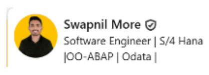
Swapnil More | erswapnilmore@gmail.com

DEFINING "UNEXPLORED" IN ABAP A JOURNEY BEYOND THE BASICS

Interview Guide



Swapnil More
Software Engineer | S/4 Hana
|OO-ABAP | Odata |

1. Core Modern ABAP Syntax Enhancements

The evolution of ABAP has introduced fundamental syntax improvements that have transformed daily coding practices. These enhancements promote conciseness, readability, and efficiency, aligning ABAP more closely with modern programming language paradigms.

Inline Data Declarations: Streamlining Variable Definitions

One of the most impactful modernizations is inline data declaration. This feature eliminates the traditional requirement for separate DATA statements at the beginning of a code block or program. Instead, variables can be declared directly at the point of their first use, often allowing the compiler to implicitly infer the data type.

For example, the classic approach of declaring a variable and then using it in a SELECT statement: DATA lv_value TYPE i. SELECT SINGLE col1 INTO lv_value FROM table WHERE col2 = 'X'.

can now be streamlined into a single line: SELECT SINGLE col1 INTO @DATA(lv_value) FROM table WHERE col2 = 'X'.

This change significantly reduces boilerplate code and enhances readability by keeping the variable's declaration close to its usage, making the code flow more intuitive. The compiler's ability to infer data types further simplifies the development process, allowing developers to focus more on logic and less on explicit type definitions. This compiler intelligence is a key factor in the increased conciseness and reduced verbosity of modern ABAP.

Constructor Expressions: Concise Object and Value Creation

Constructor expressions represent a cornerstone of modern ABAP's shift towards a more functional programming style. These expressions return a result, enabling the creation of values, objects, and complex data structures in a single, concise line of code.

- **VALUE #(): Structure and Internal Table Construction** The VALUE #() operator is exceptionally versatile, used for creating and populating various data types, including structures, internal tables, and ranges. The hash symbol (#) often allows for implicit type inference, deriving the type from the target variable or an earlier declaration.

For instance, to construct a structure: DATA(line) = VALUE tadir(obj_name = 'ZCL_CS' object = 'CLAS').

Or to populate an internal table with multiple rows: lt_itab = VALUE tt_user((user_id = 'U_PATJAG' user_name = 'Jagdish P') (user_id = 'U_DOEJOH' user_name = 'John Doe')).

It can also construct ranges, simplifying the definition of selection criteria.

- **NEW #(): Object Instantiation** The NEW #() operator provides a modern alternative to the traditional CREATE OBJECT statement for instantiating classes.

Instead of: DATA: lo_abap TYPE REF TO lcl_abap. CREATE OBJECT lo_abap. The new syntax is: DATA(lo_abap) = NEW lcl_abap( ).

Parameters for the class constructor can be passed directly within the parentheses.

- **COND #(): Conditional Value Assignment (IF/ELSE Replacement)** The COND #() operator streamlines conditional logic, serving as a powerful replacement for verbose IF/ELSE constructs. It evaluates multiple conditions (WHEN) and assigns a corresponding value (THEN), with an ELSE clause for fallback.

A nested IF/ELSE structure like:

Code snippet

```
DATA: lv_text(30).

IF lv_vehicle = '01' AND lv_type = 'C'.

   lv_text = 'Toyota'.

ELSE.

IF lv_vehicle ='02' AND lv_type = 'C'.

  lv_text = 'Chevy'

ELSE.

IF lv_vehicle ='03' AND lv_type = 'C'.

  lv_text = 'Range Rover'.

 ..

ENDIF.
```

Can be concisely written as: DATA(lv_text) = COND text30(WHEN lv_vehicle ='01' AND lv_type = 'C' THEN 'Toyota' WHEN lv_vehicle ='02' AND lv_type = 'C' THEN 'Chevy' WHEN lv_vehicle ='03' AND lv_type = 'C' THEN 'Range Rover').

- **SWITCH #(): Single-Variable Conditional Assignment (CASE Replacement)** Similar to COND, SWITCH #() is designed for conditional assignments based on a single variable, effectively replacing the traditional CASE statement.

For example, a CASE statement to determine the day of the week:

Code snippet

```
data: l_indicator like scal-indicator, l_day(10) type c.

call function 'DATE_COMPUTE_DAY' exporting date = p_date importing day = l_indicator.

case l_indicator. when 1. l_day = 'Monday'.... else. Raise exception type zcx_day_problem. endcase.
```

Becomes: DATA(L_DAY) = SWITCH char10(l_indicator WHEN 1 THEN 'Monday' WHEN 2 THEN 'Tuesday'... ELSE THROW zcx_day_problem( )).

Notably, SWITCH also allows for throwing exceptions directly, aligning with modern error handling.

- **CONV #(): Type Conversion** The CONV #() operator simplifies type conversions, eliminating the need for intermediate helper variables when converting data types, such as when passing data to method parameters.

Example: DATA(xstr) = cl_abap_codepage=>convert_to(source = CONV #(text))

- **REF #(): Obtaining Data References** This operator provides a modern and more concise syntax for obtaining data references, replacing the older GET REFERENCE OF statement.

Instead of: GET REFERENCE OF it_flights INTO dref. The new syntax is: DATA(dref) = REF #(it_flights)

- **CORRESPONDING #(): Structure and Table Assignment** The CORRESPONDING #() operator is a powerful replacement for MOVE-CORRESPONDING, offering flexible data assignment between structures and internal tables, even when field names differ. It supports mapping fields with different names, excluding specific fields (

EXCEPT), and merging content using the BASE addition.

Simple assignment: ls_new = CORRESPONDING #(ls_base)

With mapping and exclusion: lt_new = CORRESPONDING #(lt_base MAPPING b4 = a1 EXCEPT a2)

Merging structures: ls_data = CORRESPONDING #(BASE(ls_data) ls_bkpf).

- **FILTER #(): Internal Table Filtering** This operator efficiently moves rows from one internal table to another based on a specified WHERE condition or EXCEPT WHERE condition.

Example: DATA(messages_en) = FILTER #(messages WHERE sprsl = 'E')

- **REDUCE #(): Iterative Value Creation** The REDUCE #() operator is used to create a single result value of a specified type through one or more iterations (loops), making it highly useful for aggregations and complex calculations.

Example for summing numbers: DATA(lv_sum) = REDUCE #(INIT s = 0 FOR i = 1 UNTIL i > 100 NEXT s = s + i)

- **EXACT #(): Lossless Assignment** The EXACT #() operator ensures a lossless assignment between variables, particularly when their types differ. If data cannot be converted without loss, it triggers exceptions, promoting robust data handling.
- **XSDBOOL: Boolean Type Conversion** Introduced in ABAP 7.4, XSDBOOL is a built-in function that returns an ABAP_BOOL type parameter, resolving issues encountered with the older BOOLC function which returned a string rather than a true boolean. This ensures correct evaluation of boolean expressions.

Example: IF xsdbool(1 = 2) = ABAP_FALSE. WRITE: / 'YES'. ELSE. WRITE: / 'NO'. ENDIF.

The extensive array of new constructor expressions, coupled with table expressions and FOR loops, signifies a profound shift in ABAP towards more functional and declarative programming styles. This change moves the focus from explicitly detailing "how to do it" to concisely stating "what to achieve." This paradigm shift requires a different way of thinking for ABAP developers, and while it may present an initial learning curve, the resulting conciseness and readability are substantial benefits once mastered.

Table Expressions: Efficient Internal Table Access

Table expressions offer a modern, concise, and powerful way to read entries from internal tables, largely replacing the traditional READ TABLE statement.

- **itab: Direct Access** The itab syntax allows direct access to table lines using an index or key, making internal table operations more intuitive and compact.

Reading by index: ls_flight = it_flights[1].

Reading by key: ls_flight = it_flights.

A critical distinction from the READ TABLE statement is the behavior on failure. Unlike READ TABLE, which sets sy-subrc to indicate success or failure, the itab expression raises a CX_SY_ITAB_LINE_NOT_FOUND exception if the requested entry is not found. This fundamental change in error handling for internal table reads necessitates the use of

TRY...CATCH blocks or the OPTIONAL keyword to manage potential failures. This aligns ABAP more closely with modern programming language error handling conventions, promoting more robust and explicit error management.

Furthermore, itab functions as an expression, meaning it can be used at any operand position, which was not possible with the READ TABLE statement. This includes direct field access (

data(plane_type) = it_flights[connid = '0026']-planetype.) or direct inclusion within IF conditions. For sorted tables with unique keys, the system implicitly leverages binary search for optimized performance.

- **LINE_EXISTS and LINE_INDEX** While the provided information highlights the itab expression, LINE_EXISTS and LINE_INDEX are also part of modern table expressions, offering ways to check for the existence of a line or retrieve its index without necessarily reading the full line. (Detailed explanations for these specific functions are not available in the provided material, but they are integral to the broader concept of table expressions.)

FOR Loop Expressions: Modern Iteration for Internal Tables

Also known as Iteration Expressions, FOR loops provide a more concise and powerful way to iterate over and transform internal tables, serving as a modern alternative to traditional LOOP AT and APPEND statements.

A basic iteration to transfer data: lt_new_flights = VALUE #(FOR ls_flight IN lt_flights...)

FOR loops support WHERE conditions, similar to LOOP AT, but without the mandatory parentheses. They can also integrate with the

LET expression, allowing the definition of variables for calculations or transformations directly within the loop, leading to more concise and readable code. Furthermore, multiple

FOR iterations can be nested, enabling complex scenarios involving related data from multiple internal tables, akin to nested LOOP statements.

String Processing Enhancements: && and String Templates

Modern ABAP introduces significant enhancements for string manipulation, improving readability and efficiency.

- **Chaining Operator (&&)** The chaining operator && concatenates two or more character-like operands into a single string. This operator largely replaces the older

CONCATENATE statement, offering a more streamlined syntax. Example: lv_result = v_var1 && v_var2 && v_var3.

- **String Templates (|...|)** String templates provide a flexible and powerful way to create character strings from literal texts, embedded expressions, and control characters (such as \n for line feed, \r for carriage return, and \t for tabulator). A string template is defined by enclosing its content within pipe (

|) symbols. Example: WRITE: |The total is { lv_total } for { lv_name }.|

Embedded expressions, enclosed in curly brackets { expression } within the template, allow direct inclusion of variables or more complex expressions, including formatting options like ALPHA (which replaces older conversion function modules), WIDTH, ALIGN, DATE, and TIME. This capability significantly simplifies string construction and formatting, reducing the need for multiple

WRITE TO or CONCATENATE statements.

Old vs. New ABAP Syntax Comparison

Category	Old Syntax Example	New Syntax Example (ABAP 7.4+)	Benefit
Data Declaration	DATA lv_value TYPE i. SELECT SINGLE col1 INTO lv_value FROM table WHERE col2 = 'X'.	SELECT SINGLE col1 INTO @DATA(lv_value) FROM table WHERE col2 = 'X'.	Reduces boilerplate, type inference, improved readability
Internal Table Read	READ TABLE lt_table INTO ls_row WITH KEY col1 = 'X'. IF sy-subrc = 0. WRITE: ls_row-col2. ENDIF.	DATA(value) = lt_table[col1 = 'X']-col2. WRITE: value.	More concise, direct access, raises exception on failure (modern error handling)
String Concatenation	CONCATENATE v_var1 v_var2 v_var3 INTO lv_result.	lv_result = v_var1 && v_var2 && v_var3.	Simpler, more intuitive operator
Object Creation	DATA: lo_abap TYPE REF TO lcl_abap. CREATE OBJECT lo_abap.	DATA(lo_abap) = NEW lcl_abap( ).	Concise, inline instantiation
Conditional Logic (IF/ELSE)	IF lv_vehicle = '01' AND lv_type = 'C'. lv_text = 'Toyota'. ELSEIF... ENDIF.	DATA(lv_text) = COND text30(WHEN lv_vehicle ='01' AND lv_type = 'C' THEN 'Toyota'...).	More compact, single assignment for result
Conditional Logic (CASE)	CASE l_indicator. WHEN 1. l_day = 'Monday'.... ENDCASE.	DATA(L_DAY) = SWITCH char10(l_indicator WHEN 1 THEN 'Monday'... ELSE THROW zcx_day_problem( )).	More compact, allows direct exception throwing

Category	Old Syntax Example	New Syntax Example (ABAP 7.4+)	Benefit
Reference Assignment	GET REFERENCE OF it_flights INTO dref.	DATA(dref) = REF #(it_flights).	More concise syntax
Structure/Table Assignment	MOVE-CORRESPONDING ls_base TO ls_new.	ls_new = CORRESPONDING #(ls_base).	Flexible mapping, merging, exclusion options
Internal Table Loop/Populate	LOOP AT lt_old INTO wa. APPEND INITIAL LINE TO lt_new. lt_new-col1 = wa-col1. ENDLOOP.	DATA(lt_new) = VALUE #(FOR wa IN lt_old (col1 = wa-col1 col2 = wa-col2)).	Highly compact, declarative iteration and transformation

Table 2.2: Overview of Constructor Operators

Operator	Brief Description	Simple Example
VALUE #()	Creates and populates data objects (structures, internal tables, ranges)	DATA(line) = VALUE tadir(obj_name = 'ZCL_CS' object = 'CLAS').
NEW #()	Creates an instance of a class	DATA(lo_abap) = NEW lcl_abap( ).
COND #()	Conditional value assignment, similar to IF/ELSE	DATA(lv_text) = COND text30(WHEN lv_vehicle = '01' THEN 'Toyota' ELSE 'Other').
SWITCH #()	Conditional value assignment for a single variable, similar to CASE	DATA(L_DAY) = SWITCH char10(l_indicator WHEN 1 THEN 'Monday' ELSE 'Unknown').
CONV #()	Performs type conversion	DATA(xstr) = cl_abap_codepage=>convert_to(source = CONV #(text)).
REF #()	Obtains a data reference	DATA(dref) = REF #(it_flights).
CORRESPONDING #()	Assigns data between structures/tables based on corresponding names, with mapping options	ls_new = CORRESPONDING #(ls_base MAPPING b4 = a1).
FILTER #()	Filters rows from an internal table based on a condition	DATA(messages_en) = FILTER #(messages WHERE sprsl = 'E').
REDUCE #()	Creates a result through iterative processing (looping)	DATA(lv_sum) = REDUCE #(INIT s = 0 FOR i = 1 UNTIL i > 100 NEXT s = s + i).
EXACT #()	Performs lossless assignment, raising exceptions on data loss	DATA(num) = EXACT #('123.456').

Operator	Brief Description	Simple Example
XSDBOOL	Converts a boolean expression to ABAP_BOOL type	IF xsdbool(1 = 2) = ABAP_FALSE.... ENDIF.

2. Advanced Open SQL and CDS Features

The evolution of ABAP is inextricably linked with the advancements in SAP's database technology, particularly SAP HANA. This has led to significant enhancements in Open SQL and the pervasive role of Core Data Services (CDS) Views, fundamentally altering how ABAP interacts with the database and enabling efficient "code pushdown."

Enhanced SELECT Statements: The New Frontier of Database Access

Open SQL, traditionally a simpler SQL dialect, has gained substantial power and expressiveness, incorporating features common in full-fledged relational database systems. This transformation is driven by a strategic imperative from SAP to perform more logic directly at the database layer, thereby reducing overhead from looping and merging data at the application server level.

Key enhancements include:

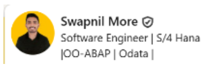
- **Host Variables (@) and Inline Declaration:** The @ escape character is now mandatory to explicitly distinguish ABAP host variables (variables, structures, or internal tables) within an SQL query from database fields. The SELECT field list is comma-separated, and inline declarations for target tables are standard, such as INTO TABLE @DATA(result). The

INTO clause has also been repositioned to the end of the statement.

- **Column Aliases:** Developers can now use aliases for column names, allowing for custom naming of fields in the resulting table, enhancing readability. For example:

SELECT carrid AS carrier, connid AS connection FROM sflight....

- **Literals as Additional Columns:** Fixed literal values can be included as new columns directly within the SELECT query.
- **Arithmetic Operations:** Open SQL now supports direct arithmetic operations (+, -, *, /, DIVISION) within the SELECT statement, pushing computation to the database.
- **String Operations:** Various built-in string functions like CONCAT, LENGTH, LEFT, LOWER, UPPER, SUBSTRING, and REPLACE can be used directly in SELECT queries.
- **Date Functions:** Integrated date-related functions such as dats_is_valid, dats_days_between, dats_add_days, and dats_add_months are available within Open SQL.
- **CASE Expressions:** Both simple and complex CASE statements can be embedded within SELECT queries for conditional logic, enabling more complex data transformations at the database level.
- **Aggregate Queries:** With HANA's code pushdown capabilities, aggregate queries are highly recommended. Expressions used in the field list can also be used in GROUP BY... HAVING clauses, providing powerful analytical capabilities directly within the database.
- **Client Handling:** The USING CLIENT clause allows specifying a particular client for the query, offering fine-grained control over data retrieval.



- **Case-Insensitive Search:** The upper() function can be used directly in the WHERE clause for efficient case-insensitive searches, eliminating the need for application-level string manipulation.
- **Arithmetic Operations in WHERE Clause:** Operations like +, -, and * are now permitted between fields directly within the WHERE clause.
- **Method Calls within WHERE/SELECT/ON:** Functional methods can be directly invoked within the WHERE clause, SELECT list, and ON conditions, enabling more dynamic and integrated data filtering and selection.
- **IS INITIAL/IS NOT INITIAL:** These conditions can be used within WHERE clauses (as of ABAP 1809) for straightforward filtering based on a field's initial value.
- **Code Completion:** The enhanced syntax, particularly SELECT FROM <tablename> FIELDS…, provides improved code completion for fields, boosting development efficiency.

Advanced Joins and COALESCE

The effective use of joins in Open SQL has become increasingly important, especially with SAP HANA, often replacing the less performant FOR ALL ENTRIES clause for many scenarios.

- **INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN:** Open SQL now effectively leverages these join types for optimal performance on HANA, allowing for complex data retrieval across multiple tables.
- **COALESCE:** This function is particularly useful with outer joins, allowing developers to specify a default value for fields where no matching record is found in the joined table, preventing null values in the result set.
- **FIELDS <tablename>-*:** This keyword allows selecting all fields from a specific table within a join, simplifying the field list definition for complex queries.

UNION in SELECT Statements

The UNION addition, available from ABAP 7.50, enables the combination of result sets from two SELECT statements.

UNION ALL retains duplicate rows, while UNION DISTINCT (the default if not specified) returns only unique records. It is important to note that

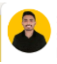
UNION queries cannot use FOR ALL ENTRIES but can incorporate aggregate functions.

SELECT FROM INTERNAL_TABLE

As of ABAP 7.52, developers can directly query an internal table using SELECT FROM @internal_table AS alias. This feature blurs the line between database and application layer processing, enabling SQL-like operations on in-memory data.

Common Table Expressions (WITH Clause)

Introduced from ABAP 7.51+, the WITH clause allows the definition of named temporary datasets (e.g., +cities) within a single query. These Common Table Expressions (CTEs) can be referenced multiple times within the same query, improving readability and modularity for complex SQL statements.



SQL Window Functions: Analytical Power

SQL Window Functions represent a significant leap in Open SQL's analytical capabilities. They allow for partitioning a query's result set and applying various functions (aggregates, ranking, value functions) to these partitions, all while retaining individual records in the output. This contrasts with standard

GROUP BY clauses, which only provide aggregate values without individual records.

- **OVER Keyword:** Initiates a window expression.
- **PARTITION BY:** Defines how data is grouped into logical partitions, similar to GROUP BY but for windowing operations.
- **ORDER BY (within OVER):** Creates a frame within the current window, crucial for applying ranking functions and analyzing trends based on a sorted sequence.
- **Frames (ROWS BETWEEN A AND B):** Defines the specific range of rows within a partition for calculation. Options include UNBOUNDED PRECEDING, CURRENT ROW, and n PRECEDING/FOLLOWING.
- **Ranking Functions:**
  - ROW_NUMBER(): Assigns a unique integer row number to each row within a partition.
  - RANK(): Assigns a rank, skipping numbers for ties.
  - DENSE_RANK(): Assigns a rank without skipping numbers for ties, providing continuous ranking.
  - NTILE(n): Splits the window into 'n' number of frames or groups.
- **Value Functions:**
  - FIRST_VALUE(), LAST_VALUE(): Provide the first and last values within a window based on the sort sequence.
  - LEAD(), LAG(): Provide the value from the next or previous record, respectively, useful for variance calculations.

Path Expressions

Path expressions are a powerful feature, particularly when working with CDS Views, allowing navigation through associations directly within Open SQL statements. They can be used in the SELECT list, FROM clause, WHERE clause, HAVING clause, and with filter conditions.

ABAP Managed Database Procedures (AMDP)

AMDP allows developers to implement database procedures and functions (e.g., HANA SQLScript) directly within ABAP classes. This enables the pushdown of complex, performance-critical logic to the database layer, where it can execute more efficiently.

Core Data Services (CDS Views): The Foundation for Modern Data Models

Introduced with NetWeaver 7.4/7.5, CDS Views provide a semantic data model layer that is central to modern ABAP development, especially in S/4HANA environments. They enable "code pushdown" to

HANA by defining data models and queries that are optimized for in-memory processing. CDS Views offer enhanced Open SQL capabilities and serve as the foundation for many new ABAP features, providing a unified and performant data access layer.

The extensive list of Open SQL enhancements, including arithmetic, string, and date functions, CASE expressions, advanced joins, UNION, CTEs, and window functions, underscores a strategic shift by SAP to execute more logic directly at the database layer. This "code pushdown" is not merely a performance optimization; it represents a fundamental architectural principle for ABAP on HANA. Developers must internalize this principle to write efficient and scalable applications, as relying on traditional ABAP processing for data manipulation after a

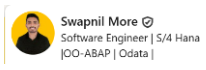
SELECT is now often an anti-pattern.

Open SQL's evolution into a richer query language, incorporating features previously found only in full-fledged SQL databases, signifies a trend towards reducing the need for complex ABAP logic post-database retrieval. This evolution will likely continue, further enhancing the database's role in application logic. The tight integration and interdependence of ABAP, Open SQL, and CDS Views are also apparent. CDS Views act as a foundational layer, exposing optimized data models and logic that Open SQL can then consume more effectively. This symbiotic relationship is critical for modern ABAP development, as developers cannot fully leverage modern Open SQL without understanding CDS, and vice-versa.

Open SQL Enhancements (7.4+)

Feature Category	Specific Enhancement	Brief Description	Example (brief)	ABAP Release
Basic Syntax	Host Variables (@)	Explicitly denotes ABAP variables in SQL	SELECT col INTO @DATA(var) FROM tab.	7.40
	Comma-Separated Field List	Fields in SELECT list separated by commas	SELECT col1, col2 FROM tab.	7.40
	INTO Clause at End	Target variable/table specified at end of SELECT	SELECT col FROM tab INTO @DATA(var).	7.40
	Column Aliases (AS)	Custom naming of fields in result set	SELECT col1 AS new_name FROM tab.	7.40
	Literals as Columns	Include fixed values as new columns	SELECT col, 'Fixed' AS literal_col FROM tab.	7.40
Data Manipulation	Arithmetic Operations	Perform +, -, *, / directly in SELECT	SELECT price + tax AS total FROM tab.	7.40
	String Operations	Use functions like CONCAT, LENGTH, SUBSTRING	SELECT CONCAT(f1, f2) AS result FROM tab.	7.40

Feature Category	Specific Enhancement	Brief Description	Example (brief)	ABAP Release
	Date Functions	Use functions like dats_add_days, dats_days_between	SELECT dats_add_days(d, 5) AS new_date FROM tab.	7.40
	CASE Expressions	Conditional logic within SELECT statement	SELECT CASE WHEN val > 10 THEN 'High' ELSE 'Low' END FROM tab.	7.40
	Aggregate Queries	SUM, AVG, MIN, MAX, COUNT with GROUP BY	SELECT SUM(price) FROM tab GROUP BY cat.	7.40
	USING CLIENT	Specify a client for the query	SELECT * FROM tab USING CLIENT '100'.	7.40
	Case-Insensitive Search	Use upper() function in WHERE clause	SELECT col FROM tab WHERE upper(col) LIKE 'TEXT%'.	7.40
	Arithmetic in WHERE	Perform +, -, * between fields in WHERE	SELECT col FROM tab WHERE (f1 - f2) > 100.	7.40
	Method Calls in SQL	Invoke functional methods in WHERE, SELECT, ON	SELECT col FROM tab WHERE col = @(meth()).	7.40
	IS INITIAL/IS NOT INITIAL	Check for initial values in WHERE clause	SELECT col FROM tab WHERE col IS NOT INITIAL.	1809
Advanced Querying	INNER JOIN, LEFT/RIGHT OUTER JOIN	Combine data from multiple tables efficiently	SELECT a~col, b~col FROM t1 AS a INNER JOIN t2 AS b ON a~id = b~id.	7.40
	COALESCE	Return first non-null expression	SELECT COALESCE(col1, 'N/A') FROM tab.	7.40
	UNION	Combine result sets of two SELECT statements	SELECT c1 FROM t1 UNION ALL SELECT c1 FROM t2.	7.50
	SELECT FROM INTERNAL_TABLE	Query an internal table directly with SQL syntax	SELECT col FROM @lt_itab AS itab.	7.52
	Common Table Expressions (WITH)	Define temporary named result sets within a query	WITH +cte AS (SELECT col FROM tab) SELECT * FROM +cte.	7.51
	SQL Window Functions	Perform analytical calculations over a set of rows	SELECT col, AVG(col) OVER(PARTITION BY grp) FROM tab.	7.40



Feature Category	Specific Enhancement	Brief Description	Example (brief)	ABAP Release
	Path Expressions	Navigate through CDS associations in SQL	SELECT entity.association.field FROM entity.	7.40

SQL Window Functions Overview

Function Type	Function Name	Description	Key Usage/Distinction
Ranking	ROW_NUMBER()	Assigns a unique, sequential integer to each row within its partition.	Provides a distinct row number for each record, regardless of ties.
	RANK()	Assigns a rank to each row within its partition, with gaps for ties.	If multiple rows have the same value, they get the same rank, and the next rank number is skipped.
	DENSE_RANK()	Assigns a rank to each row within its partition, without gaps for ties.	Similar to RANK but provides continuous ranking, even with ties.
	NTILE(n)	Divides the rows within a partition into 'n' groups and assigns a group number.	Useful for splitting data into a specified number of buckets.
Value	FIRST_VALUE()	Returns the value of the expression from the first row in the window frame.	Useful for retrieving a specific value from the beginning of a sorted window.
	LAST_VALUE()	Returns the value of the expression from the last row in the window frame.	Requires explicit frame definition (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) to get the true last value of the partition.
	LEAD()	Returns the value of the expression from a subsequent row within the partition.	Useful for comparing a row's value with a value from a "future" row.
	LAG()	Returns the value of the expression from a preceding row within the partition.	Useful for comparing a row's value with a value from a "past" row.

Function Type	Function Name	Description	Key Usage/Distinction
Aggregate	AVG()	Calculates the average of a numeric column within the window frame.	Performs average calculation while retaining individual records.
	SUM()	Calculates the sum of a numeric column within the window frame.	Performs sum calculation while retaining individual records.
	MIN()	Returns the minimum value of a column within the window frame.	Performs minimum calculation while retaining individual records.
	MAX()	Returns the maximum value of a column within the window frame.	Performs maximum calculation while retaining individual records.
	COUNT(*)	Counts the number of rows within the window frame.	Performs count calculation while retaining individual records.

3. Other Unexplored and Niche ABAP Capabilities

Beyond the core syntax and Open SQL enhancements, ABAP offers additional modern and less common features that contribute to better code structure, flexibility, and quality assurance. These capabilities, while perhaps not daily staples for every developer, are vital for advanced scenarios and maintaining robust applications.

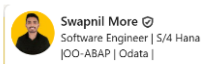
Enumerated Types (ENUM): Structured Constants

Enumerated types, or ENUMs, provide a structured way to define a data type that consists of a set of named constant values. This concept, familiar in many other programming languages, brings enhanced readability and type safety to ABAP constants.

An enumerated type can be declared directly in ABAP using TYPES: BEGIN OF ENUM... END OF ENUM or defined within CDS (Core Data Services). Once defined, it can be used for declaring enumerated variables, which can then be assigned specific named constants. For example:

Code snippet

```
TYPES: BEGIN OF ENUM colors,
  red,
  blue,
  green,
END OF ENUM colors.
DATA color1 TYPE colors VALUE red.
```



```
DATA(color3) = VALUE colors( ).

color3 = green.
```

The primary benefits of using enumerated types include better organization of constants, streamlined code, improved readability by replacing "magic numbers" or raw values with meaningful names, and enhanced debugging capabilities as the constant values are clearly labeled. This provides a compile-time type safety for constants, ensuring that only valid, predefined values are assigned to enumerated variables, which contributes to more robust code.

CASE TYPE OF: Dynamic Type Checks and RTTI

The CASE TYPE OF statement represents a specialized form of the CASE control structure that allows for branching program flow based on the runtime type of an object reference variable. This capability leverages Runtime Type Information (RTTI), a component of ABAP's Runtime Type Services (RTTS).

Unlike a regular CASE statement, which compares a single variable against a list of static values ,

CASE TYPE OF performs a dynamic type check. This is particularly crucial for handling polymorphic objects in object-oriented programming, where the exact type of an object is not known until runtime. While other languages like C# offer a dynamic type that bypasses static type checking and defers errors to runtime , ABAP's

CASE TYPE OF provides a controlled mechanism for explicitly checking types at runtime. This contributes to safer dynamic programming by allowing developers to define specific behaviors for different object types, thereby improving flexibility and maintainability in complex object-oriented designs. This demonstrates ABAP's dual approach to type management: strong static typing where possible, and robust dynamic typing where necessary.

Dynamic Programming: Field Symbols and Data References

Dynamic programming in ABAP enables programs to handle data objects whose type or structure is not determined until runtime. This is essential for building generic frameworks and flexible applications that can adapt to varying data requirements.

- **Field Symbols (FIELD-SYMBOLS):** These act as symbolic placeholders or pointers to data objects. They allow generic access and modification of data without requiring the static declaration of its type at design time.
- **Data References (DATA REF TO):** These are variables that point to other data objects. The REF #() constructor operator is the modern and concise way to obtain these references , replacing the older

GET REFERENCE OF statement.

- **Runtime Type Services (RTTS):** RTTS comprises Runtime Type Identification (RTTI), which allows querying type information at runtime, and Runtime Type Creation (RTTC), which enables the dynamic creation of new data types. These services are fundamental for implementing truly dynamic ABAP applications.

The combination of field symbols, data references, and RTTS provides the necessary tools for generic programming, framework development, and scenarios where data structures are dynamic, such as processing generic interfaces or handling user-defined fields.

ABAP Unit Tests: Ensuring Code Quality

ABAP Unit is an integrated testing framework within the ABAP development environment designed to automate unit tests. It is a critical component of modern software development practices, ensuring code integrity and reliability.

Developers can create and manage test classes within their development projects, regularly running these tests during the development cycle and prior to deployment. Developing test methods that cover both expected behavior and edge cases helps to establish robust development practices and facilitates continuous integration and delivery pipelines.

Classic Dynpro and Selection Screen Elements in a Modern Context

While modern ABAP development increasingly emphasizes Fiori and web-based user interfaces, traditional Dynpro (Transaction SE51) and Selection Screens (Transaction SE41) remain relevant for many existing SAP applications and specific niche use cases.

- **Dynpro:** Used to create the actual screens displayed to the user, including input/output fields, table controls, and push buttons.
- **Selection Screens:** Employed for defining user input parameters and selection criteria for reports.

These classic UI elements are still fully supported in traditional ABAP systems. However, it is important to note that Dynpro-related ABAP keywords and functionality are generally not available in ABAP for Cloud Development. Understanding these classic components remains crucial for maintaining and extending existing legacy codebases, even as new development shifts towards modern UI technologies.

The inclusion of modern, object-oriented concepts like ENUM and CASE TYPE OF alongside the continued relevance of classic UI elements such as Dynpro and Selection Screens highlights ABAP's unique position as a language that is rapidly modernizing while simultaneously supporting a vast legacy codebase. This necessitates that developers be proficient in both worlds: leveraging modern features for new development while possessing the knowledge to maintain and understand existing "classic" ABAP systems. This dual requirement contributes to the perceived "unexplored" areas within the language.

Furthermore, the emphasis on ENUM for compile-time type safety and CASE TYPE OF for controlled runtime type checking indicates ABAP's progression towards a more sophisticated type system. This system balances strictness with flexibility, enabling the creation of more robust and less error-prone dynamic programming solutions compared to older, less controlled methods.

4. Obsolete but Still Valid Syntax: What to Avoid and Why

A critical aspect of exploring ABAP syntax involves understanding not only what is new and recommended but also what is obsolete yet still functional. This section addresses the "unexplored" from the perspective of deprecated syntax, explaining why these elements are discouraged despite their continued validity, and outlining their modern, preferred alternatives.

Understanding Backward Compatibility and its Pitfalls

SAP's commitment to backward compatibility is a defining characteristic of ABAP. This means that older commands and syntax continue to function even after new, more modern alternatives are introduced. This approach is highly beneficial for customers, as it prevents breaking changes in their custom code following ABAP upgrades, ensuring system stability.

However, this strong backward compatibility carries a significant, often overlooked, consequence: it can inadvertently disincentivize developers from learning and adopting new, more efficient, and readable syntax. This phenomenon allows some developers to continue coding as if it were decades ago, leading to a stagnation in coding practices within parts of the ABAP community.

To counteract this inertia, SAP has implemented stricter syntax checks, particularly within the context of object-oriented programming. Many obsolete language elements are explicitly forbidden in the syntax of classes, meaning their use is restricted to code written outside of class definitions. This serves as a strategic measure to guide developers towards modern practices when engaging in new, class-based development. This restriction in the class context is a deliberate move by SAP to enforce modernization, effectively compelling developers to adopt the new syntax if they wish to write code in the recommended object-oriented style.

Common Obsolete Statements and Their Modern Replacements

Several common ABAP statements, while still valid, are now considered obsolete due to their limitations, potential for confusion, or simply because more efficient and readable alternatives exist.

- **OCCURS WITH HEADER LINE:**
    - **Obsolete:** This declaration creates both an internal table and a work area (header line) with the exact same name. This practice is widely discouraged due to its potential for confusion, reduced readability, and the implicit nature of operations that can lead to errors. It is explicitly forbidden in classes.
    - **Replacement:** The modern approach involves declaring internal tables without header lines (DATA lt_table TYPE STANDARD TABLE OF...) and using explicit work areas (LOOP AT lt_table INTO ls_line) or field symbols (LOOP AT lt_table ASSIGNING FIELD-SYMBOL(<fs_line>)) for processing individual table rows.
- **REFRESH vs. CLEAR/FREE:**
    - **Obsolete REFRESH:** This statement deletes all rows of an internal table, freeing up memory space except for the initial memory requirement. Its use is obsolete because the

CLEAR and FREE statements can achieve the same, or more precise, memory management.

-
    - **Replacement:**
        - CLEAR itab: Deletes all rows of the table body, similar to REFRESH.
        - FREE itab: Deletes all rows and releases the *entire* allocated memory space, including the initial memory requirement.
        - CLEAR itab: If the internal table has a header line, this clears the header line; for tables without a header line, it clears the table body.
- **MOVE, COMPUTE, ADD/SUBTRACT/MULTIPLY/DIVIDE (as statements):**

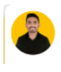
  - **Obsolete:** These traditional statements for assignments and arithmetic operations are considered outdated.
  - **Replacement:** Modern ABAP favors direct assignment (=) and arithmetic operators (+, -, *, /, and their compound assignment forms like +=, -=, *=, /=).
- **GET REFERENCE OF and MOVE-CORRESPONDING:**
  - **Obsolete:** These represent older syntax for obtaining data references and for moving data between structures or internal tables where field names correspond.
  - **Replacement:** The REF #() constructor operator is the modern way to obtain data references, and CORRESPONDING #() is the powerful, flexible replacement for MOVE-CORRESPONDING.
- **BINARY SEARCH:**
  - **Obsolete:** This addition to the READ TABLE statement requires the internal table to be sorted beforehand for correct results. If the table is not properly sorted, the results can be unpredictable, leading to subtle and hard-to-debug errors.
  - **Replacement:** Using SORTED or HASHED table types implicitly leverages optimized search algorithms (binary search for sorted tables with unique keys) without the explicit BINARY SEARCH addition. The new table expressions (

itab) also implicitly use binary search for sorted tables. There is a common misconception among some developers that new syntax or object-oriented programming is inherently slower, leading to resistance to adoption. However, empirical tests often show that modern syntax is "faster or similar" , and that the real performance bottlenecks are typically database interactions rather than ABAP processing itself.

- **FORM/FUNCTION MODULE vs. METHOD:**
  - **Obsolete:** While still functional, FORM routines and FUNCTION MODULEs represent older procedural modularization techniques.
  - **Replacement:** Modern ABAP strongly emphasizes object-oriented programming (OOP), recommending METHODs within classes for code modularization, reusability, and better representation of complex business processes. Function modules should primarily be reserved for specific scenarios like RFCs (Remote Function Calls) or update modules.
- **LIKE for Type Reference:**
  - **Obsolete in Classes:** While LIKE can generally refer to existing data objects , within classes and interfaces,

TYPE must be used to refer to ABAP Dictionary types.

  - **Replacement:** Use TYPE.
- **RANGES:**
  - **Obsolete:** This statement was used for declaring internal tables of type RANGE OF.



  - **Replacement:** The modern approach is to use DATA... TYPE RANGE OF... or to construct ranges using the VALUE #() constructor operator.

- **CATCH SYSTEM-EXCEPTIONS:**
  - **Obsolete:** This represents an older mechanism for handling system exceptions.
  - **Replacement:** Modern ABAP mandates the use of class-based exceptions (TRY...CATCH cx_sy_...), which provide a more robust and structured approach to error handling.

Impact on Code Readability, Maintainability, and Performance

The continued use of obsolete syntax often leads to code that is less readable, more verbose, and significantly harder to maintain. Modern replacements, by design, improve efficiency and readability, making programs easier to understand and debug. While older code may still execute, it may not fully leverage the performance optimizations available with SAP HANA, potentially leading to suboptimal application performance. Organizations must actively enforce modern coding standards to overcome the inertia created by backward compatibility and prevent the accumulation of technical debt.

Obsolete ABAP Statements and Modern Replacements

Obsolete Statement	Reason for Obsolescence	Modern Replacement(s)	ABAP Release (if applicable)
DATA... OCCURS... WITH HEADER LINE	Confusing, less readable, implicit operations, forbidden in classes	DATA lt_table TYPE STANDARD TABLE OF..., LOOP AT lt_table INTO ls_line, LOOP AT lt_table ASSIGNING FIELD-SYMBOL(<fs_line>)	7.40 (explicit header lines discouraged)
REFRESH itab	Less precise memory management, CLEAR and FREE offer better control	CLEAR itab, FREE itab	7.40 (obsolete due to CLEAR itab equivalence)
MOVE source TO target	Verbose, direct assignment is more concise	target = source	7.40
COMPUTE result = expr	Verbose, direct assignment with arithmetic operators is more concise	result = expr	7.40

Obsolete Statement	Reason for Obsolescence	Modern Replacement(s)	ABAP Release (if applicable)
ADD/SUBTRACT/MULTIPLY/DIVIDE (as statements)	Verbose, arithmetic operators are more concise	result = operand1 + operand2, result += operand	7.40
GET REFERENCE OF var INTO dref	Verbose, new constructor operator is more concise	DATA(dref) = REF #(var)	7.40
MOVE-CORRESPONDING s1 TO s2	Limited flexibility, new constructor operator offers more control	s2 = CORRESPONDING #(s1)	7.40
READ TABLE itab WITH KEY... BINARY SEARCH	Error-prone if table not sorted, implicit optimized search in modern types	DATA lt_table TYPE SORTED TABLE OF... WITH UNIQUE KEY..., lt_table[key_field = 'X']	7.40 (implicit binary search with table expressions)
FORM routine. / FUNCTION MODULE func.	Procedural, less aligned with modern OOP principles	METHOD method_name. (within classes)	7.00 (OOP emphasis)
DATA var LIKE dobj (in classes)	Less precise, TYPE is preferred for dictionary types in OO context	DATA var TYPE ddic_type	7.00 (stricter OO syntax)
RANGES s_range FOR field.	Verbose, modern declaration and construction methods available	DATA s_range TYPE RANGE OF field., s_range = VALUE #((sign = 'I' option = 'EQ' low = 'VAL'))	7.40
CATCH SYSTEM-EXCEPTIONS	Older, less structured exception handling	TRY... CATCH cx_sy_... (class-based exceptions)	7.00 (class-based exceptions)
REFRESH: s_bukrs.	CLEAR with square brackets or FREE is more explicit and versatile.	CLEAR: s_bukrs, s_bukrs.	7.40

Obsolete Statement	Reason for Obsolescence	Modern Replacement(s)	ABAP Release (if applicable)
LEAVE	Ambiguous, more specific LEAVE statements exist.	LEAVE PROGRAM., LEAVE TO TRANSACTION., LEAVE TO SCREEN., LEAVE SCREEN., LEAVE LIST-PROCESSING.	7.02

5. Best Practices for Modern ABAP Development

To effectively navigate the evolving landscape of ABAP and leverage its modern capabilities, developers must adopt a comprehensive set of best practices. These practices extend beyond mere syntax, encompassing coding standards, tool utilization, performance optimization, and architectural principles.

Adopting Clean ABAP Coding Standards

Clean code is paramount for creating software that is readable, understandable, maintainable, and testable. Embracing these principles ensures that ABAP applications are not only functional but also sustainable and scalable in the long term. Key guidelines for writing clean ABAP include:

- **Meaningful Naming:** Use descriptive variable, method, and class names that clearly convey their purpose. This reduces reliance on comments for basic understanding.
- **Consistent Styling:** Apply uniform coding styles across all projects to enhance readability and maintainability.
- **Purposeful Documentation:** Document complex logic with comments, but avoid using comments to explain poorly named variables or methods.
- **Simplicity and Minimizing Nesting:** Emphasize code simplicity and avoid overly complex or deeply nested IF statements, CASE WHEN THEN structures, and loops. Instead of embedding multiple exits within nested blocks, prefer early returns,

EXIT, or CONTINUE statements for a cleaner flow.

- **Proper Formatting:** Utilize consistent spacing and indentation to improve code readability and maintainability.
- **Avoid Magic Numbers and Hard-coded Constants:** Replace arbitrary numerical or string values with clearly named constants or enumerated types for better clarity and maintainability.

The emphasis on "Clean ABAP" represents a significant cultural shift within the ABAP community. It moves beyond merely creating functional code to prioritizing code quality, readability, and long-term sustainability. This is a critical evolution for an enterprise language with applications that often have extended lifespans.

Leveraging ABAP Development Tools (ADT, abapGit)

Modern ABAP development is significantly enhanced by sophisticated tooling.

- **ABAP Development Tools (ADT):** This Eclipse-based environment is indispensable for contemporary ABAP development. ADT provides advanced features such as sophisticated refactoring tools, real-time syntax checks, templates for common coding patterns, and integrated task management, all of which are not available in the Classic ABAP Workbench (SE80).
- **abapGit:** This plug-in facilitates the installation of ABAP cheat sheets and, more broadly, enables the management of ABAP code in Git repositories. This integration with Git supports modern development workflows, including version control, collaboration, and continuous integration.

The interplay of these tools, adherence to Clean ABAP standards, and mastery of new syntax are crucial for effective modern ABAP development. Adopting modern ABAP is not just about learning new keywords; it requires a holistic change in the development environment and coding philosophy. Without this integrated approach, the potential productivity gains and performance improvements may not be fully realized.

Effective Debugging Techniques

Mastering debugging is essential for any ABAP developer. Effective debugging increases efficiency and helps quickly identify and resolve issues.

- **Utilize ABAP Debugger:** Familiarize oneself with the various modes of the ABAP debugger, including Classic, New, and External Debugging.
- **Strategic Breakpoints:** Set breakpoints strategically to examine the code flow at critical points.
- **Watchpoints:** Analyze variable contents at runtime using watchpoints, which trigger a debugger stop when a variable's value changes.
- **Understanding Dynamic Behavior:** Debugging can be complex, especially with dynamic type checking and runtime object creation. A thorough understanding of how dynamic features behave is crucial for effective troubleshooting.

Performance Optimization Strategies

Performance optimization is integral to modern ABAP development, especially in SAP HANA environments. Poorly optimized code can severely impact application efficiency and user satisfaction.

- **Code Pushdown:** A core principle is to avoid unnecessary database hits by pushing down logic to the database layer. This is achieved by leveraging enhanced Open SQL capabilities, CDS Views, and ABAP Managed Database Procedures (AMDP) for in-memory processing.
- **Efficient Queries:** Construct SELECT queries effectively, utilizing advanced joins, aggregates, and window functions to retrieve and process data efficiently at the database level.
- **Parallel Processing:** Employ parallel processing techniques for computationally intensive calculations to distribute workload and improve response times.
- **Performance Trace Tools:** Utilize built-in performance trace tools to identify and analyze bottlenecks in code execution.

Security Considerations in ABAP Development

Security should be an inherent part of the development process, not an afterthought. Strengthening application defenses involves implementing best practices throughout the development lifecycle.

- **Data Encryption:** Ensure data encryption during transmission and storage to protect sensitive information.
- **Input Validation:** Validate all user input rigorously to safeguard against common vulnerabilities like injection attacks.
- **Authorization Management:** Regularly review and update authorizations and role assignments to adhere to the principle of least privilege.
- **Access Governance Platforms:** Tools like Pathlock Identity Security Platform can assist in identifying role conflicts, managing Segregation of Duties (SoD), and automating user access reviews for compliance.

Modularization and Object-Oriented Programming

SAP strongly emphasizes object-oriented programming (OOP) for developing complex business processes in modern ABAP.

- **Classes over Functions/Forms:** Developers are encouraged to use classes and methods for code modularization and reusability, reserving traditional FUNCTION MODULEs primarily for specific scenarios like RFCs or update modules.
- **Single-Responsibility Principle:** Methods should ideally adhere to the single-responsibility principle, meaning they should have only one reason to change. Complex logic should be delegated to other, more focused methods to improve maintainability.
- **Class-Based Exceptions:** Adopt modern class-based exceptions (TRY...CATCH cx_sy_...) for robust and structured error handling, moving away from obsolete system or classic exceptions.

Stay Updated with ABAP Cloud Concepts

The SAP landscape is increasingly moving towards cloud solutions and S/4HANA.

- **ABAP in Cloud:** Understanding the SAP Cloud Platform ABAP Environment and the ABAP RESTful Application Programming (RAP) model is crucial for developing modern, scalable, and cloud-compatible applications.
- **S/4HANA Migration:** Companies adopting SAP S/4HANA require developers proficient in the new ABAP syntax and modern development paradigms.

6. Conclusion: Embracing the Future of ABAP

The journey through "unexplored" ABAP syntax reveals a language that has undergone a profound evolution, driven primarily by the advent of SAP HANA and the imperative for enhanced performance, readability, and developer productivity. This report has highlighted that "unexplored syntax" encompasses not only powerful new features that remain underutilized but also older, obsolete constructs that, despite their continued functionality due to backward compatibility, should be actively avoided in new development.

Key Takeaways and the Value of Continuous Learning

Modern ABAP, particularly from releases 7.4 and 7.5 onwards, represents a significant leap forward. Features such as inline declarations, a rich set of constructor expressions, efficient table expressions, and advanced Open SQL capabilities—including window functions and Common Table Expressions—empower developers to write more concise, readable, and performant code. These advancements facilitate "code pushdown" to the database, a crucial architectural principle for maximizing performance in SAP HANA environments.

While SAP's commitment to backward compatibility ensures that older code continues to run, this also creates an inertia that developers must actively overcome to stay current. The restriction of many obsolete elements within the context of classes serves as a strategic nudge from SAP, encouraging the adoption of modern, object-oriented practices.

Adhering to best practices like Clean ABAP, leveraging modern development tools such as ADT and abapGit, and embracing object-oriented principles are essential for writing high-quality, maintainable, and future-proof ABAP applications. The ABAP community is undergoing a cultural shift, prioritizing code quality and long-term sustainability alongside immediate functionality.

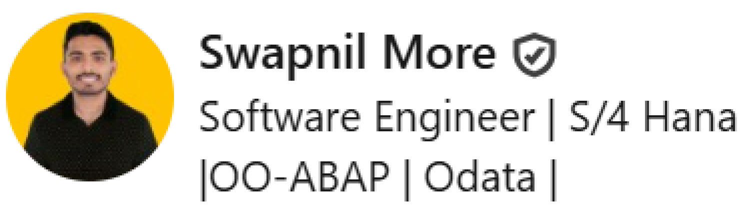
Future-Proofing ABAP Skills

For individual ABAP developers and organizations, the choice between embracing modernization and clinging to outdated practices carries significant implications. Mastering the new syntax is not merely an option but an essential requirement for success in SAP S/4HANA environments and for developing cloud-compatible applications. The ABAP language will continue its evolution, with new commands and technologies being introduced regularly. Therefore, continuous learning is not optional but a necessity for ABAP developers to remain competitive and relevant in the dynamic SAP ecosystem.

The report's exploration has demonstrated that the numerous benefits of modern ABAP, coupled with SAP's strategic push towards HANA and cloud-native development, make modernization inevitable. Stagnation, fueled by the "dangerous side effect" of backward compatibility, can lead to diminishing opportunities and increasing technical debt. By embracing modern paradigms and tools, developers can elevate ABAP from a niche, procedural language to a more versatile, contemporary enterprise programming language, enabling them to contribute effectively to the next generation of SAP applications. Learning modern ABAP not only enhances skills within the SAP ecosystem but also exposes developers to broader software engineering principles, potentially making transitions to other technologies less daunting in the long run.

CONNECT FOR MORE

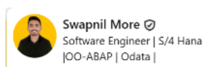
CLICK HERE



Swapnil More

Software Engineer | S/4 Hana

|OO-ABAP | Odata |



Swapnil More

Software Engineer | S/4 Hana

|OO-ABAP | Odata |