

Implement programmability objects with SQL

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/1-introduction>

Introduction

Completed

SQL Server provides several programmability objects that help you encapsulate logic, improve code reusability, and enforce business rules within your database. These objects—[views](#), [stored procedures](#), [functions](#), and [triggers](#)—each serve distinct purposes and offer unique capabilities for database development.

Scenario

You're a database developer at a growing e-commerce company. Your team manages a SQL Server database that handles customer orders, inventory, and reporting. As the application grows more complex, you notice:

- Developers write the same `JOIN` queries repeatedly across different applications
- Business logic is scattered throughout application code, making it hard to maintain
- Some data modifications need automatic validation and logging
- Complex calculations appear in multiple queries, leading to inconsistencies

You decide to create specific SQL Server objects to centralize logic, improve maintainability, and enhance security across your database applications.

What you'll learn

In this module, you'll explore the core programmability objects in SQL Server:

- **Views** - Virtual tables that simplify data access and provide security boundaries
- **Stored procedures** - Precompiled T-SQL code blocks for complex operations and data modifications

- **Scalar functions** - Reusable calculations that return single values
- **Table-valued functions** - Functions that return result sets for use in queries
- **Triggers** - Automatic responses to data modifications or database events

You'll also learn decision criteria for choosing the right programmability object based on your specific requirements.

2. Create views

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/2-create-views>

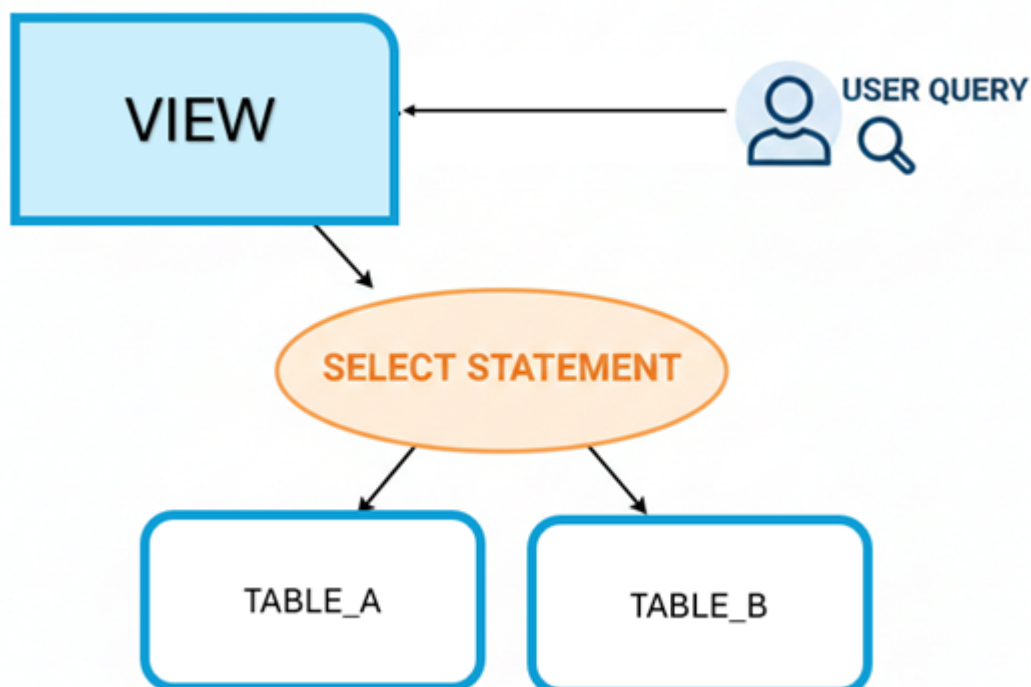
Create views

Completed

Views simplify how you access and present data in SQL Server. As a SQL developer, you create views to encapsulate complex queries, provide security boundaries, and present data in a format that matches your application's needs.

Understand views in SQL Server

A **view** is a virtual table based on a `SELECT` statement. Unlike physical tables, views don't store data themselves. Instead, they retrieve data from underlying tables each time you query them.



With views, you can hide the complexity of `JOIN` clauses, calculations, and filters from application code. For example, if your application frequently needs customer orders with product details, you can create a view that uses `JOIN` across the Customers, Orders, and Products tables. Your application then queries a single view instead of writing the same complex `JOIN` repeatedly.

Views also provide a security layer. You can grant users access to specific columns through a view while restricting access to the underlying tables. This approach lets you expose only the data users need without giving them full table access.

Create a basic view

You create a view using the `CREATE VIEW` statement followed by a `SELECT` query. The view definition determines what data appears when you query the view.

Here's a simple view that combines customer and order information:

```
CREATE VIEW Sales.CustomerOrders
AS
SELECT
    c.CustomerID,
    c.CustomerName,
    c.Email,
    o.OrderID,
    o.OrderDate,
    o.TotalAmount
FROM Sales.Customers c
INNER JOIN Sales.Orders o ON c.CustomerID = o.CustomerID;
```

After creating this view, you can query it like any table:

```
SELECT * FROM Sales.CustomerOrders
WHERE CustomerName = 'Contoso Ltd';
```

The view executes the underlying `SELECT` statement and returns results as if they came from a single table. This simplification makes your application code cleaner and easier to maintain.

You can also create views with calculated columns. The following view adds a column that categorizes orders by size:

```
CREATE VIEW Sales.OrderSummary
AS
SELECT
    OrderID,
    CustomerID,
    OrderDate,
    TotalAmount,
    CASE
        WHEN TotalAmount < 100 THEN 'Small'
        WHEN TotalAmount < 1000 THEN 'Medium'
```

```
        ELSE 'Large'
    END AS OrderSize
FROM Sales.Orders;
```

This view handles the categorization logic in one place. Every query against OrderSummary gets the same calculation without duplicating the `CASE` expression.

Apply design considerations

When designing views, consider how they'll be used and maintained. Well-designed views balance simplicity, performance, and security.

Specify column names explicitly instead of using `SELECT *`. Explicit columns make views more maintainable and prevent unexpected results when underlying tables change. If someone adds a column to a base table, your view definition stays consistent:

```
CREATE VIEW Sales.ActiveCustomers
AS
SELECT
    CustomerID,
    CustomerName,
    Email,
    Phone
FROM Sales.Customers
WHERE IsActive = 1;
```

Use the `WITH CHECK OPTION` clause when views will handle data modifications. This option ensures that `INSERT` and `UPDATE` statements through the view only affect rows visible in the view:

```
CREATE VIEW Sales.HighValueOrders
AS
SELECT
    OrderID,
    CustomerID,
    OrderDate,
    TotalAmount
FROM Sales.Orders
WHERE TotalAmount > 1000
WITH CHECK OPTION;
```

With this option, you can't insert an order with TotalAmount of 500 through the HighValueOrders view. The database rejects the operation because the new row wouldn't meet the view's `WHERE` condition.

Keep view definitions focused on a specific purpose. A view that tries to serve multiple purposes often becomes difficult to optimize and understand. Create separate views for different use cases rather than building one complex view.

Determine when to use views

Views excel at specific scenarios where their characteristics provide clear benefits. Understanding these scenarios helps you choose the right tool for each situation.

Use views to simplify complex queries that multiple applications or users need to execute. Instead of requiring everyone to understand the complexity of joining five tables with specific conditions, create a view once. This centralization also means you can optimize or modify the logic in one place.

Consider views when you need to restrict data access at the column or row level. A view can expose only the columns appropriate for a role while hiding sensitive information. Combined with appropriate permissions, views let you grant access to specific data without exposing entire tables.

Views work well for presenting data in different formats for different purposes. You might have a *Products* table with technical details, but your reporting team needs a simplified version with calculated fields. Create a view that transforms and aggregates the data appropriately.

At the same time, recognize when other objects serve better. For performance-critical queries that always return the same results, indexed views (materialized views) store the result set physically. For complex calculations that accept parameters, user-defined functions provide more flexibility. For data modification logic, stored procedures offer better control.

When you need to encapsulate reusable query logic without parameters, and you want to present data in a simplified way, views are your solution. They bridge the gap between the physical database structure and the logical view of data your applications need.

3. Create stored procedures

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/3-create-stored-procedures>

Create stored procedures

Completed

Stored procedures are one of the most powerful tools in SQL Server for encapsulating business logic and improving application performance. When you create stored procedures, you build reusable code blocks that execute on the server, reducing network traffic and centralizing data access logic.

Understand stored procedures

A stored procedure is a compiled collection of T-SQL statements that SQL Server stores and executes as a single unit. Unlike unplanned queries that you send to the server each time, stored procedures are precompiled and optimized, which means they run faster on subsequent executions.

You use stored procedures to encapsulate complex business logic, enforce data validation rules, and control how applications interact with your database. For example, instead of allowing direct table access, you can create stored procedures that validate input, apply business rules, and log changes before modifying data.

The performance benefits come from query plan caching. With unplanned queries, SQL Server must parse and optimize each query every time. With stored procedures, the execution plan is cached after the first run, reducing overhead for repeated operations.

Create basic stored procedures

Creating a stored procedure starts with the `CREATE PROCEDURE` statement followed by your T-SQL logic. You specify the procedure name using a schema-qualified identifier, which improves clarity and performance.

```
CREATE PROCEDURE dbo.GetCustomerOrders
AS
BEGIN
    SET NOCOUNT ON;

    SELECT
        OrderID,
        CustomerID,
        OrderDate,
        TotalAmount
    FROM dbo.Orders
    ORDER BY OrderDate DESC;
END
```

The `SET NOCOUNT ON` statement prevents the message about the number of rows affected from being sent to the client. This reduces network traffic and improves performance, especially when the procedure executes multiple statements.

When you create procedures, use the `BEGIN` and `END` keywords to clearly define the procedure body. This makes your code more readable and helps prevent errors when adding or modifying logic later.

Work with parameters

Parameters make stored procedures flexible and reusable. You define input parameters to accept values from the calling application, and output parameters to return values back to the caller.

Input parameters use the @ symbol followed by a parameter name and data type. You can provide default values to make parameters optional:

```
CREATE PROCEDURE dbo.GetCustomerOrdersByDate
    @CustomerID int,
    @StartDate datetime = NULL,
    @EndDate datetime = NULL
AS
BEGIN
    SET NOCOUNT ON;

    SELECT
        OrderID,
        OrderDate,
        TotalAmount
    FROM dbo.Orders
    WHERE CustomerID = @CustomerID
        AND (@StartDate IS NULL OR OrderDate >= @StartDate)
        AND (@EndDate IS NULL OR OrderDate <= @EndDate)
    ORDER BY OrderDate DESC;
END
```

Output parameters let you return values to the calling application. You define them using the **OUTPUT** keyword:

```
CREATE PROCEDURE dbo.CalculateOrderTotal
    @OrderID int,
    @TotalAmount decimal(10,2) OUTPUT
AS
BEGIN
    SET NOCOUNT ON;

    SELECT @TotalAmount = SUM(Quantity * UnitPrice)
    FROM dbo.OrderDetails
    WHERE OrderID = @OrderID;

    RETURN 0;
END
```

When you call a procedure with output parameters, you must declare a variable to receive the value and use the **OUTPUT** keyword in the **EXECUTE** statement.

Implement error handling

Robust stored procedures include error handling to manage unexpected conditions and maintain data integrity. You implement error handling using **TRY...CATCH** blocks, which work similarly to exception handling in other programming languages.

```
CREATE PROCEDURE dbo.InsertCustomerOrder
    @CustomerID int,
```

```

    @OrderDate datetime,
    @TotalAmount decimal(10,2)
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        BEGIN TRANSACTION;

        -- Validate customer exists
        IF NOT EXISTS (SELECT 1 FROM dbo.Customers WHERE CustomerID = @CustomerID)
        BEGIN
            RAISERROR('Customer does not exist.', 16, 1);
        END

        -- Insert order
        INSERT INTO dbo.Orders (CustomerID, OrderDate, TotalAmount)
        VALUES (@CustomerID, @OrderDate, @TotalAmount);

        COMMIT TRANSACTION;
        RETURN 0;
    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRANSACTION;

        DECLARE @ErrorMessage nvarchar(4000) = ERROR_MESSAGE();
        DECLARE @ErrorSeverity int = ERROR_SEVERITY();
        DECLARE @ErrorState int = ERROR_STATE();

        RAISERROR(@ErrorMessage, @ErrorSeverity, @ErrorState);
        RETURN -1;
    END CATCH
END

```

The `TRY` block contains your main logic, while the `CATCH` block handles any errors that occur. You can use system functions like `ERROR_MESSAGE()`, `ERROR_SEVERITY()`, and `ERROR_STATE()` to capture error details and pass them to the calling application.

Always check `@@TRANCOUNT` before rolling back transactions in the `CATCH` block. This prevents errors if the transaction already completed or was never started.

Apply best practices

Following established best practices when you create stored procedures ensures they're maintainable, secure, and performant.

Use schema-qualified names

Use schema-qualified names for all objects. This eliminates ambiguity and improves performance by avoiding schema resolution overhead:

```
-- Good
SELECT * FROM dbo.Orders

-- Avoid
SELECT * FROM Orders
```

Implement parameter validation

Implement parameter validation at the start of your procedure. Fail fast when inputs are invalid rather than processing bad data:

```
IF @CustomerID IS NULL OR @CustomerID <= 0
BEGIN
    RAISERROR('CustomerID must be a positive integer.', 16, 1);
    RETURN -1;
END
```

Avoid `SELECT *`

Avoid `SELECT *` in production code. Explicitly list columns to prevent issues when table structures change and to improve query performance:

```
-- Good
SELECT OrderID, CustomerID, OrderDate FROM dbo.Orders

-- Avoid
SELECT * FROM dbo.Orders
```

Use meaningful names

Use meaningful names that describe what the procedure does. Include a verb that indicates the operation (Get, Insert, Update, Delete, Calculate):

```
CREATE PROCEDURE dbo.GetActiveCustomersByRegion
CREATE PROCEDURE dbo.UpdateCustomerAddress
CREATE PROCEDURE dbo.DeleteExpiredOrders
```

Avoid the `sp_` prefix

Don't use the `sp_` prefix for your stored procedures. SQL Server reserves this prefix for system procedures stored in the `master` database. When you name a procedure with `sp_`, SQL Server first searches `master` before checking the current database, adding unnecessary overhead:

```
-- Good
CREATE PROCEDURE dbo.GetCustomerOrders
```

```
-- Avoid  
CREATE PROCEDURE dbo.sp_GetCustomerOrders
```

Building on these practices helps you create stored procedures that your team can understand, maintain, and trust to perform reliably in production environments.

4. Create scalar functions

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/4-create-scalar-functions>

Create scalar functions

Completed

[Scalar functions](#) are essential tools in SQL Server that allow you to encapsulate reusable logic and return a single value. You can use them directly in `SELECT` statements, `WHERE` clauses, and other [T-SQL](#) expressions, making your queries more maintainable and your code more modular.

Understand scalar function fundamentals

A scalar function accepts zero or more parameters and returns a single value of a specified data type. Unlike stored procedures, scalar functions can be embedded directly in SQL expressions wherever you would use a column or variable.

The key characteristics of scalar functions include their ability to accept input parameters, perform calculations or transformations, and return exactly one value. You define the return data type explicitly in the function definition, which SQL Server validates at creation time.

When you create a scalar function, you're creating a reusable piece of logic that other developers can call throughout the database. This promotes code reuse and helps maintain consistency across your applications.

Define scalar function syntax

To create a scalar function, you use the `CREATE FUNCTION` statement with specific syntax components. The basic structure includes the function name, parameters, return type, and function body.

Here's the fundamental syntax pattern:

```

CREATE FUNCTION schema_name.function_name
(
    @parameter1 datatype,
    @parameter2 datatype
)
RETURNS return_datatype
AS
BEGIN
    -- Function logic here
    RETURN @result
END

```

The `RETURNS` clause specifies the data type of the single value the function returns. Within the `BEGIN...END` block, you write your T-SQL logic and use a `RETURN` statement to send back the result.

For example, you can create a simple function that calculates sales tax:

```

CREATE FUNCTION dbo.CalculateSalesTax
(
    @Amount DECIMAL(10,2),
    @TaxRate DECIMAL(5,4)
)
RETURNS DECIMAL(10,2)
AS
BEGIN
    DECLARE @TaxAmount DECIMAL(10,2)
    SET @TaxAmount = @Amount * @TaxRate
    RETURN @TaxAmount
END

```

This function accepts two parameters and returns the calculated tax amount. You can use this function in any `SELECT` statement.

Implement scalar functions with business logic

Scalar functions excel at encapsulating business rules and calculations that you need to apply consistently across your database. With scalar functions, you centralize logic that might otherwise be duplicated in multiple queries or application code.

Consider a scenario where you need to calculate employee tenure in years. You create a scalar function that accepts a hire date and returns the number of complete years:

```

CREATE FUNCTION dbo.GetEmployeeTenure
(
    @HireDate DATE
)
RETURNS INT
AS
BEGIN

```

```
DECLARE @Tenure INT
SET @Tenure = DATEDIFF(YEAR, @HireDate, GETDATE())
RETURN @Tenure
END
```

You can use this function in queries to display tenure information:

```
SELECT
    EmployeeName,
    HireDate,
    dbo.GetEmployeeTenure(HireDate) AS YearsOfService
FROM Employees
WHERE dbo.GetEmployeeTenure(HireDate) >= 5
```

This approach ensures consistent tenure calculation throughout your database. If the business rules change, you modify the function once rather than updating multiple queries.

Note

This function uses `GETDATE()`, which makes it non-deterministic. Non-deterministic functions can't be used in indexed views or indexes on computed columns. For scenarios requiring determinism, pass the current date as a parameter instead.

Apply best practices for scalar functions

When you create scalar functions, several best practices help ensure optimal performance and maintainability. Understanding these practices helps you avoid common pitfalls and create efficient, reliable functions.

First, keep your scalar functions deterministic whenever possible. A deterministic function always returns the same result given the same input parameters. Functions that reference system date/time functions or tables are non-deterministic and may prevent certain query optimizations.

Also, avoid side effects in your functions. Scalar functions shouldn't modify database state or have dependencies on external resources. This restriction exists because SQL Server may execute functions multiple times or in different orders than you expect.

Lastly, be mindful of performance implications. When you use a scalar function in a `WHERE` clause or `SELECT` list with large tables, SQL Server may execute the function for every row. This can significantly impact query performance. For such scenarios, consider inline table-valued functions as an alternative.

Here's an example of a well-designed scalar function that follows these practices:

```
CREATE FUNCTION dbo.FormatPhoneNumber
(
    @PhoneNumber VARCHAR(10)
```

```

)
RETURNS VARCHAR(14)
AS
BEGIN
    DECLARE @FormattedNumber VARCHAR(14)

    IF LEN(@PhoneNumber) = 10
        SET @FormattedNumber = '(' + SUBSTRING(@PhoneNumber, 1, 3) + ') ' +
                                SUBSTRING(@PhoneNumber, 4, 3) + '-' +
                                SUBSTRING(@PhoneNumber, 7, 4)

    ELSE
        SET @FormattedNumber = @PhoneNumber

    RETURN @FormattedNumber
END

```

This function is deterministic, has no side effects, and performs a straightforward transformation. It handles invalid input gracefully by returning the original value when the phone number doesn't match the expected format.

Modify and manage scalar functions

After you create a scalar function, you can modify its definition using the `ALTER FUNCTION` statement. The `ALTER FUNCTION` syntax mirrors `CREATE FUNCTION` but allows you to change the function without dropping and recreating it, which preserves permissions and dependencies.

```

ALTER FUNCTION dbo.CalculateSalesTax
(
    @Amount DECIMAL(10,2),
    @TaxRate DECIMAL(5,4)
)
RETURNS DECIMAL(10,2)
AS
BEGIN
    DECLARE @TaxAmount DECIMAL(10,2)
    -- Updated logic with rounding
    SET @TaxAmount = ROUND(@Amount * @TaxRate, 2)
    RETURN @TaxAmount
END

```

To remove a scalar function, you use the `DROP FUNCTION` statement:

```

DROP FUNCTION IF EXISTS dbo.CalculateSalesTax

```

The `IF EXISTS` clause prevents errors if the function doesn't exist, which is useful in deployment scripts. Before dropping a function, verify that no other database objects depend on it by checking system views like `sys.sql_expression_dependencies`.

5. Create table-valued functions

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/5-create-table-valued-functions>

Create table-valued functions

Completed

Table-valued functions let you encapsulate complex query logic into reusable components that return result sets. You can call these functions directly in queries, just like tables or views, making your code more modular and maintainable.

When you build database applications, you often need to retrieve filtered or calculated data sets based on input parameters. Table-valued functions solve this problem by packaging query logic into functions that accept parameters and return tables. Unlike stored procedures, you can use table-valued functions in `JOIN` clauses and `SELECT` statements, giving you the flexibility to treat function results as data sources.

Understand table-valued function types

SQL Server provides two types of table-valued functions, each suited for different scenarios.

Inline table-valued function

Contains a single `SELECT` statement and returns results directly. With inline functions, you don't define the table structure—SQL Server infers it from your `SELECT` statement. The query optimizer treats inline table-valued functions like views with parameters, often producing better execution plans.

Multi-statement table-valued function

Uses a `BEGIN . . . END` block and explicitly declares the structure of the returned table. This type gives you more control when you need to execute multiple statements, perform complex calculations, or build the result set iteratively. However, this flexibility comes with a performance trade-off, as the optimizer treats these functions differently.

The choice between these functions depends on your specific requirements. For simple queries with parameters, inline functions provide better performance. When you need procedural logic or multiple steps to build your result set, multi-statement functions become necessary.

Create inline table-valued functions

Inline table-valued functions offer a concise way to parameterize queries. You define them with a single RETURN statement followed by a SELECT query.

The following example shows an inline function that retrieves orders for a specific customer:

```
CREATE FUNCTION dbo.GetCustomerOrders
(
    @CustomerID INT
)
RETURNS TABLE
AS
RETURN
(
    SELECT
        OrderID,
        OrderDate,
        TotalAmount,
        Status
    FROM Sales.Orders
    WHERE CustomerID = @CustomerID
);
```

You can now use this function in queries just like a table:

```
SELECT OrderID, OrderDate, TotalAmount
FROM dbo.GetCustomerOrders(1001)
WHERE OrderDate >= '2024-01-01';
```

The function accepts the customer ID parameter and returns only that customer's orders. You can further filter, JOIN, or aggregate the results as needed. This approach keeps your main query clean while encapsulating the customer filtering logic.

Create multi-statement table-valued functions

Multi-statement table-valued functions provide more flexibility when you need to perform multiple operations to build your result set.

Consider a function that calculates product sales summaries with multiple aggregations:

```
CREATE FUNCTION dbo.GetProductSalesSummary
(
    @StartDate DATE,
    @EndDate DATE
)
RETURNS @SalesSummary TABLE
(
    ProductID INT,
```

```

        ProductName NVARCHAR(100),
        TotalQuantity INT,
        TotalRevenue DECIMAL(18,2),
        AveragePrice DECIMAL(18,2)
    )
AS
BEGIN
    INSERT INTO @SalesSummary
    SELECT
        p.ProductID,
        p.ProductName,
        SUM(od.Quantity) AS TotalQuantity,
        SUM(od.Quantity * od.UnitPrice) AS TotalRevenue,
        AVG(od.UnitPrice) AS AveragePrice
    FROM Production.Products p
    INNER JOIN Sales.OrderDetails od ON p.ProductID = od.ProductID
    INNER JOIN Sales.Orders o ON od.OrderID = o.OrderID
    WHERE o.OrderDate BETWEEN @StartDate AND @EndDate
    GROUP BY p.ProductID, p.ProductName;

    RETURN;
END;

```

Notice how you explicitly declare the table variable `@SalesSummary` with specific columns and data types. The function body inserts data into this table variable, then returns it. This structure allows you to add additional processing logic, error handling, or conditional statements as needed.

Use table-valued functions in queries

Table-valued functions integrate seamlessly into your queries, enabling powerful data retrieval patterns.

You can join function results with other tables:

```

SELECT
    c.CustomerName,
    s.ProductName,
    s.TotalRevenue
FROM Customers c
CROSS APPLY dbo.GetProductSalesSummary('2024-01-01', '2024-12-31') s
WHERE s.TotalRevenue > 10000
ORDER BY s.TotalRevenue DESC;

```

The `CROSS APPLY` operator calls the function for each row from the Customers table, though in this example, the function parameters are constants. When you pass column values as parameters, `CROSS APPLY` becomes particularly useful:

```

SELECT
    c.CustomerName,
    o.OrderID,

```

```
o.TotalAmount
FROM Customers c
CROSS APPLY dbo.GetCustomerOrders(c.CustomerID) o
WHERE o.Status = 'Completed';
```

This query retrieves all completed orders for each customer, demonstrating how table-valued functions enable row-by-row processing within your queries. The function acts as a correlated subquery but with better readability and reusability.

For inline table-valued functions that don't require row-by-row evaluation, you can use **INNER JOIN** syntax as well:

```
SELECT
    c.CustomerName,
    o.OrderDate,
    o.TotalAmount
FROM Customers c
INNER JOIN dbo.GetCustomerOrders(c.CustomerID) o ON 1=1
WHERE YEAR(o.OrderDate) = 2024;
```

With these techniques, you can build complex queries from simpler, tested function components, improving both code maintainability and development efficiency.

6. Create triggers

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/6-create-triggers>

Create triggers

Completed

Triggers are special stored procedures that automatically execute when specific events occur in your database. You define triggers to maintain data integrity, enforce business rules, and automate database operations without requiring application-level code.

Understand trigger fundamentals

A trigger responds to data modification or schema changes in your database. When you create a trigger, you specify the event that activates it and the actions it performs.

Triggers execute automatically. Unlike stored procedures that you call explicitly, triggers fire in response to **INSERT**, **UPDATE**, **DELETE**, or DDL statements. This automatic execution makes them powerful for enforcing rules that must apply consistently across all data modifications.

SQL Server supports two main categories of triggers: **DML (Data Manipulation Language)** triggers and **DDL (Data Definition Language)** triggers. DML triggers respond to changes in table data, while DDL triggers respond to schema changes like `CREATE`, `ALTER`, or `DROP` statements.

Create DML triggers for data modifications

DML triggers monitor and respond to data changes in tables or views. You define them as either **AFTER** triggers or **INSTEAD OF** triggers.

AFTER triggers execute after the triggering statement completes. The database first performs the data modification, then runs the trigger code. You use AFTER triggers to validate changes, update related tables, or log modifications:

```
CREATE TRIGGER tr_UpdateInventory
ON Sales.OrderDetails
AFTER INSERT
AS
BEGIN
    UPDATE Inventory.Products
    SET QuantityInStock = QuantityInStock - i.Quantity
    FROM Inventory.Products p
    INNER JOIN inserted i ON p.ProductID = i.ProductID;
END;
```

INSTEAD OF triggers replace the original data modification statement. The trigger code executes instead of the `INSERT`, `UPDATE`, or `DELETE` operation. You use INSTEAD OF triggers to modify views that normally wouldn't accept direct modifications or to implement complex business logic:

```
CREATE TRIGGER tr_UpdateOrderView
ON Sales.OrderSummaryView
INSTEAD OF UPDATE
AS
BEGIN
    UPDATE Sales.Orders
    SET OrderStatus = i.OrderStatus,
        ModifiedDate = GETDATE()
    FROM Sales.Orders o
    INNER JOIN inserted i ON o.OrderID = i.OrderID;
END;
```

With DML triggers, you access the **inserted** and **deleted** pseudo-tables. These temporary tables store copies of the affected rows. `INSERT` operations populate the **inserted** table, `DELETE` operations populate the **deleted** table, and `UPDATE` operations populate both tables with old values in **deleted** and new values in **inserted**.

Implement triggers for specific events

You specify which data modification events activate your trigger. A single trigger can respond to multiple events by combining `INSERT`, `UPDATE`, and `DELETE` in the trigger definition.

For precise control, you create separate triggers for each operation. This approach simplifies your code and makes your triggers easier to maintain:

```
CREATE TRIGGER tr_LogPriceChanges
ON Products.Catalog
AFTER UPDATE
AS
BEGIN
    IF UPDATE(Price)
    BEGIN
        INSERT INTO Audit.PriceHistory (ProductID, OldPrice, NewPrice, ChangeDate)
        SELECT d.ProductID, d.Price, i.Price, GETDATE()
        FROM deleted d
        INNER JOIN inserted i ON d.ProductID = i.ProductID
        WHERE d.Price <> i.Price;
    END;
END;
```

At the same time, you might combine events when the same logic applies to multiple operations. For example, you could create a single audit trigger that responds to `INSERT`, `UPDATE`, and `DELETE`:

```
CREATE TRIGGER tr_AuditEmployeeChanges
ON HR.Employees
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
    DECLARE @Operation NVARCHAR(10);

    IF EXISTS (SELECT * FROM inserted) AND NOT EXISTS (SELECT * FROM deleted)
        SET @Operation = 'INSERT';
    ELSE IF EXISTS (SELECT * FROM inserted) AND EXISTS (SELECT * FROM deleted)
        SET @Operation = 'UPDATE';
    ELSE
        SET @Operation = 'DELETE';

    INSERT INTO Audit.EmployeeLog (EmployeeID, Operation, ChangeDate)
    SELECT COALESCE(i.EmployeeID, d.EmployeeID), @Operation, GETDATE()
    FROM inserted i
    FULL OUTER JOIN deleted d ON i.EmployeeID = d.EmployeeID;
END;
```

The `UPDATE()` function helps you determine which columns changed. You check specific columns to avoid unnecessary processing when only certain fields matter for your business logic.

Apply trigger best practices

Triggers affect database performance because they execute with every qualifying operation. You write efficient trigger code to minimize impact on transaction throughput.

Keep your trigger logic focused and minimal. Execute only essential operations within the trigger body. For complex or time-consuming operations, consider logging the event details and processing them asynchronously through a separate job:

```
CREATE TRIGGER tr_QueueLargeOrders
ON Sales.Orders
AFTER INSERT
AS
BEGIN
    INSERT INTO Processing.OrderQueue (OrderID, TotalAmount, QueuedDate)
    SELECT OrderID, TotalAmount, GETDATE()
    FROM inserted
    WHERE TotalAmount > 10000;
END;
```

Avoid recursive operations where a trigger modification causes the same trigger to fire again. Set the `RECURSIVE_TRIGGERS` database option appropriately and design your triggers to prevent endless loops.

Handle errors properly within triggers. Transaction behavior depends on your error handling. If a trigger encounters an error and you don't handle it, SQL Server rolls back both the trigger and the original statement:

```
CREATE TRIGGER tr_ValidateOrderDate
ON Sales.Orders
AFTER INSERT, UPDATE
AS
BEGIN
    IF EXISTS (SELECT * FROM inserted WHERE OrderDate > GETDATE())
    BEGIN
        THROW 50001, 'Order date cannot be in the future', 1;
    END;
END;
```

Document your triggers thoroughly. Other developers need to understand why triggers exist and what they do, since they execute invisibly during normal database operations.

Now that you understand how to create and implement triggers, you're ready to explore how they integrate with other programmability objects to build comprehensive database solutions.

7. Choose when to use each option

Choose when to use each option

Completed

SQL Server programmability objects provide different ways to encapsulate and reuse logic in your database. Each object type—[views](#), [stored procedures](#), [functions](#), and [triggers](#)—serves distinct purposes and offers unique capabilities.

Compare options

The following table summarizes key capabilities and limitations of each object type:

Capability	Views	Stored Procedures	Functions	Triggers
Accept parameters	No	Yes	Yes	No
Modify data	Limited	Yes	No	Yes
Return result sets	Yes	Yes	Yes (TVFs)	No
Use in <code>SELECT / JOIN</code>	Yes	No	Yes	No
Transaction control	No	Yes	No	Yes
Automatic execution	No	No	No	Yes
Execution plan caching	No	Yes	Varies	Yes

Views can modify data only when changes affect a single base table. Inline table-valued functions benefit from plan caching because the optimizer expands them directly into the query plan. Multi-statement TVFs and scalar functions are treated as "black boxes"—the optimizer can't see inside them, which often leads to inaccurate row estimates and suboptimal plans.

Choose based on your requirements

The right programmability object depends on what you need to accomplish. Use this decision framework to guide your selection:

Choose views when you need to:

- Simplify access to complex joins or commonly filtered data
- Provide a security layer by controlling column and row visibility
- Create a stable interface to underlying tables that might change
- Present data without accepting parameters or modifying values

Choose stored procedures when you need to:

- Execute complex business logic with multiple statements
- Modify data across multiple tables in a single transaction
- Accept input parameters and return output parameters or result sets
- Implement error handling and transaction control

Choose functions when you need to:

- Perform reusable calculations that return values for use in queries
- Return parameterized result sets (table-valued functions)
- Embed logic directly in `SELECT`, `WHERE`, or `JOIN` clauses
- Ensure deterministic results for indexing (for specific function types)

Choose triggers when you need to:

- Automatically respond to data modification events
- Enforce complex business rules that extend beyond constraints
- Maintain audit logs of data changes
- Synchronize related data across tables automatically

Apply decision scenarios

Consider these common scenarios and the recommended approach for each:

Scenario	Recommended Object	Why
Simplify a 5-table join that multiple reports use	View	Encapsulates complexity; no parameters needed
Process an order: validate stock, insert order, update inventory	Stored procedure	Multiple modifications in a transaction
Calculate shipping cost based on weight and destination	Scalar function	Reusable calculation in queries
Return all orders for a customer within a date range	Table-valued function	Parameterized result set usable in <code>JOIN</code>
Log all changes to the <code>Salary</code> column	Trigger	Automatic, transparent audit trail
Provide read-only access to employee data without SSN	View	Security layer hiding sensitive columns

Avoid common mistakes

When choosing programmability objects, watch for these pitfalls:

- **Using scalar functions in WHERE clauses on large tables**— The function executes for every row, degrading performance. Consider inline table-valued functions or rewriting the logic.
- **Creating triggers for logic that stored procedures handle better**— Triggers execute implicitly and can be hard to debug. Use them only when automatic execution is essential.
- **Building complex views that nest other views**— Deeply nested views become difficult to optimize and maintain. Keep view definitions focused and shallow.
- **Choosing stored procedures when a function would integrate better**— If you need the result in a SELECT statement, a function provides cleaner syntax than EXEC with temp tables.

With this understanding of each programmability object's strengths and trade-offs, you can select the appropriate tool for your database design and implementation tasks.

8. Exercise: Implement programmability objects in SQL Server

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/8-exercise-implement-programmability-objects>

Exercise: Implement programmability objects in SQL Server

Completed

Now it's your chance to practice implementing programmability objects in SQL Server.

In this exercise, you learn how to create views, stored procedures, scalar functions, table-valued functions, and triggers to encapsulate logic and improve code reusability.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

9. Knowledge check

Knowledge check

Completed

10. Summary

<https://learn.microsoft.com/en-us/training/modules/implement-programmability-objects/10-summary>

Summary

Completed

In this module, you explored SQL Server programmability objects and learned how to use them effectively in your database solutions.

You learned how to:

- **Create views** to simplify data access, hide complexity, and provide security boundaries by exposing only specific columns or rows from underlying tables.
- **Build stored procedures** to encapsulate complex business logic, handle transactions, implement error handling, and create reusable data modification operations.
- **Develop scalar functions** to create reusable calculations that return single values, making your queries more readable and consistent.
- **Implement table-valued functions** using both inline and multi-statement approaches to return result sets that can be used in `FROM` clauses like tables.
- **Configure triggers** to automatically respond to DML operations (`INSERT` , `UPDATE` , `DELETE`) or DDL events, enabling audit logging, data validation, and enforcement of complex business rules.
- **Choose the right programmability object** based on your specific requirements, considering factors like whether you need to modify data, return single values or result sets, or respond automatically to database events.

Next steps

Now that you understand SQL Server programmability objects, consider:

- Implementing views in your existing databases to simplify complex queries
- Converting repetitive application logic into stored procedures
- Creating functions to standardize calculations across your organization
- Adding audit triggers to track data changes in critical tables

Learn more

- [CREATE VIEW \(Transact-SQL\)](#)
- [CREATE PROCEDURE \(Transact-SQL\)](#)
- [CREATE FUNCTION \(Transact-SQL\)](#)
- [CREATE TRIGGER \(Transact-SQL\)](#)