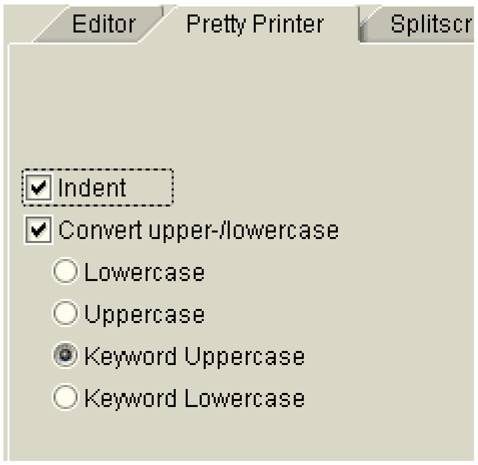
Table of Contents

General Standards

- Always follow the development process standard:
- This process facilitates development efforts, improves response time, etc.
- Program code should be easy to read
  - Use pretty printer. You should change your setting for this under Utilities/Settings.



[✓] Indent
[✓] Convert upper-/lowercase
○ Lowercase
○ Uppercase
● Keyword Uppercase
○ Keyword Lowercase

.

  - Avoid selects or if statements that span several pages.
  - Form routine size should be manageable, a few pages at most.
- SAP's Source Code Inspector should be used (see standards document "Code Review SCI tests in for details). **New** One Global **Check variant is being worked on for ABAP 7.4 and will be soon added in this document.**
- Remove all unused variable declarations. 'Extended Program Check' can be run to identify unused data variables. Tables declared at the beginning of the program may show up on this check even though they are used in the program.
- No unit-specific values such as purchasing organizations, plants, or company codes should be hard-coded into objects that we create. All unit values should be passed as parameters, variants or rows in user defined or SAP data dictionary tables.
- Clean up unnecessary objects in the development system after development is finished.
- **SAP tables should never be updated directly.**
- Do not modify SAP-delivered ABAP code, dictionary field lengths or screen layouts, except with the approved methods
  - Customer, field, and screen user exits
  - Business Add Ins (BAdIs).
- Do not re-use SAP fields for purposes not intended by the SAP software.
- The following high-risk methods of customizing SAP-delivered programs require review by a technical leader in your division: exits where you programmatically change the SAP call stack, copy-and-modify of SAP delivered code, modifying SAP code. Modifying SAP code is against standards.
- Proper customization techniques will vary based on the situation at hand, and should be chosen to best suit the design. In general, however, the preferred methods of customizing are (in order of preference):

    - Configuration Without Customization
    - Application-Specific Enhancement (customizing specific to a particular area) e.g. **BDT (Business Development Toolkit)**, **BTE (Business Transaction Events)**, Substitutions validations in **FI**. **VOFM** routine in **SD** , Module Editor in **BA (Bank Analyzer)** , **FQEVENTS** in **FICA**
    - SAPUI5 apps & Screen Personas (for **UI's**)

    - New or Kernel Badi
    - Classic Badi
    - Explicit Enhancement
    - Implicit Enhancement
    - Customer Exit
    - User Exit
    - Copy Code
    - Modification Assistant

- All names of development and dictionary objects like programs, transactions, function modules, form names, dialog modules, tables, domains etc. should have English-like names.

- Per SAP Note #**936835**, avoid reinstantiating proxy objects unnecessarily (for example, in a loop) by checking to see if the object is already bound:

```
DATA: g_web_ims TYPE REF TO zwsco_imsbridge_port_type.
IF NOT g_web_ims IS BOUND.
  TRY.
    CREATE OBJECT g_web_ims    "zwsco_imsbridge_port_type
      EXPORTING logical_port_name =
                Zbc0_msg_config-logical_port. "IMSBRIDGE
                               "Logical port on LPCONFIG
    CATCH cx_ai_system_fault into g_ref.
      g_text = g_ref->get_text ( ).
    ENDTRY.
  ENDIF.
```

  This standard also applies to objects in general; it is important to consider all object instantiations and only create a new instance if necessary.

  - For performance reasons, the recommendations in **OSS Note 3992** must be followed. In particular: (1) **INDX**-based tables should never be used for persisting data long-term (anything stored there should be able to self-recover in the case of deletion), and (2) the actual table **INDX** should never be used; instead, make a separate copy of table **INDX** for your use case.

    Per **OSS Note 3992**: “This facilitates an easier identification of the application to which the data belongs. In addition, this reduces the risk of two different applications storing their data under the same **RELID** and data being merged or overwritten.”

    Information on data cluster functionality in NetWeaver 7.0 is located here:
    http://help.sap.com/saphelp_nw70/helpdata/en/fc/eb3bb7358411d1829f0000e829fbfe/frameset.htm

    SAP Note 3992

- Check if the program can be enhanced with **Test Driven Development (TDD)** Add extra code for automated testing

Application Logging

Background & Driver

The application log is a tool that collects messages, exceptions, and errors. This information is organized and displayed in a log. **Transaction code SLG1** is used to display these log records. Transaction **SLG0** is used to define custom objects. Processing is usually done with function modules in function group **SBAL** or methods of class **CL_BAL_LOGOBJ**. The tables that contain this data are growing even though data is being deleted from the tables daily (in **PAG**). Keeping up with new objects and ensuring that all objects are being deleted requires much effort and this process needs to be simplified in order reduce manual effort and avoid unnecessary complexity.

Standard

When creating application logs a valid date should be passed in **ALDATE_DEL**. This date should not be **12/31/9999** and should be a reasonable time in the future based on input from the functional team but should not exceed **180 days** except in very unusual circumstances that are approved by the ILM team. If the application logs cannot be deleted before the expiration date, then pass **X** in field **DEL_BEFORE**. Never use date **12/31/9999** in field **ALDATE_DEL** with field **DEL_BEFORE = X**. Any data needed for a long time should not be stored in **BALHDR/BALDAT** tables. If application logs are only created to troubleshoot production issues, consider designing your logging so that the logging can be turned on/off via a switch in a control table. If application logs are no longer needed for a specific process, then this code should be removed.

> **Note:**
> Basis will run a batch job on a determined schedule and delete any application logs that have reached their expiration date.

Program Name

The program name must be in compliance with the ABAP Naming Standards document. The original language for all **Custom developed objects** (programs, function modules, etc.) must be English.

SAP Development Naming Standards

Program Template

A template has been created which shows the layout for a program. The template was created for ease of maintaining programs. The template should be copied to the new program name and the layout followed for all new programs. The main template is a program based on principles of separation of concern **z_soc_class_report**.

sample_soc_rep...

Modifying Programs

Procedure

- Compare the version in development to production to make sure there are no current pending changes.
- If the changes are complex in nature consider using the sandbox client to validate the design.
- Check if there are any other Life Cycle management changes that need to be done for this program
- Check if there are any Security updates to be done to this program based on suggestions in this document
- Check if the program is in the correct Package and whether it will be moved all across the landscape, if so move it to the correct packaging assignment
- Check if the program is applicable in **S/4 HANA** systems or will be needed there
- Run Source Code Inspector variants on this program for
  - **HANA** readiness
  - **S/4 HANA** readiness
  - If the code written in the landscape which is running on **HANA** or **not on HANA/anyDB**
  - Netweaver version when it was last changed. If it's the new version, check for new commands that can enhance performance
- Check for redundant code that is commented out from last update, whether it can be deleted to keep the program clean

Documentation

- The modification log of the program should be filled out using the transport number/Feature/User Story.
- In case of production issue related break fix please mention the SNOW ticket number in the modification log.
  For single-line changes, each line should be marked with the transport number and/or a reference that's tied back to the modification log.
- For multiple-line changes, a begin of block and end of block can be used to mark the changes. (i.e. begin of *transport number* changes)
- Consider commenting old code instead of deleting. After the program has been in production for some time, you can remove old commented code.
- If the readability of the code is affected by the commenting, delete the code but enter a comment in the program that the previous version should be retrieved for complete code changes. Include the transport number of this change in the comment.
- Add more documentation in the Program Documentation for better clarification, reasons for change, especially if it was a break fix or Enhancement.

On-line Program Documentation

The SAP on-line documentation **SHOULD** be maintained with a business description of the object. It should contain sufficient information for a user to understand the functionality of the object. Any requirement, input, output, restriction, scheduling and assumption should be mentioned. Try to add comments about the life cycle of this object, so it can be reviewed regularly and deleted once it's no longer in use. Any special notes for Version differences and **HANA DB** specific comments.

The documentation for programs can be at two levels: program and in-line. All comments should be in English.

*Program – Available through **SA38** and **SE38***

When a user displays program documentation, they should get:

- A short description of the program's functionality along with the **SDP**, iPortal number, Solution Manager, or consolidation number that requested the program development

- The **SAP functional team** that is responsible for the program
- Order dependencies
- Input needed
- Output received
- Indication of availability to run in background
- Technical information
  - Tables used – especially those z tables that must be maintained and who maintains them
  - Transfers of files

In-Line – Comments which will help a programmer follow logic

In-Line comments must be used to explain the logic of the program. The comments should include but not be limited to:

- Explanation of any complex logic including examples where needed
- Any change in logic flow (i.e. continuation or exit of loops or programs)
- Functional description of all conditional logic (i.e. if statements, do while, etc.)
- Consider using comments (especially for complex situations) connecting end statements to the beginning. Including: endif, endcase, endloop, etc.

Declarative Elements

- Wherever possible, reference to data dictionary fields and structures should be used to create work variables and flags.
- In-line comments should be provided for explaining the purpose of the work fields

REPORT

Any associated parameters should be consistently indented to the right of the ***REPORT*** element, for the purposes of readability:

Format:

```
REPORT ZNA0_ABAPSTANDRD [no standard page heading]
                        [Line-size 132]
                        [Line-count 65]
                        [Message-id zz].
```

1. Include '**NO STANDARD PAGE HEADING'** parameter for those programs that are type 1 and produce one and more reports as output.

TABLES

The following is the format for declaring *TABLES:*

Format:

```
TABLES: table1,       "Table description as in-line comment
        table2,       "Table description as in-line comment
        table3,       "Table description as in-line comment
```

```
For example:-
   TABLES: MARA,         "Material Master Data
           MARC,         "Plant data for Material.
```

TYPES

Type definitions include: Simple, structured & internal tables

```
TYPES: begin of t_lfa1,
         Lifnr TYPE lfa1-lifnr,  “vendor
         Land1 TYPE lfa1-land1,  “Country Key
       End of t_lfa1.
```

SELECT-OPTIONS

- Selection texts should always be used over hard-coding text elements of the selection screen. This is for maintenance and translation reasons.
- The *fieldname* should reflect the SAP field name for ease of maintaining the program.
- Select-options and parameters should be used as much as possible instead of hard coded values.
- Make use of blocks.
- Make use of block text descriptions.
- Make use of tabs when appropriate.

The following is the format for declaring *SELECT-OPTIONS*:

Format:
```
SELECT-OPTIONS:
   s_usnam FOR bkpf-usnam, “appropriate in-line comment
   s_budat FOR bkpf-budat. “appropriate in-line comment
```

PARAMETERS

The following is the format for declaring selection *PARAMETERS*:

Format:
```
PARAMETERS:
   p_bukrs TYPE bkpf_bukrs OBLIGATORY "company code
                           MEMORY ID buk,
   p_usnam TYPE bkpf-usnam,        "appropriate in-line comment
   r_text1 RADIOBUTTON GROUP radi  "comment for radio button
           DEFAULT ‘x’,
   r_text2 RADIOBUTTON GROUP radi. "comment for radio button
c_text3 AS CHECKBOX DEFAULT ''  "comment for checkbox
```

RANGES

The following is the format for declaring *RANGES*:

Format:
```
*Appropriate comment for range
DATA: r_usnam TYPE RANGE OF bkpf-usnam.
DATA: r_usname_line LIKE LINE OF r_usnam.
```

GLOBAL FIELDS

The following is the suggested format for declaring global work variables and flags. Group like fields together under a descriptive comment:

Use meaningful assignment of names and descriptive inline comments.
Don’t use **LIKE** statement, Reference Data Dictionary fields
Data field with same name shall not be declared in global and local field

For example:
`G_MATNR & L_MATNR.`

Format:
```
DATA:
  g_usnam TYPE bkpf-usnam                    "Appropriate in-line comment

*transaction dates used
DATA: g_reverse_date    TYPE sy-datum,                         "GB01 date
      g_reverse_year (4) TYPE c.          "fiscal year for GB01 date

*data for distribution of amounts based on percentages entered
```

```
DATA: g_hsl_dis TYPE hslxx9,                         "Amount to distribute
      g_tot_per TYPE konv-kdiff,                             "Total percent
    g_ag_amount TYPE hslxx9,                                 "AG Div amount
    g_cc_amount TYPE hslxx9.                               "CC&E Div amount

*flags used
DATA: flag_success (1) TYPE c.
```

CONSTANTS

The following is the format for declaring constants:

Format:

```
CONSTANTS:
   c_usnam TYPE bkpf-usnam VALUE 'raok'. "in-line comment-
```

INTERNAL TABLES

Internal tables should be used as often as possible to save data selected from the database.
With ABAP 7.4 we have inline declarations for internal tables so we should consider this option while declaring internal tables.

- Define internal tables using table types. SAP does not recommend the use of header rows as they are incompatible with ABAP objects.
- Appropriate description should precede the table.
- The table fields should only be the fields needed for output (especially for large tables).
- Whenever an internal table is no longer needed in a program, use the statement "FREE 'itab'".
- When reading entries from an internal table, make sure the table is sorted in order, use the key (sort fields) to read the entry and use a binary search.
  Exceptions:
  - Table holds a structure that will be lost when sorting.
  - Table holds sequential information.

Naming (recommended)

- I_*table* – for input tables
- O_*table* – for output tables
- E_*table* – for error tables

Format:

*Material and Material type from MARA

```
TYPES: BEGIN OF t_mara,
               matnr LIKE mara-matnr,
               mtart LIKE mara-mtart,
            END OF t_mara.
DATA: i_mara TYPE STANDARD TABLE OF t_mara.
DATA: i_mara_line LIKE LINE OF i_mara.
```

sample_soc_report.txt

Constructor Operator in ABAP

With release **7.40 ABAP** supports constructor operators. The main objective of a constructor operator is to create a result that can be used at operand positions in a constructor expression. The syntax for constructor expression is

... Operator type (...) ...

Here, *operator* : constructor operator
Type : either the explicit name of data type or the character #.
If character # is used, data type is derived from the *operand*.

#	Operator	Name
1	NEW	Instantiation Operator
2	VALUE	Value Operator
3	CONV	Conversion Operator
4	CAST	Casting Operator
5	REF	Reference Operator
6	EXACT	Lossless Operator
7	COND	Condition Operator
8	SWITCH	Condition Operator

Table 1 : List of constructor operator in ABAP

Instantiation Operator NEW

Description and Usage

Instantiation operator **NEW** creates an anonymous data object or to instantiate an object. It is also used to new table, new work area, etc.

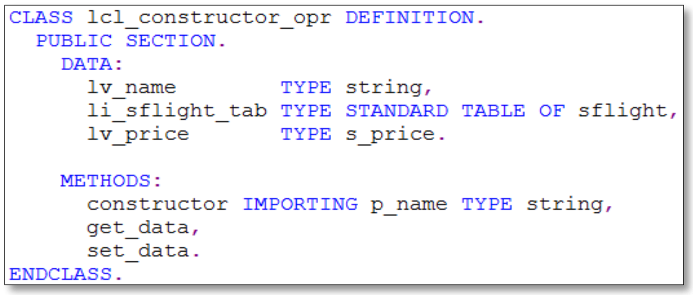
NEW operator results into code minimization by clubbing steps to create or instantiate object in one statement. For a class type, **NEW** operator works in the same way as the statement **CREATE OBJECT** oref **TYPE** class, where oref stands the result that points to the new object.

Syntax and example

*. . . **NEW** dtype (. . .) . . .*

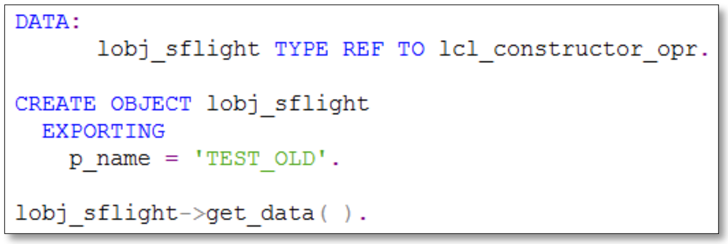
Dtype could be a non-generic data type or a class or symbol #. In case of symbol # the result of NEW takes the type based on the operand type.

1. Declaration



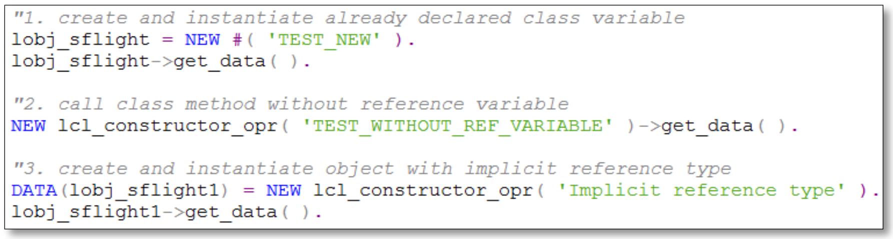
```
CLASS lcl_constructor_opr DEFINITION.
  PUBLIC SECTION.
    DATA:
      lv_name         TYPE string,
      li_sflight_tab  TYPE STANDARD TABLE OF sflight,
      lv_price        TYPE s_price.

    METHODS:
      constructor IMPORTING p_name TYPE string,
      get_data,
      set_data.
ENDCLASS.
```

2. Old way of creating and instantiating object.



```
DATA:
      lobj_sflight TYPE REF TO lcl_constructor_opr.

CREATE OBJECT lobj_sflight
  EXPORTING
    p_name = 'TEST_OLD'.

lobj_sflight->get_data( ).
```

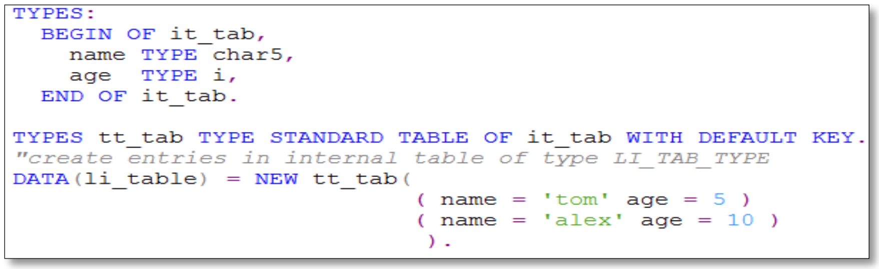
3. Using NEW operator



```
"1. create and instantiate already declared class variable
lobj_sflight = NEW #( 'TEST_NEW' ).
lobj_sflight->get_data( ).

"2. call class method without reference variable
NEW lcl_constructor_opr( 'TEST_WITHOUT_REF_VARIABLE' )->get_data( ).

"3. create and instantiate object with implicit reference type
DATA(lobj_sflight1) = NEW lcl_constructor_opr( 'Implicit reference type' ).
lobj_sflight1->get_data( ).
```

Case 1:
The class object **LOBJ_SFLIGHT** is already declared and is instantiated using the **NEW** operator. As the object is already typed so we don't need to mention the class name explicitly in the syntax.
Case 2:
With **NEW** operator we can eliminate the step of declaring a variable explicitly. The operator **DATA** and class name could be used to access the class methods and variables.
Case 3:
Combination of **NEW** and **DATA** operator can be used for incline declaration and instantiation of object variable.

4. New operator for creating internal table entries.



```
TYPES:
  BEGIN OF it_tab,
    name TYPE char5,
    age  TYPE i,
  END OF it_tab.

TYPES tt_tab TYPE STANDARD TABLE OF it_tab WITH DEFAULT KEY.
"create entries in internal table of type LI_TAB_TYPE
DATA(li_table) = NEW tt_tab(
                            ( name = 'tom' age = 5 )
                            ( name = 'alex' age = 10 )
                             ).
```

Value Operator VALUE

Description and Use

A value operator is used to construct an initial value for any non-generic data type. It is also used to construct content of structured types and table types.

Syntax and examples

*…**VALUE** dtype | # ( )…*

1. Construct a structure where for each component a value can be assigned
   a. Declaration

```
CLASS lcl_constructor_opr DEFINITION.
  PUBLIC SECTION.
    TYPES: BEGIN OF t_struct,
             col1 TYPE i,
             col2 TYPE i,
           END OF t_struct.

    CLASS-DATA:
          lv_struct TYPE t_struct.

    CLASS-METHODS val_opr_example IMPORTING p_struct TYPE t_struct.
ENDCLASS.

CLASS lcl_constructor_opr IMPLEMENTATION.
  METHOD val_opr_example.

    "method body

    ENDMETHOD.
  ENDCLASS.
```

b. Creating initial value using **VALUE** operator

```
 "in-line use of VALUE operator
 lcl_constructor_opr=>val_opr_example( VALUE #( col1 = 1 col2 = 2 )  ).

 "independent use of VALUE operator
 lcl_constructor_opr=>lv_struct = VALUE #( col1 = 1 col2 = 2 ).

 "using data type in the syntax
 li_struct_line = VALUE t_struct( col1 = 1 col2 = 1 ).
```

2. Construct an internal table, where for each line a value can be assigned

a. Declaration

```
PUBLIC SECTION.
  TYPES: BEGIN OF t_struct,
           col1 TYPE i,
           col2 TYPE i,
         END OF t_struct.

  CLASS-DATA:
        lv_struct TYPE t_struct,
        li_table  TYPE STANDARD TABLE OF t_struct.
```

b. Creating initial values in the table

```
 "initialization an internal table with different rows
 lcl_constructor_opr=>li_table = VALUE #( ( col1 = 1 col2 = 2 )
                                          ( col1 = 4 col2 = 5 ) ).

 "initialization an internal table with same column 1
 lcl_constructor_opr=>li_table = VALUE #(  col1 = 1 ( col2 = 2 )
                                                    ( col2 = 5 ) ).
```

Note

1. Arithmetic calculations with the results of **VALUE** for constructing values are not possible (except when creating an initial value). This means that constructor expressions of this type cannot be

specified directly in the operand positions of arithmetic expressions. Any constructor expressions with **VALUE** used to control table expressions are not affected by this (if the table expressions can be used for calculations using valid results).

2. If a constructor expression with **VALUE** is not used as the source of an assignment to a data object, the value operator **VALUE** creates a new temporary data object whose data type is determined by the specified type and whose content is determined by the parameters passed. This data object is used as the operand of a statement and then deleted. It is deleted either when the current statement is closed or when a relational expression is evaluated after the truth value is determined.

Conversion Operator CONV

Description and use

A constructor expression with conversion operator **CONV** converts the argument *dobj* to the data type specified using *type* and creates the appropriate result.

… **CONV** *type* (dobj)…

This operator comes handy when calling function module or subroutines. It's also useful in complicated arithmetic expression. In these situation we require another temporary variable (helper variable) to convert the value of a given operation or variable into required type and then pass the helper variable to the method or function module. **CONV** operator eliminates the use of these helper variables.

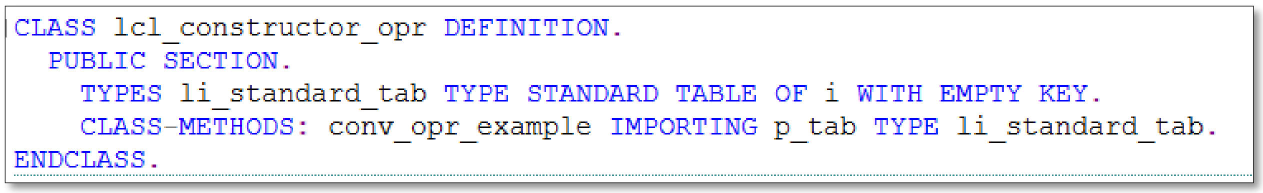
Syntax and examples

The **CONV** operator like any other constructor operator does not need an operand and equal to sign, it can be directly use as a part of arithmetic or relational expression.

… **CONV** type (dobj) …

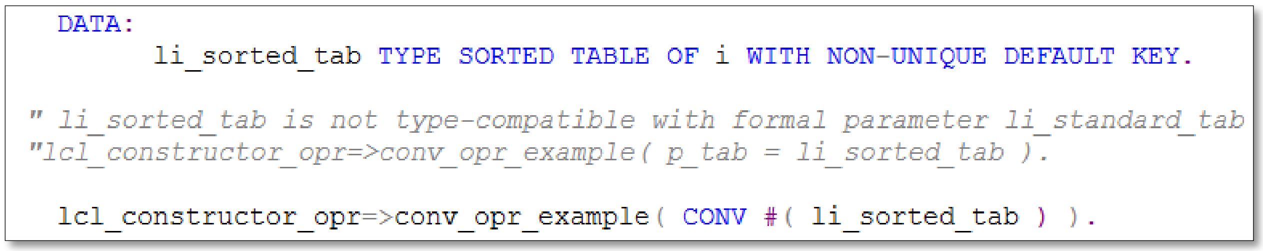
Examples:

1. While calling a method
   In the below example, the method **CONV_OPR_EXAMPLE** expects an importing parameter of type standard table of I.



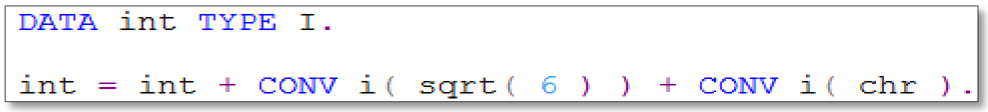
```
CLASS lcl_constructor_opr DEFINITION.
  PUBLIC SECTION.
    TYPES li_standard_tab TYPE STANDARD TABLE OF i WITH EMPTY KEY.
    CLASS-METHODS: conv_opr_example IMPORTING p_tab TYPE li_standard_tab.
ENDCLASS.
```

The call to the method has variable of type sorted table of I. With the help of **CONV** operator the value of variable **LI_SORTED_TAB** is type converted. Without the conversion operator (commented code) the compilation throws a type-compatible error.



```
   DATA:
         li_sorted_tab TYPE SORTED TABLE OF i WITH NON-UNIQUE DEFAULT KEY.

  " li_sorted_tab is not type-compatible with formal parameter li_standard_tab
  "lcl_constructor_opr=>conv_opr_example( p_tab = li_sorted_tab ).

   lcl_constructor_opr=>conv_opr_example( CONV #( li_sorted_tab ) ).
```

2. In arithmetic expressions



```
DATA int TYPE I.

int = int + CONV i( sqrt( 6 ) ) + CONV i( chr ).
```

With **CONV** operator the result of statement **SQRT** (6), which is type **F** is type converted to type **I** and value of variable **CHR**, which of type **CHAR** is type converted to type **I**.

Note

1. The use of **CONV** operator along with value operator **VALUE** helps to close the gap where the value operator cannot be used to construct values for elementary data objects except the initial value.
2. **CONV** operator is different from **CAST** operator. Conversion operator converts a value into a type mentioned in the syntax of Conversion Expression whereas **CAST** operator is used for casting of reference variables, which could be down cast or up cast. For more details check casting operator **CAST**.

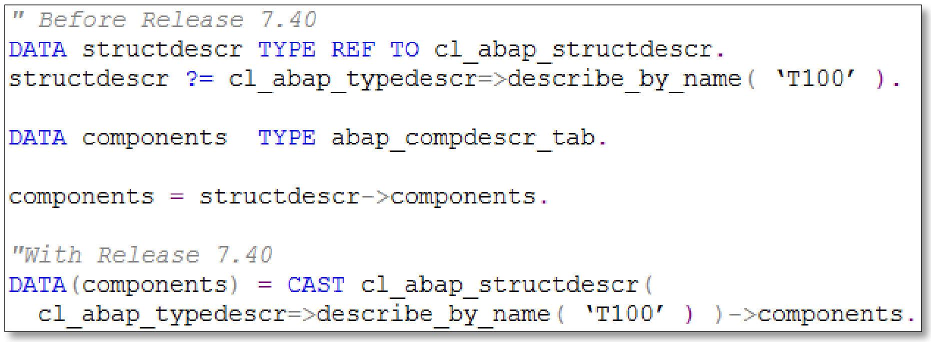
Casting Operator CAST

Description and use

The casting operator **CAST** performs a down cast or up cast for the argument and creates a reference variable. **CAST** operator eliminates use of helper variable in down casting (**old**) where it is needed in order to cast with ?**=** to a requested reference type. **CAST** operator for an up cast in order to construct more general type.

Syntax and example

*. . . **CAST** dtype | class | interface | # (. . .) . . .*



```
" 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.
```

Note

1. It is recommended to handle exception CX_SY_MOVE_CAST_ERROR when performing casting operation.

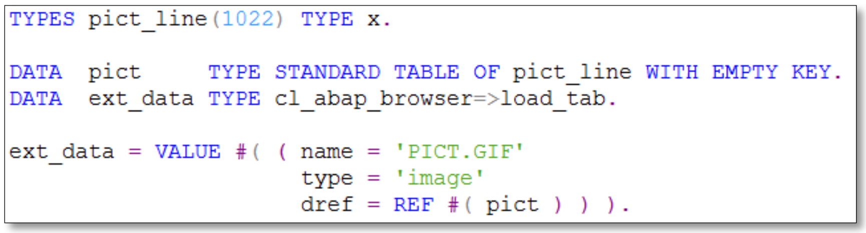
Reference Variable REF

Description and use

REF is the short form for **GET REFERENCE OF** dobj **INTO**. It eliminates the use of helper variable (temporary variable) for assignment of an object reference to another object.
The **REF** operator creates either a data reference variable that points to the argument dobj or controls a table expression.

Syntax and example

. . . REF type (dobj | *table expression) . . .*



```
TYPES pict_line(1022) TYPE x.

DATA  pict     TYPE STANDARD TABLE OF pict_line WITH EMPTY KEY.
DATA  ext_data TYPE cl_abap_browser=>load_tab.

ext_data = VALUE #( ( name = 'PICT.GIF'
                      type = 'image'
                      dref = REF #( pict ) ) ).
```

Note
No empty parentheses can be specified after **REF**.

Lossless Operator EXACT

Description and use
The constructor operator **EXACT** executes either a lossless calculation or lossless assignment and creates the result with the data type **TYPE**, which could be a non-generic data type or the symbol #. In case of the symbol # the type is derived from the operand on the left side of the assignment operator.

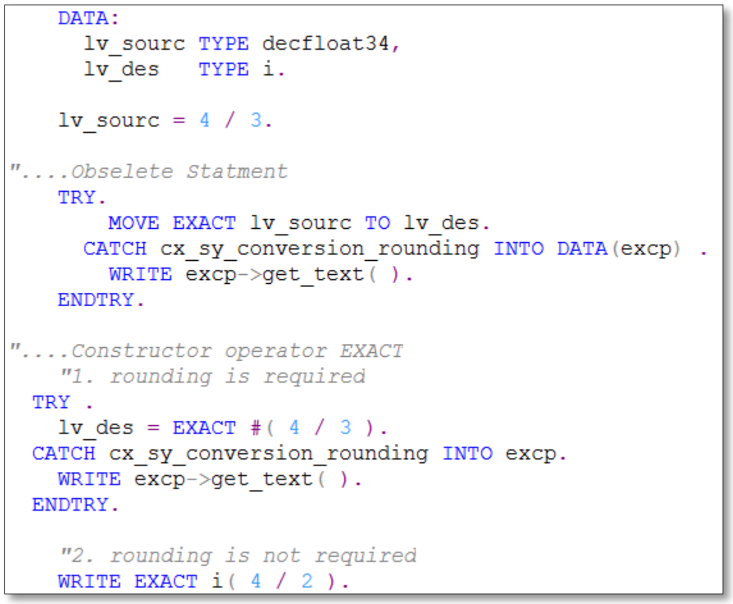
Lossless calculation: Arithmetic calculation that does not perform any rounding operations. It was introduced by **SAP** for decimal floating point numbers in Release **7.02** with key word **COMPUTE** with addition of **EXACT**.
The key word **COMPUTE** is obsolete and replaced by construction operator **EXACT**.

Lossless assignment: is an assignment between incompatible data type in which the conversion is checked to see whether data is lost. This was introduced by SAP in Release 7.02 with key word **MOVE** with addition of **EXACT**. This syntax is obsolete and is replaced by constructor operator **EXACT**.

Syntax and example

*... **EXACT** type ([dobj)...*

Here the argument ***dobj*** must be convertible to specified ***type***. In case ***dobj*** is an arithmetic expression, the expression is calculated in accordance with the rules for a lossless assignment and the result (with calculation type **DECFLOAT34**) is converted to the data type '***type***'.



```
    DATA:
      lv_sourc TYPE decfloat34,
      lv_des   TYPE i.

    lv_sourc = 4 / 3.

"....Obselete Statment
    TRY.
        MOVE EXACT lv_sourc TO lv_des.
      CATCH cx_sy_conversion_rounding INTO DATA(excp) .
        WRITE excp->get_text( ).
    ENDTRY.

"....Constructor operator EXACT
    "1. rounding is required
  TRY .
    lv_des = EXACT #( 4 / 3 ).
  CATCH cx_sy_conversion_rounding INTO excp.
    WRITE excp->get_text( ).
  ENDTRY.

    "2. rounding is not required
    WRITE EXACT i( 4 / 2 ).
```

In this example, the variable **LV_SOURC** is of type **DECFLOAT** and when assigned to variable **LV_DES** (integer type) the decimal part is ignored and only the integer part is accepted (loss of data). With key world **EXACT** the operation checks if any data would be lost. If yes, a catchable exception (**CX_SY_CONVERSION_ROUNDING**) is thrown.

The first part of the code snippet shows the obsolete statement **MOVE EXACT**.

Note

The lossless operator **EXACT** replaces the identically named addition of the obsolete statements **MOVE** and **COMPUTE**.

Condition Operator COND

Description and use

Condition operator could be read as IF statement. It constructs a result based on the logic expression coded in the syntax (conditional expression) of the **COND** operator. Like other constructor operators, **COND** operator also has *type* in its syntax. The result of the logical expression is converted to the mentioned type.

Syntax and example

```
        . . . COND type ( [let_expression]
  WHEN log_exp1 THEN [let_expression] THEN result1
[ WHEN log_exp2 THEN [let_expression] THEN result2 ]
. . .
[ ELSE [let_expression] resultn ]
```

```
DATA(time) = COND #( LET t = '120000' IN
          WHEN sy-timlo < t THEN
            |{ sy-timlo TIME = ISO } AM|
          WHEN sy-timlo > t AND sy-timlo < '240000' THEN
            |{ CONV t( sy-timlo - 12 * 3600 ) TIME = ISO } PM|
          WHEN sy-timlo = t THEN
            |High Noon|
          ELSE
            | test | ) .

WRITE: time.
```

Here, the variable **TIME** gets the value from the **THEN** part of the condition based on the **WHEN** clause. The addition **LET** is used to define variables as local auxiliary fields and assign values to them. In the example above **LET** defines variable t and assign value '**120000'** to it.

Condition Operator SWITCH

Description and use

SWITCH is conditional operator has a result that is specified by a case distinction. Constructor operator **SWITCH** can be read as **CASE** statement. It differs from **CASE** statement as **SWITCH** could be used as an operand.

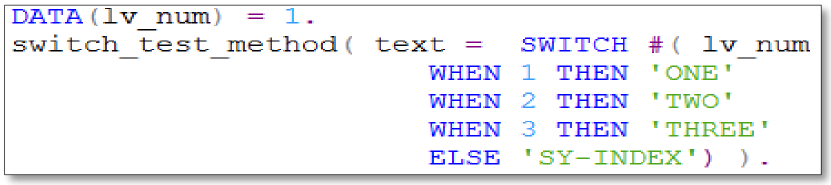
Note: *In computer programming, an operand is a term used to describe any object that is capable of being manipulated. For example, in "1 + 2" the "1" and "2" are the operands and the plus symbol is the operator*.

Syntax and example

```
SWITCH type( WHEN operand = const1 THEN result1
      [ WHEN operand = const2 THEN result2 ]
      ...
      [ ELSE resultn ] )
```

The type could be a non-generic data type or the symbol #.

```
CLASS-METHODS:
  switch_test_method IMPORTING text TYPE string.
```



```
DATA(lv_num) = 1.
switch_test_method( text =  SWITCH #( lv_num
                                WHEN 1 THEN 'ONE'
                                WHEN 2 THEN 'TWO'
                                WHEN 3 THEN 'THREE'
                                ELSE 'SY-INDEX') ).
```

In the example, the method **SWITCH_TEST_METHOD**, the importing parameter is of type **STRING**. The # symbol requires the result of **SWITCH** operation to be string. This makes it necessary the variable or expression after **THEN** statement to yield string type result.

Note

1. All operands specified after THEN must be convertible to the data type determined by type. In the case of reference variables, an up cast must be possible.

   Please refer to References section in the last page for additional resource links for ***constructor operator.***

Data Selection

General

There are many do's and don'ts for select statements. Please follow this section carefully.

- Use of **native SQL** must be justified and documented in the program.
- Use of **dynamic open SQL** is not recommended. If used, it must be justified and well documented in the program next to its use. Using dynamic code to Update Modify or Delete entries in a table can be extremely dangerous and lead to data loss or corruption. At the very least, ensure your dynamic variables are not blank in your condition, as this will bring back unwanted rows. Using an alternate technique to fulfill the requirement, such as using existing function modules, is highly preferred. Divisional review processes/teams must be used if moving forward with a use of **dynamic open SQL**.

  An example of **Dynamic Open SQL** is:

```
UPDATE (p_table)
   SET (l_set_expr)
 WHERE (l_condition).
```

- Use **SELECT SINGLE** whenever appropriate. If not sure about the complete key for using 'select single' but sure that only one row matches your selection criteria use 'select ... up to n rows' where n is a small number.
- Do not use **SELECT SINGLE** in a loop. Consider using an array fetch.
- Filter ***ASAP***, filter ***AMAP***, filter ***AOAP***.
  - As Soon As Possible - **WHERE** clause is used that invokes index(s).
  - As Much As Possible - **SELECT** only needed columns where the value is not already known.
  - As Often As Possible - more WHERE clause predicates even if un-indexed.
- Avoid Group by, Order by, Distinct, and like operations.
  - Until **HANA** is live on all systems, let the programs do the heavy lifting for sorting and grouping.
- Put the **AND**, **OR** and **NOT** in front of the condition to make it more readable.
- Use parenthesis to separate where conditions if it makes the document more readable. If an '**OR'** condition is used, the selection from the database may not be correct if it is put in the wrong order without parenthesis.
- In complex select statements, consider adding inline comments to the conditions to make it easier for future additions or debugging.

Structure

- Recommended Format (for readability):

```
SELECT matnr
  INTO TABLE i_cp_matnr
  FROM mkal
 WHERE bukrs = p_bukrs  “company code
   AND werks IN s_werks “plant
   AND verid = p_verid  “product version
   AND bdatu > sy-datum “after current date
   AND mdv01 = p_mdv01. “Production line
```

Performance

Performance problems can occur in both **SQL**, **ABAP** coding, and User Interface.

Database (SQL)

- **Filter.** Avoid retrieving rows that are later discarded – put all conditions into the **WHERE** clause unless performance is severely affected.

```
SELECT * FROM sflight
  INTO xflight
  WHERE carrid = 'LH ' AND connid = '0300'.
    CHECK xflight-fldate(4) = '2002'.
    WRITE: / xflight-fldate.
ENDSELECT.
```

✗

```
SELECT * FROM sflight
  INTO xflight
  WHERE carrid = 'LH '  AND
        connid = '0300' AND
        fldate LIKE '2002%'.
    WRITE: / xflight-fldate.
ENDSELECT.
```

✓

- **Index.** Frequently executed **SQL** or **SQL** on large tables needs to be supported by an index that provides highly selective access for the **WHERE** clause. Use the primary key when available.
- **Views.** For complementary that are often being read in conjunction, consider creating a view that contains the items from each table. This will need to be discussed in your area, but could make selecting items much easier and efficient.
- **Select *. Select only columns being used.** All columns are processed individually anyway by the database and the **SAP** interface, so nothing is saved by selecting un-needed columns.

```
SELECT * FROM sflight
  INTO xflight
  WHERE carrid = 'LH '  AND
        connid = '0300' AND
        fldate LIKE '2002%'.
    WRITE: / xflight-fldate.
ENDSELECT.
```

✗

```
SELECT fldate FROM sflight
  INTO (xflight-fldate)
  WHERE carrid = 'LH '  AND
        connid = '0300' AND
        fldate LIKE '2002%'.
    WRITE: / xflight-fldate.
ENDSELECT.
```

✓

- **Select --- INTO CORRESPONDING and APPENDING CORRESPONDING.**

It is more efficient to name the list of receiving fields.

- **OSS Notes 185530, 187906 and 191492, and 1516684**
    - Tables **VBAK, VBAP, VMVA, LIKP, LIPS, VBRK, VBRP, VBFA** and **VEPO** often need to be accessed by columns that are not indexed. **OSS Note 185530** shows how to use other **SAP** tables to find the primary keys.

    - Tables **MSEG, LTAP** and **EBAN** often need to be accessed by columns that are not indexed. **OSS Note 187906** shows how to use other **SAP** tables to find the primary keys.

    - Tables **AFRU, AFKO, AUFK, CAUFV** and **RESB** often need to be accessed by columns that are not indexed. **OSS Note 191492** shows how to use other **SAP** tables to find the primary keys.

    - **OSS Note 1516684** duplicates **MKPF** (material header) fields into **MSEG** (material item), thereby reducing the need for join operations and providing a significant performance improvement opportunity. This note already is (or could be) enabled in landscapes working with these tables.

      The following redundant fields are provided in **MSEG** after the note is applied: **VGART_MKPF, BUDAT_MKPF, CPUDT_MKPF, CPUTM_MKPF, USNAM_MKPF, XBLNR_MKPF, and TCODE2_MKPF.**

      Joins on **MKPF** like the following could be eliminated, and subsequent code could be changed easily to use the redundant **MSEG** fields instead:

```
FROM mseg AS a
    INNER JOIN mara AS b ON a~matnr = b~matnr
    INNER JOIN marc AS c ON a~matnr = c~matnr
                         AND a~werks = c~werks
    INNER JOIN MKPF AS d ON a~mblnr = d~mblnr
                         AND a~mjahr = d~mjahr
    INNER JOIN MBEW AS e ON a~matnr = e~matnr
                         AND a~werks = e~bwkey
    INNER JOIN t006 AS t ON b~gewei = t~msehi
```

- **ALL ENTRIES cautions**

  - When using "**FOR ALL ENTRIES IN**" and the "**INTO**" clause, you must use different internal tables.
    - *NOT ACCEPTABLE*

```
SELECT matnr vbeln
     INTO TABLE itab2
   FROM ebew
               FOR ALL ENTRIES IN itab2
               WHERE matnr = itab2-matnr.
```

Since the example above is selecting rows into the same table it is using as 'FOR ALL ENTRIES IN', it will get stuck in a loop as the table grows larger and larger

    - *ACCEPTABLE*

```
SELECT matnr vbeln
     INTO TABLE itab1
   FROM ebew
   FOR ALL ENTRIES IN itab2
   WHERE matnr = itab2-matnr.
```

  - If there are multiple condition columns in the **ALL ENTRIES** table, consider using a join or sub-select instead.

  - Before using a "for all entries" in a select, make sure:
    - ➢ That the (for all entries) internal table is not empty. This leads to a full table scan of the table being selected from.
    - ➢ The internal table is sorted into the order of the fields in the where clause.
    - ➢ That the internal table that is used does not contain duplicates by the key used in the where clause. In many cases, we cannot simply delete duplicates from the internal table because we need to consider those duplicates later in the program. Therefore, it is recommended that, unless specified in the technical spec, the program use a temporary table with the same layout and delete the duplicates from the temporary table.
    - ➢ Once the SQL is finished, the temporary table should be freed.

```
REPORT  z_test_remote_tran.
IF resb_tab [] IS NOT INITIAL.
  PERFORM get_material_info.
ENDIF.

*-----------------------------------------------------------------------------------
-----------*
*                                                              form  get_material_info
*
*  Fetch MRP controller (DISPO) for all materials selected in resb_tab    *
*-----------------------------------------------------------------------------------
-----------*
FORM get_material_info.
  get_matnr[ ] = resb_tab[ ].
  SORT get_matnr BY matnr werks.
  DELETE ADJACENT DUPLICATES FROM get_matnr COMPARING matnr werks.

  SELECT matnr werks dispo
     FROM marc
        FOR ALL ENTRIES IN get_matnr
  WHERE matnr = get_matnr-matnr
     AND werks = get_matnr-werks.

  FREE get_matnr.
```

```
ENDFORM.                        "get_material_info
```

Current Database Standards per SAP:

Efficient Database Programming with A

- **Leading columns of indexes**

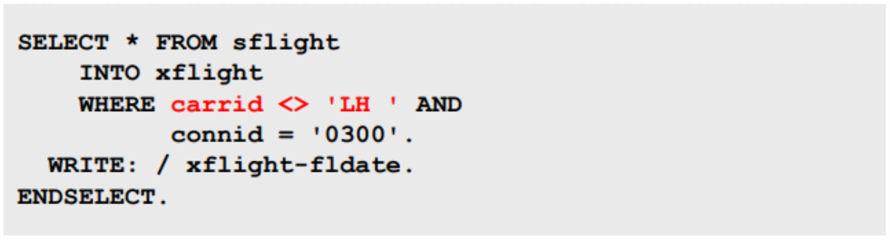
  - **DB2** can use an index to filter rows even if leading columns are not used in the **WHERE** clause. However, this will result in a full-index scan. In most cases, a full-index scan is better than a full-table scan.

  - In many cases, the leading column of an index might be **BUKRS, WERKS, VKORG** or other common, low cardinality data items. It would be worth determining the range of values (in a sub-select or **ALL ENTRIES)** to fill in the **WHERE** clause.

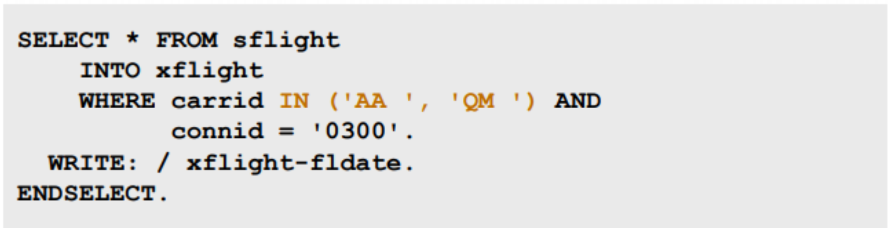
In the near future, our SAP systems will move to the HANA DB, please refer to HANA DB specific guidelines elsewhere in this document. The DB2 relevant sections are retained in this document based on the fact that there are still a few years left for the complete migration to HANA DB.

- **Negative conditions cannot be indexed**

  - Negative conditions cannot be used in index matching or filtering. Whenever possible, recast the condition in a positive way. Note: if the condition cannot be recast in a positive way, keep it in the **WHERE** clause. It is still better to do the filtering within the database than to return un-needed rows to **ABAP**.



```
SELECT * FROM sflight
    INTO xflight
    WHERE carrid <> 'LH ' AND
          connid = '0300'.
  WRITE: / xflight-fldate.
ENDSELECT.
```



```
SELECT * FROM sflight
    INTO xflight
    WHERE carrid IN ('AA ', 'QM ') AND
          connid = '0300'.
  WRITE: / xflight-fldate.
ENDSELECT.
```

  -

- **Conditions with leading wildcard cannot be indexed**

  Whenever possible, recast a leading wildcard as a list of values.

  Example:

  ```
  LIKE ‘%ABC’ might become
  IN (‘1ABC’, ‘2ABC’, ‘3ABC’)
  ```

  Note: if the condition cannot be recast as a list, keep it in the **WHERE** clause. It is still better to do the filtering within the database than to return un-needed rows to **ABAP**.

- **Avoid bypassing the SAP buffer**

  SAP memory buffering of database tables keeps much work off the database server. Candidate tables are 1) frequently accessed, 2) seldom changed and 3) small (or a small working set)

  - Explicitly **BYPASSING BUFFER**
  - **ORDER BY** (except by the primary key)
  - **GROUP BY**
  - Single-rows buffered but not using **SELECT SINGLE**
  - Generic buffering but full generic key not specified
  - **SELECT FOR UPDATE**
  - Aggregate functions (**SUM, AVG, MIN, MAX**)
  - **SELECT DISTINCT**
  - **Native SQL**
  - WHERE clause with **IS NULL**

- **SELECT --- ENDSELECT Loop**

  - **SELECT --- INTO TABLE** is more efficient. This is preferable except in those rare cases where processing must be done to each record as it is selected.

- **SELECT --- UP TO 1 ROWS**

  - This is **OK** to check for existence, but may be dangerous to use data retrieved (what if some other row had different values). This coding method can be used when you are sure there is only 1 occurrence of the value in the table or when the data retrieved does not matter, only the existence of that data matters.

- **SELECT --- Date Ranges**

  Frequently in **SAP**, **DATE** and **TIME** are stored in separate columns. Selecting a date-time range is then usually coded as:

  ```
  WHERE REC_DATE = LOW_DATE AND REC_TIME > LOW_TIME
    OR
  REC_DATE > LOW_DATE AND REC_DATE < HIGH_DATE
    OR
  REC_DATE = HIGH_DATE AND REC_TIME <= HIGH_TIME
  ```

  This coding gets the correct answer but is not index able due to the **OR's**. This can lead to tablespace scans.

  A more efficient method is to use separate **SELECT** statements and **APPEND** to the internal table. Using a **WHERE** clause that avoids "**OR**" allows for the index to be used which yields greater efficiency:

  For upper and lower date range, use:

  ```
  IF LOW_DATE < HIGH_DATE.
    SELECT ----
  ```

```
    INTO itab
    WHERE REC_DATE = LOW_DATE AND REC_TIME > LOW_TIME
  SELECT ----
    APPENDING itab
    WHERE REC_DATE > LOW_DATE and REC_DATE < HIGH_DATE
  SELECT ----
    APPENDING itab
    WHERE REC_DATE = HIGH_DATE AND REC_TIME <= HIGH_TIME


  SELECT ----
    INTO itab
    WHERE ( REC_DATE = LOW_DATE AND REC_TIME > LOW_TIME )
       OR ( REC_DATE > LOW_DATE AND REC_DATE < HIGH_DATE )
       OR ( REC_DATE = HIGH_DATE AND REC_TIME <= HIGH_TIME ).

ELSE. "(LOW_DATE equals HIGH_DATE)
  SELECT ----
    INTO itab
    WHERE REC_DATE = LOW_DATE
      AND REC_TIME > LOW_TIME
      AND REC_TIME <= HIGH_TIME
```

For open-ended upper range, use:

```
SELECT ----
  INTO itab
  WHERE REC_DATE = LOW_DATE
    AND REC_TIME > LOW_TIME
SELECT ----
  APPENDING itab
  WHERE REC_DATE > LOW_DATE
```

For open-ended lower range, use:

```
SELECT ----
  INTO itab
  WHERE REC_DATE < HIGH_DATE
SELECT ----
  APPENDING itab
  WHERE REC_DATE = HIGH_DATE AND REC_TIME <= HIGH_TIME
```

ABAP

- **General**
  - **MOVE-CORRESPONDING** is less efficient than coding a move for each field.

- **Loops**
  - Every statement within a loop should be acting on loop variables. If not, pull it outside the loop.
  - Use the statement '**LOOP AT i_tab WHERE'** only with a table of type **SORTED**. With other table types, this is a full internal table scan.
  - With a **STANDARD** internal table that has been sorted by an **ABAP SORT** statement, use **READ** with **BINARY SEARCH**. This avoids the full table search.
  - With a table of type **SORTED, READ** automatically uses binary search.

- For loops that require nested loops, these can be optimized by reading the inner loop table by the specified key to get the index. Once the index is found, loop at the table from the index.
  Instead of:
  ```
        LOOP AT resb_tab.
          LOOP AT matnr_tab WHERE matnr = resb_tab-matnr
                              AND werks = resb_tab-werks.
          ENDLOOP.
        ENDLOOP.
  ```
  Use:
  ```
        SORT matnr_tab BY matnr werks.
        LOOP AT resb_tab.
          READ TABLE matnr_tab WITH KEY matnr = resb_tab-matnr
                                        werks = resb_tab-werks
                               BINARY SEARCH TRANSPORTING NO FIELDS.
          IF sy-subrc EQ 0.
            t_tabix = sy-tabix.
            LOOP AT matnr_tab FROM t_tabix.
              IF ( matnr_tab-werks NE resb_tab-werks ) OR
               ( matnr_tab-matnr NE resb_tab-matnr ).
                EXIT.
              ENDIF.
            ENDLOOP.
          ENDIF.
        ENDLOOP. "end loop at resb_tab
  ```

- **Locking Issues**

  - Proper locking is essential to guarantee data consistency.

  - Minimizing lock duration is essential to prevent timeouts.
    - Move activities ahead of locking.
    - Release locks as soon as possible.

  - Standard sequencing of locking is essential to prevent deadlocks.

  - Use **SAP** Number Range Buffering for all number ranges unless prevented by legal requirements.

  - For each Number Range buffered, be sure that the quantity buffered is large enough to prevent multiple buffer refreshes per job.

- **Recursion**

  - Recursive processing (common with **BOM** explosions) **MUST** be guaranteed to end even if the data contains an endless loop.

- **Memory**

  - Large internal tables should be cleared as soon as possible using **FREE I_TAB**.

Scalability

Scalability refers to the change in performance (response time) as a result of changes in:
- Number of users
- Size of the database
- Size of the result set
- Size of the input

Scalability with Respect to the Number of Users (Workload)

Within broad limits, the number of users should not have any effect on performance. However, every transaction, report, function, etc. contributes to the overall workload and overall resource utilization. It is well known that high levels of resource utilization lead to degraded performance. Therefore, every inefficient program, in addition to hurting its own performance, causes some amount of slow down across the entire system.

This is also the reason to avoid **ORDER BY, GROUP BY** and **DISTINCT** in **SQL** statements. These transfer load from the application servers to the less scalable database resource.

This would still be continued until we use **HANA** as our underlying **DATABASE.**

Commit Frequency

Any program that holds database locks longer than a threshold (currently **600 seconds**, but we intend to drive this down) may cause other programs to incur a database timeout. For this reason, every long-running program should commit at least every **30 seconds**.

On the other hand, every commit consumes resources – do not commit too frequently. Try to commit every **10 – 30 seconds**.

Even read-only programs should be designed with a commit frequency. **DBA** tasks can be timed-out by read-only long-running programs.

Any update program with intermediate commits must address rollback as well. The program could be written to detect its last commit point and automatically restart. If this is not done, the user must be able to purge the committed input and then rerun the job.

SAP Locking Strategy

SAP locks (enqueue) can span multiple program steps. This means that other programs can be blocked for an extended period of time. The strategy is to delay taking SAP locks as long as possible and then releasing the SAP locks as soon as possible.

Scalability with Respect to the Size of the Database

Some programming techniques scale linearly with respect to the size of the database, such as full table scans or full index scans. **These must be avoided**; there are better techniques that scale logarithmically. Generically, this is why:

- Indexed access is preferred over full table scans for database access
- Binary search is preferred over full table scans for internal tables

Some programming techniques scale **even worse** than linearly with respect to the size of the database.

- Nested selects are less efficient than joins, sub-selects or **FOR ALL ENTRIES**.
    - Sub-selects are useful when no actual data is required from a table, only keys.
    - Joins work best for **1:1** or **few:1** situations.
    - **FOR ALL ENTRIES** works best for **many:1** situations to prevent the 1 from being retrieved many times.
- Nested loop joins (database) need index access on the inner table(s).
- Nested loops (internal tables) are always full table scans unless the inner table(s) are type **SORTED** tables.

Scalability with Respect to the Size of the Result Set and/or the Size of the Input

This is where linear scalability is the standard. If the program does more work, it is expected to take proportionately more time.

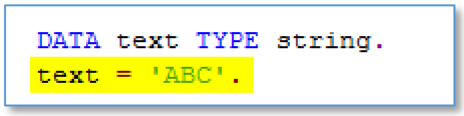
Unfortunately, some techniques lead to exponential scaling. In general, these involve re-processing prior items every time a new item is processed. Typically, this is caused by the failure to clear an internal table.

ABAP 7.4 Features – New Syntax & Keywords

Inline Declaration

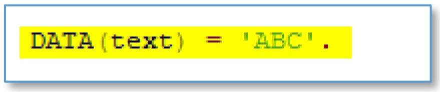
1. Data statement

Before 7.4



```
DATA text TYPE string.
text = 'ABC'.
```

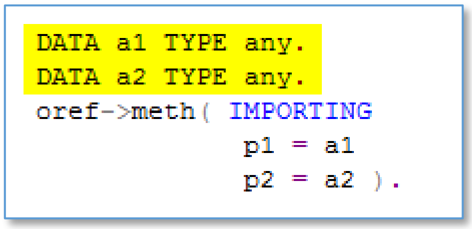
After 7.4



```
DATA(text) = 'ABC'.
```

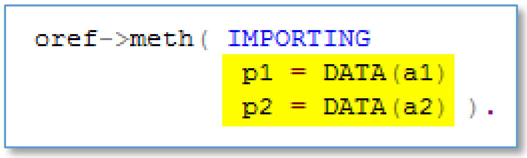
2. *Call method*

Before 7.4



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

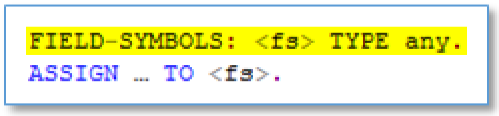
After 7.4



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

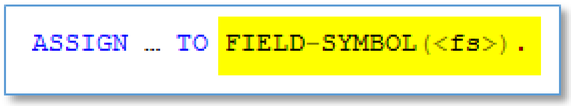
3. *Field Symbols*

Before 7.4



```
FIELD-SYMBOLS: <fs> TYPE any.
ASSIGN … TO <fs>.
```

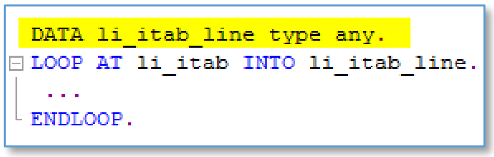
After 7.4



```
ASSIGN … TO FIELD-SYMBOL(<fs>).
```

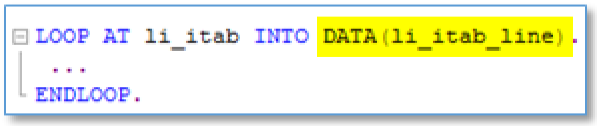
4. *Loop at into work area*

Before 7.4



```
DATA li_itab_line type any.
LOOP AT li_itab INTO li_itab_line.
  ...
ENDLOOP.
```

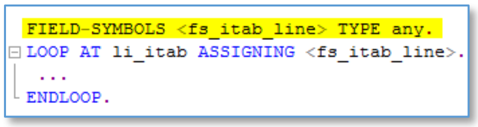
After 7.4



```
LOOP AT li_itab INTO DATA(li_itab_line).
  ...
ENDLOOP.
```

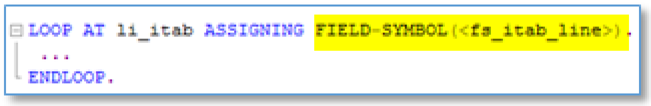
5. *Loop at assigning*

Before 7.4



```
FIELD-SYMBOLS <fs_itab_line> TYPE any.
LOOP AT li_itab ASSIGNING <fs_itab_line>.
  ...
ENDLOOP.
```

After 7.4

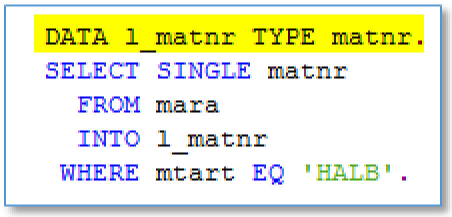


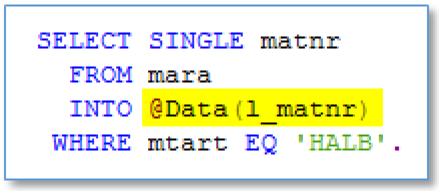
```
LOOP AT li_itab ASSIGNING FIELD-SYMBOL(<fs_itab_line>).
  ...
ENDLOOP.
```

6. *Select single into*

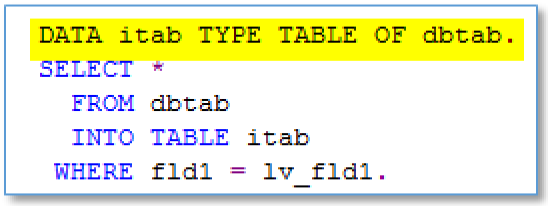
Before 7.4

After 7.4

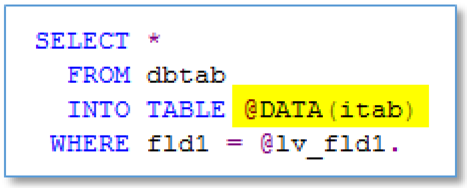

```
DATA l_matnr TYPE matnr.
SELECT SINGLE matnr
  FROM mara
  INTO l_matnr
 WHERE mtart EQ 'HALB'.
```


```
SELECT SINGLE matnr
  FROM mara
  INTO @Data(l_matnr)
 WHERE mtart EQ 'HALB'.
```

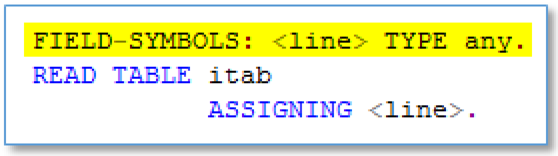
7. *Select into table*
   Before 7.4


```
DATA itab TYPE TABLE OF dbtab.
SELECT *
  FROM dbtab
  INTO TABLE itab
 WHERE fld1 = lv_fld1.
```

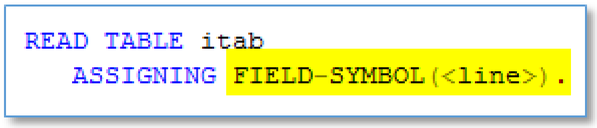
After 7.4


```
SELECT *
  FROM dbtab
  INTO TABLE @DATA(itab)
 WHERE fld1 = @lv_fld1.
```

8. *Read assigning*
   Before 7.4

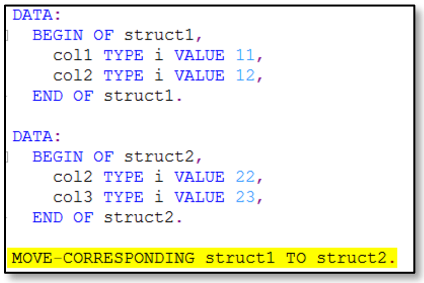

```
FIELD-SYMBOLS: <line> TYPE any.
READ TABLE itab
           ASSIGNING <line>.
```

After 7.4

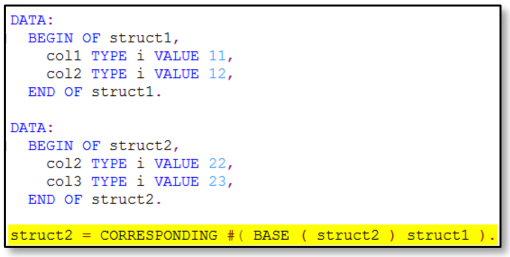

```
READ TABLE itab
   ASSIGNING FIELD-SYMBOL(<line>).
```

Internal Tables & Structures

1. *Structure Assignment*
   Before 7.4

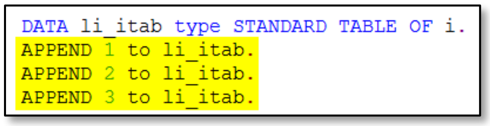

```
DATA:
  BEGIN OF struct1,
    col1 TYPE i VALUE 11,
    col2 TYPE i VALUE 12,
  END OF struct1.

DATA:
  BEGIN OF struct2,
    col2 TYPE i VALUE 22,
    col3 TYPE i VALUE 23,
  END OF struct2.

MOVE-CORRESPONDING struct1 TO struct2.
```

After 7.4

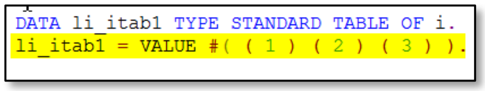

```
DATA:
  BEGIN OF struct1,
    col1 TYPE i VALUE 11,
    col2 TYPE i VALUE 12,
  END OF struct1.

DATA:
  BEGIN OF struct2,
    col2 TYPE i VALUE 22,
    col3 TYPE i VALUE 23,
  END OF struct2.

struct2 = CORRESPONDING #( BASE ( struct2 ) struct1 ).
```

2. *Internal table Initialization*
   Before 7.4

After 7.4

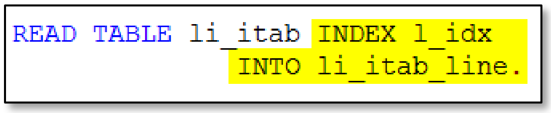


```
DATA li_itab type STANDARD TABLE OF i.
APPEND 1 to li_itab.
APPEND 2 to li_itab.
APPEND 3 to li_itab.
```

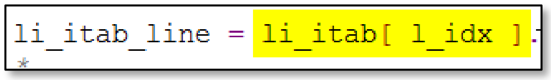


```
DATA li_itab1 TYPE STANDARD TABLE OF i.
li_itab1 = VALUE #( ( 1 ) ( 2 ) ( 3 ) ).
```

3. *Read table index*
   Before 7.4 After 7.4

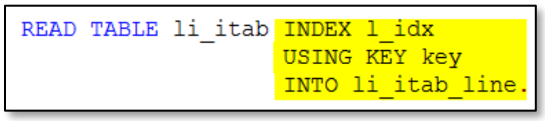


```
READ TABLE li_itab INDEX l_idx
                   INTO li_itab_line.
```

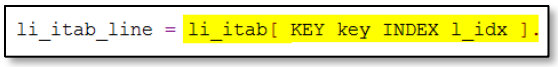


```
li_itab_line = li_itab[ l_idx ].
```

4. *Index access using a secondary index*
   Before 7.4 After 7.4

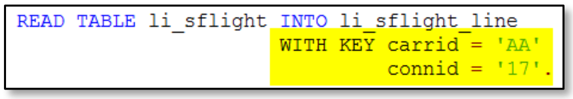


```
READ TABLE li_itab INDEX l_idx
                   USING KEY key
                   INTO li_itab_line.
```

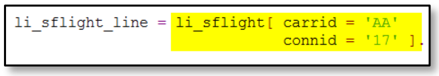


```
li_itab_line = li_itab[ KEY key INDEX l_idx ].
```

5. *Read table record using key*
   Before 7.4 After 7.4

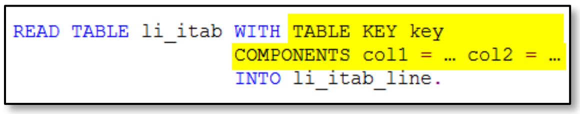


```
READ TABLE li_sflight INTO li_sflight_line
                      WITH KEY carrid = 'AA'
                               connid = '17'.
```

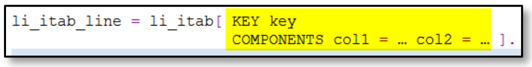


```
li_sflight_line = li_sflight[ carrid = 'AA'
                              connid = '17' ].
```

6. *Read table with key components*
   Before 7.4 After 7.4

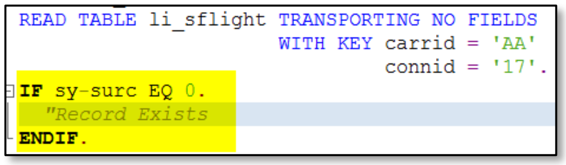


```
READ TABLE li_itab WITH TABLE KEY key
                   COMPONENTS col1 = … col2 = …
                   INTO li_itab_line.
```

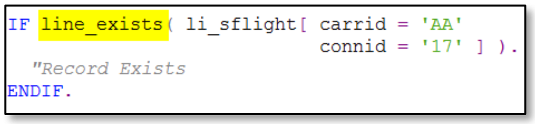


```
li_itab_line = li_itab[ KEY key
                        COMPONENTS col1 = … col2 = … ].
```

7. *Line/Record existence in internal table*
   Before 7.4 After 7.4

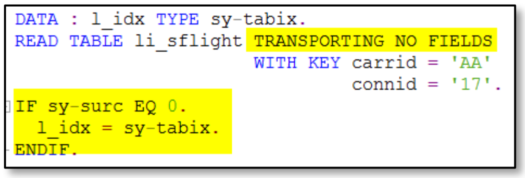


```
READ TABLE li_sflight TRANSPORTING NO FIELDS
                      WITH KEY carrid = 'AA'
                               connid = '17'.
IF sy-surc EQ 0.
  "Record Exists
ENDIF.
```

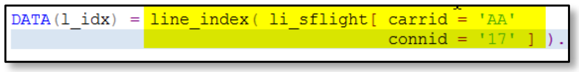


```
IF line_exists( li_sflight[ carrid = 'AA'
                            connid = '17' ] ).
  "Record Exists
ENDIF.
```

8. *Get the table index*
   Before 7.4 After 7.4



```
DATA : l_idx TYPE sy-tabix.
READ TABLE li_sflight TRANSPORTING NO FIELDS
                      WITH KEY carrid = 'AA'
                               connid = '17'.
IF sy-surc EQ 0.
  l_idx = sy-tabix.
ENDIF.
```



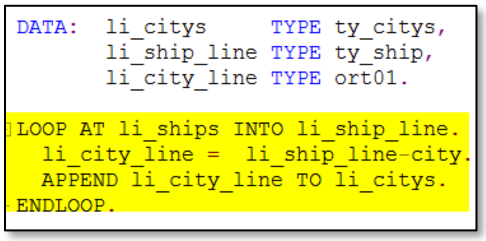
```
DATA(l_idx) = line_index( li_sflight[ carrid = 'AA'
                                      connid = '17' ] ).
```

NOTE:
If a table line is not found, the exception CX_SY_ITAB_LINE_NOT_FOUND is raised.

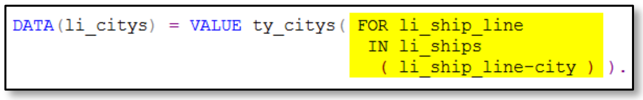
9. ***FOR** operator for iterating through the internal table*

Before 7.4

After 7.4



```
DATA:  li_citys      TYPE ty_citys,
       li_ship_line TYPE ty_ship,
       li_city_line TYPE ort01.

LOOP AT li_ships INTO li_ship_line.
  li_city_line =  li_ship_line-city.
  APPEND li_city_line TO li_citys.
ENDLOOP.
```

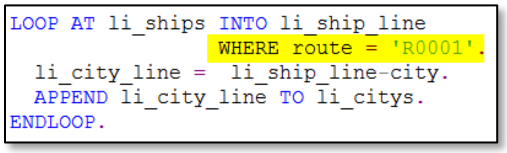


```
DATA(li_citys) = VALUE ty_citys( FOR li_ship_line
                                  IN li_ships
                                   ( li_ship_line-city ) ).
```

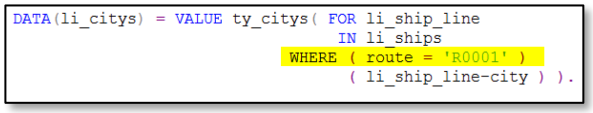
Before 7.4

After 7.4

With where condition in LOOP…ENDLOOP.



```
LOOP AT li_ships INTO li_ship_line
                 WHERE route = 'R0001'.
  li_city_line =  li_ship_line-city.
  APPEND li_city_line TO li_citys.
ENDLOOP.
```



```
DATA(li_citys) = VALUE ty_citys( FOR li_ship_line
                                  IN li_ships
                                 WHERE ( route = 'R0001' )
                                   ( li_ship_line-city ) ).
```

Program Modularization

- Event **START-OF-SELECTION** should be a series of perform statements or call class methods with in-line documentation briefly describing what the subroutine will do.
- Large processing blocks of code should be broken down into logical units and performed as subroutines/ class methods. Typically, no processing block should be more than **1-2 pages** (**60-120 lines**) long.
- If the same block of code is used in more than one program, the following should be considered:
  - ➢ Function module.
  - ➢ Include program.
  - ➢ Subroutine pool.
  - ➢ Global Class with methods

Forms/Class methods (either local or global) should be used in programs to help keep programs readable and keep logical units of work together.

Use **ABAP OOPS** concepts and Principle of Separation of concerns to write your code. A sample code snippet is embedded in this document.

sample_soc_report.txt

Error Checking

- If notification of errors is required, it is preferred that an email be sent to a distribution list.
- Error messages should be logged using **SLG1** logs as described in section application logging above.
- Always check the return code (sy-subrc).
- Case statements should always have a ‘when others’ condition.
- Case statements should always have a ‘when others’ condition.
- **COND** and **SWITCH** should always have **throw** statement to handle exceptions
- Capture all messages when possible. (i.e. open files, call transaction, etc.)
- Messages for errors should include return code for debugging purposes

If no return code check is done, there must be documentation as to why.

Usage of SAP delivered error messages

Don’t use generic messages that consist only of placeholders together with text-elements.

```
Not Acceptable: MESSAGE E899 (8A) WITH text-001 text-002 wa_matnr.
       Acceptable: MESSAGE E899 (8A) WITH ‘Error in selection from MARA’ (001)
                                      ‘Part number not found’ (002)
                                      wa_matnr.
```

This makes it really hard to locate the source of an error message and does not allow to attach long texts to an error message.

You can create new **Z**-message classes as needed or create new messages in existing **Z**-message classes, see naming convention extensions. If you create a new message without documentation you should select the “self-explanatory” check box.

Text Elements

The standard convention of text-elements should be followed.

- All text elements should be translated in standard SAP languages.
- All literal values for output should have a text element.
- The in-line method of defining text elements should be used when possible. Example: 'End of File'**(001).**
- If the in-line method of defining text elements is used, use the adjust utility under the text elements to check the consistency of the text elements.
- Suggestion: Use the first character of the text element to differentiate between types:

Examples:	Enn = Errors
	Lnn = Print line texts
	Cnn = Column headers
	Hnn = Report headers
	Snn = Selection screen texts
	Dnn = Download texts
	Inn = Informational messages
	Pnn = Popup texts

Security

Appropriate authority-checks should be used based upon security needs.

Transaction codes should be created for all programs. Authority checks for transaction codes must be made in the **INITIALIZATION** section of a program. Authority checks for SAP-delivered programs are not required; however if this functionality is desired it can be implemented via an enhancement or custom report that wrappers the SAP-delivered program (and has very little function other than to perform the necessary authority checks).

```
INITIALIZATION.
  SELECT tcode UP TO 1 ROWS
    FROM tstc
    INTO g_tcode
    WHERE pgmna = sy-repid.
  ENDSELECT.

IF sy-subrc NE 0.
  MESSAGE a001(zsfx) WITH sy-repid.
ENDIF.

AUTHORITY-CHECK OBJECT 'S_TCODE'
            ID 'TCD' FIELD g_tcode.

IF sy-subrc NE 0.
  MESSAGE e002(zsfx) WITH g_tcode.
ENDIF.
```

Organizational security checks should be done in the **AT SELECTION SCREEN** area, by using the custom security objects for each functional area:

- FI:
  - Z_FI_CCODE (company code)
  - Z_FI_PLANT (plant)
- CO:
  - Z_CO_CCODE (company code)
  - Z_CO_PLANT (plant)
- MM:
  - Z_MM_CCODE (company code)
  - Z_MM_PLANT (plant)

    - Z_MM_PORG (purchase organization)
- PP:
    - Z_PP_CCODE (company code)
    - Z_PP_PLANT (plant)
- SD:
    - Z_SD_CCODE (company code)
    - Z_SD_PLANT (plant)
    - Z_SD_SORG (sales org)
    - Z_SD_SHIP (shipping point)

Each of these objects has the fields ACTVT (activity) and TCD (transaction) and one of the fields BUKRS, WERKS, VKORG, EKORG or VSTEL depending on org level.

An example of this coding is as follows:

```
AT SELECTION-SCREEN ON p_werks.
* Validation for Plant SELECT SINGLE WERKS
  SELECT SINGLE werks
    FROM t001w
    INTO g_werks
    WHERE werks EQ p_werks.
  IF sy-subrc EQ 0.
*   Authorization check for the plant
    AUTHORITY-CHECK  OBJECT 'Z_MM_PLANT'
           ID 'ACTVT' FIELD c_act_auth_03
           ID 'WERKS' FIELD g_werks
           ID 'TCD' FIELD g_tcode.
    IF sy-subrc NE 0.
      MESSAGE e004(co) WITH g_werks.
    ENDIF.                            " IF SY-SUBRC NE 0.....
  ELSE.
    MESSAGE e174(q3) WITH g_werks.
  ENDIF.                                                  "  IF  SY-SUBRC  EQ  0.....
```

NOTE:

- *Use of variables and constants for the OBJECT and ID sections should be avoided.*
- *Use minimal authorization for the requirements defined. For instance use activity ‘03’ when only display access is needed. Common activity types are:*
    - *01 = create*
    - *02 = change*
    - *03 = display*
    - *06 = delete*

For these new objects there are these common error messages predefined in message-class ZSFX that should be used:

003	You are not authorized to use this tran for plant &
004	You are not authorized to use this tran for company &
005	You are not authorized to use this tran for sales org. &
006	You are not authorized to use this tran for purch. org &

All security checks (standard and custom) must be added by **SAP Security** into tran **SU24**. Resulting transports will be assigned to the appropriate functional security agent who will be responsible for moving it across their respective landscape.

Function calls

If a program consists of function calls (includes **SAP** function calls), which, in turn, includes other auth-checks other than those in the main program, **SAP Security** should be made aware of these extra auth-checks. This information is required to ensure complete and proper linking of auth-objects to transactions. If not sure of what auth-checks are brought in through a function call, a security trace (**ST01**) should be performed during the testing stage.

The preferred method of communication for this information would be in the technical specs for newly developed transactions / programs.

Table Access

All tables that are required for display or maintenance should be assigned to an appropriate auth-group. Requests for new table auth-groups must be submitted to SAP Security. Creating table auth-groups will be performed by **SAP Security**. Assignment of tables into an auth-group will be performed by the **AD group**.

Two proven methods of providing table access to users:

- Z-transactions which uses the ZZ: TABLMNP auth-check to provide access to a specific table.
- Transaction ZTAB which restricts access to a group of tables (S_TABU_DIS), and then further restriction to a specific table (ZZ: TABLMNP) within a group of tables. This transaction should be used mainly for tables that are maintained by SAP support groups not the users since there is no specific authority checks being done for company code, plant, etc.

Directory Traversals

Physical file names can be specified as the content of a character-like data object in the statements and system class of the **ABAP** file interface. If some or all of this content originates outside of the calling program, there is a risk that files or file paths are accessed by unauthorized sources (this is known as directory traversal). The following are potential security risks when using input from outside to access the **ABAP** file interface:

- A file name used in the statements **OPEN DATASET** and **DELETE DATASET** originates either partly or in full from outside the program.
- A file name passed to the method **CREATE_UTF8_FILE_WITH_BOM** of the system class **CL_ABAP_FILE_UTILITIES** originates either partly or in full from outside the program.

  To act against this security risk, the file names must be validated. There are two known ways of validations:
    - It can be a self-programmed validation Using **cl_abap_dyn_prg=>check_whitelist_tab ( )** method against a set of valid file names (hardcoded/maintained in table).
    - The function module **FILE_VALIDATE_NAME** can be used. This function module checks whether a physical file name matches a logical file name or whether it is a valid directory. One prerequisite is that the matching file names or logical paths were created using the transactions **FILE** or **SF01**.

NOTE:

- If a program uses logical file names exclusively, instead of physical file names, the physical file names or paths required by the statements are constructed using the function module **FILE_GET_NAME** only. In this case, validation is not usually necessary.
- Alongside the validation of file names, adequate checks should be made on the authorizations for file access.

If your program uses any kind of file name that originates from outside the program, then it is at a potential risk for directory traversal.

- If you use any **DATASET** commands on filenames **OR**
- If you use Class **ABAP** File utilities to handle file data

Then you must validate the file name before using it in the program. This can be done in two ways

- Create a Logical File name for the Physical File path and use only Logical File name in the program. Then use **FILE_GET_NAME** to get the actual physical file at runtime

- If you must use Physical filenames in the program then create a Logical Filename of expected file names using **transaction FILE**. Then use **FILE_VALIDATE_NAME** to verify input against allowed filenames. ONLY process those that are valid.

  Along with these validations you should also perform adequate authorization checks against the data being processed through the files.

SQL Injection

If dynamically specified database tables for the **SELECT** Statements originate in full or in part from outside the program, users could potentially access databases for which they usually do not have authorization. If the use of external input in dynamically specified database tables is unavoidable, the input must be properly checked. The class **CL_ABAP_DYN_PRG** can be used to make a comparison with a whitelist.

```
TRY.
    dbtab =
      cl_abap_dyn_prg=>check_table_name_str(
        val = to_upper( dbtab )
        packages = 'SAPBC_DATAMODEL' ).
  CATCH cx_abap_not_a_table cx_abap_not_in_package.
    cl_demo_output=>display( 'Wrong input' ).
    LEAVE PROGRAM.
ENDTRY.
```

In the above snippet the variable dbtab contains the table name which could be compared to the hardcoded package(datamodel) to ensure the program is accessing data only from the intentional tables.

- If the dynamically specified table columns in the **SELECT** list of the statement **SELECT** originate fully or in part from outside the program, users could potentially access table columns for which they usually do not have authorization. Users could also rename columns without permission or use aggregate functions to perform unauthorized calculations. If the use of external input in a dynamically specified table columns is unavoidable, the input must be properly checked. For example, the class **CL_ABAP_DYN_PRG** can be used to make a comparison with a whitelist.

```
TRY.
    comp =
      cl_abap_dyn_prg=>check_whitelist_tab(
        val = to_upper( comp )
        whitelist = VALUE string_hashed_table( ( `CITYFROM` )
                                               ( `CITYTO` ) ) ).
  CATCH cx_abap_not_in_whitelist.
    cl_demo_output=>display( 'Not allowed' ).
    LEAVE PROGRAM.
ENDTRY.
```

  In the above code snippet the variable comp could contain the dynamic field and so is checked against hardcoded values of Column names from the intended table (**SPFLI**).

- Dynamic Where Clause: Use **CL_ABAP_DYN_PRG**=>quote method to verify the dynamic where clause. Let's consider the below snippet where the value of the where clause is getting generated from outside the program.

```
PARAMETERS: column   TYPE c LENGTH 30 DEFAULT 'CARRID',
            value  TYPE c LENGTH 30.
```

```
AT SELECTION-SCREEN.
  TRY.
      cl_abap_dyn_prg=>check_column_name( column ).
  CATCH cx_abap_invalid_name.
      MESSAGE 'Not allowed' TYPE 'E'.
  ENDTRY.

START-OF-SELECTION.

  DATA(cond_syntax) = column && ` = '` && value && `'`.

  TRY.
      SELECT *
             FROM spfli
             WHERE (cond_syntax)
             INTO TABLE @DATA (spfli_tab).
  CATCH cx_sy_dynamic_osql_error.
      MESSAGE `Wrong WHERE condition!` TYPE 'I'.
  ENDTRY.
```

If this code was written with malicious intent then what would be the value of cond_syntax if value = AA' AND CARRID <> 'AA. Your code is hacked, you retrieve all the contents of the table. :(

How do we stop it?   We can do this by using One statement change as shown below:

```
 DATA(cond_syntax) = column && ` = '` && cl_abap_dyn_prg=>quote( value ) && `'`.


 TRY.
      SELECT *
             FROM spfli
             WHERE (cond_syntax)
             INTO TABLE @DATA (spfli_tab).
    CATCH cx_sy_dynamic_osql_error.
      MESSAGE `Wrong WHERE condition!` TYPE 'I'.
 ENDTRY.
```

Cl_abap_dyn_prg=>quote puts a `'` at the start and end of the parameter passed to it. It makes the cond_syntax variable an incorrect syntax which results in runtime error.

Dynamic Calls

In order to avoid the Dynamic Calls metioned above, in order to avoid any transaction/program not to be called unintentionally, the best way is to have a set of allowed programs/transactions and checking the input against it. As mentioned earlier it could be done using the cl_abap_dyn_prg=>check_whitelist_tab( ) method.

In addition to that with 7.5 we could utilise the WITH AUTHORITY-CHECK clause in CALL Transaction.

Check the below snippet:

```
DATA whitelist TYPE HASHED TABLE OF string
               WITH UNIQUE KEY table_line.
SELECT obj_name
       FROM tadir
       WHERE pgmid    = 'R3TR' AND
```

```
             object   = 'TRAN' AND
             devclass = 'SABAPDEMOS'
       INTO TABLE @whitelist.

DATA transaction TYPE sy-tcode.
cl_demo_input=>request( CHANGING field = transaction ).

TRY.
    transaction = cl_abap_dyn_prg=>check_whitelist_tab(
      val = transaction
      whitelist = whitelist ).
  CATCH cx_abap_not_in_whitelist INTO DATA(exc).
    cl_demo_output=>display( exc->get_text( ) ).
    LEAVE PROGRAM.
ENDTRY.

TRY.
    CALL TRANSACTION transaction WITH AUTHORITY-CHECK.
  CATCH cx_sy_authorization_error ##NO_HANDLER.
ENDTRY.
```

Background Programs

- Success messages should be output for progress tracking in the job log.
- GUI_UPLOAD and GUI_DOWNLOAD cannot be used in background jobs.
- If a program should not be run in background, check **sy-batch** not = ‘**X**’.

Update Programs

SAP tables should never be updated directly. When writing programs that need to update SAP tables, the method of update should be in the following priority:

- Find BAPI, BAdI or function module.
- Find an IDoc that can be used.
- Do not use BDC methods any more except the code which already exists.
- Below points is applicable only to the existing code.
    - Use ‘call transaction’, use include ZSSVL_BDCINCLUDE.
    - BDC programs may be SAP version specific. Sometimes the screen fields change.
- Find Direct Input program (caution: some error processing is bypassed, and change logs are not always consistently updated).

Update program should have the ability to be rerun or restarted in case of failure.

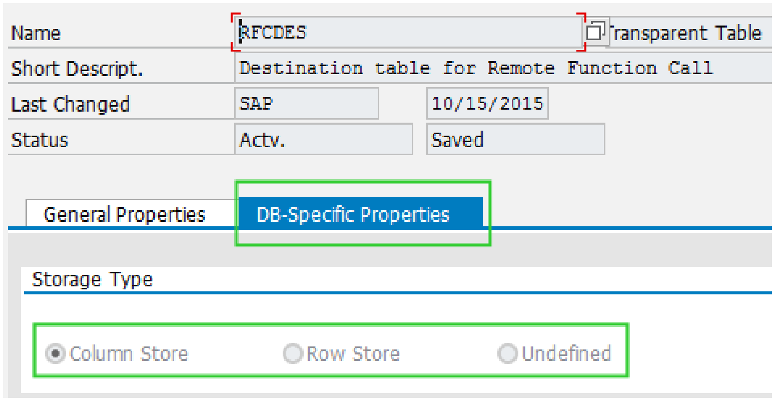
- Restart point detectable.
  Or
- One single unit of work.

Data Dictionary Objects

Table Design

- If you find the need to extend a SAP delivered table, you may do so using an APPEND or an INCLUDE. These can only be created in the customer namespace ZZ and will ensure that the extensions are not lost during an upgrade
- Data retention needs (keep online vs. purge vs. archive) should be discussed and documented.
- Appropriate purge or archive programs if needed should be written.

- Design should be reviewed by authorized reviewer.
- The table's technical attributes must be defined appropriately. This includes delivery class and size category.
- New tab added in technical setting as “DB-Specific Properties”
- This setting is only relevant when the underlying database is HANA.
- "The storage type only applies if the current database is an SAP HANA database. The specified storage type is ignored by other database systems. In this case, the platform-specific storage type is used."



Name	RFCDES	Transparent Table
Short Descript.	Destination table for Remote Function Call	
Last Changed	SAP	10/15/2015
Status	Actv.	Saved

Storage Type

[✓ Column Store] Row Store Undefined

- Consider adding documentation using the SAP table documentation functionality (go to → Documentation within the ABAP dictionary)
- Commonly used and rarely changing data should be buffered. Data that changes frequently and is not used very commonly should not be buffered.
- Check if buffering should be switched on with authorized reviewer, typically this should be switched on for all control / customizing type tables. Tables with only a few entries should be 'fully buffered', for larger tables with direct access specify 'single records buff'. For tables with generic access check 'generic area buffered' and specify the number of key fields.
- Custom developed tables will be maintained via a custom developed table maintenance program (copy and modify template ZSSVT_USR_TBL_MAINTENANCE).
- Custom tables may also be maintained via transaction ZTAB, however this provides less security than the method recommended above and should be used only if the SAP support team will be maintaining the table.
- All custom tables must be assigned to a custom table authorization group, never use SAP authorization groups. Currently these groups are defined.
- For all the control / customizing tables that are either maintained via ZTAB or via a custom maintenance transaction, the flag 'log data changes' must be set. This is required for audit purposes.

Data Elements and Domains

- The table must include the client (mandt) field due to SOX requirements
- Each field should use either a standard SAP data element or a defined created data element.
- Generic data elements such as CHAR1 should not be used. Since semantic information for the field such as text labels are contained in the data element, it needs to be as specific as possible.
- Each data element should use either a standard SAP domain or a defined created domain.
- Add documentation to data elements when the element will be visible to an end user.
- Consider adding documentation domains.
- Generic domains in the absence of existing suitable domains are acceptable.
- Field names should be as descriptive as possible.
  - Use SAP field names when appropriate.
- If available, Foreign Keys should be established.
- If appropriate, Value Tables should be established.

Indexes

- Since indexes cause additional writes during creates/updates/deletes (once for the table, another time for each index), indexes should only be created as needed. Try to use the indexes that exist whenever possible.
- Prior to defining a new index, review with a DBA.
- Primary key should be defined to guarantee uniqueness.
- Table added to proper lock object, or lock object should be created.
- No existing index is reused.
- An index only contains the necessary fields.
- The fields of the new index are in best order, the most selective first.
- The index contains the fields to make retrieval 'index only'.
- Only direct entry type data fields are defined without a data element.

Views

- Consult a DBA when creating or changing a view.
- Consider performance when using views.

ABAP List Viewer (ALV)

- There are many ALV frameworks available in SAP now:

Type	**Class / Function Module**
ALV List	REUSE_ALV_LIST_DISPLAY
Hierarchical sequential list	REUSE_ALV_HIERSEQ_LIST_DISPLAY
Full screen grid	REUSE_ALV_GRID_DISPLAY
ALV Grid Control	CL_GUI_ALV_GRID
ALV Tree	CL_GUI_ALV_TREE
From Basis 6.4 and onwards use	
Simple, two-dimensional tables	CL_SALV_TABLE
hierarchical sequential tables	CL_SALV_HIERSEQ_TABLE
Tree structures	CL_SALV_TREE

- Use the SALV reporting framework to show your data in ALVs when possible.
- SALV framework does not require you to declare a Dynpro or a field catalog
- Sorting, coloring, subtotaling etc. should be done in the SALV framework and not in the program code as this provides more flexibility
- Refer SAP Note 551605 - ALV FAQ - general information

Forms

- Copy Smart Forms or SAP Script, if available, to Z forms.
- Original language for all forms should be English
- Do not copy and modify the SAP delivered Print program or Application Program. Instead use a Perform program using the Perform command (SAP Script), or use Program Lines (SmartForm).
    - Copy/Modify of a SAP program will result in a loss of SAP support for that program. Any code fixes, OSS Notes, SAP package updates and new functionality will be applied only to the original program not the created/modified program. To take advantage of these changes, the original program then would have to be copied, modified and tested again after application of

the updates. Through previous experience we have validated each of these issues and experienced great pain and effort to return to the SAP delivered print program.

- The standard program documentation ZSSVT_PROGRAM_DOCUMENTATION must be filled out and inserted at the beginning of the MAIN window in the original language of the Form. Indicate its print program or Application Program, the IMG transaction where it's configured, any ABAP programs it may call and SAPs original form name it was copied from.
- Standard Logo is in top left corner.
- 'Test Print Only!" logic is added to print at top of page on non-production clients.
- Form name is printed at top of page (few exceptions).
- Changes have been applied to every language of the Form unless specified differently.
- Code is written to handle different companies and plants on a client (few exceptions).
- Code is documented.
  - Form check performed
  - SAP Scripts -> Form - Check - Texts - Copy.
- SmartForms -> Activate the form.
- Comments are added when a unique condition does not follow the normal course of logic.
- When retrieving a company address, pull it from the CAM or from a table instead of creating a standard-text.
- Every defined variable is commented to indicate its purpose, especially flags.
- There is only one ABAP 'Perform' program for each SAP Module. If an additional function is needed it should be added to the already existing perform program. These are named z<appcode>sv_scrpfrm and are called from the SAPScript
- SmartForms copied from SAP standard – Do not change the interface of the form.
- SmartForms – do not use Complex Sections.
- Form names should be name_a4 for European paper and name_ltr for US 8 ½ X 11.
- In a Smart Form, every node has a description (called 'meaning'). When you create a new node, SAP defaults the description to New Window 1, New Line 1, etc. A meaningful description is to be used instead of leaving the default text.

BAPIs

- When enhancing SAP supplied BAPIs, all extensions should be made using the ExtensionIn and ExtensionOut parameters based on the data structure called BAPIPAREX in the data dictionary.
- Every BAPI must have a return parameter that is either an export parameter or table.
- All BAPI interface parameters should be non-scalar and reference data structures in the data dictionary even if the parameter consists of one field.
- DO NOT use appends or includes in BAPI data structures.
- DO NOT use exceptions in BAPI function module interfaces.
- Use the function modules BAPI_CURRENCY_CONV_TO_INTERNAL and BAPI_CURRENCY_CONV_TO_EXTERNAL to convert currency amounts.
- When using instance methods, the business object's key fields should be required import parameters in the BAPI function module interface.
- When using create methods, the business object's key fields should be export parameters in the BAPI function module interface.
- Key parameters/fields should have the same name as key fields for the business object.
- BAPI function modules MUST BE RFC enabled.
- A BAPI cannot cause program termination, it can only return the corresponding message (message type A, termination message) in the return parameter.
- BAPIs SHOULD NOT use call transactions, unless a strong business case dictates otherwise.
- BAPIs SHOULD NOT execute an explicit or implicit commit-work, unless a strong business case dictates otherwise.
- When using BAPIs, use BAPI_TRANSACTION_COMMIT if the BAPI was successful and the BAPI_TRANSACTION_ROLLBACK if the BAPI was not successful.
- The BAPI return parameter should be based on the data structure BAPIRET2.

- All currency fields should use the BAPICURR or BAPICUREXT domain.
- If both an internal and external keys are stored in the database, then the external key must be used for the BAPI interface.
- Create specific data structures for BAPIs that are independent of the data structures that are generally used in R/3 applications.
- For ISO related fields (currency, language, quantity and country) use SAP additional ISO codes when necessary.

Data Transfer

The landscapes use a variety of methods to transfer data. The specific method of data transfer will be explained in the Functional and Technical Specification document.
Care should be taken to ensure that outbound test files are not used in production systems. The best way to do this is to include the sysid as part of the file name. Another alternative is to append .test to the filename when running in non-production clients. In this case, use the Z_SSV_GET_CLIENT_CATEGORY function module to determine if the system is test or production.

ABAP on HANA

Optimization of custom code for HANA
CDS – Core Data Services
AMDP – ABAP Managed Database Procedure

OSS Note 1912445 - ABAP custom code migration for SAP HANA - recommendations and Code Inspector variants for SAP HANA migration

For New Developments:

- If the requirement fits the ABAP for HANA coding using CDS view or AMDP
  - Using SQL to push code to data base to retrieve data can significantly reduce the runtime of ABAP program. But it has limitation. A SQL statement always has one result set and the calculation must be done in one single step.
- Review when creating custom table to decide if it should be defined as row or column Table.
- ABAP Test cockpit to be used as code review process to run code inspector check with variant FUNCTIONAL_DB and PERFORMANCE_DB.

For already developed objects:

- Run ATC check and work on fixes on priority/ availability.
- Monitoring custom code performance in production/ test system by running ATC in batch job periodically and reviewing the result.

To identify objects for AMDP or CDS

AMDP- If you have complex calculations which you want to execute on the database, then choice is to implement ABAP managed database procedure .Calculations executed while retrieving the data from the database can significantly reduce the runtime of an ABAP program, especially when the calculations can be parallelized.

<u>Limitations</u>

The parameters have to meet the following prerequisites:
- Exporting, importing and changing parameters are allowed

- Methods with returning parameters cannot be implemented as AMDPs
- Method parameters have to be tables or scalar types
- Method parameters have to be passed as values

ABAP for HANA
next steps.xlsx

Helpful document: http://scn.sap.com/docs/DOC-51612

Core Data Service- if you have a need for reusable view with logic for selecting data , CDS is the right solution.

Helpful documents

http://scn.sap.com/community/abap/blog/2014/10/10/abap-news-for-740-sp08--abap-core-data-services-cds

Also please take a look in this location, directory Run_S4H, filename S4H231 which shows you more options for ABAP on HANA.

References:

Constructor Operator

1. https://blogs.sap.com/2013/05/27/abap-news-for-release-740-constructor-operator-value/
2. https://help.sap.com/http.svc/rc/abapdocu_751_index_htm/7.51/en-US/abenconstructor_expression_value.htm
3. https://help.sap.com/doc/abapdocu_750_index_htm/7.50/en-US/abenconstructor_expression_conv.htm
4. https://blogs.sap.com/2013/05/28/abap-news-for-release-740-constructor-operators-cond-and-switch/
5. https://www.computerhope.com/jargon/o/operand.htm
6. https://help.sap.com/http.svc/rc/abapdocu_751_index_htm/7.51/en-US/abenlossless_move.htm
7. http://zevolving.com/2014/09/abap-740-new-operator-create-itab-entries/
8. http://zevolving.com/2014/08/abap-740-new-operator-instantiate-objects/
9. https://blogs.sap.com/2013/05/27/abap-news-for-release-740-constructor-operators-conv-and-cast/