ABSTRACT

To leverage off the fantastic new functionality available to you in ABAP 7.40!

Jangid, Aditya

ABAP 7.4

By Aditya

Expression Warm Up

Example 1.

```
DATA itab TYPE TABLE OF scarr.

SELECT * FROM scarr INTO TABLE itab.

DATA wa LIKE LINE OF itab.

READ TABLE itab WITH KEY carrid = ‘LH’ INTO wa.

DATA output TYPE string.

CONCATENATE ‘Carrier:’ wa–carrname INTO output SEPARATED BY space.

cl_demo_output=>display( output ).
```

- ***inline declarations in declaration positions. Let’s rewrite the above code in 7.40***

```
DATA itab TYPE TABLE OF scarr.

SELECT * FROM scarr INTO TABLE itab.

READ TABLE itab WITH KEY carrid = ‘LH’ INTO DATA(wa).

cl_demo_output=>display( |Carrier: { wa–carrname }| ).
```

- ***Let’s go one step further using a table expression:***

```
DATA itab TYPE TABLE OF scarr.

SELECT * FROM scarr INTO TABLE itab.

cl_demo_output=>display( |Carrier: { itab[ carrid = ‘LH’ ]–carrname }| ).
```

Gee! Square brackets within curly brackets and no `READ`-statement any more 😉

Inline Declarations

Inline declarations are a new way of declaring variables and field symbols at operand positions.

Before 7.40

```
DATA text TYPE string.
text = 'Aditya's code before 7.4'.
```

With 7.40

```
DATA(text) = 'Aditya's code after 7.4'.
```

Declaration of table work areas.

Before 7.40

```
DATA wa like LINE OF itab.
LOOP AT itab INTO wa.
  ...
ENDLOOP.
```

With 7.40

```
LOOP AT itab INTO DATA(wa).
  ...
ENDLOOP.
```

Declaration of a helper variable

Before 7.40

```
DATA cnt TYPE i.

FIND … IN … MATCH COUNT cnt.
```

With 7.40

```
FIND … IN … MATCH COUNT DATA(cnt).
```

Declaration of a result

Before 7.40

```
DATA xml TYPE xstring.
CALL TRANSFORMATION … RESULT XML xml.
```

With 7.40

```
CALL TRANSFORMATION … RESULT XML DATA(xml).
```

Declaration of actual parameters

Before 7.40

```
DATA a1 TYPE …
DATA a2 TYPE …
oref->meth( IMPORTING p1 = a1
            IMPORTING p2 = a2
            … )
```

With 7.40

```
oref->meth( IMPORTING p1 = DATA(a1)
            IMPORTING p2 = DATA(a2)
            … ).
```

Declaration of reference variables for factory methods

Before 7.40

```
DATA ixml           TYPE REF TO if_ixml.
DATA stream_factory TYPE REF TO if_ixml_stream_factory.
DATA document       TYPE REF TO if_ixml_document.

ixml           = cl_ixml=>create( ).

stream_factory = ixml->create_stream_factory( ).

document       = ixml->create_document( ).
```

With 7.40

```
DATA(ixml)           = cl_ixml=>create( ).

DATA(stream_factory) = ixml->create_stream_factory( ).

DATA(document)       = ixml->create_document( ).
```

Field Symbols:

For field symbols there is the new declaration operator `FIELD-SYMBOL(<Field_symbol>)` that you can use at exactly three declaration positions.

- `ASSIGN <dataobject> TO FIELD-SYMBOL(<fs>).`
- ```
  LOOP AT itab ASSIGNING FIELD-SYMBOL(<line>).
  …
  ENDLOOP.
  ```
- `READ TABLE itab ASSIGNING FIELD-SYMBOL(<line>) …`
  ```

Constructor Operator NEW

7.40 ABAP supports constructor operators.

Constructor operators are used in constructor expressions to create a result that can be used at operand positions. The syntax for constructor expressions is

operator <datatype>/#( )

operator is a constructor operator. type is either the explicit name of a data type or the character #. With # the data type can be derived from the operand position if the operand type is statically known. Inside the parentheses specific parameters can be specified.

Instantiation Operator NEW

The instantiation operator NEW is a constructor operator that creates an object (anonymous data object or instance of a class).

- … NEW dtype(value) …

creates an anonymous data object of data type dtype and passes a value to the created object. The value construction capabilities cover structures and internal tables (same as those of the VALUE operator).

- … NEW class(p1 = a1 p2 = a2 …) …

creates an instance of class class and passes parameters to the instance constructor.

- … NEW #(…) …

creates either an anonymous data object or an instance of a class depending on the operand type.

You can write a component selector -> directly behind NEW type(…).

Example for data objects

Before Release 7.40

```
FIELD-SYMBOLS <fS> TYPE data.
DATA dref TYPE REF TO data.

CREATE DATA dref TYPE i.

ASSIGN dref->* TO <fs>.
<fs> = 555.
```

With Release 7.40

```
DATA dref TYPE REF TO data.

dref = NEW i( 555 ).
```

Example for instances of classes

Before Release 7.40

DATA oref TYPE REF TO class.

CREATE OBJECT oref EXPORTING …

With Release 7.40

Either

DATA oref TYPE REF TO class.
oref = NEW #(…).

or with an inline declaration

DATA(oref) = NEW class(…).

This is the kind of statement NEW is made for. You can also pass it to methods expecting references.

```
TYPES: BEGIN OF t_struct1,
         col1 TYPE i,
         col2 TYPE i,
       END OF t_struct1,

       BEGIN OF t_struct2,
         col1 TYPE i,
         col2 TYPE t_struct1,
         col3 TYPE TABLE OF t_struct1 WITH EMPTY KEY,
       END OF t_struct2,

       t_itab TYPE TABLE OF t_struct2 WITH EMPTY KEY.

DATA(DREF) =  NEW t_itab( ( col1 = 1        >>> 1st record
                            col2-col1 = 1
                            col2-col2 = 2
                            col3 = VALUE #( ( col1 = 1 col2 = 2 )
                                            ( col1 = 3 col2 = 4 ) ) )

                           ( col1 = 2  >>> 2nd record
                             col2-col1 = 2
                             col2-col2 = 4
                             col3 = VALUE #(( col1 = 2 col2 = 4 )
                                            ( col1 = 6 col2 = 8 ) ) ) ).
```

Constructor Operator VALUE

The value operator `VALUE` is a constructor operator that constructs a value for the type specified with `type`.

- `VALUE dtype|#( )` constructs an initial value for any data type.

```
VALUE dtype|#( comp1 = a1 comp2 = a2 … ) …
```

constructs a structure where for each component a value can be assigned.

```
VALUE dtype|#( ( … ) ( … ) … ) …
```

constructs an internal table, where for each line a value can be assigned. Inside inner parentheses you can use the syntax for structures but not the syntax for table lines directly. But you can nest VALUE operators.

Note that you cannot construct elementary values (which is possible with instantiation operator NEW) – simply because there is no need for it.

For internal tables with a structured line type there is a short form that allows you to fill columns with the same value in subsequent lines

```
VALUE dtype|#( col1 = dobj11 … ( col2 = dobj12 col3 = dobj13 … )
                               ( col2 = dobj22 col3 = dobj23 … )
                                   …
               col1 = dobj31 col2 = dobj32 … ( col3 = dobj33 … )
                                             ( col3 = dobj43 … )
               … ).
```

```
CLASS c1 DEFINITION.
  PUBLIC SECTION.
    TYPES: BEGIN OF t_struct,
             col1 TYPE i,
             col2 TYPE i,
           END OF t_struct.
    CLASS-METHODS m1 IMPORTING p TYPE t_struct.
ENDCLASS.
CLASS c1 IMPLEMENTATION.
  METHOD m1.
    …
  ENDMETHOD.
ENDCLASS.
START-OF-SELECTION.
  c1=>m1( VALUE #( ) ).
```

Example for structures

Three different ways to construct the same nested structure:

```
TYPES:  BEGIN OF t_col2,
           col1 TYPE i,
           col2 TYPE i,
        END OF t_col2.

TYPES: BEGIN OF t_struct,
         col1 TYPE i,
         col2 TYPE t_col2,
       END OF t_struct.

DATA: struct TYPE t_struct,
      col2   TYPE t_col2.

"1
struct = VALUE t_struct( col1 = 1
                         col2-col1 = 1
                         col2-col2 = 2 ).

"2

col2   = VALUE   t_col2( col1 = 1
                         col2 = 2 ).

struct = VALUE t_struct( col1 = 1
                         col2 = col2 ).

"3

struct = VALUE t_struct( col1 = 1
                         col2 = VALUE #( col1 = 1
                                         col2 = 2 ) ).
```

Examples for internal tables:

Elementary line type:

```
TYPES t_itab TYPE TABLE OF i WITH EMPTY KEY.

DATA itab TYPE t_itab.

itab = VALUE #( ( ) ( 1 ) ( 2 ) ).
```

Structured line type (RANGES table):

```
DATA itab TYPE RANGE OF i.

itab = VALUE #( sign = 'I'  option = 'BT'( low = 1  high = 10)
                                          ( low = 21 high = 30)
                                          ( low = 41 high = 50)
                            option = 'GE' ( low = 61 )  ).
```

Other expressions in VALUE operator

The arguments of `VALUE` can be expressions or function calls:

```
TYPES t_date_tab TYPE TABLE OF string WITH EMPTY KEY.

DATA(date_tab) = VALUE t_date_tab(
                    ( |{ CONV d( sy-datlo - 1 ) DATE = ENVIRONMENT }| )
                    ( |{         sy-datlo       DATE = ENVIRONMENT }| )
                    ( |{ CONV d( sy-datlo + 1 ) DATE = ENVIRONMENT }| )
).
```

Constructor Operator REF

The reference operator REF constructs a data reference at operand positions.

… REF dtype|#(dobj) …

results in a data reference pointing to dobj with the static type specified by type. with other words, REF is the short form for GET REFERENCE OF dobj INTO.

Example for dynamic method call

Using `REF` for filling the parameter table:

```
CLASS class DEFINITION.
  PUBLIC SECTION.
    METHODS meth
              IMPORTING p1 TYPE string
                        p2 TYPE i.
ENDCLASS.

CLASS class IMPLEMENTATION.
  METHOD meth.
    …
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
  DATA(arg1) = `blah`.
  DATA(arg2) = 111.

  DATA(ptab) = VALUE abap_parmbind_tab(
    ( name = 'P1' kind = cl_abap_objectdescr=>exporting value = REF #( arg1 ) )
    ( name = 'P2' kind = cl_abap_objectdescr=>exporting value = REF #( arg2 ) ) ).

DATA(oref) = NEW class( ).

  CALL METHOD oref->('METH') PARAMETER-TABLE ptab.
```

Example for ADBC

Parameter binding made easy!

```
DATA(key) = 'LH'.
DATA(sql) = NEW cl_sql_statement( ).

sql->set_param( REF #( sy-mandt ) ).
sql->set_param( REF #( key ) ).

DATA(result) = sql->execute_query(
      `SELECT carrname ` &&
      `FROM scarr ` &&
      `WHERE mandt  = ? AND carrid = ?` ).

DATA name TYPE scarr-carrname.
result->set_param( REF #( name ) ).
result->next( ).
Believe it or not: name contains the carrier name now.
```

Operators CONV and CAST

The conversion operator `CONV` is a constructor operator that converts a value into the type specified in `type`.

… CONV dtype|#(…) …

You use `CONV` where you needed helper variables before in order to achieve a requested data type.

Example for parameter passing

Method `cl_abap_codepage=>convert_to` expects a string but you want to convert a text field.

Before release 7.40

```
DATA text TYPE c LENGTH 255.
DATA helper TYPE string.
DATA xstr   TYPE xstring.

helper = text.

xstr = cl_abap_codepage=>convert_to( source = helper ).
```

With release 7.40

```
DATA text TYPE c LENGTH 255.

DATA(xstr) = cl_abap_codepage=>convert_to( source = CONV string( text ) ).
```

In such cases it is even simpler to write

```
DATA text TYPE c LENGTH 255.

DATA(xstr) = cl_abap_codepage=>convert_to( source = CONV #( text ) ).
```

Casting Operator CAST

The casting operator `CAST` is a constructor operator that executes an up or down cast for reference varaibles with the type specified in `type`.

`… CAST dtype|class|interface|#( … ) …`

- You use `CAST` for a down cast where you needed helper variables before in order to cast with `?=`to a requested reference type.
- You use `CAST` for an up cast, e,g, with an inline declaration, in order to construct a more general type.

You can write a compnent selector `->` directly behind `CAST type( … ).`

Example from RTTI

Common example where a down cast is needed.

Before release 7.40

```
DATA structdescr TYPE REF TO cl_abap_structdescr.
structdescr ?= cl_abap_typedescr=>describe_by_name( 'T100' ).
DATA components  TYPE abap_compdescr_tab.
components = structdescr->components.
```

With release 7.40

```
DATA(components) = CAST cl_abap_structdescr(
  cl_abap_typedescr=>describe_by_name( 'T100' ) )->components.
```

Example with up cast

The static type of the reference variable iref declared inline should be the interface not the class.

```
INTERFACE if.
  …
ENDINTERFACE.
CLASS cl DEFINITION CREATE PRIVATE.
  PUBLIC SECTION.
    INTERFACES if.
    CLASS-METHODS factory RETURNING value(ref) TYPE REF TO cl.
    …
ENDCLASS.

CLASS cl IMPLEMENTATION.
  METHOD factory.
    ref = NEW #( ).
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
  DATA(iref) = CAST if( cl=>factory( ) ).
```

Example with data objects

A constructor expression with `CAST` followed by `->` is an LHS-expression, you can assign values to it.

```
TYPES: BEGIN OF t_struc,
        col1 TYPE i,
        col2 TYPE i,
       END OF t_struc.
DATA dref  TYPE REF TO data.
DATA struc TYPE t_struc.
dref = NEW t_struc( ).
CAST t_struc( dref )->col1 = struc–col1.
```

Constructor Operator EXACT

Lossless Operator EXACT

The lossless operator `EXACT` is a constructor operator that exceutes either a lossless calculation or a lossless assignment.

- `… EXACT dtype|#( arith_exp ) …`
  `arith_exp` is an arithmetic expression that is calculated lossless with calculation type `decfloat34` and the result is converted to the specified type.
- `… EXACT dtype|#( arg ) …`
  `arg` is **not** an arithmetic expression and its value is assigned to the result of the specified type according to the rules for lossless assignments.

Lossless calculations

Lossless calculations were introduced for decimal floating point numbers in Release 7.02 with the addition `EXACT` to `COMPUTE`. This addition (better the whole statement `COMPUTE`) became obsolete now.
A lossless calculation must not perform any roundings. If it does, an exception occurrs.

Example

```
TRY.
    DATA(r1) = EXACT decfloat34( 3 / 2 ).
    cl_demo_output=>write( |Exact: { r1 }| ).
  CATCH cx_sy_conversion_rounding INTO DATA(e1).
    cl_demo_output=>write( |Not exact: { e1->value }| ).
ENDTRY.
TRY.
    DATA(r2) = EXACT decfloat34( 3 / 7 ).
    cl_demo_output=>write( |Exact: { r2 }| ).
  CATCH cx_sy_conversion_rounding INTO DATA(e2).
    cl_demo_output=>write( |Not exact: { e2->value }| ).
ENDTRY.
cl_demo_output=>display( ).
```

The output is:

```
Exact: 1.5
Not exact: 0.4285714285714285714285714285714286
```

You see that the non-exact result can be found in the exception object.

Lossless assignments

Lossless assignments were introduced for conversions in Release 7.02 with the addition `EXACT` to `MOVE`. This addition (better the whole statement `MOVE`) became obsolete now.
A lossless assignment is an assignment where

- the value of the source is valid for its type
- there are no data losses during the assignment
- the converted value of the target is valid for its type

Example

```
TYPES numtext TYPE n LENGTH 10.

cl_demo_output=>write( CONV numtext( ‘4 Apples + 2 Oranges’ ) ).
TRY.
    DATA(number) = EXACT numtext( ‘4 Apples + 2 Oranges’ ).
  CATCH cx_sy_conversion_error INTO DATA(exc).
  cl_demo_output=>write( exc->get_text( ) ).
ENDTRY.
cl_demo_output=>display( ).
```

The result is
`0000000042`
`The argument ‘4 Apples + 2 Oranges’ cannot be interpreted as a number`
The infamous conversion rule from `c` to `n` is not supported by `EXACT`.

Operators COND and SWITCH

Last but not least, two nice ones, the conditionals `COND` and `SWITCH`.

- ```
  … COND dtype|#( WHEN log_exp1 THEN result1
                  [ WHEN log_exp2 THEN result2 ]
                  …
                  [ ELSE resultn ] ) …
  ```

  constructs a result of the specified type that depends on logical expressions.

- ```
  … SWITCH dtype|#( operand
                      WHEN const1 THEN result1
                    [ WHEN const2 THEN result2 ]
                    …
                    [ ELSE resultn ] ) …
  ```

constructs a result of the specified type that depends on a case differentiation.

With other words: `IF` and `CASE` as expressions in operand positions!

Example for COND

```
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 cx_cant_be( ) ).
```

Note the `THROW`. Now you can raise **and** throw exceptions …

Example for SWITCH

```
CLASS cx_langu_not_supported DEFINITION INHERITING FROM cx_static_check.
ENDCLASS.
CLASS class DEFINITION.
  PUBLIC SECTION.
    METHODS meth IMPORTING iso_langu   TYPE string
                 RETURNING VALUE(text) TYPE string.
ENDCLASS.
CLASS class IMPLEMENTATION.
  METHOD meth.
    …
  ENDMETHOD.
ENDCLASS.
…
  DATA(text) =
    NEW class(
      )->meth(
          SWITCH #( sy-langu
                    WHEN ‘D’ THEN `DE`
                    WHEN ‘E’ THEN `EN`
                    ELSE THROW cx_langu_not_supported( ) ) ).
```