ABAP 7.40

Contents

1. Inline Declarations

Description	Before 7.40	With 7.40
Datastatement	DATA text TYPE string. text = `ABC`.	DATA(text) = `ABC`.
Loop at into work area	DATA wa like LINE OF itab. LOOP AT itab INTO wa. … ENDLOOP.	LOOP AT itab INTO DATA(wa). … ENDLOOP.
Call method	DATA a1 TYPE … DATA a2 TYPE … oref->meth(IMPORTING p1 = a1 IMPORTING p2 = a2).	oref->meth(IMPORTING p1 = DATA(a1) IMPORTING p2 = DATA(a2)).
Loop at assigning	FIELD-SYMBOLS: <line> type … LOOP AT itab ASSIGNING <line>. … ENDLOOP.	LOOP AT itab ASSIGNING FIELD-SYMBOL(<line>). … ENDLOOP.
Read assigning	FIELD-SYMBOLS: <line> type … READ TABLE itab ASSIGNING <line>.	READ TABLE itab ASSIGNING FIELD-SYMBOL(<line>).

Select into table	DATA itab TYPE TABLE OF dbtab.
	SELECT * FROM dbtab
	INTO TABLE itab
	WHERE fld1 = lv_fld1.
		SELECT * FROM dbtab
		INTO TABLE DATA(itab)
		WHERE fld1 = @lv_fld1.

Select single into	SELECT SINGLE f1 f2
	FROM dbtab
	INTO (lv_f1, lv_f2)
	WHERE …
	WRITE: / lv_f1, lv_f2.
		SELECT SINGLE f1 AS my_f1,
		F2 AS abc
		FROM dbtab
		INTO DATA(ls_structure)
		WHERE …
		WRITE: / ls_structure-my_f1, ls_structure-abc.

2. Table Expressions

If a table line is not found, the exception `CX_SY_ITAB_LINE_NOT_FOUND` is raised. No `sy-subrc`.

Description	Before 7.40	With 7.40
Read Table index	READ TABLE itab INDEX idx INTO wa.	wa = itab[idx].
Read Table using key	READ TABLE itab INDEX idx USING KEY key INTO wa.	wa = itab[KEY key INDEX idx].
Read Table with key	READ TABLE itab WITH KEY col1 = … col2 = … INTO wa.	wa = itab[col1 = … col2 = …].

Read Table with key components	READ TABLE itab
	WITH TABLE KEY key
	COMPONENTS col1 = …
	col2 = …
	INTO wa.
		wa = itab[KEY key col1 = …
		col2 = …].

Does record exist?	READ TABLE itab …
	TRANSPORTING NO FIELDS.
	IF sy-subrc = 0.
	…
	ENDIF.
		IF line_exists(itab[…]).
		…
		ENDIF.

Get table index	DATA idx type sy-tabix. READ TABLE … TRANSPORTING NO FIELDS. idx = sy-tabix.	DATA(idx) = line_index(itab[…]).

NB: There will be a short dump if you use an inline expression that references a non-existent record.

SAP says you should therefore assign a field symbol and check sy-subrc.

ASSIGN `lt_tab[ 1 ]` to FIELD–SYMBOL(`<ls_tab>`).

IF `sy–subrc` = 0.

…

ENDIF.

NB: Use itab [`table_line = … ] for untyped tables.`

3. Conversion Operator CONV

I. Definition

CONV dtype|#(…)

dtype = Type you want to convert to (explicit)

= compiler must use the context to decide the type to convert to (implicit)

II. Example

Method cl_abap_codepage=>convert_to expects a string

Before 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 7.40

```
DATA text TYPE c LENGTH 255.
DATA(xstr) = cl_abap_codepage=>convert_to( source = CONV string( text ) ).
                                OR
DATA(xstr) = cl_abap_codepage=>convert_to( source = CONV #( text ) ).
```

4. Value Operator VALUE

I. Definition

Variables: VALUE dtype|#( )

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

Tables: VALUE dtype|#((…) (…) …) …

II. Example for structures

```
TYPES:  BEGIN OF ty_columns1, “Simple structure
          cols1 TYPE i,
          cols2 TYPE i,
        END OF ty_columns1.
TYPES: BEGIN OF ty_columnns2,   “Nested structure
          coln1 TYPE i,
          coln2 TYPE ty_columns1,
       END OF ty_columns2.
```

```
  DATA: struc_simple TYPE ty_columns1,
           struc_nest    TYPE ty_columns2.
 struct_nest   = VALUE t_struct(coln1 = 1
                                coln2-cols1 = 1
                                coln2-cols2 = 2 ).
```

OR

```
 struct_nest   = VALUE t_struct(coln1 = 1
                                 coln2 = VALUE #( cols1 = 1
                                 cols2 = 2 ) ).
```

III. 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 )  ).
```

5. FOR operator

I. Definition

```
  FOR wa|<fs> IN itab [INDEX INTO idx] [cond]
```

II. Explanation

This effectively causes a loop at itab. For each loop the row read is assigned to a work area (wa) or field-symbol(<fs>).

This wa or <fs> is local to the expression i.e. if declared in a subrourine the variable wa or <fs> is a local variable of

that subroutine. Index like SY-TABIX in loop.

Given:

```
TYPES: BEGIN OF ty_ship,
           tknum TYPE tknum,     "Shipment Number
           name  TYPE ernam,     "Name of Person who Created the Object
           city  TYPE ort01,     "Starting city
           route TYPE route,     "Shipment route
       END OF ty_ship.
TYPES: ty_ships TYPE SORTED TABLE OF ty_ship WITH UNIQUE KEY tknum.
TYPES: ty_citys TYPE STANDARD TABLE OF ort01 WITH EMPTY KEY.
```

GT_SHIPS type ty_ships. -> has been populated as follows:

Row	TKNUM[C(10)]	Name[C(12)]	City[C(25)]	Route[C(6)]
1	001	John	Melbourne	R0001
2	002	Gavin	Sydney	R0003
3	003	Lucy	Adelaide	R0001
4	004	Elaine	Perth	R0003

III. Example 1

Populate internal table GT_CITYS with the cities from GT_SHIPS.

Before 7.40

```
DATA: gt_citys TYPE ty_citys,

       gs_ship  TYPE ty_ship,

       gs_city  TYPE ort01.

LOOP AT gt_ships INTO gs_ship.
  gs_city =  gs_ship-city.
  APPEND gs_city TO gt_citys.
ENDLOOP.
```

With 7.40

```
DATA(gt_citys) = VALUE ty_citys( FOR ls_ship IN gt_ships ( ls_ship-city ) ).
```

IV. Example 2

Populate internal table GT_CITYS with the cities from GT_SHIPS where the route is R0001.

Before 7.40

```
DATA: gt_citys TYPE ty_citys,

       gs_ship  TYPE ty_ship,

       gs_city  TYPE ort01.

LOOP AT gt_ships INTO gs_ship WHERE route = ‘R0001’.
  gs_city =  gs_ship-city.
  APPEND gs_city TO gt_citys.
ENDLOOP.
```

With 7.40

```
DATA(gt_citys) = VALUE ty_citys( FOR ls_ship IN gt_ships
                               WHERE ( route = ‘R0001’ ) ( ls_ship-city ) ).
```

Note: ls_ship does not appear to have been declared but it is declared implicitly.

V. FOR with THEN and UNTIL|WHILE

```
FOR i = … [THEN expr] UNTIL|WHILE log_exp
```

Populate an internal table as follows:

```
TYPES:

  BEGIN OF ty_line,

    col1 TYPE i,

    col2 TYPE i,

    col3 TYPE i,

  END OF ty_line,

  ty_tab TYPE STANDARD TABLE OF ty_line WITH EMPTY KEY.
```

Before 7.40

```
DATA: gt_itab TYPE ty_tab,

      j       TYPE i.

FIELD-SYMBOLS <ls_tab> TYPE ty_line.

j = 1.
DO.
j = j + 10.
IF j > 40. EXIT. ENDIF.
APPEND INITIAL LINE TO gt_itab ASSIGNING <ls_tab>.
<ls_tab>-col1 = j.
<ls_tab>-col2 = j + 1.
<ls_tab>-col3 = j + 2.
ENDDO.
```

With 7.40

Before 7.40

```
DATA(gt_itab) = VALUE ty_tab( FOR j = 11 THEN j + 10 UNTIL j > 40

                             ( col1 = j col2 = j + 1 col3 = j + 2  ) ).
```

6. Reduction operator REDUCE

I. Definition

… REDUCE type(

INIT result = start_value

    …

FOR for_exp1

FOR for_exp2

…

NEXT …

    result = iterated_value

…)

II. Note

While VALUE and NEW expressions can include FOR expressions, REDUCE must include at least one FOR expression. You can use all kinds of FOR expressions in REDUCE:

- with IN for iterating internal tables
- with UNTIL or WHILE for conditional iterations

III. Example 1

Count lines of table that meet a condition (field F1 contains “XYZ”).

Before 7.40

```
DATA: lv_lines TYPE i.
LOOP AT gt_itab INTO ls_itab where F1 = ‘XYZ’.
  lv_lines= lv_lines + 1.
ENDLOOP.
```

With 7.40

```
DATA(lv_lines) = REDUCE i( INIT x = 0 FOR wa IN gt_itab
                    WHERE( F1 = ‘XYZ’ ) NEXT x = x + 1 ).
```

IV. Example 2

Sum the values 1 to 10 stored in the column of a table defined as follows

```
DATA gt_itab TYPE STANDARD TABLE OF i WITH EMPTY KEY.

gt_itab = VALUE #( FOR j = 1 WHILE j <= 10 ( j ) ).
```

Before 7.40

```
DATA: lv_line TYPE i,
      lv_sum  TYPE i.
LOOP AT gt_itab INTO lv_line.
  lv_sum = lv_sum + lv_line.
ENDLOOP.
```

With 7.40

```
DATA(lv_sum) = REDUCE i( INIT x = 0 FOR wa IN itab NEXT x = x + wa ).
```

V. Example 3

Using a class reference – works because “write” method returns reference to instance object

With 7.40

```
TYPES outref TYPE REF TO if_demo_output.

DATA(output) = REDUCE outref( INIT out  = cl_demo_output=>new( )
                              text = `Count up:`
                              FOR n = 1 UNTIL n > 11
                              NEXT out = out->write( text )
                              text = |{ n }| ).
output->display( ).
```

7. Conditional operators COND and SWITCH

I. Definition

```
… COND dtype|#( WHEN log_exp1 THEN result1

[ WHEN log_exp2 THEN result2 ]

…

[ ELSE resultn ] ) …

… SWITCH dtype|#( operand

WHEN const1 THEN result1

[ WHEN const2 THEN result2 ]

…

[ ELSE resultn ] ) …
```

II. 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( ) ).
```

III. Example for SWITCH

```
DATA(text) =

NEW class( )->meth(

                     SWITCH #( sy-langu

                              WHEN 'D' THEN `DE`

                              WHEN 'E' THEN `EN`

                               ELSE THROW cx_langu_not_supported( ) ) ).
```

8. Corresponding Operator

I. Definition

… CORRESPONDING type([BASE (base)] struct|itab [mapping|except])

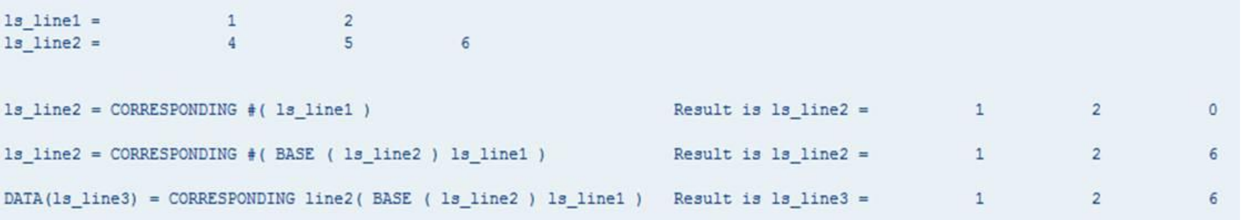
II. Example Code

With 7.40

```
TYPES: BEGIN OF line1, col1 TYPE i, col2 TYPE i, END OF line1.
TYPES: BEGIN OF line2, col1 TYPE i, col2 TYPE i, col3 TYPE i, END OF line2.
DATA(ls_line1) = VALUE line1( col1 = 1 col2 = 2 ).
WRITE: / 'ls_line1 =' ,15 ls_line1-col1, ls_line1-col2.
DATA(ls_line2) = VALUE line2( col1 = 4 col2 = 5 col3 = 6 ).
WRITE: / 'ls_line2 =' ,15 ls_line2-col1, ls_line2-col2, ls_line2-col3.
SKIP 2.
ls_line2 = CORRESPONDING #( ls_line1 ).
WRITE: / 'ls_line2 = CORRESPONDING #( ls_line1 )'
      ,70 'Result is ls_line2 = '
         ,ls_line2-col1, ls_line2-col2, ls_line2-col3.
SKIP.
ls_line2 = VALUE line2( col1 = 4 col2 = 5 col3 = 6 ).   "Restore ls_line2
ls_line2 = CORRESPONDING #( BASE ( ls_line2 ) ls_line1 ).
WRITE: / 'ls_line2 = CORRESPONDING #( BASE ( ls_line2 ) ls_line1 )'
         , 70 'Result is ls_line2 = ', ls_line2-col1
         , ls_line2-col2, ls_line2-col3.
SKIP.
ls_line2 = VALUE line2( col1 = 4 col2 = 5 col3 = 6 ).   "Restore ls_line2
DATA(ls_line3) = CORRESPONDING line2( BASE ( ls_line2 ) ls_line1 ).
WRITE: / 'DATA(ls_line3) = CORRESPONDING line2( BASE ( ls_line2 ) ls_line1 )'
```

With 7.40
, 70 'Result is ls_line3 = ' , ls_line3-col1 , ls_line3-col2, ls_line3-col3.

III. Output



```
ls_line1 =              1           2
ls_line2 =              4           5           6


ls_line2 = CORRESPONDING #( ls_line1 )                           Result is ls_line2 =          1           2           0

ls_line2 = CORRESPONDING #( BASE ( ls_line2 ) ls_line1 )         Result is ls_line2 =          1           2           6

DATA(ls_line3) = CORRESPONDING line2( BASE ( ls_line2 ) ls_line1 )  Result is ls_line3 =       1           2           6
```

IV. Explanation

Given structures ls_line1 & ls_line2 defined and populated as above.

	Before 7.40	With 7.40
1	CLEAR ls_line2. MOVE-CORRESPONDING ls_line1 TO ls_line2.	ls_line2 = CORRESPONDING #(ls_line1).
2	MOVE-CORRESPONDING ls_line1 TO ls_line2.	ls_line2 = CORRESPONDING # (BASE (ls_line2) ls_line1).
3	DATA: ls_line3 like ls_line2. ls_line3 = ls_line2. MOVE-CORRESPONDING ls_line1 TO ls_line2.	DATA(ls_line3) = CORRESPONDING line2 (BASE (ls_line2) ls_line1).

1. The contents of ls_line1 are moved to ls_line2 where there is a matching column name. Where there is no

match the column of ls_line2 **is initialised.**

2. This uses the existing contents of ls_line2 as a base and overwrites the matching columns from ls_line1.

This is exactly like MOVE-CORRESPONDING.

3. This creates a third and new structure (ls_line3) which is based on ls_line2 but overwritten by matching columns of ls_line1.

V. Additions MAPPING and EXCEPT

MAPPING allows you to map fields with non-identically named components to qualify for the data transfer.

```
… MAPPING  t1 = s1 t2 = s2
```

EXCEPT allows you to list fields that must be excluded from the data transfer

```
… EXCEPT  {t1 t2 …}
```

9. Strings

I. String Templates

A string template is enclosed by two characters “|” and creates a character string.

Literal text consists of all characters that are not in braces {}. The braces can contain:

- data objects,
- calculation expressions,
- constructor expressions,
- table expressions,
- predefined functions, or
- functional methods and method chainings

Before 7.40

```
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 ).
```

With 7.40

```
SELECT * FROM scarr INTO TABLE @DATA(lt_scarr).

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

II. Concatenation

Before 7.40

```
DATA lv_output TYPE string.

CONCATENATE ‘Hello’ ‘world’ INTO lv_output SEPARATED BY space.
```

With 7.40

```
DATA(lv_out) = |Hello| & | | & |world|.
```

III. Width/Alignment/Padding

```
WRITE / |{ ‘Left’     WIDTH = 20 ALIGN = LEFT   PAD = ‘0’ }|.

WRITE / |{ ‘Centre’   WIDTH = 20 ALIGN = CENTER PAD = ‘0’ }|.

WRITE / |{ ‘Right’    WIDTH = 20 ALIGN = RIGHT  PAD = ‘0’ }|.
```

IV. Case

```
WRITE / |{ ‘Text’ CASE = (cl_abap_format=>c_raw) }|.

WRITE / |{ ‘Text’ CASE = (cl_abap_format=>c_upper) }|.

WRITE / |{ ‘Text’ CASE = (cl_abap_format=>c_lower) }|.
```

V. ALPHA conversion

```
DATA(lv_vbeln) = ‘0000012345’.

WRITE / |{ lv_vbeln  ALPHA = OUT }|.     “or use ALPHA = IN to go in other
direction
```

VI. Date conversion

```
WRITE / |{ pa_date DATE = ISO }|.           “Date Format YYYY-MM-DD

WRITE / |{ pa_date DATE = User }|.          “As per user settings

WRITE / |{ pa_date DATE = Environment }|.   “Formatting setting of language
environment
```

10. Loop at Group By

I. Definition

```
LOOP AT itab result [cond] GROUP BY key ( key1 = dobj1 key2 = dobj2 …

       [gs = GROUP SIZE] [gi = GROUP INDEX] )

       [ASCENDING|DESCENDING [AS TEXT]]

       [WITHOUT MEMBERS]

       [{INTO group}|{ASSIGNING <group>}]

       …

       [LOOP AT GROUP group|<group>

       …
```

```
        ENDLOOP.]
        …
ENDLOOP.
```

II. Explanation

The outer loop will do one iteration per key. So if 3 records match the key there will only be one iteration for these 3 records. The structure “group” (or “<group>”) is unusual in that it can be looped over using the “LOOP AT GROUP” statement. This will loop over the 3 records (members) of the group. The structure “group” also contains the current key as well as the size of the group and index of the group (if GROUP SIZE and GROUP INDEX have been assigned a field name). This is best understood by an example.

III. Example

With 7.40

```
TYPES: BEGIN OF ty_employee,
  name TYPE char30,
  role    TYPE char30,
  age    TYPE i,
END OF ty_employee,
ty_employee_t TYPE STANDARD TABLE OF ty_employee WITH KEY name.
DATA(gt_employee) = VALUE ty_employee_t(
( name = ‘John‘     role = ‘ABAP guru‘       age = 34 )
( name = ‘Alice‘     role = ‘FI Consultant‘   age = 42 )
( name = ‘Barry‘     role = ‘ABAP guru‘       age = 54 )
( name = ‘Mary‘     role = ‘FI Consultant‘   age = 37 )
( name = ‘Arthur‘   role = ‘ABAP guru‘       age = 34 )
( name = ‘Mandy‘  role = ‘SD Consultant‘  age = 64 ) ).
DATA: gv_tot_age TYPE i,
          gv_avg_age TYPE decfloat34.
“Loop with grouping on Role
LOOP AT gt_employee INTO DATA(ls_employee)
  GROUP BY ( role  = ls_employee-role
                      size  = GROUP SIZE
                     index = GROUP INDEX )
  ASCENDING
  ASSIGNING FIELD-SYMBOL(<group>).
  CLEAR: gv_tot_age.
  “Output info at group level
  WRITE: / |Group: { <group>-index }    Role: { <group>-role WIDTH = 15 }|
            & |     Number in this role: { <group>-size }|.
   “Loop at members of the group
   LOOP AT GROUP <group> ASSIGNING FIELD-SYMBOL(<ls_member>).
     gv_tot_age = gv_tot_age + <ls_member>-age.
     WRITE: /13 <ls_member>-name.
   ENDLOOP.
   “Average age
   gv_avg_age = gv_tot_age / <group>-size.
   WRITE: / |Average age: { gv_avg_age }|.
```

With 7.40

```
  SKIP.
ENDLOOP.
```

IV. Output

```
Group: 1    Role: ABAP guru         Number in this role: 3
            John
            Barry
            Arthur
Average age: 40.666666666666666666666666666667
Group: 2    Role: FI Consultant      Number in this role: 2
             Alice
             Mary
Average age: 39.5
Group: 3    Role: SD Consultant      Number in this role: 1
             Mandy
Average age: 64
```

11. Classes/Methods

I. Referencing fields within returned structures

Before 7.40

```
DATA: ls_lfa1  TYPE lfa1,

      lv_name1 TYPE lfa1-name1.

ls_lfa1  = My_Class=>get_lfa1( ).
lv_name1 = ls_lfa1-name1.
```

With 7.40

```
DATA(lv_name1) = My_Class=>get_lfa1( )-name1.
```

II. Methods that return a type BOOLEAN

Before 7.40

```
IF My_Class=>return_boolean( ) = abap_true.
…
ENDIF.
```

With 7.40

```
IF My_Class=>return_boolean( ).
…
ENDIF.
```

NB: The type “BOOLEAN” is not a true Boolean but a char1 with allowed values X,- and <blank>.

Using type “FLAG” or “WDY_BOOLEAN” works just as well.

III. NEW operator

This operator can be used to instantiate an object.

Before 7.40

```
DATA: lo_delivs TYPE REF TO zcl_sd_delivs,
      lo_deliv TYPE REF TO zcl_sd_deliv.
CREATE OBJECT lo_delivs.
CREATE OBJECT lo_deliv.
lo_deliv = lo_delivs->get_deliv( lv_vbeln ).
```

With 7.40

```
DATA(lo_deliv) = new zcl_sd_delivs( )->get_deliv( lv_vbeln ).
```

12. Meshes

Allows an association to be set up between related data groups.

I. Problem

Given the following 2 internal tables:

```
TYPES: BEGIN OF t_manager,

name    TYPE char10,

salary TYPE int4,

END OF t_manager,

tt_manager TYPE SORTED TABLE OF t_manager WITH UNIQUE KEY name.

TYPES: BEGIN OF t_developer,

name     TYPE char10,

salary  TYPE int4,

manager TYPE char10,    “Name of manager

END OF t_developer,

tt_developer TYPE SORTED TABLE OF t_developer WITH UNIQUE KEY name.
```

Populated as follows:

Row	**Name[C(10)]**	Salary**[I(4)]**
1	Jason	3000
2	Thomas	3200

Row	**Name[C(10)]**	Salary**[I(4)**	Manager**[C(10)]**
1	Bob	2100	Jason
2	David	2000	Thomas

Row	Name[C(10)]	Salary[I(4)]	
3	Jack	1000	Thomas
4	Jerry	1000	Jason
5	John	2100	Thomas
6	Tom	2000	Jason

Get the details of Jerry’s manager and all developers managed by Thomas.

II. Solution

With 7.40

```
TYPES: BEGIN OF MESH m_team,
         managers   TYPE tt_manager  ASSOCIATION my_employee TO developers
                         ON manager = name,
         developers TYPE tt_developer ASSOCIATION my_manager TO managers
                         ON name = manager,
       END OF MESH m_team.
DATA: ls_team TYPE m_team.
ls_team-managers   = lt_manager.
ls_team-developers = lt_developer.
*Get details of Jerry’s manager *
“get line of dev table
ASSIGN lt_developer[ name = ‘Jerry’ ] TO FIELD-SYMBOL(<ls_jerry>).
DATA(ls_jmanager) =  ls_team-developers\my_manager[ <ls_jerry> ].
WRITE: / |Jerry‘s manager: { ls_jmanager-name }|,30
                  |Salary: { ls_jmanager-salary }|.

“Get Thomas’ developers
SKIP.
WRITE: / |Thomas‘ developers:|.
“line of manager table
ASSIGN lt_manager[ name = ‘Thomas’ ] TO FIELD-SYMBOL(<ls_thomas>).
LOOP AT ls_team-managers\my_employee[ <ls_thomas> ]
        ASSIGNING FIELD-SYMBOL(<ls_emp>).
  WRITE: / |Employee name: { <ls_emp>-name }|.
ENDLOOP.
```

III. Output

```
Jerry’s manager: Jason        Salary: 3000

Thomas’ developers:

Employee name: David

Employee name: Jack

Employee name: John
```

13. Filter

Filter the records in a table based on records in another table.

I. Definition

… FILTER type(itab [EXCEPT] [IN ftab] [USING KEY keyname]

WHERE c1 op f1 [AND c2 op f2 […]])

II. Problem

Filter an internal table of Flight Schedules (SPFLI) to only those flights based on a filter table that contains the fields Cityfrom and CityTo.

III. Solution

With 7.40

```
TYPES: BEGIN OF ty_filter,
         cityfrom TYPE spfli-cityfrom,
         cityto   TYPE spfli-cityto,
         f3       TYPE i,
       END OF ty_filter,
       ty_filter_tab TYPE HASHED TABLE OF ty_filter
                     WITH UNIQUE KEY cityfrom cityto.
DATA: lt_splfi TYPE STANDARD TABLE OF spfli.
SELECT * FROM spfli APPENDING TABLE lt_splfi.
DATA(lt_filter) = VALUE ty_filter_tab( f3 = 2
           ( cityfrom = ‘NEW YORK’  cityto  = ‘SAN FRANCISCO’ )
             ( cityfrom = ‘FRANKFURT’ cityto  = ‘NEW YORK’ )  ).
DATA(lt_myrecs) = FILTER #( lt_splfi IN lt_filter
                                  WHERE cityfrom = cityfrom
                                    AND cityto = cityto ).
“Output filtered records
LOOP AT lt_myrecs ASSIGNING FIELD-SYMBOL(<ls_rec>).
  WRITE: / <ls_rec>-carrid,8 <ls_rec>-cityfrom,30
           <ls_rec>-cityto,45 <ls_rec>-deptime.
ENDLOOP.
```

Note: using the keyword “EXCEPT” (see definition above) would have returned the exact opposite records i.e all records EXCEPT for those those returned above.