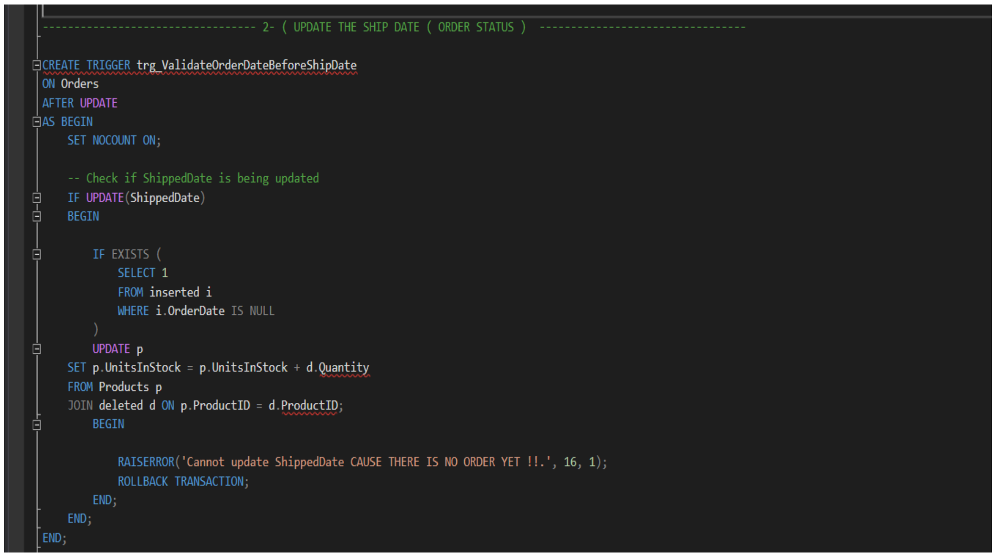
```sql
------------------------------- 1- ( ValidatePurchaseOrderStatus )  -------------------------------------

CREATE TRIGGER trg_ValidatePurchaseOrderStatus
ON PurchaseOrders
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    -- Check if the Status column was updated
    IF UPDATE(Status)
    BEGIN
        -- Declare variables to hold old and new status values
        DECLARE @ProductID INT;
        DECLARE @OldStatus NVARCHAR(50);
        DECLARE @NewStatus NVARCHAR(50);

        -- Fetch the updated row(s) using the INSERTED and DELETED tables
        SELECT
            @ProductID = i.ProductID,
            @OldStatus = d.Status, -- Old status (before update)
            @NewStatus = i.Status  -- New status (after update)
        FROM
            INSERTED i
        INNER JOIN
            DELETED d ON i.ProductID = d.ProductID;

        -- Validate status transitions
        IF @OldStatus = 'Pending' AND @NewStatus NOT IN ('Approved', 'Cancelled')
        BEGIN
            RAISERROR('Invalid status transition from Pending. CHECK YOUR STATUS !', 16, 1);
            ROLLBACK TRANSACTION; -- Undo the update
            RETURN;
        END;

        IF @OldStatus = 'Approved' AND @NewStatus NOT IN ('Shipped', 'Cancelled')
        BEGIN
            RAISERROR('Invalid status transition from Approved. CHECK YOUR STATUS !', 16, 1);
            ROLLBACK TRANSACTION; -- Undo the update
            RETURN;
        END;

        IF @OldStatus = 'Shipped' AND @NewStatus NOT IN ('Received')
        BEGIN
            RAISERROR('Invalid status transition from Shipped. CHECK YOUR STATUS !', 16, 1);
            ROLLBACK TRANSACTION; -- Undo the update
            RETURN;
        END;

        -- If the status transition is valid, allow the update
        PRINT 'Status updated successfully '
    END;
END;
```

12. Update Ship Date (Order Status)

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.

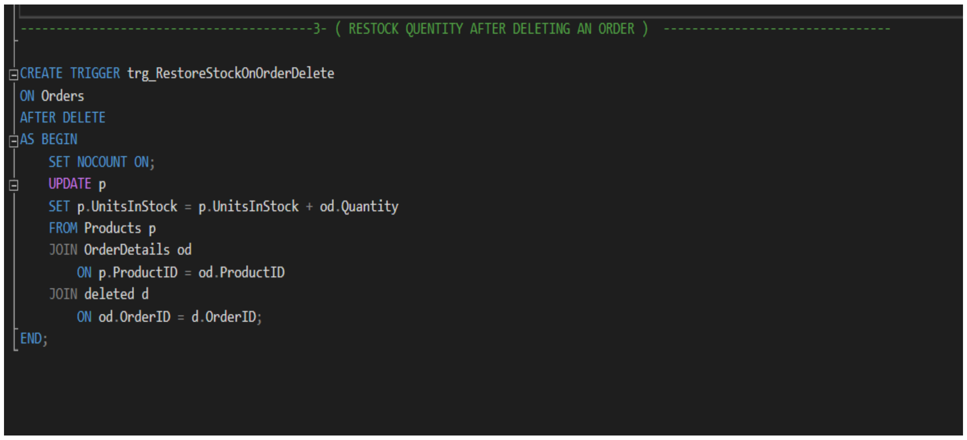


```sql
--------------------------------- 2- ( UPDATE THE SHIP DATE ( ORDER STATUS )  --------------------------------

CREATE TRIGGER trg_ValidateOrderDateBeforeShipDate
ON Orders
AFTER UPDATE
AS BEGIN
    SET NOCOUNT ON;

    -- Check if ShippedDate is being updated
    IF UPDATE(ShippedDate)
    BEGIN

        IF EXISTS (
            SELECT 1
            FROM inserted i
            WHERE i.OrderDate IS NULL
        )
        UPDATE p
    SET p.UnitsInStock = p.UnitsInStock + d.Quantity
    FROM Products p
    JOIN deleted d ON p.ProductID = d.ProductID;
        BEGIN

            RAISERROR('Cannot update ShippedDate CAUSE THERE IS NO ORDER YET !!.', 16, 1);
            ROLLBACK TRANSACTION;
        END;
    END;
END;
```

13. Restock Quantity After Deleting an Order

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

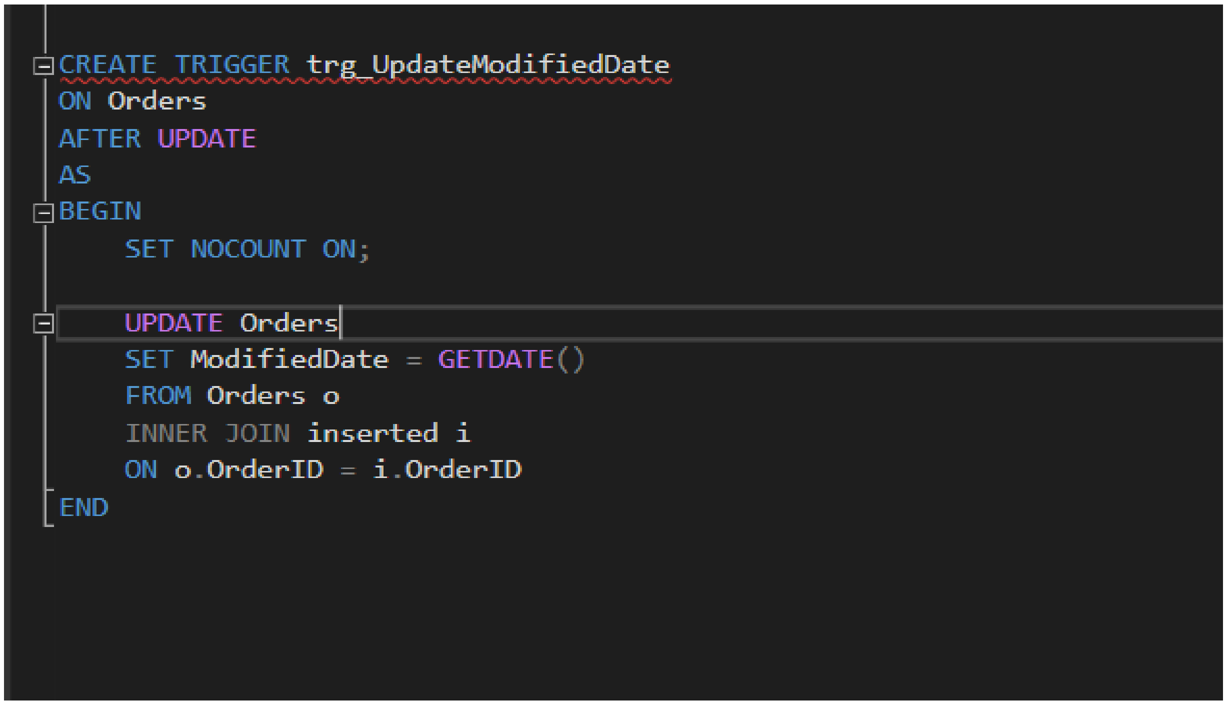
- The trigger executes **AFTER a DELETE** operation on Orders.
- It updates the UnitsInStock in the Products table by adding back the quantity of products from the deleted order.
- This ensures that when an order is removed, the associated products are returned to stock.



```sql
----------------------------------------3- ( RESTOCK QUENTITY AFTER DELETING AN ORDER )  --------------------------------

CREATE TRIGGER trg_RestoreStockOnOrderDelete
ON Orders
AFTER DELETE
AS BEGIN
    SET NOCOUNT ON;
    UPDATE p
    SET p.UnitsInStock = p.UnitsInStock + od.Quantity
    FROM Products p
    JOIN OrderDetails od
        ON p.ProductID = od.ProductID
    JOIN deleted d
        ON od.OrderID = d.OrderID;
END;
```

14. Updating Modified Date

This trigger ensures that whenever any row in the **`Orders`** table is **updated**, the `ModifiedDate` column is automatically set to the current timestamp **(`GETDATE()`).** This helps track data changes for **incremental loading** in a data warehouse, allowing ETL processes to efficiently identify and extract only new or updated records based on the **`ModifiedDate`**. To fully support **incremental load**, a similar trigger for **inserts** should also be implemented to set **`ModifiedDate`** when new rows are added.



```
CREATE TRIGGER trg_UpdateModifiedDate
ON Orders
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    UPDATE Orders
    SET ModifiedDate = GETDATE()
    FROM Orders o
    INNER JOIN inserted i
    ON o.OrderID = i.OrderID
END
```

4. Constraints

Constraints are rules applied to table columns to **maintain data integrity and consistency** in a database. They ensure that invalid data **cannot** be inserted, updated, or deleted.

1. chk_UnitPrice

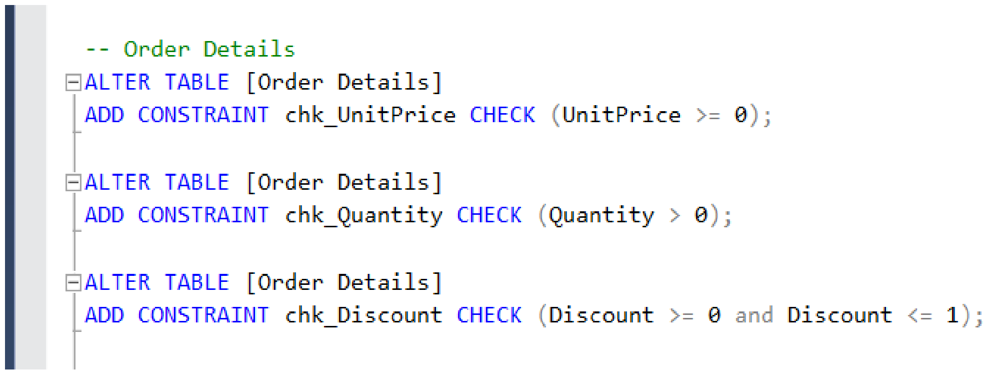
Ensures that the **UnitPrice** in the **[Order Details]** table is non-negative (greater than or equal to 0), preventing invalid pricing data.

2. chk_Quantity

Ensures that the **Quantity** in the **[Order Details]** table is positive (greater than 0), enforcing valid order quantities.

3. chk_Discount

Ensures that the **Discount** in the **[Order Details]** table is between 0 and 1 (inclusive), representing a valid discount range (0% to 100%).



```
-- Order Details
ALTER TABLE [Order Details]
ADD CONSTRAINT chk_UnitPrice CHECK (UnitPrice >= 0);

ALTER TABLE [Order Details]
ADD CONSTRAINT chk_Quantity CHECK (Quantity > 0);

ALTER TABLE [Order Details]
ADD CONSTRAINT chk_Discount CHECK (Discount >= 0 and Discount <= 1);
```

4. chk_UnitPrice_Product

Ensures that the **UnitPrice** in the **Products** table is non-negative, maintaining valid pricing for products.

5. chk_UnitsInStock

Ensures that **UnitsInStock** in the **Products** table is non-negative, preventing negative stock levels.

6. chk_UnitsOnOrder

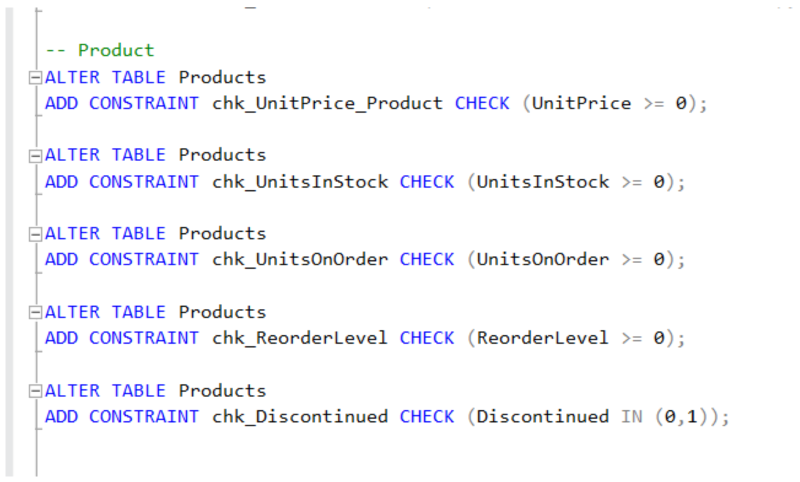
Ensures that **UnitsOnOrder** in the **Products** table is non-negative, enforcing valid order quantities.

7. chk_ReorderLevel

Ensures that **ReorderLevel** in the **Products** table is non-negative, maintaining valid stock reorder thresholds.

8. chk_Discontinued

Ensures that the **Discontinued** flag in the **Products** table is either 0 or 1, representing a valid binary state.



```
-- Product
ALTER TABLE Products
ADD CONSTRAINT chk_UnitPrice_Product CHECK (UnitPrice >= 0);

ALTER TABLE Products
ADD CONSTRAINT chk_UnitsInStock CHECK (UnitsInStock >= 0);

ALTER TABLE Products
ADD CONSTRAINT chk_UnitsOnOrder CHECK (UnitsOnOrder >= 0);

ALTER TABLE Products
ADD CONSTRAINT chk_ReorderLevel CHECK (ReorderLevel >= 0);

ALTER TABLE Products
ADD CONSTRAINT chk_Discontinued CHECK (Discontinued IN (0,1));
```

9. chk_Fright

Ensures that the **Freight** cost in the **Orders** table is non-negative, preventing invalid shipping costs.

10.uniq_ComapnyName (Suppliers)

Ensures that **CompanyName** in the **Suppliers** table is unique, preventing duplicate supplier names.

11.chk_HireDate

Ensures that **HireDate** in the **Employees** table is not in the future (less than or equal to current date), maintaining realistic hire dates.

12.chk_BirthDate

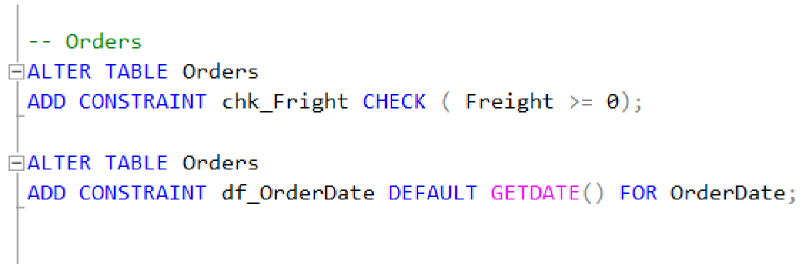
Ensures that **BirthDate** in the **Employees** table is not in the future and is before **HireDate**, enforcing logical age and employment timelines.

13. uniq_CategoryName

Ensures that **CategoryName** in the **Categories** table is unique, prevent duplicate category names

14. df_OrderDate

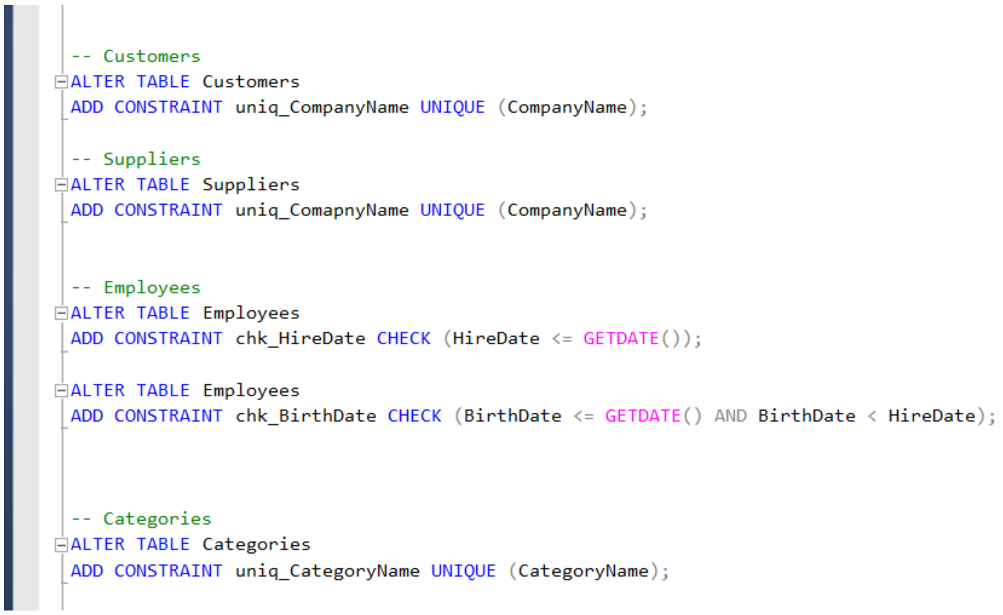
Brief: Sets the default value of **OrderDate** in the **Orders** table to the current date (`GETDATE()`) if not specified, ensuring a timestamp for new orders.



```sql
-- Orders
ALTER TABLE Orders
ADD CONSTRAINT chk_Fright CHECK ( Freight >= 0);

ALTER TABLE Orders
ADD CONSTRAINT df_OrderDate DEFAULT GETDATE() FOR OrderDate;
```

15.uniq_CompanyName (Customers)

Brief: Ensures that **CompanyName** in the **Customers** table is unique, preventing duplicate customer company names.



```sql
-- Customers
ALTER TABLE Customers
ADD CONSTRAINT uniq_CompanyName UNIQUE (CompanyName);

-- Suppliers
ALTER TABLE Suppliers
ADD CONSTRAINT uniq_ComapnyName UNIQUE (CompanyName);


-- Employees
ALTER TABLE Employees
ADD CONSTRAINT chk_HireDate CHECK (HireDate <= GETDATE());

ALTER TABLE Employees
ADD CONSTRAINT chk_BirthDate CHECK (BirthDate <= GETDATE() AND BirthDate < HireDate);



-- Categories
ALTER TABLE Categories
ADD CONSTRAINT uniq_CategoryName UNIQUE (CategoryName);
```

5. Indexes

An **Index** in a database **improves query performance** by allowing **faster data retrieval** from a table. It works like a book's index, making it quicker to find specific rows **without scanning the entire table**.

1. idx_Orders_CustomerID

A non-clustered index on **CustomerID** in the **Orders** table to improve query performance for customer-specific order lookups.

2. idx_Orders_OrderDate

A non-clustered index on **OrderDate** in the **Orders** table to enhance performance for date-based order queries.

3. idx_OrderDetails_ProductID

A non-clustered index on **ProductID** in the **[Order Details]** table to optimize product-specific order detail retrieval.

4. idx_Products_CategoryID

A non-clustered index on **CategoryID** in the **Products** table to speed up category-based product queries.

5. idx_Customers_Country

A non-clustered index on **Country** in the **Customers** table to improve performance for country-specific customer searches.