—BONUSKAPITEL—

Aus alt mach neu

Die wunderbare neue ABAP-Syntax

In ABAP hat sich in den letzten Jahren einiges getan. Falls du die letzten Jahre im Dornröschenschlaf verbracht hast, fasse ich hier alles für dich zusammen. Die neue ABAP-Syntax macht uns das Leben ziemlich einfach. Von den neuen Inline-Deklarationen bis hin zu Erweiterungen beim Initialisieren gibt es so viel zu erzählen, dass ich mich dazu entschlossen habe, eine Auswahl zu treffen. Ich zeige dir die Features, die aus meiner Sicht für dich besonders nützlich sind, speziell für deinen Start in die neue ABAP-Welt.

Danke für deine Fürsorge. Du darfst mich aber gerne etwas überfordern. Du weißt ja: Nur durch Überforderung lerne ich etwas!

Wow, Schrödinger, der Didakt!

String Templates

Die **String Templates** (Textschablonen) dienen der Aufbereitung und Formatierung von Texten und Variablen.

Altes, ups, 'tschuldigung, klassisches ABAP	New Style ABAP	Mein Kommentar

`DATA: ld_time TYPE t,`
`ld_time_text TYPE text10,`
`ld_output TYPE string.`
 	
`ld_time = sy-uzeit.`
`WRITE ld_time TO ld_time_text`
`USING EDIT MASK '__:__:__'.`
 	
`CONCATENATE ld_time_text`
`` `Uhr` INTO ld_output``
`SEPARATED BY space.`
	`DATA(ld_output) =`
	`|{ sy-uzeit TIME = ISO }`
	`Uhr|`
		Mit den String Templates wird das Formatieren von Texten total einfach. Die String Templates werden von senkrechten Strichen begrenzt. In den geschwungenen Klammern steht die Variable, die du gerne formatieren möchtest. Außerhalb der Klammern steht ein Text, der zu dem formatierten Text ergänzt wird.

Inline-Deklarationen

Die **Inline-Deklarationen** helfen dir dabei, weniger **Variablen** am Anfang eines Programms zu deklarieren. Du kannst die Variablen damit einfach direkt an der Stelle deklarieren, an der du sie benötigst.

Klassisches ABAP	New Style ABAP	Mein Kommentar
`DATA: gd_anz_kaffee TYPE i.`	`DATA(gd_anz_kaffee) = 9.`	Inline-Deklaration. Du kannst, wo immer du willst, eine Variable im Programm deklarieren, praktisch also „in der Zeile", daher heißt es Inline-Deklaration.

`DATA: gs_bohne`
`LIKE LINE OF gt_bohne.`
`LOOP AT gt_bohne`
`INTO gs_bohne.`
`*`
`ENDLOOP.`
	`LOOP AT gt_bohne`
	`INTO DATA(gs_bohne).`
	`*`
	`ENDLOOP.`
		Schon wieder eine Inline-Deklaration, aber hier bei einer Schleife. Das geht auch mit Feldsymbolen (kommt zwei Zeilen weiter unten).

Klassisches ABAP	New Style ABAP	Mein Kommentar
`DATA: gs_bohne LIKE LINE OF gt_bohne. READ TABLE gt_bohne INTO gs_bohne.`	`READ TABLE gt_bohne INTO DATA(gs_bohne).`	interne Tabelle, Einzelsatz
`FIELD-SYMBOLS: <gs_bohne> LIKE LINE OF gt_bohne. LOOP AT gt_bohne ASSIGNING <gs_bohne>. * ENDLOOP.`	`LOOP AT gt_bohne INTO ASSIGNING FIELD-SYMBOL(<gs_bohne>). * ENDLOOP.`	schon wieder eine Inline-Deklaration, aber hier bei einer Schleife mit Feldsymbol
`FIELD-SYMBOLS: <gs_bohne> LIKE LINE OF gt_bohne. READ TABLE gt_bohne ASSIGNING <gs_bohne>.`	`READ TABLE gt_bohne ASSIGNING FIELD-SYMBOL(<gs_bohne>).`	interne Tabelle, Einzelsatz
`DATA: gt_bohne TYPE TABLE OF bohne. SELECT * FROM bohne INTO TABLE gt_bohne WHERE spalte = variable.`	`SELECT * FROM bohne INTO TABLE DATA(gt_bohne) WHERE spalte = @variable.`	Lesen mehrerer Sätze aus der Datenbanktabelle `bohne`. In der neuen Variante siehst du, dass vor dem Wert ein `@`-Symbol steht. Damit musst du nun die Vergleichsvariablen aus dem Programm in der `SELECT`-Anweisung markieren.

Tabellenausdrücke

Das **Lesen** ist mit der neuen Syntax wirklich einfach. Achtung gebührt der Ausnahmebehandlung, die musst du etwas anders implementieren, als gewohnt.

Klassisches ABAP	New Style ABAP	Mein Kommentar
`READ TABLE gt_bohne INDEX index.`	`gt_bohne[ index ].`	Da kannst du jetzt ganz einfach den Index direkt hinter dem Namen der internen Tabelle in eckigen Klammern angeben, voll praktisch. Aufpassen: Der Index beginnt bei 1. Schieß nicht darüber hinaus, sonst erhältst du einen Fehler (Ausnahmeklasse `CX_SY_ITAB_LINE_NOT_FOUND`).

Klassisches ABAP	New Style ABAP	Mein Kommentar

`READ TABLE gt_bohne WITH KEY comp1 = wert1 comp2 = wert2 INTO gs_bohne.`	`gs_bohne = gt_bohne[ comp1 = wert1 comp2 = wert2 ].`	Lesen einer Tabelle mit Schlüsselkomponenten. Wenn da ein Fehler bei der Verwendung des Inline-Codes passiert, weil zum Beispiel nichts gefunden wird, dann kracht es wieder. Eine Lösung dafür wäre:
		`ASSIGN gt_bohne[ 1 ] to FIELD-SYMBOL(<ls_bohne>).`
		`IF sy-subrc = 0.`
		`...`
		`ENDIF.`

Typkonvertierung mit CONV

Willst du von einem Typ einer Variablen in einen anderen **Typ wechseln**, dann ist **CONV** die richtige Anweisung für dich.

Klassisches ABAP	New Style ABAP	Mein Kommentar

`DATA: gd_result TYPE i,`
`gd_sqrt1`
`TYPE i`
`gd_sqrt2`
`TYPE i.`
`gd_sqrt1 = sqrt( 2 ).`
`gd_sqrt2 = sqrt( 3 ).`
`gd_result = gd_sqrt1 + gd_sqrt2.`
	`DATA: gd_result TYPE i,.`
	`gd_result =`
	`CONV i( sqrt( 2 ) ) +`
	`CONV i( sqrt( 3 ) ).`
		Du kannst Typkonvertierungen mit dem Schlüsselwort CONV durchführen. Danach gibst du an, in welchen Zieltyp du die Variable konvertieren möchtest. In unserem Beispiel steht das `i` für Integer. Dabei kannst du alle Typen verwenden. Wenn du willst, dass der Zieltyp automatisch erkannt wird, zum Beispiel bei einem Schnittstellenparameter, verwende ein `#` statt des Typnamens.

Inline-Wertzuweisung mit VALUE

Falls du total einfach **initialisieren** möchtest, verwende **VALUE**, um Inhalte für Strukturen und interne Tabellen anzulegen.

<table>
<tr><th>Klassisches ABAP</th><th>New Style ABAP</th><th>Mein Kommentar</th></tr>
<tr><td>Das gibt es im klassischen ABAP nicht.</td><td></td><td>Variable:
VALUE datentyp|#( )
Du kannst mit VALUE direkt einen Wert zuweisen.</td></tr>
<tr><td></td><td>gs_bohne =
VALUE gts_bohne(
name = 'Arabica'
groesse = 5).</td><td>Struktur:
VALUE datentyp|#(comp1 = wert1 comp2 = wert2 ...)
Da gibst du einfach die Komponenten mit den Werten an.</td></tr>
<tr><td></td><td>DATA gtr_bohne
TYPE RANGE OF groesse.
gtr_bohne = VALUE #(
sign = 'I'
option = 'BT' (
low = 1 high = 10)).</td><td>Interne Tabelle:
VALUE datenyp|#((...) (...) ...) ...
Mit dem Beispiel füllst du die interne Tabelle gtr_bohne mit einer Zeile:
<table><tr><th>sign</th><th>option</th><th>low</th><th>high</th></tr><tr><td>I</td><td>BT</td><td>1</td><td>10</td></tr></table>
So einfach geht das!</td></tr>
</table>

FOR-Schleife

Die **FOR**-Schleife ist optimiert, um **Einträge in internen Tabellen** anzulegen.

Klassisches ABAP	New Style ABAP	Mein Kommentar

`DATA: gs_bohne LIKE LINE OF gt_bohne, gt_bohnen_groesse TYPE TABLE OF ty_groesse, gs_bohnen_groesse LIKE LINE OF gt_bohnen_groesse.`
 	
`LOOP AT gt_bohne INTO gs_bohne WHERE groesse < 42.`
`gs_bohnen_groesse = gs_bohne-groesse.`
`APPEND gs_bohnen_groesse TO gt_bohnen_groesse.`
`ENDLOOP.`
	`DATA(gt_bohnen_groesse) = VALUE gts_bohne( FOR ls_bohne IN gt_bohne WHERE ( groesse < 42 ) ( groesse = ls_bohne-groesse ) ).`	Mit der FOR-Anweisung kannst du Schleifen über eine interne Tabelle ausführen. Die ganze Syntax sieht so aus:
		`FOR wa|<fs> IN itab [INDEX INTO idx] [cond]`
		Zuerst musst du eine Struktur definieren, auch **Workarea** genannt, oder ein Feldsymbol. Die sind nur sichtbar in der FOR-Schleife und sonst nirgends, also kannst du sie auch nur in der FOR-Schleife verwenden. Dafür musst du keinen Strukturtyp festlegen, der wird von der Tabelle abgeleitet.
		Dann kommt der Name der internen Tabelle.
		Mit `INDEX INTO` kannst du dir den Zeilenindex `idx` merken. Die Bedingung `cond` sieht so aus, wie bei einer WHERE-Bedingung, nix Spektakuläres.
		Im Beispiel erhältst du die interne Tabelle `gt_bohnen_groesse` mit einer Spalte `groesse` und allen Werten pro Zeile, die kleiner als 42 sind:
		
		`groesse`
		1
		3
		4
		17
		41

Klassisches ABAP	New Style ABAP	Mein Kommentar
DATA: gt_bohne TYPE ty_bohne, gd_count TYPE I, FIELD-SYMBOLS <gs_bohne> LIKE LINE OF gt_bohne. gd_count= 1. DO. gd_count = gd_count + 10. IF gd_count > 40. EXIT. ENDIF. APPEND INITIAL LINE TO gt_bohne ASSIGNING <gs_bohne>. <gs_bohne>-name = `Name ` && gd_count. <gs_bohne>-groesse = gd_count. ENDDO.	DATA(gt_bohne) = VALUE ty_bohne(FOR gd_count = 11 THEN gd_count + 10 UNTIL gd_count > 40 (name = `Name ` && gd_count groesse = gd_count)).	Die FOR-Schleife gibt es auch noch mit THEN und UNTIL|WHILE. Die Syntax dazu sieht folgendermaßen aus: FOR i = ... [THEN expr] UNTIL|WHILE log_exp

COND und SWITCH

Bedingte Zuweisung, das wäre wohl die treffende Zusammenfassung für die Anweisungen **COND** und **SWITCH**, mit denen du Logik und Werteausprägungen kombinieren kannst.

Klassisches ABAP	New Style ABAP	Mein Kommentar
Das gibt es im klassischen ABAP nicht.	DATA(time) = COND string(WHEN sy-timlo < '120000' THEN |{ sy-timlo TIME = ISO } AM| WHEN sy-timlo > '120000' THEN |{ CONV t(sy-timlo - 12 * 3600) TIME = ISO } PM| WHEN sy-timlo = '120000' THEN |High Noon| ELSE THROW zcx_time_confusion( )).	Der COND-Ausdruck liefert einen Wert zurück. Dabei ist die Rückgabe von logischen Bedingungen abhängig, zum Beispiel wenn die Tageszeit kleiner als 12:00 Uhr ist. Dann ist nämlich **Vormittag**, und in manchen Teilen der Erde würde man die Uhrzeit mit AM kennzeichnen. Die ganze Syntax lautet: COND datentyp|#(WHEN log_exp1 THEN result1 [WHEN log_exp2 THEN result2] * ... [ELSE resultn]) ...

Klassisches ABAP	New Style ABAP	Mein Kommentar
	`DATA(text) =` `NEW class( )->meth(` `SWITCH #( sy-langu` `WHEN 'D' THEN `DE` `WHEN 'E' THEN `EN` `ELSE THROW cx_langu_not_supported( ) ) ).`	Der SWITCH-Ausdruck leistet das Gleiche.

Objektorientierung

Was sollte ich da noch sagen? Eleganter und einfacher geht fast nicht.

Klassisches ABAP	New Style ABAP	Mein Kommentar

`DATA: ls_bohne`
`TYPE ty_bohne, ld_name`
`LIKE ls_bohne-name.`
`ls_bohne=`
`zcl_bohne=>get_bohne( ).`
`ld_name = ls_bohne-name.`
	`DATA(ld_name) = zcl_bohne=>get_bohne( )`
	`-name.`
		Felder werden direkt in Strukturen angesprochen, die von Methoden mittels RETURNING-Parameter zurückgeliefert werden.

	Variante 1 `DATA(go_bohne) =` `NEW zcl_bohne( `Arabica` ).` Variante 2 `DATA: go_bohne` `TYPE REF TO zcl_bohne.` `go_bohne = NEW #(` `Robusta`).`  Variante 3 `NEW zcl_bohne(`Excelsa`)->get_bohne( ).`	Den NEW-Operator kannst du alternativ zur Anweisung CREATE OBJECT verwenden, um ein neues Objekt anzulegen.