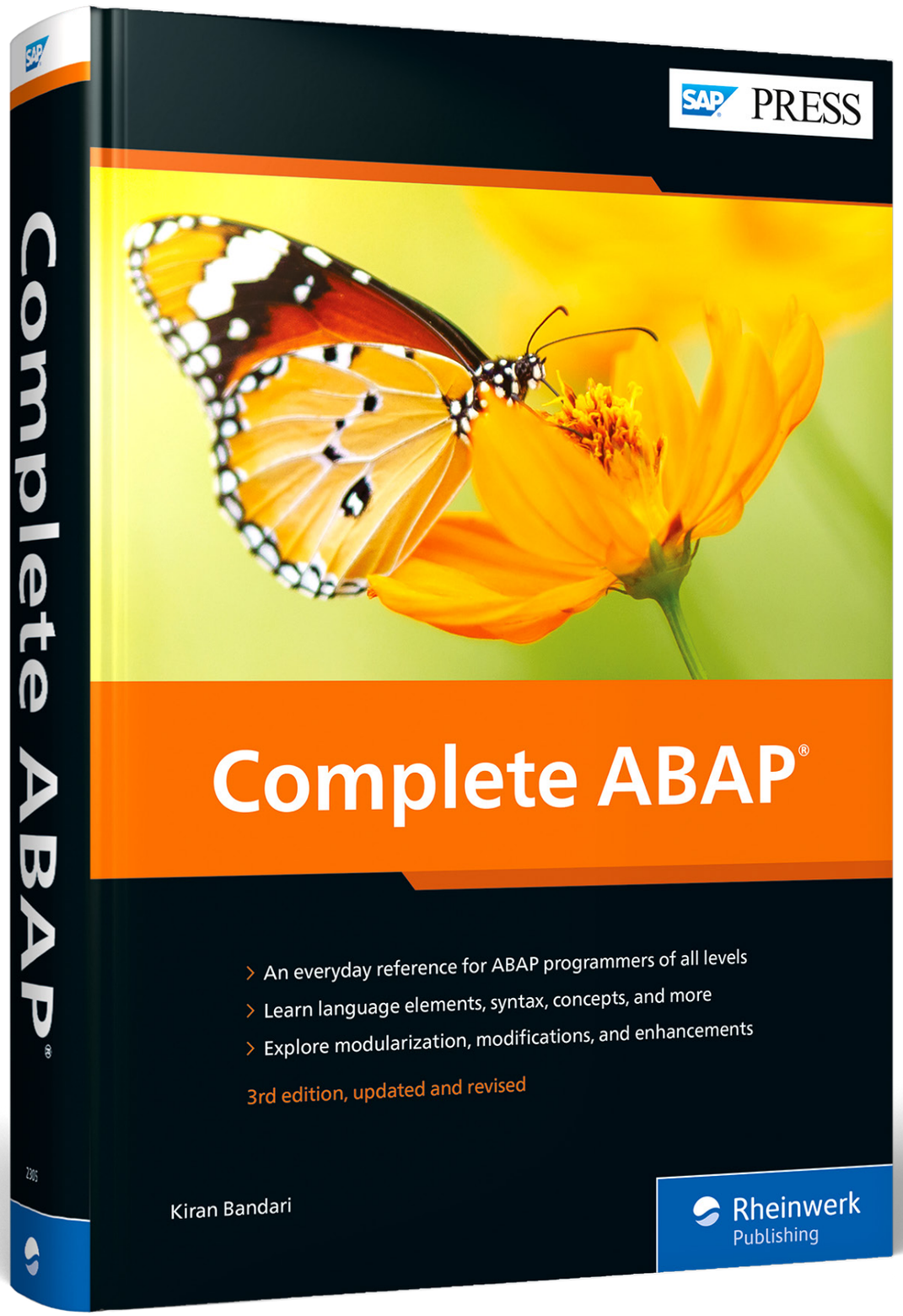
SAP PRESS
First-hand knowledge.
Rheinwerk Publishing



Reading Sample

In this sample chapter, you'll learn how to enable dynamic programs in ABAP. Dynamic programs return results based on user inputs. This chapter teaches you how to use field symbols, data references, specifications, procedure calls, and more in a dynamic way.

Kiran Bandari
Complete ABAP

912 pages, 2022, $89.95
ISBN 978-1-4932-2305-3

www.sap-press.com/5567

Chapter 16

Dynamic Programming

A dynamic program returns results based on specific user input that changes and can only be determined at runtime. This chapter explains how to enable dynamic programs in ABAP.

Software applications are developed to solve a problem or to help users with their day-to-day activities. Most of the time, the problem is known beforehand, and the design and development of the application focuses on solving the problem at hand. For example, assume your company follows a vendor-managed inventory model in which if goods are unsold for 30 days, the company becomes billable to the supplier. At this point, the company can either pay to keep the inventory or return it to the supplier.

To accommodate this business process, you need to develop a set of applications that allows a user to capture information when goods arrive in the warehouse and a report that can show the age of the inventory. The report helps the supervisor check the goods that are reaching the end of the supplier contract period.

To develop applications for this requirement, we know all the information that we need to process in the program at design time. For example, we know which tables to update the data in, and we know which tables to query to calculate the age of the inventory for the report. All this information is available statically (beforehand) at design time.

To show the age of the inventory, we could write the code shown in Listing 16.1.

```
SELECT item_id item_name in_date INTO it_inv FROM ztabinv WHERE
in_date LE p_date.
WRITE : / 'The following items exceed 30 days'.
LOOP AT it_inv INTO wa_inv.
WRITE: / wa_inv-item_id, wa_inv-item_name, wa_inv-in_date.
ENDLOOP.
```

Listing 16.1 Sample Code for Inventory Age

To write the code as shown in Listing 16.1, we need to know the table name and the fields of the table statically. In the program, we statically define the data objects and access them statically by addressing the name of the data object.

However, sometimes we may not have all the information until runtime. What if the table to which we need to write the select query depends on the result of the previous statement? It isn't possible to include the table name statically in the query; to handle such scenarios, we use *dynamic programming*.

You can handle dynamic programming in various ways. Sometimes, a program may have mostly static elements, but some part of it may be dynamic, or the complete program itself could be dynamic. Sometimes, you may even need to create new programs on the fly dynamically or call procedures dynamically. In this chapter, we'll explore how to handle all these scenarios.

ABAP supports various concepts to make ABAP programs dynamic. We'll begin by discussing field symbols in Section 16.1. Field symbols act as pointers and can be used to point to any data object of the program dynamically, allowing you to determine dynamically which data object should be accessed at runtime.

We'll discuss data references in Section 16.2, which, similarly to field symbols, allow you to point dynamically to existing data objects in the program. However, data references also allow you to create data objects dynamically. Another important aspect of dynamic programming is the ability to determine the type information of an existing data object dynamically or to create a data type dynamically.

In Section 16.3, we'll discuss Runtime Type Services (RTTS) and explore identifying and creating data types dynamically.

Business requirements sometimes demand that you determine dynamically the database table from which the data is fetched, which program/transaction should be called during the program flow, or something similar. ABAP allows you to determine the database table, procedure, program, transaction, and so on dynamically by providing the token dynamically in the code.

You'll learn about dynamic token specification in Section 16.4 and dynamic procedure calls in Section 16.5. Not only can you make the ABAP code within an ABAP program dynamic, but you also can generate the complete ABAP program itself dynamically. We'll conclude this chapter with a discussion of dynamic program generation in Section 16.6.

16.1 Field Symbols

A *field symbol* is similar to a pointer, which points to an existing data object. A field symbol isn't a data object; it doesn't hold any memory on its own. Instead, when a data object is assigned to a field symbol, it points to the memory location of the assigned data object.

A field symbol serves as a label for its assigned data object, which you can access as you would access the data object itself. For example, assume there is an internal table with

three fields, `FIELD1`, `FIELD2`, and `FIELD3`, and we want to modify the third row of the internal table. The code shown in Listing 16.2 handles this scenario.

```
DATA wa LIKE LINE OF itab.
wa-field1 = 'ABC'.
wa-field2 = 'XYZ'.
wa-field3 = 123.
MODIFY itab FROM wa INDEX 3.
```

Listing 16.2 Modifying an Internal Table Record Using a Work Area

In Listing 16.2, we're using work area `wa` to modify the internal table record. Here, `wa` and `itab` are two separate memory locations, and we're using the `MODIFY` statement to copy the contents of `wa` to `itab`.

Listing 16.3 shows how the same end can be achieved using field symbols.

```
FIELD-SYMBOLS: <wa> LIKE LINE OF itab.
READ TABLE itab ASSIGNING <wa> INDEX 3.
IF <wa> IS ASSIGNED.
<wa>-field1 = 'ABC'.
<wa>-field2 = 'XYZ'.
<wa>-field3 = 123.
ENDIF.
```

Listing 16.3 Modifying an Internal Table Record Using a Field Symbol

In Listing 16.3, we defined a `<wa>` field symbol as a line of `itab`. The `READ` statement assigns the third row of the internal table to field symbol `<wa>`. At this point, field symbol `<wa>` points to the third row of the internal table. When we change a field, through the field symbol, the field of the third row is changed immediately because the field symbol points to the third row directly.

We don't have to copy the contents manually using the `MODIFY` statement as we did in Listing 16.2 because the `<wa>` field symbol in Listing 16.3 doesn't take a separate memory space but points to the existing memory, unlike work area `wa` in Listing 16.2.

After the data object is assigned to the field symbol, we can work with the field symbol just like the assigned data object and perform operations that can usually be performed on the assigned data object. Almost any data object can be assigned to a field symbol.

As you can see in Listing 16.3, there is a way to define field symbols, assign a data object to a field symbol, check if the field symbol is assigned before accessing the field symbol, and more. Before we discuss these aspects, however, let's look at how field symbols can help make programs dynamic.

16.1.1 Using Field Symbols to Make Programs Dynamic

For our example, we'll use a report to show some basic information about a material. In this report, we'll use the SAP List Viewer (ALV) object model to display the output. We'll provide functionality for the user to double-click in any cell to filter the records based on the value of the cell and show the result in a new container below the displayed output.

Most of the components of this report are static as we already know the data to be displayed in the output and the user input for the report. However, filtering the records based on user action is dynamic. Let's look at how we can achieve this in a static way and then look at how to make it dynamic.

Listing 16.4 shows the code for the report using the ALV object model. The code is long, but very straightforward. Take some time to go through the code and try it out in your system.

```
REPORT zdemo_salv_static.
CLASS  lcl_event_handler DEFINITION DEFERRED.
DATA: custom_container TYPE REF TO cl_gui_custom_container,
      splitter         TYPE REF TO cl_gui_splitter_container,
      container_1      TYPE REF TO cl_gui_container,
      so_alv           TYPE REF TO cl_salv_table,
      mo_alv           TYPE REF TO cl_salv_table,
      gr_event_handler TYPE REF TO lcl_event_handler,
      gr_event         TYPE REF TO cl_salv_events_table.
TYPES: BEGIN OF ty_mara,
         matnr TYPE mara-matnr,
         ersda TYPE ersda,
         ernam TYPE ernam,
         laeda TYPE laeda,
         mtart TYPE mtart,
       END OF ty_mara.
DATA: it_mara  TYPE STANDARD TABLE OF ty_mara,
      gv_matnr TYPE matnr.
SELECT-OPTIONS: s_matnr FOR gv_matnr.


*&---------------------------------------------------------------------*
*&       Class LCL_EVENT_HANDLER
*&---------------------------------------------------------------------*
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    METHODS : on_double_click FOR EVENT double_click OF cl_salv_events_
table IMPORTING row column.
ENDCLASS.               "LCL_EVENT_HANDLER
```



```
*---------------------------------------------------------------------*
*       CLASS lcl_event_handler IMPLEMENTATION
*---------------------------------------------------------------------*
CLASS lcl_event_handler IMPLEMENTATION.
  METHOD on_double_click.
    DATA : lo_sel      TYPE REF TO cl_salv_selections,
           ls_cell     TYPE salv_s_cell,
           container_2 TYPE REF TO cl_gui_container,
           lt_mara     TYPE STANDARD TABLE OF ty_mara,
           lw_mara     TYPE ty_mara.
*Get reference to the user selection.
    lo_sel = mo_alv->get_selections( ).
*Get the value of the cell
    ls_cell = lo_sel->get_current_cell( ).
*Logic to filter records
    IF ls_cell-columnname EQ 'MATNR'.
      LOOP AT it_mara INTO lw_mara WHERE matnr EQ ls_cell-value.
        APPEND lw_mara TO lt_mara.
      ENDLOOP.
    ELSEIF ls_cell-columnname EQ 'ERSDA'.
      LOOP AT it_mara INTO lw_mara WHERE ersda EQ ls_cell-value.
        APPEND lw_mara TO lt_mara.
      ENDLOOP.
    ELSEIF ls_cell-columnname EQ 'ERNAM'.
      LOOP AT it_mara INTO lw_mara WHERE ernam EQ ls_cell-value.
        APPEND lw_mara TO lt_mara.
      ENDLOOP.
    ELSEIF ls_cell-columnname EQ 'LAEDA'.
      LOOP AT it_mara INTO lw_mara WHERE laeda EQ ls_cell-value.
        APPEND lw_mara TO lt_mara.
      ENDLOOP.
    ELSEIF ls_cell-columnname EQ 'MTART'.
      LOOP AT it_mara INTO lw_mara WHERE mtart EQ ls_cell-value.
        APPEND lw_mara TO lt_mara.
      ENDLOOP.
    ENDIF.
    CALL METHOD splitter->get_container
      EXPORTING
        row       = 2
        column    = 1
      RECEIVING
        container = container_2.
```

```
TRY.
        IF so_alv IS BOUND.
          so_alv->set_data( CHANGING t_table = lt_mara ).
        ELSE.
          CALL METHOD cl_salv_table=>factory
            EXPORTING
              list_display = if_salv_c_bool_sap=>false
              r_container  = container_2
            IMPORTING
              r_salv_table = so_alv
            CHANGING
              t_table      = lt_mara.
        ENDIF.
      CATCH cx_salv_no_new_data_allowed.
      CATCH cx_salv_msg .
    ENDTRY.

    so_alv->display( ).
  ENDMETHOD.                    "on_double_click
ENDCLASS.               "lcl_event_handler

START-OF-SELECTION.
  PERFORM fill_table.
  CALL SCREEN 100.


*&---------------------------------------------------------------------*
*&      Module  STATUS_100  OUTPUT
*&---------------------------------------------------------------------*
MODULE status_100 OUTPUT.
* Create container object
  CREATE OBJECT custom_container
    EXPORTING
      container_name = 'CONTAINER'.
*Create splitter object
  CREATE OBJECT splitter
    EXPORTING
      parent  = custom_container
      rows    = 2
      columns = 1.
*Split the container
  CALL METHOD splitter->get_container
    EXPORTING
```



```
      row       = 1
      column    = 1
    RECEIVING
      container = container_1.
*Get ALV Object reference
  TRY.
      CALL METHOD cl_salv_table=>factory
        EXPORTING
          list_display = if_salv_c_bool_sap=>false
          r_container  = container_1
        IMPORTING
          r_salv_table = mo_alv
        CHANGING
          t_table      = it_mara.
    CATCH cx_salv_msg .
  ENDTRY.
*  Set the event handler here
  gr_event = mo_alv->get_event( ).
  CREATE OBJECT gr_event_handler.
  SET HANDLER gr_event_handler->on_double_click FOR gr_event.
*Display output
  mo_alv->display( ).
ENDMODULE.                 " STATUS_100  OUTPUT
*&---------------------------------------------------------------------*
*&      Form  FILL_TABLE
*&---------------------------------------------------------------------*
FORM fill_table .
  SELECT matnr ersda ernam laeda mtart FROM mara INTO TABLE it_
mara WHERE matnr IN s_matnr.
ENDFORM.                    " FILL_TABLE
```

Listing 16.4 Report to Display Material and Filter Functionality

In Listing 16.4, we're selecting certain field data from table MARA and displaying the output. We're using the ALV object model to generate the ALV output.

We've defined a screen 100 for our program in which we've created a custom control. Because we want to display the filtered records on the same screen, we're using the CL_GUI_SPLITTER_CONTAINER class to split the container into two rows.

We show the complete selected data in the first row and display the filtered data in the second row. Figure 16.1 shows the initial display of the report.

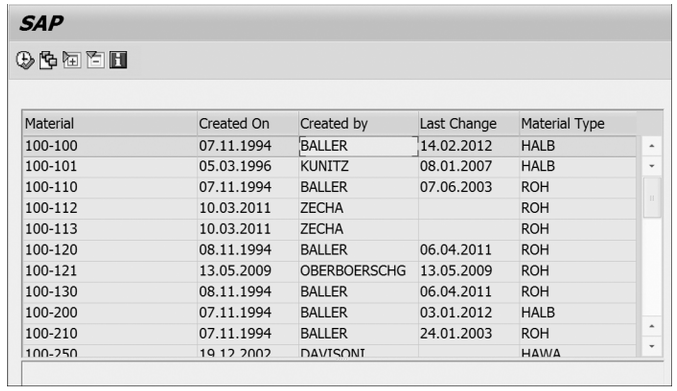


Figure 16.1 Initial ALV Display

The user can double-click any cell to show the filtered records in a new container below the main display. For example, if the user double-clicks material type **HALB** in Figure 16.1, the output would look like Figure 16.2.

To display the filtered data in a new container per the user's selection, we need to do a couple of things:

1. Identify the user action (double-click).
2. Identify the column and value of the cell to filter the records.

The ALV object model raises the `ON_DOUBLE_CLICK` event when the user double-clicks the cell. In the program, we're setting the event handler and have defined the `ON_DOUBLE_CLICK` event handler method in the `LCL_EVENT_HANDLER` local class. In this method, we're filtering the records based on the value and displaying the filtered records, as shown in Figure 16.2.

Figure 16.2 Filtered Output in the New Container

When the user double-clicks, the `ON_DOUBLE_CLICK` method of the `LCL_EVENT_HANDLER` local class is called. The code in this method is where our interest lies. If you look at the code in this method, you'll notice that we used the `IF... ELSEIF... ENDIF` block to check the field name manually and accordingly filter the internal table data.

Because there's no way to know statically which column the user will double-click, we had to construct the `IF…ELSEIF` block to include all the fields of the internal table. Luckily, we only had five fields in the internal table, so it wasn't that difficult.

However, what if the internal table has 30 fields or 50 fields? What if some new fields are added to the internal table as part of future enhancement? We have no other option but to update this logic to include the additional fields. Not only does listing all the fields statically in an `IF…ELSEIF` block make the code cumbersome, but hard-coding also isn't a good programming practice and should be avoided as much as possible.

There must be a better way to handle this, right? Of course, this is where field symbols come to the rescue. Using field symbols, we can modify the code in the `ON_DOUBLE_CLICK` method of the `LCL_EVENT_HANDLER` local class, as shown in Listing 16.5. Nothing much has changed in Listing 16.5, except how we're filtering the records.

```
METHOD on_double_click.
    FIELD-SYMBOLS: <field> TYPE any.
    DATA : lo_sel      TYPE REF TO cl_salv_selections,
           ls_cell     TYPE salv_s_cell,
           container_2 TYPE REF TO cl_gui_container,
           lt_mara     TYPE STANDARD TABLE OF ty_mara,
           lw_mara     TYPE ty_mara.

  lo_sel = mo_alv->get_selections( ).
  ls_cell = lo_sel->get_current_cell( ).

  LOOP AT it_mara INTO lw_mara.
    ASSIGN COMPONENT ls_cell-columnname OF STRUCTURE lw_mara TO <field>.
    IF <field> IS ASSIGNED.
      IF <field> EQ ls_cell-value.
        APPEND lw_mara TO lt_mara.
      ENDIF.
    ENDIF.
  ENDLOOP.
  CALL METHOD splitter->get_container
    EXPORTING
      row       = 2
      column    = 1
    RECEIVING
      container = container_2.
```

```
    TRY.
        IF so_alv IS BOUND.
          so_alv->set_data( CHANGING t_table = lt_mara ).
        ELSE.
          CALL METHOD cl_salv_table=>factory
            EXPORTING
              list_display = if_salv_c_bool_sap=>false
              r_container  = container_2
            IMPORTING
              r_salv_table = so_alv
            CHANGING
              t_table      = lt_mara.
        ENDIF.
      CATCH cx_salv_msg .
    ENDTRY.
    so_alv->display( ).
  ENDMETHOD.                    "on_double_click
```

Listing 16.5 Filter Functionality Using Field Symbols

In Listing 16.5, we've defined a <FIELD> field symbol as a generic type (we'll discuss typing field symbols in a moment). Within the loop, we're dynamically assigning the component of the table to the field symbol. Based on which column the user double-clicks, the LS_CELL-COLUMNNAME field will have that field name. The ASSIGN COMPONENT OF statement will assign that field to the <FIELD> field symbol.

From here, all we need to do is check the cell value and filter the records. As you can see, now our filtering logic is dynamic. The code will filter the records irrespective of which column the user double-clicks. This code will be equally effective when new fields are added to the structure as part of future enhancement because we're no longer hard-coding anything.

Now that you understand how field symbols can be used to make programs dynamic, let's go into further detail about how to define field symbols.

16.1.2 Defining Field Symbols

Field symbols are declared using the FIELD-SYMBOLS statement. The name of the field symbol should be enclosed between angled brackets (< and >). You can type the field symbol just like any other data object using the TYPE reference or LIKE reference.

You can also define a generic field symbol by typing it as ANY. When you define a fully typed field symbol, only a data object of the same type can be assigned to the field symbol. This is useful if you plan to access the components of a structure statically.

For example, if you want to work with the individual components of a structure as shown in Listing 16.6, then the field symbol should be fully typed. Without typing the field symbol, there's no way for the compiler to know what fields it contains when the code is compiled.

```
DATA st_mara TYPE mara.
FIELD-SYMBOLS: <fs_mara> TYPE mara,
               <fs_matnr> TYPE matnr.
ASSIGN st_mara TO <fs_mara>.
<fs_mara>-matnr = '100'.
ASSIGN st_mara-matnr TO <fs_matnr>.
```

Listing 16.6 Working with Structure Components

If you plan to use the field symbol to point to any field or structure dynamically, you can define the field symbol as a generic type using the TYPE ANY addition, as shown in Listing 16.7. However, with generic types, the components are known only at runtime, so you can't access them statically.

```
DATA st_mara TYPE mara.
FIELD-SYMBOLS: <fs_any> TYPE ANY.
ASSIGN st_mara TO <fs_any>.
* <fs_any>-matnr = '100'. "This will result in syntax error
ASSIGN st_mara-matnr TO <fs_any>.
```

Listing 16.7 Generic Type

You can also assign an internal table to the field symbol. To assign an internal table, the field symbol should be defined as a table. You can type the field symbol fully as shown:

```
FIELD-SYMBOLS <fs_mara> TYPE STANDARD TABLE OF mara.
```

You can define it as any kind of internal table of any type, just like a data object. You can also perform all the actions and use all the statements that are applicable to an internal table. For example, the syntax checker won't complain if you use a LOOP statement with a field symbol.

If you plan to assign any internal table dynamically (for which the line type is known only at runtime), you can define the field symbol as a generic table using the ANY TABLE syntax, as shown:

```
FIELD-SYMBOLS <fs_table> TYPE ANY TABLE.
```

16.1.3 Assigning a Data Object

The ASSIGN statement is used to assign the data object to a field symbol, as shown:

```
ASSIGN dobj TO <fs>.
```

The assignment operation can be performed both statically and dynamically.

If we know the data object name to assign to the field symbol, we can assign the data object statically, as shown in Listing 16.8.

```
DATA st_mara TYPE mara.
FIELD-SYMBOLS: <fs_mara> TYPE mara.
ASSIGN st_mara TO <fs_mara>.
```

Listing 16.8 Static Assignment of a Field Symbol

Sometimes, we may know the data object that we want to assign dynamically only at runtime. For example, say that you're maintaining some change history in a table in which you're storing the field name with its old and new values. In your program, you want to process only those fields for which a record is available in the change history table.

You can assign the data object dynamically, as shown:

```
ASSIGN (name) TO <fs>.
```

Here, the `name` variable contains the name of the data object. The variable should be enclosed within parentheses.

Listing 16.9 shows an example of dynamic assignment. Here, the data object name stored in the `FIELD_NAME` variable will be assigned to the field symbol. If the assignment is successful (i.e., the data object with that field name exists in the program), the `SY-SUBRC` system field will be set to `0`.

When using dynamic assignments, it's important to check if the field symbol is assigned before accessing the field symbol. Trying to access an unassigned field symbol will result in a runtime error.

```
DATA field_name TYPE string.
FIELD-SYMBOLS <fs_field> TYPE ANY.
SELECT SINGLE ... FROM ... INTO field_name WHERE ....
ASSIGN (field_name) TO <fs_field>.
```

Listing 16.9 Dynamic Field Assignment

You can use dynamic assignment for the attributes of a class as shown:

```
ASSIGN oref->(attribute_name) TO <fs>.
```

In this assignment, the name of the attribute is derived dynamically. If you're accessing a static attribute, you can even derive the class name dynamically, as shown in Listing 16.10.


```
ASSIGN (class_name)=>(attribute_name) TO <fs>.
ASSIGN (class_name)=>attribute TO <fs>.
```

Listing 16.10 Accessing Class Attributes Dynamically

The first line in Listing 16.10 addresses both the class name and the attribute name dynamically, whereas the second line addresses the class name dynamically and the attribute name statically.

You can also assign the components of a structure dynamically with the following syntax:

```
ASSIGN comp OF STRUCTURE struc TO <fs>.
```

With this statement, the `comp` component of the `struc` structure is assigned to the `<fs>` field symbol. If the `comp` data object is set as `TYPE c` or `TYPE string`, then the content of the `comp` field is interpreted to be the component name. For example, in Listing 16.11, the `matnr` field of the `st_mara` structure will be assigned to the field symbol.

```
DATA : st_mara TYPE mara,
       l_field TYPE string VALUE 'MATNR'.
FIELD-SYMBOLS: <fs_any> TYPE any.
ASSIGN l_field OF STRUCTURE st_mara TO <fs_any>.
```

Listing 16.11 Assigning a Structure Field

However, if the `comp` data object is of a different type and has a number, then the system will interpret it as a field position and assign that field to the field symbol.

For example, in Listing 16.12, the third field, `mtart`, of the `st_mara` structure will be assigned to the field symbol because the `l_field` data object contains the value `3`.

```
TYPES: BEGIN OF ty_mara,
       matnr TYPE matnr,
       ersda TYPE ersda,
       mtart TYPE mtart,
      END OF ty_mara.
DATA : st_mara TYPE ty_mara,
       l_field TYPE n VALUE 3.
FIELD-SYMBOLS: <fs_any> TYPE any.
ASSIGN l_field OF STRUCTURE st_mara TO <fs_any>.
```

Listing 16.12 Dynamic Assignment Based on Field Position

You can use the field symbol as a work area when working with internal tables. The internal table record is assigned to the field symbol using the `ASSIGNING` addition with the `LOOP` or `READ` statement, as shown in Listing 16.13.

```
READ TABLE itab ASSIGNING <fs>...
LOOP AT itab ASSIGNING <fs>..
ENDLOOP.
```

Listing 16.13 Using the ASSIGNING Addition

When an internal table record is assigned to a field symbol using the LOOP or READ statement, the field symbol will directly point to the assigned row of the table. For example, in Listing 16.14, we're assigning the first row of the internal table IT_MARA to the <FS_MARA> field symbol.

After this assignment, the field symbol will point to the first row of the internal table, and you can change any field in the row directly using the field symbol.

In this example, the <FS_MARA>-MTART = 'ROH' statement will change the value of the MTART field in the first row of the internal table. Unlike a work area, when using a field symbol, you don't need to use the MODIFY statement to copy the contents of the work area back to the internal table.

```
TYPES: BEGIN OF ty_mara,
       matnr TYPE matnr,
       ersda TYPE ersda,
       mtart TYPE mtart,
      END OF ty_mara.
DATA : it_mara TYPE TABLE OF ty_mara.
FIELD-SYMBOLS: <fs_mara> LIKE LINE OF it_mara.
PARAMETERS p_matnr TYPE matnr.
SELECT matnr ersda mtart FROM mara INTO TABLE it_mara WHERE matnr EQ p_matnr.
READ TABLE it_mara ASSIGNING <fs_mara> INDEX 1.
IF <fs_mara> IS ASSIGNED.
<fs_mara>-mtart = 'ROH'.
ENDIF.
```

Listing 16.14 Assigning an Internal Table Record to a Field Symbol

16.1.4 Checking If a Field Symbol Is Assigned

When a data object is assigned to a field symbol, the SY-SUBRC system field is set to 0. If you want to check immediately after the assignment, you can check the SY-SUBRC field, as shown in Listing 16.15.

```
ASSIGN dobj TO <fs>.
IF sy-subrc IS INITIAL.
<fs> = 123.
ENDIF.
```

Listing 16.15 Checking the SY-SUBRC Field

However, in most situations, you won't need to access the field symbol immediately after the assignment but may need to access it later in the code. In such situations, checking the SY-SUBRC system field isn't useful because SY-SUBRC is updated by many statements, and it always contains the result of the last statement that updated it.

To be sure that the field symbol is assigned before accessing it, use the IS ASSIGNED statement with the following syntax:

```
IF <fs> IS ASSIGNED.
ENDIF.
```

We recommend that you always check if the field symbol is assigned before accessing it. Trying to access a field symbol that isn't assigned will result in an untreatable exception.

16.1.5 Unassigning a Field Symbol

If you want to remove the assignment of the field symbol to ensure that no data object is assigned to it, use the UNASSIGN statement as follows:

```
UNASSIGN <fs>.
```

After the field symbol is unassigned, it will be initialized, so it won't point to any data object. You shouldn't access the field symbol after it's unassigned unless another data object is assigned to the field symbol.

Using the CLEAR statement won't initialize the field symbol; it will simply clear the contents of the data object assigned to the field symbol.

16.1.6 Casting

When a data object is assigned to a field symbol, the field symbol should be of a type compatible with the data object. However, you can perform a cast on any data type when the data object is assigned to the field symbol.

Casting allows you to treat a particular data type as another data type. This means that any area of memory can be viewed as having assumed a given type. For example, you can interpret the value of a character field as a date field when the data object is assigned to the field symbol by performing a cast.

CASTING is used to perform a cast, as shown:

```
ASSIGN dobj TO <fs> CASTING.
```

When you use CASTING, a data object of a different type can be assigned to the field symbol. Casting allows you to assign a data object to the field symbol that isn't compatible with the field symbol.

Listing 16.16 shows sample code in which a field of TYPE C is assigned to the field symbol, which is of TYPE D. After the data object is assigned, the contents of the data object will be interpreted as TYPE D.

```
DATA text(8) TYPE c VALUE '20111201'.
FIELD-SYMBOLS <fs> TYPE sy-datum.
ASSIGN text TO <fs> CASTING.
WRITE <fs>.
```

Listing 16.16 Casting

Casting can be performed implicitly or explicitly. If the field symbol is fully typed or is typed generically using one of the built-in ABAP types (c, n, p, or x), then CASTING can be used with the ASSIGN statement, which performs an implicit casting to cast the assigned memory to the type of the field symbol.

If the field symbol is typed generically, then you can perform an explicit casting by specifying the data type in the ASSIGN statement as follows:

```
ASSIGN dobj TO <fs> CASTING [TYPE data_type|LIKE dobj].
```

After the CASTING addition, you can specify the data type or a data object. The casting will be performed per the specified type. Your specified type should be compatible with the generic type of the field symbol, and you shouldn't perform an explicit typing for a fully typed field symbol. Casting allows you to interpret data differently per type conversion.

In this section, we discussed the use of field symbols in dynamic programming. In the next section, we'll look at using data references to point to a data object in memory.

16.2 Data References

Similar to object references, a *data reference* points to a data object in memory. Data references can be used as alternate access references to existing data objects, similar to field symbols. However, with data references, it's possible to define anonymous data objects. The data reference is stored in the data reference variable defined using the TYPE REF TO addition of the DATA statement.

Now, we'll explain how to define and work with data references. We'll also discuss how to define data objects dynamically using data references and assignments between data references.

16.2.1 Defining Reference Variables

The syntax to define a data reference variable is shown in Listing 16.17.


```
DATA dref TYPE REF TO dtype
DATA dref LIKE REF TP dobj
DATA dref TYPE REF TO DATA
```

Listing 16.17 Defining a Data Reference Variable

In Listing 16.17, the first statement uses the data type reference, and the second statement uses the data object reference. The data reference variables are fully typed and can point to data objects of this type. The third statement is defined as a generic type, DATA, and can point to any data object.

Similar to object references, a data reference can be a static type or a dynamic type. A static type of data reference should be a fully typed or fully generic type, as shown in Listing 16.17. A dynamic type should be either equal to the static type or more special than its static type (not generic).

When the data reference variable is defined, it's initial; in other words, the data reference variable doesn't point to any data object. A reference to a data object should be supplied to the data reference variable.

After the data reference is supplied, it holds the memory location of the referenced data object and not its content. To access the contents of the referenced data object, the reference variable should be dereferenced using the ->* dereference selector.

Figure 16.3 shows the data reference variable in debug mode for better visualization. As shown, when the data reference variable DREF is accessed, it holds the reference to the V_MATNR data object. To access the contents of V_MATNR, we dereference the reference variable using the ->* selector.

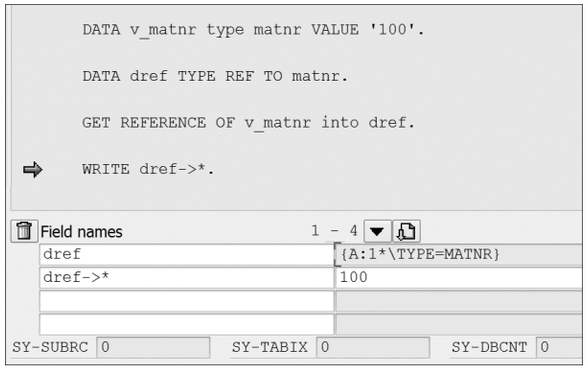


Figure 16.3 Data Reference Variable in Debug Mode

16.2.2 Getting Data References

The GET REFERENCE OF statement is used to obtain a reference to an existing data object. You can obtain a reference using the GET REFERENCE OF statement from an existing data

object, an assigned field symbol, or a dereferenced data reference variable, as shown in Listing 16.18.

```
DATA v_matnr type matnr VALUE '100'.
DATA dref TYPE REF TO matnr.
DATA dref1 TYPE REF TO matnr.
FIELD-SYMBOLS <fs> TYPE matnr.
ASSIGN v_matnr TO <fs>.
*Getting reference from data object
GET REFERENCE OF v_matnr into dref.
*Getting reference from dereferenced reference variable
GET REFERENCE OF dref->* INTO dref1.
*Getting reference from field symbol
GET REFERENCE OF <fs> INTO dref1.
WRITE dref->*.
```

Listing 16.18 Getting Data Object References

After the data object reference is obtained, the reference variable should be dereferenced to access the contents of the referenced data object. For example, in Listing 16.18, we're dereferencing the reference variable using the `->*` dereference selector to access the contents of the referenced data object with the `WRITE` statement.

After the reference variable is dereferenced, the reference variable can be used to perform any actions or use any statements that are possible with the referenced data object. You can dereference the data reference variable directly only if it's fully typed. If the reference variable is generically typed, then it should be assigned to a field symbol to access the contents of the referenced data object.

Listing 16.19 shows an example using generic reference variables.

```
DATA v_matnr type matnr VALUE '100'.
DATA dref TYPE REF TO data.
FIELD-SYMBOLS <fs> TYPE any.

GET REFERENCE OF v_matnr into dref.
ASSIGN dref->* TO <fs>.
* WRITE dref->*. " Results in syntax error
WRITE <fs>.
```

Listing 16.19 Dereferencing Generic Type

In Listing 16.19, the `DREF` reference variable is declared as a generic type. The reference of data object `V_MATNR` is obtained using the `GET REFERENCE` statement. However, because the `DREF` reference variable is a generic type, we can't access the contents of `V_MATNR` by directly dereferencing the reference variable as we did in the `WRITE` statement in Listing 16.18; in fact, dereferencing a generic type results in a syntax error.

To dereference the generic type, we assign it to a field symbol as shown in Listing 16.19. If the reference variable has an empty reference, then the field symbol won't be assigned. You can check if the field symbol is assigned before accessing it to avoid an untreatable exception.

16.2.3 Anonymous Data Objects

Anonymous data objects are used when the data type is unknown statically. When we define data objects using the `DATA` statement, all the data objects are created by the runtime environment when the program starts. All these data objects can be accessed in the program statically using the name of the data object. These are called *named data objects*.

Anonymous data objects are created on demand in the program as required and can be assigned a suitable type per requirements. You use the `CREATE DATA` statement to instantiate an anonymous data object with the following syntax:

```
CREATE DATA dref TYPE|LIKE dtype.
```

The `CREATE DATA` statement creates an anonymous data object and assigns the reference of the data object to the reference variable. Anonymous data objects can't be accessed directly by their names. The data object needs to be dereferenced to access its contents.

Listing 16.20 shows an example using an anonymous data object.

```
FIELD-SYMBOLS <fs> TYPE DATA.
DATA dref TYPE REF TO DATA.
CREATE DATA dref TYPE i.
ASSIGN dref->* TO <fs>.
<fs> = 5.
```

Listing 16.20 Anonymous Data Object

The data type can be specified dynamically using a field in parentheses, for example, `CREATE DATA dref TYPE (name)`. Here, `name` is a data object that contains the data type.

With the release of SAP NetWeaver 7.4, a new instantiation operation, `NEW`, has been introduced to instantiate both data references and object references in lieu of the `CREATE` statement. The `NEW` statement, as shown in Listing 16.21, takes its cue from other programming languages, such as Java.

```
FIELD-SYMBOLS <fs> TYPE DATA.
DATA dref TYPE REF TO data.
dref = NEW i( 5 ).
ASSIGN dref->* TO <fs>.
```

Listing 16.21 Using the NEW Instantiation Operator

Similar to accessing reference objects, you can access the components of a data reference using the -> selector, for example, DREF->COMP. If no TYPE specification is provided with the CREATE DATA statement, the anonymous data object is created as a static type of the data reference variable.

Listing 16.22 shows example code to access components of the anonymous data object.

```
TYPES: BEGIN OF ty_mara,
         matnr TYPE matnr,
         mtart TYPE mtart,
       END OF ty_mara.
DATA dref TYPE REF TO ty_mara.
CREATE DATA dref.
dref->matnr = '100'.
```

Listing 16.22 Accessing Components of Anonymous Data Objects

You can use the IS BOUND statement to check if the data reference variable contains a valid reference and is dereferenceable, for example, IF dref IS BOUND.

16.2.4 Assignment between Reference Variables

As with object references, you can perform assignments between data references. When a data reference is assigned, the target reference variable will point to the same data object as the source reference variable.

A *reference variable* can be assigned only to another reference variable. In other words, you can't assign a reference variable to a data object or a reference object, and vice versa.

Like object references, data references also support upcasting and downcasting. When assigning a reference variable, if the static type of the source reference variable is similar to or more special than the static type of the target reference variable, then an upcast is performed. You can use the = assignment operator or the MOVE statement to perform the assignment.

An upcast is performed when the target reference variable is a generic type and the source reference variable is fully typed, as shown in Listing 16.23.

```
TYPES: BEGIN OF ty_mara,
         matnr TYPE matnr,
         mtart TYPE mtart,
       END OF ty_mara.
DATA : dref TYPE REF TO ty_mara, " Fully Typed
       dref1 TYPE REF TO DATA. " Generic type
```



```
CREATE DATA dref.
dref1 = dref. " Upcast
```

Listing 16.23 Upcast

If the static type of the source reference variable is more general than the static type of the target reference variable, it leads to a downcast. The assignment can only be validated at runtime because the type of the source reference variable will be known only at runtime.

For this assignment, the syntax checker can't validate statically and will throw a syntax error if the = assignment operator or the MOVE statement is used for the assignment.

For such assignments, we have to use the "?=" casting operator or the MOVE ?TO statement. An upcast is performed if the source reference variable is of a generic type and the target reference variable is fully typed.

Listing 16.24 shows an example of a downcast.

```
TYPES: BEGIN OF ty_mara,
         matnr TYPE matnr,
         mtart TYPE mtart,
       END OF ty_mara.
DATA : s_mara TYPE ty_mara,
       dref TYPE REF TO ty_mara, " Fully Typed
       dref1 TYPE REF TO DATA. " Generic type
CREATE DATA dref.
GET REFERENCE OF s_mara INTO dref1.
TRY,
dref ?= dref1. " Down Cast
CATCH cx_sy_move_cast_error.
ENDTRY.
```

Listing 16.24 Downcast

16.3 Runtime Type Services

ABAP *Runtime Type Services (RTTS)* is an object-oriented framework that consists of the following elements:

- **Runtime Type Information (RTTI)**
  Using RTTI, you can determine the type information about data objects or instances of classes at runtime during program execution.
- **Runtime Type Creation (RTTC)**
  RTTC allows you to define new data types during the program execution.

The RTTS framework implements a hierarchy of classes, as shown in Figure 16.4, via which you can determine the type information of data objects and define new data types at runtime. The instances of these type classes are known as *type objects*.

A type class exists for each ABAP type, such as elementary types, reference types, complex types (structures and tables), and so on.

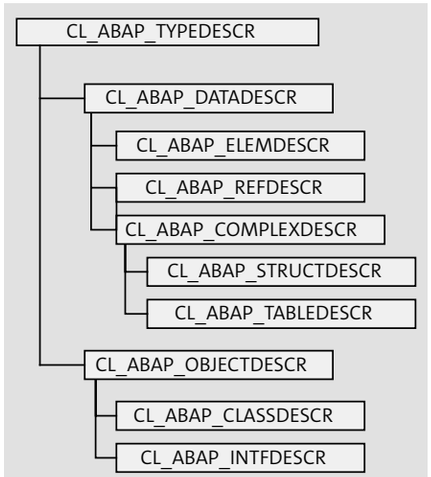


Figure 16.4 Type Class Hierarchy

For each ABAP type, exactly one type object can exist in the program, and you can create new type objects for new types. The type objects contain attributes that describe the type. The class documentation provides information on each type class.

In the following subsections, we'll discuss RTTI and RTTC in detail.

16.3.1 Runtime Type Information

Often, you'll need to determine the type information of an object dynamically. For example, before using a generically typed parameter of a procedure or a generically typed anonymous data object in an arithmetic operation, you may want to check its type, length, and number of decimal places. In such cases, you can use type objects to identify the type information of a data object.

Type objects are created using the methods of the type class. The attributes of the type object contain the type information. Listing 16.25 shows example code to identify the type information.

```
REPORT ZDEMO_EXAMPLES.
TYPES: ty_type TYPE p DECIMALS 2.
DATA: v_data   TYPE ty_type,
      r_typedescr TYPE REF TO cl_abap_typedescr.

START-OF-SELECTION.
r_typedescr = cl_abap_typedescr=>describe_by_data( v_data ).

WRITE: / 'Kind:',     r_typedescr->type_kind.
WRITE: / 'Length:',   r_typedescr->length.
WRITE: / 'Decimals:', r_typedescr->decimals.
```

Listing 16.25 Querying Type Information

In Listing 16.25, we've defined an `R_TYPEDESCR` type object as an instance of type class `CL_ABAP_TYPEDESCR`. The type object is instantiated when the `DESCRIBE_BY_DATA` static method of the type class is called. To this static method, we're passing the data object for which the type information is required.

After the type object is instantiated with the type information, we can access its attributes to determine the type information of the data object. The type object also contains constants that can be used in an `IF` or `CASE` structure to compare the `TYPE_KIND` attribute.

16.3.2 Runtime Type Creation

As discussed in the previous section, type objects are created using RTTI methods of RTTS; these type objects can be used to create new types using the RTTC methods of RTTS. The RTTC methods allow you to create data types dynamically that can be used to define data objects dynamically.

For example, say you want to dynamically create a structure on the fly and use it in a `SELECT` statement. You'll only know the fields of the structure at runtime. In such a situation, you use the RTTC methods to define a data type and use it to create the data object dynamically. The `GET` factory method of the `CL_ABAP_STRUCTDESCR`, `CL_ABAP_TABLEDESCR`, and `CL_ABAP_REFDESCR` classes allows you to create a type object.

The required information to create the type object is passed to the factory method of these classes. For example, the component details are passed to the factory method of the `CL_ABAP_STRUCTDESCR` class to create the type object for structure. Similarly, the structure type object is passed to the factory method of the `CL_ABAP_TABLEDESCR` class to create the table type object. Listing 16.26 shows how the RTTC methods can be used to dynamically define data types and create data objects using these types.

In Listing 16.26, we've defined two selection screen fields to take the table name and the `WHERE` clause as input. The program logic is built to display data from any table given as the input.

To achieve this, in the `GET_TABLE` subroutine, we're dynamically identifying the type information of the given structure to get its components. We're building an internal table on the fly using these components and writing a dynamic `SELECT` statement to get

the data from the requested table. Finally, we're displaying the output using ALV in the DISPLAY_TABLE subroutine.

```
PARAMETERS: p_table TYPE string,
            p_where TYPE string.

DATA:     r_typedescr   TYPE REF TO cl_abap_typedescr,
          r_structdescr TYPE REF TO cl_abap_structdescr,
          r_tabledescr  TYPE REF TO cl_abap_tabledescr,
          r_table       TYPE REF TO data,
          components   TYPE cl_abap_structdescr=>component_table,
          component     LIKE LINE OF components,
          ro_alv          TYPE REF TO cl_salv_table.

FIELD-SYMBOLS : <table> TYPE ANY TABLE.
START-OF-SELECTION.
  PERFORM get_table.
  PERFORM display_table.
*&---------------------------------------------------------------------*
*&      Form  GET_TABLE
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
FORM get_table .
  r_typedescr = cl_abap_typedescr=>describe_by_name( p_table ).
  TRY.
      r_structdescr ?= r_typedescr.
    CATCH cx_sy_move_cast_error.
  ENDTRY.
  components = r_structdescr->get_components( ).
  TRY.
      r_structdescr = cl_abap_structdescr=>get( components ).
      r_tabledescr = cl_abap_tabledescr=>get( r_structdescr ).
    CATCH cx_sy_struct_creation.
    CATCH cx_sy_table_creation .
  ENDTRY.
  TRY.
      CREATE DATA r_table TYPE HANDLE r_tabledescr.
      ASSIGN r_table->* TO <table>.
    CATCH cx_sy_create_data_error.
  ENDTRY.
  TRY.
      SELECT *
      FROM (p_table)
```



```
      INTO CORRESPONDING FIELDS OF TABLE <table> WHERE (p_where).
    CATCH cx_sy_sql_error.
  ENDTRY.
ENDFORM.
*&---------------------------------------------------------------------*
*&      Form  DISPLAY_TABLE
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
FORM display_table .
  TRY.
      CALL METHOD cl_salv_table=>factory
        IMPORTING
          r_salv_table = ro_alv
        CHANGING
          t_table      = <table>.
    CATCH cx_salv_msg.
  ENDTRY.
  ro_alv->display( ).
ENDFORM.
```

Listing 16.26 Dynamic Type Creation

In the GET_TABLE subroutine of Listing 16.26, we're first creating the R_TYPEDESCR type object based on the table name entered on the selection screen. We're downcasting the R_TYPEDESCR type object to R_STRUCTDESCR. The R_STRUCTDESCR type object is an instance of the CL_ABAP_STRUCTDESCR class that contains the RTTC methods to create new type objects.

We're calling the GET_COMPONENTS method to get the components of the structure. Using this information, we call the GET factory method of the CL_ABAP_STRUCTDESCR class to create the structure type object. We pass this structure type object to the GET factory method of the CL_ABAP_TABLEDESCR class to create the table type object.

After we have the table type object, we use the TYPE HANDLE addition with the CREATE statement to instantiate the data reference with the type object. This will create the R_TABLE data reference, the type definition of which is defined in the R_TABLEDESCR type object.

Although the TYPE or LIKE addition allows you to specify a data type or data object, respectively, the TYPE HANDLE addition with the CREATE DATA statement allows you to specify a type object.

After the internal table is defined dynamically, we use it in the SELECT statement to query the given database table and subsequently display the output. As you can see, we're handling the errors at each step; error handling is vital to dynamic programming.

If the user completes the fields as shown in Figure 16.5, the code in Listing 16.26 will generate the output shown in Figure 16.6. We can enhance the code further to select only specific fields of the table and create the structure with only specific fields.

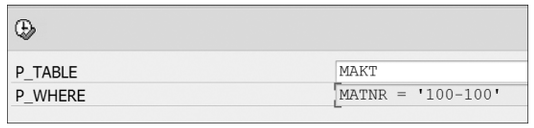


Figure 16.5 Selection Screen for Data Browser

Client	Material	Language	Description	Material description
800	100-100	ZH	外壳	外壳
800	100-100	KO	케이스	케이스
800	100-100	RO	SHOUKAT	SHOUKATFAROOQI
800	100-100	SL	Casings	CASINGS
800	100-100	HR	Casings	CASINGS
800	100-100	DE	Gehäuse	Gehäuse
800	100-100	EN	Casings	CASINGS
800	100-100	FR	Carters	Carters
800	100-100	EL	Casings	CASINGS
800	100-100	JA	ケーシング	ケーシング
800	100-100	PT	Carcaças	Carcaças
800	100-100	RU	Kopnyca	Kopnyca
800	100-100	ES	Cajas	Cajas
800	100-100	BG	Casings	CASINGS
800	100-100	SH	Casings	CASINGS

Figure 16.6 Data Browser Output

In this section, we looked at both RTTI and RTTC. In the next section, we'll look at using dynamic tokens.

16.4 Dynamic Token Specification

With many ABAP statements, you can specify individual operands as tokens dynamically by using a data object enclosed within parentheses. The content of the data object will then be used as the operand of the statement. For example, in `PERFORM (dobj)`, `dobj` (a token) usually is a character-type elementary data object.

The statement will be evaluated at runtime, and if there's a syntax error, it will result in a runtime error. For this reason, catching exceptions is very important while using dynamic tokens in a statement.

The dynamic assignment of operands can cause security risks, especially if the operands are supplied externally. For this reason, the `CL_ABAP_DYN_PRG` class should be used to check the operand thoroughly before using it in the statement. The class documentation for `CL_ABAP_DYN_PRG` provides more information about different scenarios in which the methods of the class can be used.

Some of the statements in which operands can be specified dynamically include the following:

- **`SUBMIT (dobj)`**
  With this statement, you supply the program name dynamically. The following code example shows using the dynamic token with a `SUBMIT` statement:

```
DATA lv_progname TYPE progname.
SELECT SINGLE progname FROM zprogdir INTO lv_progname WHERE ...
SUBMIT (lv_progname).
```

- **`WRITE (dobj) TO..`**
  With this statement, you can pass formatting options dynamically. For example, in the following code, we're passing the formatting of the `SY-DATUM` system date field to the `DATE` character field.

  After the `WRITE` statement is executed, the contents of the `DATE` field will have the formatting of the `SY-DATUM` field:

```
DATA format_date TYPE c LENGTH 8 VALUE 'SY-DATUM'.
DATA date TYPE c LENGTH 10.
date = sy-datum.
WRITE (format_date) TO date.
```

- **`CALL FUNCTION`**
  The `CALL FUNCTION` statement only supports dynamic calls. We can either pass the data object or use a literal for the function module name.

  For example:

```
DATA form_name TYPE C LENGTH 30 VALUE 'ZDEMO_FM'.
CALL FUNCTION (form_name)
EXPORTING
.....
IMPORTING
.....
```

- **`CALL TRANSACTION`**
  Similar to `CALL FUNCTION`, the `CALL TRANSACTION` statement also supports only dynamic calls. The name of the transaction should be supplied dynamically using a data object or statically using a literal.

  For example:

```
DATA tcode TYPE C LENGTH 30 VALUE 'MM03'.
CALL TRANSACTION (tcode).
```

- **`ASSIGN (dobj)`**
  This dynamically assigns the field to a field symbol.

  For example:

```
DATA field TYPE C LENGTH 30 VALUE 'V_MATNR'.
FIELD-SYMBOLS <fs> TYPE ANY.
ASSIGN (field) TO <fs>.
```

- **READ, MODIFY, DELETE, SORT**
  You can specify internal table fields dynamically with these statements.

  For example:

  ```
  DATA field TYPE C LENGTH 30 VALUE 'MATNR'
  SORT itab BY (field).
  ```

- **CLASSES**
  You can access class attributes dynamically. For an instance attribute, you can specify the attribute name dynamically, as follows:

  ```
  ASSIGN oref->(attribute_name) TO <fs>
  ```

  For a static attribute, you can even specify the class name dynamically, as follows:

  ```
  ASSIGN (class_name)=>(attribute_name) TO <fs>
  ```

Apart from operands, the WHERE clause for an Open SQL statement can also be constructed dynamically, as shown in Listing 16.26. With the WHERE clause, the complete statement can be constructed dynamically without any restrictions. The dynamic WHERE clause is supported for internal tables as well.

In this section, we looked at dynamic tokens. In the next section, we'll look at calling procedures dynamically.

16.5 Dynamic Procedure Calls

Procedures can be called dynamically using *dynamic tokens*. For example, if you can determine the subroutine that needs to be called only at runtime, then you can pass the subroutine name using a dynamic token. When calling an external subroutine, the program name can also be passed dynamically as follows:

```
PERFORM (l_name) IN PROGRAM (l_program) IF FOUND.
```

When calling a subroutine dynamically, the system will generate a runtime error if the specified subroutine or program isn't found. To avoid the runtime error, you should use the IF FOUND addition.

Methods can also be called dynamically by passing the class name and the method name with a dynamic token. In fact, the parameters of the method can also be supplied dynamically using the PARAMETER-TABLE addition as follows:

```
CALL METHOD class_name=>method_name PARAMETER-TABLE para.
```


In the preceding statement, we're passing the parameters to the method dynamically using the PARAMETER-TABLE addition. Here, an internal table of type ABAP_PARMBIND_TAB is expected. The line type of ABAP_PARMBIND_TAB consists of three fields: NAME, KIND, and VALUE. The NAME field should contain the formal parameter name.

The KIND field should be filled with the parameter type (whether exporting, importing, returning, etc., from a caller's point of view). The VALUE field should contain the reference of the actual parameter.

We can use the constants of the cl_abap_objectdescr class to fill the KIND field. As with PARAMETER-TABLE, nonclass-based exceptions can be handled using the EXCEPTION-TABLE addition.

As with methods, the parameters for a function module can also be supplied dynamically using PARAMETER-TABLE. The only difference is that PARAMETER-TABLE when used for the function module is of type ABAP_FUNC_PARMBIND_TAB.

Listing 16.27 shows example code using a parameter table to pass parameters dynamically.

```
DATA: fm_name TYPE STRING,
      filename TYPE STRING,
      filetype TYPE C LENGTH 80,
      text_tab LIKE STANDARD TABLE OF LINE,
      fleng    TYPE I,
      ptab TYPE abap_func_parmbind_tab,
      ptab_line LIKE LINE OF ptab,
      etab TYPE abap_func_excpbind_tab,
      etab_line LIKE LINE OF etab.

fm_name = 'GUI_DOWNLOAD'.
filename = 'C:\text.txt'.
filetype = 'ASC'.

ptab_line-NAME = 'FILENAME'.
ptab_line-KIND = abap_func_exporting.
GET REFERENCE OF filename INTO ptab_line-VALUE.
INSERT ptab_line INTO TABLE ptab.

ptab_line-NAME = 'FILETYPE'.
ptab_line-KIND = abap_func_exporting.
GET REFERENCE OF filetype INTO ptab_line-VALUE.
INSERT ptab_line INTO TABLE ptab.

ptab_line-NAME = 'DATA_TAB'.
ptab_line-KIND = abap_func_tables.
GET REFERENCE OF text_tab INTO ptab_line-VALUE.
```

```
INSERT ptab_line INTO TABLE ptab.

ptab_line-NAME = 'FILELENGTH'.
ptab_line-KIND = abap_func_importing.
GET REFERENCE OF fleng INTO ptab_line-VALUE.
INSERT ptab_line INTO TABLE ptab.

...

etab_line-NAME = 'OTHERS'.
etab_line-VALUE = 10.
INSERT etab_line INTO TABLE etab.

CALL FUNCTION fm_name
PARAMETER-TABLE
ptab
EXCEPTION-TABLE
etab.
```

Listing 16.27 Parameter Table

In Listing 16.27, we defined parameter table PTAB as type ABAP_FUNC_PARMBIND_TAB. We're filing the parameter of function module GUI_DOWNLOAD dynamically in internal table PTAB.

The function module is called dynamically, and the parameter table is passed using the PARAMETER-TABLE addition. We're also passing exceptions table ETAB, which is defined as type ABAP_FUNC_EXCPBIND_TAB.

This section looked at the process of calling procedures dynamically. In the next section, we look at a more advanced feature of ABAP: dynamic program generation.

16.6 Dynamic Program Generation

In addition to all the dynamic programming concepts discussed in this chapter, which can be used within an ABAP program, we can also generate ABAP programs dynamically. Generating a program dynamically should be used as a last resort if other dynamic programming techniques don't satisfy your requirements.

Dynamic program generation is for expert programmers and should be used with caution; it can cause serious problems if you don't know what you're doing. The downsides of programs generated dynamically are that they can't be tested like regular programs and can involve serious security risks.

Two types of programs can be generated dynamically:

- **Transient program**
  A *transient program* exists temporarily in the internal session memory and can be accessed only from the current internal session. The GENERATE SUBROUTINE POOL itab NAME prog statement is used to generate a temporary subroutine pool program. Internal table itab should be a character type that contains the source code of the program. The prog variable contains the name of the generated program by which the program can be accessed.

  The generated subroutine pool program can contain local classes or subroutines, which can be accessed from outside of the program. If the source code of the program contains any syntax errors, the program isn't generated.

  Listing 16.28 shows sample code to generate a temporary subroutine pool program. In this example, we're appending internal table IT_SOURCE with the source code of the subroutine pool program and generating the program using the GENERATE SUBROUTINE POOL statement.

```
DATA: it_source  TYPE TABLE OF string,
      program TYPE string ,
      mesg TYPE string.

APPEND 'PROGRAM.' TO it_source.
APPEND 'FORM subr.' TO it_source.
APPEND 'WRITE / ''This is dynamic subroutine''.' TO it_source.
APPEND 'ENDFORM.' TO it_source.

GENERATE SUBROUTINE POOL it_source NAME program MESSAGE mesg.
IF sy-subrc = 0.
PERFORM subr IN PROGRAM (PROGRAM).
ENDIF.
```

  Listing 16.28 Dynamic Subroutine Pool

- **Persistent program**
  Persistent programs exist permanently in the repository and can be accessed just like other programs that are created manually. The INSERT REPORT prog FROM itab statement generates a persistent program. Here, internal table itab contains the source code of the program, and the prog variable contains the program name.

  You should be careful when using this statement because if a program with the same name already exists, it will be overwritten without any warning. To ensure you don't overwrite existing programs, we recommend checking database table TRDIR to see if a program with the same name already exists.

  Listing 16.29 shows example code to generate a program dynamically. In this example, we're filling internal table IT_SOURCE with the source code line by line and then using the INSERT REPORT keyword to generate the program.

```
REPORT ZDYN_PROG.

DATA: it_source TYPE TABLE OF rssource-line.

APPEND  'REPORT ZDYN_EXAMPLE.' TO it_source.

APPEND  'WRITE / ''This program is generated dynamically''.'
         TO it_source.

INSERT REPORT 'ZDYN_EXAMPLE' FROM it_source.
```

Listing 16.29 Dynamic Program Generation

When generating programs dynamically, it's a good idea to create a program statically and use that program as a template to generate new programs. You can use the `READ REPORT INTO itab` statement to read the source code of an existing program, which can then be used as a template to generate a new program.

16.7 Summary

In this chapter, we explored the various concepts that can be used to make ABAP programs dynamic. Dynamic programming may seem intimidating, especially for a beginner, but it opens up many possibilities, such as developing applications that can handle dynamic requirements and work across multiple business functions.

This chapter looked at how field symbols and data references can be used to dynamically point to existing memory of data objects, and you saw how anonymous data objects can be created using data references.

We also discussed the RTTS framework and how it helps to find the type information of program data objects dynamically and to create new data types in the program during runtime. We concluded the chapter with a discussion of calling procedures dynamically and how to generate programs dynamically. We hope this knowledge will help you approach program development from a different perspective.

In the next chapter, we'll discuss the process of debugging, which plays an important role in identifying problems within a program, especially if dynamic elements are involved.

Contents

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

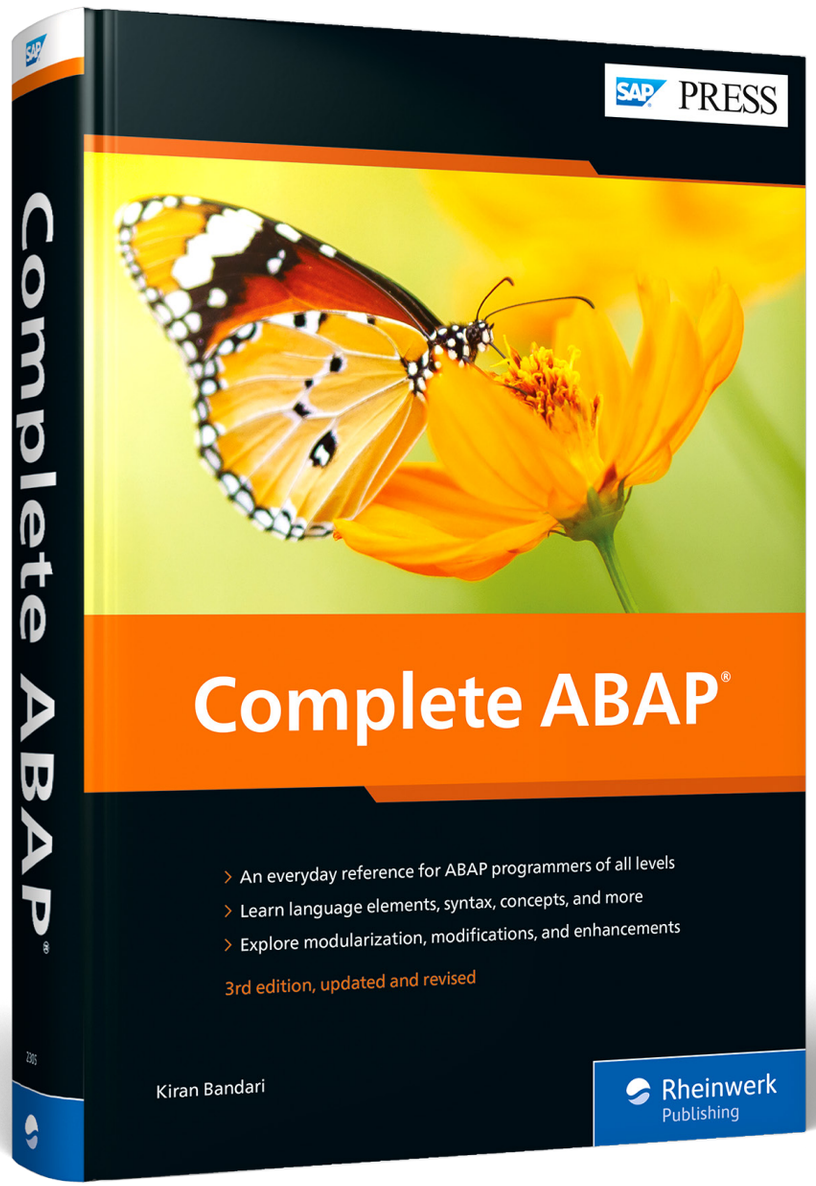
T

U

V

W

X



Kiran Bandari is a solution architect and has been working with ABAP for more than 12 years. He has worked as a lead ABAP consultant on multiple SAP implementations, roll outs, and upgrade projects with a specific focus on custom development using ABAP Objects, Web Dynpro for ABAP, SAP HANA, SAP Fiori, and SAPUI5. He is also an industry trainer and has conducted ABAP training workshops for major clients, including IBM, Accenture, Capgemini, Cognizant, Deloitte, and more. Currently, he offers consulting and training services through his firm Ivyprotech Consulting.

Kiran Bandari

Complete ABAP

912 pages, 2022, $89.95
ISBN 978-1-4932-2305-3

www.sap-press.com/5567