C5275218 initialization ba7cbab 3 days ago

0 contributors

587 lines (435 sloc) 13.8 KB

Modern ABAP Language Elements

As the ABAP language evolves and gets new features, they often go unnoticed by developers. This document highlights some of the additions which improve readability of the code and therefore belong in the toolset of a professional clean code ABAP developer.

The new ABAP statements are not described to full detail. For more information please consult the ABAP online documentation.

Table of Contents

Functional statements and expressions in ABAP

Functional statements and expressions have the advantage that they follow the principle of an assignment, meaning the result is returned as a returning parameter, thus can be directly used in assignments and can be chained.

Data type information in functional statements can be abbreviated by a `#` if the compiler can determine the data type from the context.

> **Caution**: Some of the functional statements are very powerful and versatile in their usage. When using them always keep the clean code principles in mind.

Inline declaration of variables

Use the operators `data` and `field-symbol` to combine the declaration and (initial) value assignment of a variable or field symbol. Allowed at write positions for which a type can be determined statically from the context. This *inferred* type is given to the declared symbol.

```
data(text) = `This is a string`.

" -- Old style
data text type string.
text = `This is a string`.


data(return) = method_with_returning( ).

" -- Old style
" data return type accounts_table.
" return = method_with_returning( ).


method_with_exporting( importing return = data(export) ).

" -- Old style
" data export type accounts_table.
" method_with_exporting( importing return = export ).


loop at accounts into data(acount).
endloop.

" -- Old style
" data account type accounts_table.
" loop at accounts into acount.
" endloop.


read table accounts into data(acount_sap) with key id = 5.

" -- Old style
" data account_sap type account_structure.
" read table account into data(acount_sap) with key id = 5.


loop at accounts assigning field-symbol(<account>).
endloop.

" -- Old style
" field-symbol <account> type account_structure.
" loop at accounts assigning (<account>.
" endloop.


assign component id of acount_sap to field-symbol(<account_id>).

" -- Old style
" field-symbol <account_id> type account_structure.
" assign component id of acount_sap to <account_id>.


select * from t000 into table @data(clients).

" -- Old style
" data clients type standard table of t000.
" select * from t000 into table clients.
```

Booleans

Comparison with the type `abap_bool` can be omitted in conditions.

```
if method_returns_abap_bool( ).
  "method returned abap_true
else.
  "method returned abap_false
endif.
```

Enumerations

Define enumerations instead of using constants.

```
types: begin of enum scrum_status_type,
         open,
         in_progress,
         blocked,
         done,
       end if enum scrum_status_type.

data(scrum_status) = open.
```

Old style:

```
constants scrum_status_open       type i value 1.
constants scrum_status_in_process type i value 2.
constants scrum_status_blocked    type i value 3.
constants scrum_status_done       type i value 4.
data scrum status type i.

scrum_status = scrum_status_open.
```

Internal Tables

Count table lines

Count the number of lines of an internal table use the function `lines( )`.

```
data(number_of_lines) = lines( accounts ).
```

Old style:

```
data number_of_lines type i.
describe table accounts lines number_of_lines.
```

Check existence of a table line

Check the existence of a line in an internal table, use the function `line_exists( )` within an if-clause.

```
if line_exists( accounts[ id = 4711 ] ).
  "line has been found
endif.
```

Old style:

```
read table accounts with key id = 4711 transporting no fields.
if sy-subrc = 0.
  "line has been found
endif.
```

Access table key with uncertain result

```
data(account) = value #( accounts[ id = '4711' ] optional ).
```

By default, failing functional key accesses throw an exception. The addition `value ... optional` suppresses this.

Old style:

```
try.
    account = accounts[ id = '4711' ]
  catch cx_sy_itab_line_not_found.
endtry.
```

Access table index

Access a specific index of an internal table directly, use the bracket notation `table_name[ ]`.

```
data(id_of_account_5) = accounts[ 5 ]-id.
```

Old style:

```
read table accounts index 5 into data(account_5).
if sy-subrc = 0.
  data(id_of_account_5) = account_5-id.
endif.
```

Conditions

Conditional distinction

To evaluate conditions, use the `cond #( )` operator.

```
data(value) = cond #( when status = open then 1
                      when status = blocked then 3
                      else 7 ).
```

Old style:

```
data value type i.
if status = open.
  value = 1.
elseif status = blocked.
  value = 3.
else.
  value = 7.
endif.
```

> Alternatively you may use the function `xsdbool( )` described here

Case distinction

Evaluate case distinction with the `switch #( )` operator

```
data(status) = switch #( scrum_status
                         when scrum_status_open then status_waiting
                         when scrum_status_in_process then status_busy
                         when scrum_status_blocked then status_alarm
                    when scrum_status_done then status_ok ).
```

Old style:

```
data status type status_enum.
case scrum_status.
  when scrum_status_open.
    status = status_waiting.
```

```
  when scrum_status_in_process.
    status = status_busy.
  when scrum_status_blocked.
    status = status_alarm.
  when scrum_status_done.
    status = status_ok.
endcase.
```

Case distinctions of reference types class and interface

Switch on a reference types class and interface using the `case` extension `type of`.

```
case type of account.
  when type bank_account into data(bank_account).
    " some processing ...
  when others.
    " some processing ...
endcase.
```

In a condition e.g. in an `if` statement the `is in stance of` operator can be used.

```
if account is instance of bank_account.
  " some processing ...
endif.
```

Conversions

Cast data references

Cast reference types to other reference types use the `cast #( )` operator.

```
data(my_account) = cast account( NEW bank_account( ) ).
```

Old style:

```
data my_account type ref to account.
create object my_account type bank_account.
```

or

```
data my_account type ref to account.
data my_bank_account type ref to bank_account.
create object my_bank_account.
my_account ?= bank_account.
```

Create data references

Create data references to structures and tables with the operator `ref #( )`.

```
data accounts type accounts_table.
import_accounts_references( ref #( accounts ) ).
```

Old style:

```
data accounts type accounts_table.
data accounts_reference type ref to accounts_type.
get reference of accounts into accounts_reference.
import_accounts_references( accounts_reference ) ).
```

Convert data types

Use the operator `conv #( )` to convert data types and save temporary variables.

```
method_takes_string( conv #( a_char ) ).
```

Old style:

```
data a_string type string.
a_string = a_char.
method_takes_string( a_string ).
```

Copy fields with matching names

Copy fields with matching names from one data type to another with `corresponding #( )`.

```
target_structure = corresponding #( source_structure ).
```

Old style:

```
move-corresponding source_structure to target_structure.
```

Constructors

Construct objects and data

Construct objects and data with the `new #( )` operator.

```
data(acount) = new cl_account( ).

data(dref) = new struct_type( component_1 = 10
                              component_2 = 'a' ).


data(account) = cast if_account( new cl_account( ) ).

data data_structure TYPE REF TO struct_type.
create data data_structure.
data_reference->component_1 = 10.
data_reference->component_2 = 'a'.
```

Old style:

```
data account type ref to cl_account.
create object account.


data account type ref to if_account.
create object account type cl_account.
```

Construct data types

Construct structures and tables using the `value #( )` operator. It also constructs initial values for most data types.

> This statement is a life saver when writing ABAP unit tests.

Structure:

```
data(account) = value account_structure( id = 5
                                         name = 'SAP' ).
```

Old style:

```
data account type account_structure.
account-id = 5.
account-name = 'SAP'.
```

Table:

```
data(accounts) = value accounts_table( ( id = 5  name = 'SAP' )
                                       ( id = 6  name = 'ABCDE' ) ).
```

Old style:

```
data accounts type accounts_table.
data account type account_structure.
account-id = 5.
account-name = 'SAP'.
insert account into table accounts.
account-id = 6.
account-name = 'ABCDE'.
insert account into table accounts.
```

Construct tables based on other tables:

```
result = value #( for row in input ( row-text ) ).
```

Old style:

```
loop at input into data(row).
  insert row-text into table result.
ENDLOOP.
```

Filter tables

Construct a table as a subset of another stable using `filter #( )`.

```
bank_accounts = filter #( accounts
                          where account_type = 'B' ).
```

Old style:

```
data bank_account type bank_account.
loop at accounts into bank_account where account_type = 'B'.
  insert bank_account into table bank_accounts.
endloop.
```

Character Handling

Lower and upper case conversion

Convert characters between cases using `to_upper( )` or `to_lower( )`.

```
data(uppercase) = to_upper( lowercase ).
data(lowercase) = to_lower( uppercase ).
```

Old style:

```
translate lowercase to upper case.
```

Flow control

Loop with control break

Process data on defined groups using the new features with the `loop` statement extension `group by ...` and `loop at group`.

```
loop at accounts into data(account) group by grouping_id.
  " once per group before group ...
  loop at group account into data(account_group).
    " for each group member ...
  endloop.
  " once per group after group ...
endloop.
```

Old style:

```
data previous_grouping_id type i.
data last_account type account.
loop at accounts into data(account).
  if account-grouping_id <> previous_grouping_id.
    previous_grouping_id = account-grouping_id
    " once per group before group ...
    if last_account is not initial.
      " once per group after group ...
    endif.
  endif.
  " for each group member ...
  last_account = account.
endloop.
if last_account is not initial.
  " once per group after group
endif.
```

Interface implementation

Behavior on not implemented interface methods

Define the effect on not implemented interface methods.

Add `default ignore` to advise ABAP to handle the call to this method if not implemented as a call to an empty implementation.

```
interface account.
  methods new_method default ignore.
endinterface.
```

Add `default fail` to advise ABAP to raise an exception `CX_SY_DYN_CALL_ILLEGAL_METHOD` if the not implemented method is called. This is the default behavior.

```
interface account.
  methods new_method default fail.
endinterface.
```

Partially implement interfaces in tests

Implement in tests for e.g. test doubles only the interface methods which you need and skip the not needed with the `partially implemented` extension of the `interfaces` statement.

```
interface account.
  methods add_account importing account type account.
  methods delete_account importing account_id type account_id.
  methods get_account importing account_id type account_id
                      returning value(result) type account.
endinterface.

class test_double definition for testing.
  public section.
  interfaces account partially implemented.
  data account_stub type account.
endclass.

class test_double implementation.
  method productive~get.
    result = account_stub.
```

```
    endmethod.
endclass.
```