FREE eBook

LEARNING ABAP

Free unaffiliated eBook created from
Stack Overflow contributors.

#abap

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: abap

It is an unofficial and free ABAP ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official ABAP.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with ABAP

Remarks

ABAP is a programming language developed by SAP for programming business applications in the SAP environment.

Previously only procedural, ABAP is now also an object-oriented language thanks to the ABAP Objects enhancement.

Versions

Version	Release Date
ABAP 7.50	2015-10-20
ABAP 7.40	2012-11-29
ABAP 7.0	2006-04-01
ABAP 6.40	2004-04-01
ABAP 6.20	2002-04-01
ABAP 6.10	2001-07-01
ABAP 4.6C	2001-04-01
ABAP 4.6A	1999-12-01
ABAP 4.5	1999-03-01
ABAP 4.0	1998-06-01
ABAP 3.0	1997-02-20

Examples

Hello World

```
PROGRAM zhello_world.
START-OF-SELECTION.
    WRITE 'Hello, World!'.
```

Instead of printing to the console, ABAP writes values to a list which will be displayed as soon as the main logic was executed.

Hello World in ABAP Objects

```
PROGRAM zhello_world.

CLASS main DEFINITION FINAL CREATE PRIVATE.
  PUBLIC SECTION.
    CLASS-METHODS: start.
ENDCLASS.

CLASS main IMPLEMENTATION.
  METHOD start.
    cl_demo_output=>display( 'Hello World!' ).
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
  main=>start( ).
```

Read Getting started with ABAP online: https://riptutorial.com/abap/topic/1196/getting-started-with-abap

Chapter 2: ABAP GRID List Viewer (ALV)

Examples

Creating and Displaying an ALV

This example portrays the most simple ALV creation using the `cl_salv_table` class and no additional formatting options. Additional formatting options would be included after the `TRY ENDTRY` block and before the `alv->display( )` method call.

All subsequent examples using the ABAP Objects approach to ALV creation will use this example as a starting point.

```
DATA: t_spfli       TYPE STANDARD TABLE OF spfli,
      alv           TYPE REF TO cl_salv_table,
      error_message TYPE REF TO cx_salv_msg.

" Fill the internal table with example data
SELECT * FROM spfli INTO TABLE t_spfli.

" Fill ALV object with data from the internal table
TRY.
    cl_salv_table=>factory(
      IMPORTING
        r_salv_table = alv
      CHANGING
        t_table      = t_spfli ).
  CATCH cx_salv_msg INTO error_message.
    " error handling
ENDTRY.

" Use the ALV object's display method to show the ALV on the screen
alv->display( ).
```

Optimize ALV Column Width

This example shows how to optimize the column width so that column headings and data are not chopped off.

```
alv->get_columns( )->set_optimize( ).
```

Hide Columns in an ALV

This example hides the `MANDT` (client) field from the ALV. Note that the parameter passed to `get_column( )` *must* be capitalized in order for this to work.

```
alv->get_columns( )->get_column( 'MANDT' )->set_visible( if_salv_c_bool_sap=>false ).
```

Rename Column Headings in an ALV

The column text may change upon the horizontal resizing of a column. There are three methods to accomplish this:

Method Name	Maximum Length of Heading
`set_short_text`	10
`set_medium_text`	20
`set_long_text`	40

The following example shows usage of all three. A `column` object is declared and instantiated as a reference to the result of `alv->get_columns( )->get_column( 'DISTID' )`. The column name *must* be in all capital letters. This is so that this method chaining is only called once in its instantiation, instead of being executed every time a column heading is changed.

```
DATA column TYPE REF TO cl_salv_column.
column = alv->get_columns( )->get_column( 'DISTID' ).

column->set_short_text( 'Dist. Unit' ).
column->set_medium_text( 'Unit of Distance' ).
column->set_long_text( 'Mass Unit of Distance (kms, miles)' ).
```

Enable ALV Toolbar Functionality

The following method call enables usage of many advanced features such as sorting, filtering, and exporting data.

```
alv->get_functions( )->set_all( ).
```

Enabling Every Other Row Striping in ALV

This method increases readability by giving consecutive rows alternating background color shading.

```
alv->get_display_settings( )->set_striped_pattern( if_salv_c_bool_sap=>true ).
```

Setting the Title of a Displayed ALV

By default, when an ALV is displayed, the title at the top is just the program name. This method allows the user to set a title of up to 70 characters. The following example shows how a dynamic title can be set that displays the number of records displayed.

```
alv->get_display_settings( )->set_list_header( |Flight Schedule - { lines( t_spfli ) }
records| ).
```

Read ABAP GRID List Viewer (ALV) online: https://riptutorial.com/abap/topic/4660/abap-grid-list-

viewer--alv-

Chapter 3: ABAP Objects

Examples

Class declaration

ABAP Classes can be declared Globally or Locally. A global class can be used by any object within the ABAP repository. By contrast, a local class can only be used within the scope it is declared.

```
CLASS lcl_abap_class DEFINITION.
  PUBLIC SECTION.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS lcl_abap_class IMPLEMENTATION.
ENDCLASS.
```

Constructor, methods

Class implementation:

```
CLASS lcl_abap_class DEFINITION.
  PUBLIC SECTION.
    METHODS: constructor,
             method1.
  PROTECTED SECTION.
  PRIVATE SECTION.
    METHODS: method2,
             method3.
ENDCLASS.

CLASS lcl_abap_class IMPLEMENTATION.
    METHOD constructor.
        "Logic
    ENDMETHOD.

    METHOD method1.
        "Logic
    ENDMETHOD.

    METHOD method2.
        "Logic
        method3( ).
    ENDMETHOD.

    METHOD method3.
        "Logic
```

```
    ENDMETHOD.
ENDCLASS.
```

Method call example:

```
DATA lo_abap_class TYPE REF TO lcl_abap_class.
CREATE OBJECT lo_abap_class. "Constructor call
lo_abap_class->method1( ).
```

Method with parameters (Importing, Changing, Exporting)

Class implementation:

```
CLASS lcl_abap_class DEFINITION.
  PRIVATE SECTION.
    METHODS method1 IMPORTING iv_string TYPE string
                      CHANGING cv_string TYPE string
                     EXPORTING ev_string TYPE string.
ENDCLASS.

CLASS lcl_abap_class IMPLEMENTATION.
    METHOD method1.
        cv_string = iv_string.
        ev_string = 'example'.
    ENDMETHOD.
ENDCLASS.
```

Method call example:

```
method1 (
  EXPORTING iv_string = lv_string
  IMPORTING ev_string = lv_string2
   CHANGING cv_string = lv_string3
).
```

Method with returning parameter

Class implementation:

```
CLASS lcl_abap_class DEFINITION.
  PRIVATE SECTION.
    METHODS method1 RETURNING VALUE(rv_string) TYPE string.
ENDCLASS.

CLASS lcl_abap_class IMPLEMENTATION.
    METHOD method1.
        rv_string = 'returned value'.
    ENDMETHOD.
ENDCLASS.
```

Method call example:

```
lv_string = method1( ).
```

Note that parameters declared with `RETURNING` are passed by value only.

Inheritance - definition

Information

Inheritance allows you to derive a new class from an existing class. You do this using the **INHERITING FROM** addition in the

CLASS subclass **DEFINITION INHERITING FROM** superclass.

statement. The new class subclass inherits all of the components of the existing class superclass. The new class is called the subclass of the class from which it is derived. The original class is called the superclass of the new class. A class can have more than one direct subclass, but it may only have one direct superclass.

Class implementation

```
CLASS lcl_vehicle DEFINITION.
ENDCLASS.

CLASS lcl_vehicle IMPLEMENTATION.
ENDCLASS.

CLASS lcl_car DEFINITION INHERITING FROM lcl_vehicle.
ENDCLASS.

CLASS lcl_car IMPLEMENTATION.
ENDCLASS.
```

Inheritance - Abstract and Final Methods and Classes

Information

The **ABSTRACT** and **FINAL** additions to the **METHODS** and **CLASS** statements allow you to define abstract and final methods or classes.

An abstract method is defined in an abstract class and cannot be implemented in that class. Instead, it is implemented in a subclass of the class. Abstract classes cannot be instantiated.

A final method cannot be redefined in a subclass. Final classes cannot have subclasses. They conclude an inheritance tree.

Class implementation:

```
CLASS lcl_abstract DEFINITION ABSTRACT.
```

```
    PUBLIC SECTION.
        METHODS: abstract_method ABSTRACT,
                 final_method FINAL
                 normal_method.

ENDCLASS.

CLASS lcl_abstract IMPLEMENTATION.
    METHOD final_method.
        "This method can't be redefined in child class!
    ENDMETHOD.

    METHOD normal_method.
        "Some logic
    ENDMETHOD.

        "We can't implement abstract_method here!

ENDCLASS.

CLASS lcl_abap_class DEFINITION INHERITING FROM lcl_abstract.
    PUBLIC SECTION.
        METHODS: abstract_method REDEFINITION,
                 abap_class_method.
ENDCLASS.

CLASS lcl_abap_class IMPLEMENTATION.
    METHOD abstract_method.
        "Abstract method implementation
    ENDMETHOD.

    METHOD abap_class_method.
        "Logic
    ENDMETHOD.
ENDCLASS.
```

Method call example:

```
DATA lo_class TYPE REF TO lcl_abap_class.
CREATE OBJECT lo_class.

lo_class->abstract_method( ).
lo_class->normal_method( ).
lo_class->abap_class_method( ).
lo_class->final_method( ).
```

Read ABAP Objects online: https://riptutorial.com/abap/topic/2244/abap-objects

Chapter 4: Comments

Examples

End of Line

Any text following a " character on the same line is commented out:

```
DATA ls_booking TYPE flightb. " Commented text
```

Full Line

The * character comments out an entire line. The * must be the first character in the line.

```
* DATA ls_booking TYPE flightb. Nothing on this line will be executed.
```

Read Comments online: https://riptutorial.com/abap/topic/1644/comments

Chapter 5: Control Flow Statements

Examples

IF/ELSEIF/ELSE

```
IF lv_foo = 3.
  WRITE: / 'lv_foo is 3'.
ELSEIF lv_foo = 5.
  WRITE: / 'lv_foo is 5'.
ELSE.
  WRITE: / 'lv_foo is neither 3 nor 5'.
ENDIF.
```

CASE

```
CASE lv_foo.
  WHEN 1.
    WRITE: / 'lv_foo is 1'.
  WHEN 2.
    WRITE: / 'lv_foo is 2'.
  WHEN 3.
    WRITE: / 'lv_foo is 3'.
  WHEN OTHERS.
    WRITE: / 'lv_foo is something else'.
ENDCASE
```

CHECK

`CHECK` is a simple statement that evaluates a logical expression and exits the current processing block if it is false.

```
METHOD do_something.
  CHECK iv_input IS NOT INITIAL. "Exits method immediately if iv_input is initial

  "The rest of the method is only executed if iv_input is not initial
ENDMETHOD.
```

ASSERT

`ASSERT` is used in sensitive areas where you want to be absolutely sure, that a variable has a specific value. If the logical condition after `ASSERT` turns out to be false, an unhandleable exception (`ASSERTION_FAILED`) is thrown.

```
ASSERT 1 = 1. "No Problem - Program continues

ASSERT 1 = 2. "ERROR
```

COND/SWITCH

`SWITCH` and `COND` offer a special form of conditional program flow. Unlike `IF` and `CASE`, they respresent different values based on an expression rather than executing statements. That's why they count as functional.

COND

Whenever multiple conditions have to be considered, `COND` can do the job. The syntax is fairly simple:

```
COND <type>(
    WHEN <condition> THEN <value>
    ...
    [ ELSE <default> | throw <exception> ]
).
```

Examples

```
" Set screen element active depending on radio button
screen-active = COND i(
    WHEN p_radio = abap_true THEN 1
    ELSE 0 " optional, because type 'i' defaults to zero
).

" Check how two operands are related to each other
" COND determines its type from rw_compare
rw_compare = COND #(
    WHEN op1 < op2 THEN 'LT'
    WHEN op1 = op2 THEN 'EQ'
    WHEN op1 > op2 THEN 'GT'
).
```

SWITCH

`SWITCH` is a neat tool for mapping values, as it checks for equality only, thus being shorter than `COND` in some cases. If an unexpected input was given, it is also possible to throw an exception. The syntax is a little bit different:

```
SWITCH <type>(
    <variable>
    WHEN <value> THEN <new_value>
    ...
    [ ELSE <default> | throw <exception> ]
).
```

Examples

```
DATA(lw_language) = SWITCH string(
    sy-langu
    WHEN 'E' THEN 'English'
    WHEN 'D' THEN 'German'
    " ...
    ELSE THROW cx_sy_conversion_unknown_langu( )
).
```

Read Control Flow Statements online: https://riptutorial.com/abap/topic/7289/control-flow-statements

Chapter 6: Data Declaration

Examples

Inline Data Declaration

In certain situations, data declarations can be performed inline.

```
LOOP AT lt_sflight INTO DATA(ls_sflight).
    WRITE ls_sflight-carrid.
ENDLOOP.
```

Single Variable Declaration

```
DATA begda TYPE sy-datum.
```

Multiple Variable Declaration

```
DATA: begda TYPE sy-datum,
      endda TYPE sy-datum.
```

Inline Data Declaration in SELECT Statement

When using an inline data declaration inside of a `SELECT...ENDSELECT` block or `SELECT SINGLE` statement, the `@` character must be used as an escape character for the `DATA(lv_cityto)` expression. Once the escape character has been used, all further host variables must also be escaped (as is the case with `lv_carrid` below).

```
DATA lv_carrid TYPE s_carr_id VALUE 'LH'.
SELECT SINGLE cityto FROM spfli
       INTO @DATA(lv_cityto)
       WHERE carrid = @lv_carrid
       AND   connid = 2402.
WRITE: / lv_cityto.
```

Outputs `BERLIN`.

Variable Declaration Options

Different types of variables may be declared with special options.

```
DATA: lv_string   TYPE string, " standard declaration
      lv_char     TYPE c,      " declares a character variable of length 1
      lv_char5(5) TYPE c,      " declares a character variable of length 5
      l_packed TYPE p LENGTH 10 DECIMALS 5 VALUE '1234567890.123456789'. " evaluates to
1,234,567,890.12346
```

Read Data Declaration online: https://riptutorial.com/abap/topic/1646/data-declaration

Chapter 7: Dynamic Programming

Examples

Field-Symbols

Field-Symbols are ABAP's equivalent to pointers, except that Field-Symbols are always dereferenced (it is not possible to change the actual address in memory).

Declaration

To declare a Field-Symbol the keyword `FIELD-SYMBOLS` must be used. Types can be generic (`ANY [... TABLE]`) to handle a wide variety of variables.

```
FIELD-SYMBOLS: <fs_line>     TYPE any,    "generic
               <fs_struct>   TYPE kna1.   "non-generic
```

Assigning

Field-Symbols are `unassigned` on declaration, which means that they are pointing to nothing. Accessing an unassigned Field-Symbol will lead to an exception, and, if uncaught, to a short dump. Therefore, the state should be checked with `IS ASSIGNED`:

```
IF <fs> IS ASSIGNED.
*... symbol is assigned
ENDIF.
```

As they are only references, no real data can be stored inside. So, declared `DATA` is needed in every case of use.

```
DATA: w_name  TYPE string VALUE `Max`,
      w_index TYPE i      VALUE 1.

FIELD-SYMBOLS <fs_name> TYPE any.

ASSIGN w_name TO <fs_name>. "<fs_name> now gets w_name
<fs_name> = 'Manni'.        "Changes to <fs_name> now also affect w_name

* As <fs_name> is generic, it can also be used for numbers

ASSIGN w_index TO <fs_name>. "<fs_name> now refers to w_index.
ADD 1 TO <fs_name>.          "w_index gets incremented by one
```

Unassigning

Sometimes it could be useful to reset a Field-Symbol. This can be done using `UNASSIGN`.

```
UNASSIGN <fs>.
* Access on <fs> now leads to an exception again
```

Use for internal tables

Field-Symbols may be used to modify internal tables.

```
LOOP AT itab INTO DATA(wa).
* Only modifies wa_line
    wa-name1 = 'Max'.
ENDLOOP.

LOOP AT itab ASSIGNING FIELD-SYMBOL(<fs>).
* Directly refers to a line of itab and modifies its values
    <fs>-name1 = 'Max'.
ENDLOOP.
```

Attention! Field-Symbols stay assigned even after leaving the loop. If you want to reuse them safely, unassign them immediately.

Data references

Essential for data references is the addition `REF TO` after `TYPE`.

Dynamic Creation of Structures

If the type of a structure should be decided on runtime, we can define our target structure as reference to the generic type `data`.

```
DATA wa TYPE REF TO data.
```

To give `wa` a type we use the statement `CREATE DATA`. The addition `TYPE` can be specified by:

Reference:

```
CREATE DATA wa TYPE kna1
```

- *Static checks are active so it's not possible to create an unknown type*

Name:

```
CREATE DATA wa TYPE (lw_name_as_string)
```

- *The parentheses are needed and `lw_name_as_string` contains the types name as string.*
- *If the type was not found, an exception of type `CX_SY_CREATE_DATA_ERROR` will be thrown*

For instancing dynamically created types the `HANDLE` addition can be specified. `HANDLE` receives an object which inherits from `CL_ABAP_DATADESCR`.

```
CREATE DATA dref TYPE HANDLE obj
```

- `obj` can be created using the **R**un**T**ime **T**ype **S**ervices
- because `dref` is still a datareference, it has to be dereferenced (`->*`) to be used as

datacontainer (normally done via Field-Symbols)

RunTime Type Services

RunTime Type Services (*short:* **RTTS**) are used either for:

- creating types (RunTime Type Creation; *short:* **RTTC**)
- analysing types (RunTime Type Identification; *short:* **RTTI**)

Classes

```
CL_ABAP_TYPEDESCR
  |
  |--CL_ABAP_DATADESCR
  |    |
  |    |--CL_ABAP_ELEMDESCR
  |    |--CL_ABAP_REFDESCR
  |    |--CL_ABAP_COMPLEXDESCR
  |         |
  |         |--CL_ABAP_STRUCTDESCR
  |         |--CL_ABAP_TABLEDESCR
  |
  |--CL_ABAP_OBJECTDESCR
     |
     |--CL_ABAP_CLASSDESCR
     |--CL_ABAP_INTFDESCR
```

`CL_ABAP_TYPEDESCR` is the base class. It implements the needed methods for describing:

- `DESCRIBE_BY_DATA`
- `DESCRIBE_BY_NAME`
- `DESCRIBE_BY_OBJECT_REF`
- `DESCRIBE_BY_DATA_REF`

Read Dynamic Programming online: https://riptutorial.com/abap/topic/4442/dynamic-programming

Chapter 8: Internal Tables

Examples

Types of Internal tables

```
DATA: <TABLE NAME> TYPE <SORTED|STANDARD|HASHED> TABLE OF <TYPE NAME>
      WITH <UNIQUE|NON-UNIQUE> KEY <FIELDS FOR KEY>.
```

Standard Table

This table has all of the entries stored in a linear fashion and records are accessed in a linear way. For large table sizes, table access can be slow.

Sorted Table

Requires the addition `WITH UNIQUE|NON-UNIQUE KEY`. Searching is quick due to performing a binary search. Entries cannot be appended to this table as it might break the sort order, so they are always inserted using the `INSERT` keyword.

Hashed Table

Requires the addition `WITH UNIQUE|NON-UNIQUE KEY`. Uses a proprietary hashing algorithm to maintain key-value pairs. Theoretically searches can be as slow as `STANDARD` table but practically they are faster than a `SORTED` table taking a constant amount of time irrespective of the size of the table.

Declaration of ABAP Internal Tables

Internal Table Declaration Based on Local Type Definition

```
" Declaration of type
TYPES: BEGIN OF ty_flightb,
         id        TYPE fl_id,
         dat       TYPE fl_date,
         seatno    TYPE fl_seatno,
         firstname TYPE fl_fname,
         lastname  TYPE fl_lname,
         fl_smoke  TYPE fl_smoker,
         classf    TYPE fl_class,
         classb    TYPE fl_class,
         classe    TYPE fl_class,
         meal      TYPE fl_meal,
         service   TYPE fl_service,
         discout   TYPE fl_discnt,
       END OF lty_flightb.
```

```
" Declaration of internal table
DATA t_flightb TYPE STANDARD TABLE OF ty_flightb.
```

Declaration based on Database Table

```
DATA t_flightb TYPE STANDARD TABLE OF flightb.
```

Inline Internal Table Declaration

Requires ABAP version > 7.4

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

DATA(t_inline) = VALUE t_itab( ( 1 ) ( 2 ) ( 3 ) ).
```

Internal Table with Header Lines Declaration

In ABAP there are tables with header lines, and tables without header lines. Tables with header lines are an older concept and should not be used in new development.

Internal Table: Standard Table with / without header line

This code declares the table `i_compc_all` with the existing structure of `compc_str`.

```
DATA: i_compc_all TYPE STANDARD TABLE OF compc_str WITH HEADER LINE.
DATA: i_compc_all TYPE STANDARD TABLE OF compc_str.
```

Internal Table: Hashed Table with / without header line

```
DATA: i_map_rules_c TYPE HASHED TABLE OF /bic/ansdomm0100 WITH HEADER LINE
DATA: i_map_rules_c TYPE HASHED TABLE OF /bic/ansdomm0100
```

Declaration of a work area for tables without a header

A work area (commonly abbreviated *wa*) has the exact same structure as the table, but can contain only one line (a WA is a structure of a table with only one dimension).

```
DATA: i_compc_all_line LIKE LINE OF i_compc_all.
```

Read, Write and Insert into Internal Tables

Read, write and insert into internal tables with a header line:

```
" Read from table with header (using a loop):
```

```
LOOP AT i_compc_all.               " Loop over table i_compc_all and assign header line
  CASE i_compc_all-ftype.          " Read cell ftype from header line from table i_compc_all
    WHEN 'B'.                      " Bill-to customer number transformation
      i_compc_bil = i_compc_all.   " Assign header line of table i_compc_bil with content of
header line i_compc_all
      APPEND i_compc_bil.          " Insert header line of table i_compc_bil into table
i_compc_bil
    " ... more WHENs
  ENDCASE.
ENDLOOP.
```

> Reminder: Internal tables with header lines are forbidden in object oriented contexts. Usage of internal tables *without* header lines is always recommended.

Read, write and insert into internal tables without a header line:

```
" Loop over table i_compc_all and assign current line to structure i_compc_all_line
LOOP AT i_compc_all INTO i_compc_all_line.
  CASE i_compc_all_line-ftype.                " Read column ftype from current line (which as
assigned into i_compc_all_line)
    WHEN 'B'.                                 " Bill-to customer number transformation
      i_compc_bil_line = i_compc_all_line.    " Copy structure
      APPEND i_compc_bil_line TO i_compc_bil. " Append structure to table
    " more WHENs ...
  ENDCASE.
ENDLOOP.


" Insert into table with Header:
INSERT TABLE i_sap_knb1.                      " insert into TABLE WITH HEADER: insert table
header into it's content
insert i_sap_knb1_line into table i_sap_knb1. " insert into HASHED TABLE: insert structure
i_sap_knb1_line into hashed table i_sap_knb1
APPEND p_t_errorlog_line to p_t_errorlog.     " insert into STANDARD TABLE: insert structure /
wa p_t_errorlog_line into table p_t_errorlog_line
```

Read Internal Tables online: https://riptutorial.com/abap/topic/1647/internal-tables

Chapter 9: Loops

Remarks

When looping over internal tables, it is generally preferable to `ASSIGN` to a field symbol rather than loop `INTO` a work area. Assigning field symbols simply updates their reference to point to the next line of the internal table during each iteration, whereas using `INTO` results in the line of the table being copied into the work area, which can be expensive for long/wide tables.

Examples

Internal Table Loop

```
LOOP AT itab INTO wa.
ENDLOOP.

FIELD-SYMBOLS <fs> LIKE LINE OF itab.
LOOP AT itab ASSIGNING <fs>.
ENDLOOP.

LOOP AT itab ASSIGNING FIELD-SYMBOL(<fs>).
ENDLOOP.

LOOP AT itab REFERENCE INTO dref.
ENDLOOP.

LOOP AT itab TRANSPORTING NO FIELDS.
ENDLOOP.
```

Conditional Looping

If only lines that match a certain condition should be taken into the loop, addition `WHERE` can be added.

```
LOOP AT itab INTO wa WHERE f1 = 'Max'.
ENDLOOP.
```

While Loop

ABAP also offers the conventional `WHILE`-Loop which runs until the given expression evaluates to false. The system field `sy-index` will be increased for every loop step.

```
WHILE condition.
* do something
ENDWHILE
```

Do Loop

Without any addition the `DO`-Loop runs endless or at least until it gets explicitly exited from inside. The system field `sy-index` will be increased for every loop step.

```
DO.
* do something... get it?
* call EXIT somewhere
ENDDO.
```

The `TIMES` addition offers a very convenient way to repeat code (`amount` represents a value of type `i`).

```
DO amount TIMES.
* do several times
ENDDO.
```

General Commands

To break loops, the command `EXIT` can be used.

```
DO.
    READ TABLE itab INDEX sy-index INTO DATA(wa).
    IF sy-subrc <> 0.
        EXIT. "Stop this loop if no element was found
    ENDIF.
    " some code
ENDDO.
```

To skip to the next loop step, the command `CONTINUE` can be used.

```
DO.
    IF sy-index MOD 1 = 0.
        CONTINUE. " continue to next even index
    ENDIF.
    " some code
ENDDO.
```

The `CHECK` statement is a `CONTINUE` with condition. If the condition turns out to be **false**, `CONTINUE` will be executed. In other words: *The loop will only carry on with the step if the condition is true*.

This example of `CHECK` ...

```
DO.
    " some code
    CHECK sy-index < 10.
    " some code
ENDDO.
```

... is equivalent to ...

```
DO.
    " some code
    IF sy-index >= 10.
```

```
        CONTINUE.
    ENDIF.
    " some code
ENDDO.
```

Read Loops online: https://riptutorial.com/abap/topic/2270/loops

Chapter 10: Message Classes/MESSAGE keyword

Introduction

The `MESSAGE` statement may be used to interrupt program flow to display short messages to the user. Messages content may be defined in the program's code, in the program's text symbols, or in an independent message class defined in `SE91`.

Remarks

The maximum length of a message, including parameters passed to it using `&`, is 72 characters.

Examples

Defining a Message Class

```
PROGRAM zprogram MESSAGE-ID sabapdemos.
```

System-defined message may be stored in a message class. The `MESSAGE-ID` token defines the message class `sabapdemos` for the entire program. If this is not used, the message class must be specified on each `MESSAGE` call.

MESSAGE with Predefined Text Symbol

```
PROGRAM zprogram MESSAGE-ID za.
...
MESSAGE i000 WITH TEXT-i00.
```

A message will display the text stored in the text symbol `i00` to the user. Since the message type is i (as seen in `i000`), after the user exits the dialog box, program flow will continue from the point of the `MESSAGE` call.

Although the text did not come from the message class `za`, a `MESSAGE-ID` must be specified.

Message without Predefined Message Class

```
PROGRAM zprogram.
...
MESSAGE i050(sabapdemos).
```

It may be inconvenient to define a message class for the entire program, so it is possible to define the message class that the message comes from in the `MESSAGE` statement itself. This example will display message `050` from the message class `sabapdemos`.

Dynamic Messaging

```
DATA: msgid TYPE sy-msgid VALUE 'SABAPDEMOS',
      msgty TYPE sy-msgty VALUE 'I',
      msgno TYPE sy-msgno VALUE '050'.

MESSAGE ID mid TYPE mtype NUMBER num.
```

The `MESSAGE` call above is synonymous to the call `MESSAGE i050(sapdemos).`.

Passing Parameters to Messages

The `&` symbol may be used in a message to allow parameters to be passed to it.

Ordered Parameters

Message `777` of class `sabapdemos`:

```
Message with type &1 &2 in event &3
```

Calling this message with three parameters will return a message using the parameters:

```
MESSAGE i050(sabapdemos) WITH 'E' '010' 'START-OF-SELECTION`.
```

This message will be displayed as `Message with type E 010 in event START-OF-SELECTION`. The number next to the `&` symbol designates the order in which the parameters are displayed.

Unordered Parameters

Message `888` of class `sabapdemos`:

```
& & & &
```

The calling of this message is similar:

```
MESSAGE i050(sabapdemos) WITH 'param1' 'param2' 'param3' 'param4'.
```

This will output `param1 param2 param3 param4`.

Read Message Classes/MESSAGE keyword online:
https://riptutorial.com/abap/topic/10691/message-classes-message-keyword

Chapter 11: Naming Conventions

Syntax

- Characters, numbers and _ can be use for variable name.
- Two character using for variable state and object type.
- Local variables start with L.
- Global variables start with G.
- Function input parameter start with I (import).
- Function output parameter start with E (export).
- Structures symbol is S.
- Table symbol is T.

Examples

Local variable

```
data: lv_temp type string.
data: ls_temp type sy.
data: lt_temp type table of sy.
```

Global variable

```
data: gv_temp type string.
data: gs_temp type sy.
data: gt_temp type table of sy.
```

Read Naming Conventions online: https://riptutorial.com/abap/topic/6770/naming-conventions

Chapter 12: Open SQL

Examples

SELECT statement

SELECT is an Open-SQL-statement for reading data from one or several database tables into data objects.

1. Selecting All Records

```
* This returns all records into internal table lt_mara.
SELECT * FROM mara
         INTO lt_mara.
```

2. Selecting Single Record

```
* This returns single record if table consists multiple records with same key.
SELECT SINGLE * INTO TABLE lt_mara
                FROM mara
                WHERE matnr EQ '400-500'.
```

3. Selecting Distinct Records

```
* This returns records with distinct values.
SELECT DISTINCT * FROM mara
                  INTO TABLE lt_mara
                  ORDER BY matnr.
```

4. Aggregate Functions

```
* This puts the number of records present in table MARA into the variable lv_var
SELECT COUNT( * ) FROM mara
                  INTO lv_var.
```

Read Open SQL online: https://riptutorial.com/abap/topic/6885/open-sql

Chapter 13: Regular Expressions

Examples

Replacing

The `REPLACE` statement can work with regular expressions directly:

```
DATA(lv_test) = 'The quick brown fox'.
REPLACE ALL OCCURRENCES OF REGEX '\wo' IN lv_test WITH 'XX'.
```

The variable `lv_test` will evaluate to `The quick bXXwn XXx`.

Searching

The `FIND` statement can work with regular expressions directly:

```
DATA(lv_test) = 'The quick brown fox'.

FIND REGEX '..ck' IN lv_test.
" sy-subrc == 0

FIND REGEX 'a[sdf]g' IN lv_test.
" sy-subrc == 4
```

Object-Oriented Regular Expressions

For more advanced regex operations it's best to use `CL_ABAP_REGEX` and its related classes.

```
DATA: lv_test  TYPE string,
      lo_regex TYPE REF TO cl_abap_regex.

lv_test = 'The quick brown fox'.
CREATE OBJECT lo_regex
  EXPORTING
    pattern = 'q(...)\w'.

DATA(lo_matcher) = lo_regex->create_matcher( text = lv_test ).
WRITE: / lo_matcher->find_next( ).      " X
WRITE: / lo_matcher->get_submatch( 1 ). " uic
WRITE: / lo_matcher->get_offset( ).     " 4
```

Evaluating Regular Expressions with a Predicate Function

The predicate function `matches` can be used to evaluate strings on the fly without use of any object declarations.

```
IF matches( val = 'Not a hex string'
            regex = '[0-9a-f]*' ).
```

```
  cl_demo_output=>display( 'This will not display' ).
ELSEIF matches( val = '6c6f7665'
                regex = '[0-9a-f]*' ).
  cl_demo_output=>display( 'This will display' ).
ENDIF.
```

Getting SubMatches Using OO-Regular Expressions

By using the method `GET_SUBMATCH` of class `CL_ABAP_MATCHER`, we can get the data in the groups/subgroups.

Goal: get the token to the right of the keyword 'Type'.

```
DATA: lv_pattern TYPE string VALUE 'type\s+(\w+)',
      lv_test TYPE string VALUE 'data lwa type mara'.

CREATE OBJECT ref_regex
  EXPORTING
        pattern     = lv_pattern
        ignore_case = c_true.

ref_regex->create_matcher(
    EXPORTING
        text    = lv_test
    RECEIVING
        matcher = ref_matcher
       ).

ref_matcher->get_submatch(
            EXPORTING
                 index = 0
            RECEIVING
                 submatch = lv_smatch.
```

The resulting variable `lv_smatch` contains the value `MARA`.

Read Regular Expressions online: https://riptutorial.com/abap/topic/5113/regular-expressions

Chapter 14: Strings

Examples

Literals

ABAP offers three different operators for declaring string- or char-like-variables

Symbols	Internal Type	Length	Name
'...'	C	1-255 Chars	text field literals
`...`	CString	0-255 Chars	text string literals
|...|	CString	0-255 Chars	template literals

Note that the length-range only applies to hard coded values. Internally `CString`-variables have arbitrary length while variables of type `C` always have a fixed length.

String templates

String templates are a convenient way of mixing literal strings with values from variables:

```
WRITE |Hello, { lv_name }, nice to meet you!|.
```

It can also format things like dates. To use the logged on user's date format:

```
WRITE |The order was completed on { lv_date DATE = USER } and can not be changed|.
```

Functional method calls and expressions are supported:

```
WRITE |Your token is { to_upper( lv_token ) }|.
WRITE |Version is: { cond #( when lv_date < sy-datum then 'out of date' else 'up to date' )
}|.
```

> **Attention!** Directly implementing temporary results (like method-calls) inside of string templates can lead to massive performance problems (read more about it here). While using it inside of rarely executed statements is okay, it causes your program to rapidly slow down in loops.

Concatenating strings

String and char-like variables can be concatenated using ABAP `CONCATENATE` command. An extra variable for storing the results is required.

Example:

```
CONCATENATE var1 var2 var3 INTO result.
"result now contains the values of var1, var2 & var3 stringed together without spaces
```

Shorthand

Newer versions of ABAP offer a very short variant of concatenation using && (Chaining operator).

```
DATA(lw_result) = `Sum: ` && lw_sum.
```

> **Attention!** It's worth noticing, that using temporary results in combination with the Chaining operator inside of loops can lead to massive performance problems due to growing copy instructions (read more about it here).

Read Strings online: https://riptutorial.com/abap/topic/3531/strings

Chapter 15: Template Programs

Syntax

- CLASS DEFINITION ABSTRACT FINAL makes the program class essentially static as instance methods could never be used. The intention is to keep the class minimal.

Examples

OO Program with essential event methods

```
REPORT z_template.

CLASS lcl_program DEFINITION ABSTRACT FINAL.

  PUBLIC SECTION.

    CLASS-METHODS start_of_selection.
    CLASS-METHODS initialization.
    CLASS-METHODS end_of_selection.

ENDCLASS.

CLASS lcl_program IMPLEMENTATION.

  METHOD initialization.

  ENDMETHOD.

  METHOD start_of_selection.

  ENDMETHOD.

  METHOD end_of_selection.

  ENDMETHOD.

ENDCLASS.

INITIALIZATION.

  lcl_program=>initialization( ).

START-OF-SELECTION.

  lcl_program=>start_of_selection( ).

END-OF-SELECTION.

  lcl_program=>end_of_selection( ).
```

Read Template Programs online: https://riptutorial.com/abap/topic/10552/template-programs

Chapter 16: Unit testing

Examples

Structure of a test class

Test classes are created as local classes in a special unit test include.

This is the basic structure of a test class:

```
CLASS lcl_test DEFINITION
               FOR TESTING
               DURATION SHORT
               RISK LEVEL HARMLESS.

PRIVATE SECTION.
  DATA:
    mo_cut TYPE REF TO zcl_dummy.

  METHODS:
    setup,

  "********* 30 chars *********|
  dummy_test                    for testing.
ENDCLASS.

CLASS lcl_test IMPLEMENTATION.
  METHOD setup.
    CREATE OBJECT mo_cut.
  ENDMETHOD.

  METHOD dummy_test.
    cl_aunit_assert=>fail( ).
  ENDMETHOD.
ENDCLASS.
```

Any method declared with `FOR TESTING` will be a unit test. `setup` is a special method that is executed before each test.

Separate data access from logic

An important principle for unit testing is to separate data access from business logic. One efficient technique for this is to define interfaces for data access. Your main class always use a reference to that interface instead of direct reading or writing data.

in production code the main class will be given an object that wraps actual data access. This could be select statement, function mudule calls, anything really. The important part is that this class should not perform anything else. No logic.

When testing the main class, you give it an object that serves static, fake data instead.

An example for accessing the `SCARR` table

Data access interface `ZIF_DB_SCARR`:

```
INTERFACE zif_db_scarr
  PUBLIC.
    METHODS get_all
      RETURNING
        VALUE(rt_scarr) TYPE scarr_tab .
ENDINTERFACE.
```

Fake data class and test class:

```
CLASS lcl_db_scarr DEFINITION.
  PUBLIC SECTION.
    INTERFACES: zif_db_scarr.
ENDCLASS.

CLASS lcl_db_scarr IMPLEMENTATION.
  METHOD zif_db_scarr~get_all.
    " generate static data here
  ENDMETHOD.
ENDCLASS.

CLASS lcl_test DEFINITION
  FOR TESTING
  DURATION SHORT
  RISK LEVEL HARMLESS.

  PRIVATE SECTION.
    DATA:
      mo_cut TYPE REF TO zcl_main_class.

    METHODS:
      setup.
ENDCLASS.

CLASS lcl_test IMPLEMENTATION.
  METHOD setup.
    DATA: lo_db_scarr TYPE REF TO lcl_db_scarr.

    CREATE OBJECT lo_db_scarr.

    CREATE OBJECT mo_cut
      EXPORTING
        io_db_scarr = lo_db_scarr.
  ENDMETHOD.
ENDCLASS.
```

The idea here is that in production code, `ZCL_MAIN_CLASS` will get a `ZIF_DB_SCARR` object that does a `SELECT` and returns the whole table while the unit test runs against a static dataset defined right there in the unit test include.

Read Unit testing online: https://riptutorial.com/abap/topic/3999/unit-testing

Credits

S. No	Chapters	Contributors
1	Getting started with ABAP	Christian, Community, gkubed, Jagger, mkysoft
2	ABAP GRID List Viewer (ALV)	Achuth hadnoor, gkubed
3	ABAP Objects	Community, Michał Majer, Thomas Matecki
4	Comments	4444, Christian, gkubed
5	Control Flow Statements	Community, gkubed, maillard
6	Data Declaration	Christian, gkubed
7	Dynamic Programming	Community, gkubed
8	Internal Tables	Community, gkubed, Michał Majer, Rahul Kadukar, Thorsten Niehues
9	Loops	Christian, Community, gkubed, Stu G
10	Message Classes/MESSAGE keyword	gkubed
11	Naming Conventions	mkysoft
12	Open SQL	AKHIL RAJ, gkubed
13	Regular Expressions	AKHIL RAJ, gkubed, maillard
14	Strings	Achuth hadnoor, Community, maillard, nexxus, Suncatcher
15	Template Programs	nath
16	Unit testing	maillard