

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.

-  **"Dynamic Programming"**
-  **Contents**
-  **Index**
-  **The Author**

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.

Material	Created On	Created by	Last Change	Material Type
100-100	07.11.1994	BALLER	14.02.2012	HALB
100-101	05.03.1996	KUNITZ	08.01.2007	HALB
100-110	07.11.1994	BALLER	07.06.2003	ROH
100-112	10.03.2011	ZECHA		ROH
100-113	10.03.2011	ZECHA		ROH
100-120	08.11.1994	BALLER	06.04.2011	ROH
100-121	13.05.2009	OBERBOERSCHG	13.05.2009	ROH
100-130	08.11.1994	BALLER	06.04.2011	ROH
100-200	07.11.1994	BALLER	03.01.2012	HALB
100-210	07.11.1994	BALLER	24.01.2003	ROH
100-250	19.12.2002	DAVTSOMT		HAWA

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.

Material	Created On	Created by	Last Change	Material Type
100-100	07.11.1994	BALLER	14.02.2012	HALB
100-101	05.03.1996	KUNITZ	08.01.2007	HALB
100-200	07.11.1994	BALLER	03.01.2012	HALB
100-300	07.11.1994	BALLER	31.03.2015	HALB
100-301	23.01.1996	KUNITZ	08.01.2007	HALB
100-302	20.03.2002	KUNITZ	24.01.2003	HALB
100-400	07.11.1994	BALLER	09.07.2014	HALB
100-401	02.10.1996	KUNITZ	24.01.2003	HALB
100-500	07.11.1994	BALLER	19.12.2011	HALB

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 `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.

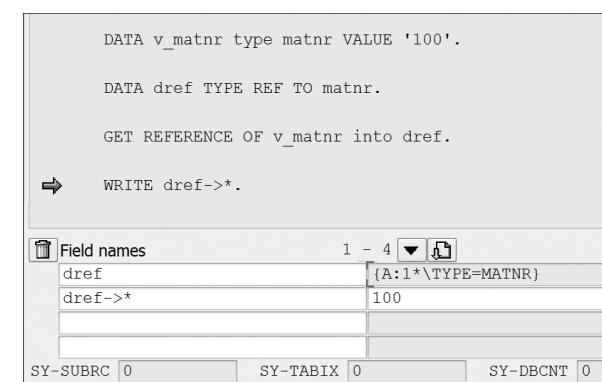


Figure 16.3 Data Reference Variable in Debug Mode

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.

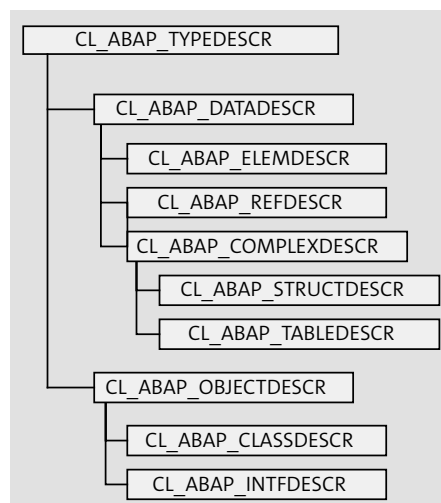


Figure 16.4 Type Class Hierarchy

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_TPEDESCR` type object as an instance of type class `CL_ABAP_TPEDESCR`. 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 zprogrid 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_PARM_BIND_TAB` is expected. The line type of `ABAP_PARM_BIND_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_PARM_BIND_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_excpcbind_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_PARM_BIND_TAB. We're filling 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_EXCP_BIND_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 rsource-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

Acknowledgments	21
Preface	23
1 Introduction to ERP and SAP	27
1.1 Historical Overview	27
1.2 Understanding an ERP System	30
1.2.1 What Is ERP?	30
1.2.2 ERP versus Non-ERP Systems	30
1.2.3 Advantages of an ERP System	32
1.3 Introduction to SAP	33
1.3.1 Modules in SAP	33
1.3.2 Types of Users	34
1.3.3 Role of an ABAP Consultant	35
1.3.4 Changing and Adapting the Data Structure	36
1.4 ABAP Overview	38
1.4.1 Types of Applications	39
1.4.2 RICEF Overview	39
1.5 System Requirements	42
1.6 Summary	43
2 Architecture of an SAP System	45
2.1 Introduction to the Three-Tier Architecture	45
2.2 SAP Implementation Overview	47
2.2.1 SAP GUI: Presentation Layer	47
2.2.2 Application Servers and Message Servers: Application Layer	49
2.2.3 Database Server/Relational Database Management System: Database Layer	55
2.2.4 SAP HANA Introduction	57
2.3 Data Structures	60
2.3.1 Client Overview	61
2.3.2 Client-Specific Data and Cross-Client Data	61

2.3.3	Repository	64
2.3.4	Packages	64
2.3.5	Transport Organizer	65
2.4	Summary	70
3	Introduction to the ABAP Environment	71
<hr/>		
3.1	SAP Environment	71
3.1.1	ABAP Programming Environment	72
3.1.2	Logging On to the SAP Environment	72
3.1.3	Elements of the SAP Screen	73
3.1.4	Transaction Codes	75
3.1.5	Opening and Navigating with Transactions	76
3.2	ABAP Workbench Overview	80
3.2.1	ABAP Editor	80
3.2.2	Function Builder	82
3.2.3	Class Builder	83
3.2.4	Screen Painter	83
3.2.5	Menu Painter	85
3.2.6	ABAP Data Dictionary	86
3.2.7	Object Navigator	88
3.3	Eclipse IDE Overview	89
3.4	SAP Business Technology Platform, ABAP Environment	94
3.5	Summary	95
4	ABAP Programming Concepts	97
<hr/>		
4.1	General Program Structure	98
4.1.1	Global Declarations	98
4.1.2	Procedural Area	99
4.2	ABAP Syntax	99
4.2.1	Basic Syntax Rules	100
4.2.2	Chained Statements	101
4.2.3	Comment Lines	102
4.3	ABAP Keywords	102
4.4	Introduction to the TYPE Concept	103

4.4.1	Data Types	104
4.4.2	Data Elements	117
4.4.3	Domains	120
4.4.4	Data Objects	123
4.5	ABAP Statements	126
4.6	Creating Your First ABAP Program	128
4.7	Summary	133
5	Structures and Internal Tables	135
<hr/>		
5.1	Defining Structures	136
5.1.1	When to Define Structures	138
5.1.2	Local Structures	138
5.1.3	Global Structures	141
5.1.4	Working with Structures	144
5.1.5	Use Cases	145
5.2	Internal Tables	146
5.2.1	Defining Internal Tables	146
5.2.2	Types of Internal Tables	149
5.2.3	Table Keys	152
5.2.4	Working with Internal Tables	156
5.2.5	Control Break Statements	162
5.3	Introduction to Open SQL Statements	167
5.3.1	Database Overview	168
5.3.2	Selecting Data from Database Tables	176
5.3.3	Selecting Data from Multiple Tables	178
5.4	Processing Data from Databases via Internal Tables and Structures	181
5.5	Introduction to the Debugger	183
5.6	Practice	186
5.7	Summary	186
6	User Interaction	187
<hr/>		
6.1	Selection Screen Overview	187
6.1.1	PARAMETERS	189
6.1.2	SELECT-OPTIONS	195

6.1.3	SELECTION-SCREEN	202
6.1.4	Selection Texts	203
6.2	Messages	204
6.2.1	Types of Messages	205
6.2.2	Messages Using Text Symbols	206
6.2.3	Messages Using Message Classes	207
6.2.4	Dynamic Messages	209
6.2.5	Translation	210
6.3	Summary	211
7	Modularization Techniques	213
7.1	Modularization Overview	213
7.2	Program Structure	217
7.2.1	Processing Blocks	217
7.2.2	Event Blocks	229
7.2.3	Dialog Modules	230
7.2.4	Procedures	231
7.3	Events	231
7.3.1	Program Constructor Events	231
7.3.2	Reporting Events	232
7.3.3	Selection Screen Events	238
7.3.4	List Events	238
7.3.5	Screen Events	239
7.4	Procedures	240
7.4.1	Subroutines	242
7.4.2	Function Modules	250
7.4.3	Methods	258
7.5	Inline Declarations	264
7.5.1	Assigning Values to Data Objects	264
7.5.2	Using Inline Declarations with Table Work Areas	265
7.5.3	Avoiding Helper Variables	265
7.5.4	Declaring Actual Parameters	265
7.6	Summary	266

8	Object-Oriented ABAP	267
8.1	Procedural Programming versus Object-Oriented Programming	267
8.2	Principles of Object-Oriented Programming	271
8.2.1	Attributes	273
8.2.2	Static and Instance Components	274
8.2.3	Methods	274
8.2.4	Objects	277
8.2.5	Constructor	278
8.3	Encapsulation	279
8.3.1	Component Visibility	280
8.3.2	Friends	282
8.3.3	Implementation Hiding	283
8.4	Inheritance	286
8.4.1	Defining Inheritance	287
8.4.2	Abstract Classes and Methods	290
8.4.3	Final Classes and Methods	293
8.4.4	Composition	294
8.4.5	Refactoring Assistant	295
8.5	Polymorphism	296
8.5.1	Static and Dynamic Types	296
8.5.2	Casting	298
8.5.3	Dynamic Binding with the Call Method	302
8.5.4	Interfaces	304
8.5.5	Events	311
8.6	Working with the Extensible Markup Language	313
8.6.1	XML Overview	314
8.6.2	XML Processing Concepts	315
8.7	Summary	317
9	Exception Handling	319
9.1	Exceptions Overview	319
9.2	Procedural Exception Handling	320
9.2.1	Maintaining Exceptions Using Function Modules	320
9.2.2	Maintaining Exceptions Using Methods	322
9.2.3	Maintaining Exceptions for Local Classes	323
9.3	Class-Based Exception Handling	324

9.3.1	Raising Exceptions	325
9.3.2	Catchable and Noncatchable Exceptions	326
9.3.3	Defining Exception Classes Globally	330
9.3.4	Defining Exception Classes Locally	333
9.4	Messages in Exception Classes	333
9.4.1	Using the Online Text Repository	334
9.4.2	Using Messages from a Message Class	337
9.4.3	Using the MESSAGE Addition to Raise an Exception	339
9.5	Summary	339
10	ABAP Data Dictionary	341
10.1	Database Tables	342
10.1.1	Creating a Database Table	344
10.1.2	Indexes	353
10.1.3	Table Maintenance Generator	357
10.1.4	Foreign Keys	361
10.1.5	Include Structure	364
10.1.6	Append Structure	366
10.2	Views	367
10.2.1	Database Views	368
10.2.2	Projection Views	370
10.2.3	Maintenance Views	372
10.2.4	Help Views	374
10.2.5	ABAP Core Data Services Views	374
10.3	Data Types	378
10.3.1	Data Elements	378
10.3.2	Structures	382
10.3.3	Table Types	384
10.4	Type Groups	388
10.5	Domains	389
10.6	Search Helps	392
10.6.1	Elementary Search Helps	393
10.6.2	Collective Search Helps	396
10.6.3	Assigning a Search Help	399
10.6.4	Search Help Exits	400
10.7	Lock Objects	400
10.8	Summary	404

11	Persistent Data	405
11.1	Working with Data in Databases	406
11.1.1	Open SQL	407
11.1.2	Logical Unit of Work	417
11.2	ABAP Object Services	423
11.2.1	Persistence Service Overview	423
11.2.2	Building Persistent Classes	424
11.2.3	Working with Persistent Objects	427
11.3	File Interfaces	428
11.3.1	Working with Files in the Application Server	428
11.3.2	Working with Files in the Presentation Layer	431
11.4	Data Clusters	432
11.4.1	Exporting Data Clusters to Databases	434
11.4.2	Importing Data Clusters	434
11.5	Security Concepts	434
11.6	Summary	436
12	Dialog Programming	439
12.1	Screen Events	440
12.2	Screen Elements and Flow Logic	443
12.2.1	Components of a Dialog Program	444
12.2.2	Screens	448
12.2.3	Screen Elements	452
12.3	Basic Screen Elements	455
12.3.1	Text Fields	456
12.3.2	Checkboxes and Radio Buttons	457
12.3.3	Push Buttons	459
12.4	Input/Output Fields	460
12.5	List Box	461
12.6	Table Controls	462
12.6.1	Create a Table Control without a Wizard	463
12.6.2	Create a Table Control with a Wizard	467
12.7	Tabstrip Controls	468
12.8	Subscreens	470

12.9 Working with Screens	471
12.9.1 Screen Flow Logic	472
12.9.2 GUI Status	474
12.9.3 GUI Title	476
12.9.4 Modifying Screen Fields Dynamically	476
12.9.5 Field Help and Input Help	478
12.9.6 Screen Sequence	479
12.9.7 Assigning Transaction Codes	480
12.10 Control Framework	483
12.10.1 Using Container Controls	483
12.10.2 Implementing Custom Controls	484
12.11 Practice	486
12.11.1 Application Flow	487
12.11.2 Delete Functionality	488
12.11.3 Validations and Autofills	488
12.12 Summary	489
13 List Screens	491
<hr/>	
13.1 Program Types	491
13.1.1 Executable Programs	492
13.1.2 Module Pool Programs	493
13.1.3 Function Groups	494
13.1.4 Class Pools	494
13.1.5 Interface Pools	494
13.1.6 Subroutine Pools	495
13.1.7 Type Pools	495
13.1.8 Include Programs	495
13.2 Program Execution	495
13.2.1 Executable Program Flow	496
13.2.2 Module Pool Program Flow	497
13.2.3 Calling Programs Internally	497
13.3 Memory Organization	498
13.4 List Events	502
13.4.1 TOP-OF-PAGE	502
13.4.2 END-OF-PAGE	504
13.4.3 AT LINE-SELECTION	505
13.4.4 AT USER-COMMAND	506

13.5 Basic Lists and Detail Lists	507
13.6 Classical Reports	511
13.7 Interactive Reports	511
13.7.1 HIDE	511
13.7.2 READ LINE	515
13.7.3 GET CURSOR	515
13.7.4 DESCRIBE LIST	516
13.8 Practice	516
13.9 Summary	517
14 Selection Screens	519
<hr/>	
14.1 Defining Selection Screens	520
14.2 Selection Screen Events	521
14.3 Input Validations	523
14.4 Selection Screen Variants	525
14.4.1 Creating a Variant	526
14.4.2 Variant Attributes	529
14.4.3 Table Variables from Table TVARVC	530
14.4.4 Dynamic Date Calculation	532
14.4.5 Dynamic Time Calculation	533
14.4.6 User-Specific Variables	534
14.5 Executing Programs in the Background	534
14.6 Displaying and Hiding Screen Elements Dynamically	536
14.7 Calling Programs via Selection Screens	538
14.8 Summary	538
15 SAP List Viewer Reports	541
<hr/>	
15.1 Standard ALV Reports Using the Reuse Library	542
15.1.1 List and Grid Display: Simple Reports	543
15.1.2 Block Display	551
15.1.3 Hierarchical Sequential Display	555
15.2 Interactive Reports	559
15.2.1 Loading a Custom GUI Status	560

15.2.2	Reacting to User Actions	564
15.2.3	Printing TOP-OF-PAGE	565
15.3	ALV Reports Using the Control Framework	566
15.4	ALV Object Model	568
15.4.1	Table Display	569
15.4.2	Hierarchical Display	570
15.4.3	Tree Object Model	574
15.5	SAP List Viewer with Integrated Data Access	576
15.6	Summary	578
16	Dynamic Programming	579
<hr/>		
16.1	Field Symbols	580
16.1.1	Using Field Symbols to Make Programs Dynamic	582
16.1.2	Defining Field Symbols	588
16.1.3	Assigning a Data Object	589
16.1.4	Checking If a Field Symbol Is Assigned	592
16.1.5	Unassigning a Field Symbol	593
16.1.6	Casting	593
16.2	Data References	594
16.2.1	Defining Reference Variables	594
16.2.2	Getting Data References	595
16.2.3	Anonymous Data Objects	597
16.2.4	Assignment between Reference Variables	598
16.3	Runtime Type Services	599
16.3.1	Runtime Type Information	600
16.3.2	Runtime Type Creation	601
16.4	Dynamic Token Specification	604
16.5	Dynamic Procedure Calls	606
16.6	Dynamic Program Generation	608
16.7	Summary	610
17	Debugging	611
<hr/>		
17.1	Classic Debugger	612
17.1.1	Activating and Using the Classic Debugger	612

17.1.2	Field View	616
17.1.3	Table View	616
17.1.4	Breakpoints View	617
17.1.5	Watchpoints View	617
17.1.6	Calls View	618
17.1.7	Overview View	619
17.1.8	Settings View	619
17.1.9	Additional Features	620
17.2	New Debugger	623
17.2.1	User Interface and Tools	623
17.2.2	Layout and Sessions	625
17.3	ABAP Managed Database Procedures Debugger	627
17.4	Using the Debugger to Troubleshoot	627
17.5	Using the Debugger as a Learning Tool	629
17.6	Summary	630
18	Forms	631
<hr/>		
18.1	SAP Scripts	633
18.1.1	Overview and Layout	633
18.1.2	Creating the Form Layout	636
18.1.3	Maintaining Window Details	642
18.1.4	Processing Forms with Function Modules	647
18.2	Smart Forms	654
18.2.1	Overview and Layout	654
18.2.2	Maintaining the Global Settings	657
18.2.3	Maintaining Elements	659
18.2.4	Driver Program	674
18.3	SAP Interactive Forms by Adobe	677
18.3.1	Form Interface	678
18.3.2	Form Context and Layout	684
18.3.3	Driver Program	695
18.3.4	Downloading the Form as a PDF	696
18.4	Summary	697

19 Interfaces	699
19.1 Batch Data Communication	700
19.1.1 Direct Input	701
19.1.2 Batch Input	703
19.2 Business Application Programming Interfaces	713
19.2.1 Business Object Types and Business Components	713
19.2.2 BAPI Development via BAPI Explorer	714
19.2.3 Standardized Business Application Programming Interfaces	717
19.2.4 Standardized Parameters	718
19.2.5 Implementing Business Application Programming Interfaces	719
19.3 EDI/ALE/IDocs	727
19.3.1 Electronic Data Interchange	727
19.3.2 Application Link Enabling	732
19.3.3 Intermediate Documents	734
19.3.4 System Configurations	744
19.3.5 Inbound/Outbound Programs	751
19.4 Legacy System Migration Workbench	754
19.4.1 Getting Started	755
19.4.2 Migration Process Steps	756
19.5 Web Services	765
19.5.1 Creating a Web Service	767
19.5.2 Consuming Web Services	771
19.6 OData Services	777
19.6.1 Data Model Definition	779
19.6.2 Service Maintenance	782
19.6.3 Service Implementation	784
19.7 Extensible Stylesheet Language Transformations	788
19.7.1 Serialization	789
19.7.2 Deserialization	789
19.8 XML and JSON Data Representation	790
19.9 WebSockets (ABAP Channels and Messages)	792
19.9.1 Creating an ABAP Messaging Channel	793
19.9.2 Creating a Producer Program	795
19.9.3 Creating a Consumer Program	796
19.10 Summary	799

20 Modifications and Enhancements	801
20.1 Customization Overview	801
20.2 Modification Overview	803
20.3 Using the Modification Assistant	803
20.3.1 Modifications to Programs	804
20.3.2 Modifications to the Class Builder	805
20.3.3 Modifications to the Screen Painter	807
20.3.4 Modifications to the Menu Painter	808
20.3.5 Modifications to the ABAP Data Dictionary	808
20.3.6 Modifications to Function Modules	808
20.3.7 Resetting to Original	809
20.4 Using the Modification Browser	810
20.5 Enhancements Overview	811
20.6 User Exits	812
20.7 Customer Exits	813
20.7.1 Create a Customer Exit	816
20.7.2 Function Module Exits	818
20.7.3 Screen Exits	818
20.7.4 Menu Exits	820
20.8 Business Add-Ins	822
20.8.1 Overview	822
20.8.2 Defining a BAdI	823
20.8.3 Implementing a Business Add-In	830
20.8.4 Implementing a Fallback Class	832
20.8.5 Calling a Business Add-In	833
20.9 Enhancement Points	834
20.9.1 Explicit Enhancements	834
20.9.2 Implicit Enhancements	837
20.10 Business Transaction Events	840
20.10.1 Implementing a Business Transaction Event	841
20.10.2 Testing a Custom Function Module	844
20.11 Summary	845

21 Test and Analysis Tools	847
21.1 Overview of Tools	847
21.2 ABAP Unit	849
21.2.1 Eliminating Dependencies	850
21.2.2 Implementing Mock Objects	852
21.2.3 Writing and Implementing Unit Tests	853
21.3 Code Inspector	860
21.4 Selectivity Analysis	862
21.5 Process Analysis	864
21.6 Memory Inspector	866
21.6.1 Creating Memory Snapshots	867
21.6.2 Comparing Memory Snapshots	868
21.7 Table Call Statistics	869
21.8 Performance Trace	871
21.8.1 SQL Trace	874
21.8.2 Remote Function Call Trace	876
21.8.3 Enqueue Trace	877
21.8.4 Buffer Trace	878
21.9 ABAP Trace/Runtime Analysis	879
21.9.1 Running ABAP Trace	879
21.9.2 Analyzing the Results	881
21.10 Single-Transaction Analysis	883
21.11 Dump Analysis	886
21.12 Summary	887
The Author	889
Index	891

Index

\$TMP	131	ABAP Data Dictionary (Cont.)	
A		<i>object</i>	87
ABAP	38	<i>screen</i>	86, 341
<i>addition</i>	100	<i>search help</i>	392
<i>conversion</i>	40	<i>selection text</i>	204
<i>environment</i>	71	<i>smart forms</i>	658
<i>extension</i>	41	<i>table control</i>	464
<i>forms</i>	42	<i>type group</i>	388
<i>interfaces</i>	40	<i>type pools</i>	495
<i>reports</i>	39	<i>view</i>	367
<i>RICEF</i>	39	ABAP Debugger	611
<i>statement</i>	100, 126	<i>learning tool</i>	629
<i>System requirements</i>	42	ABAP Developer View	886
abap_false	285	ABAP Development Tools (ADT)	71, 90, 374
ABAP CDS view	342, 368	ABAP Dispatcher	50, 51
<i>access</i>	377	ABAP Editor	71, 80, 129, 158, 493, 775
<i>create</i>	374	<i>back-end editor</i>	81
<i>data definition</i>	375	<i>creating variant</i>	526
<i>define</i>	374	<i>front-end editor (new)</i>	80
<i>replacement objects</i>	377	<i>front-end editor (old)</i>	80
<i>template</i>	376	<i>modify programs</i>	805
ABAP CDS views	374	<i>program attributes</i>	128
ABAP channel	794	<i>settings</i>	81
<i>consumer program</i>	796	<i>your first program</i>	128
<i>producer program</i>	795	ABAP in Eclipse	
ABAP channels	793	ABAP CDS views	374
ABAP consultant	35	ABAP keyword documentation	103
ABAP Data Dictionary	71, 78, 86,	ABAP Managed Database Procedures	
341, 404, 460		(AMDP)	612, 627
ABAP CDS views	368	ABAP messaging channel	793
BAPI	720	<i>create</i>	793
<i>create data element</i>	117	ABAP Object	215
<i>create domain</i>	121	ABAP Objects	216, 230, 242, 840
<i>database table</i>	342	ABAP Objects Control Framework	483
<i>data type</i>	378	ABAP Object Services	423, 436
DDL	406	ABAP processor	441
<i>domain</i>	389	ABAP program	97, 444
<i>elementary search help</i>	393	<i>declaration</i>	217
<i>global table type</i>	249	<i>global declarations area</i>	98
GTT	346	<i>procedural area</i>	98, 99
I_STRUCTURE_NAME	545	<i>statement</i>	223
<i>interface</i>	679	<i>structure</i>	98, 217
<i>lock object</i>	400	ABAP programming environment	72
<i>modification</i>	808	ABAP push channel	793, 872
<i>Modification Browser</i>	811	ABAP runtime environment	
<i>nongeneric data type</i>	191	<i>event block</i>	226
		<i>processing block</i>	219
		<i>processing blocks</i>	214

ABAP runtime environment (Cont.)
 selection screen 519
 ABAP statement
 attributes 129
 call 127
 control 127
 declarative 127
 modularization 127
 Open SQL 127
 operational 127
 processing blocks 215
 ABAP System Central Services (ASCS) 50
 ABAP Trace 849, 876, 879
 analysis 885
 analyze results 881
 call 885
 results screen 881
 run 879
 ABAP Unit 848–850
 code example 859
 eliminate dependencies 850
 fixture 853
 implement mock object 852
 predefined method 854
 SETUP 854
 TEARDOWN 854
 unit test 853
 ABAP Workbench 42, 70, 71, 77, 80, 250
 ABAP Data Dictionary 341
 Class Builder 83
 Function Builder 82
 Menu Painter 85
 object 131
 Object Navigator 88
 Screen Painter 83, 439
 ABSTRACT 291
 Abstract class 290
 Access control 407
 ActiveX 545
 Actual parameter 242
 Addition 100, 103
 Address
 element 662
 node 688
 Add\|u003csubobject\|>() 717
 Adobe Document Server (ADS) 677
 Adobe LiveCycle Designer 677
 Layout tab 691
 Advance Business Application
 Programming (ABAP) 39
 Agent class 424, 427
 Aggregation function 410
 ALE 732
 inbound process 734
 layer 732
 outbound process 732
 RFC destination 745
 system configuration 744
 tRFC 746
 Alias 310
 Alternative node 691
 ALV 865
 block display 551
 CL_GUI_ALV_GRID 541
 component 542
 Control Framework 566
 display 585
 dynamic subroutine 560
 grid display 546
 hierarchical sequential display 555
 layout 551
 library 541
 list and grid display 543
 object model 541
 parameter 544
 report 541
 reuse library 542
 simple report 543
 standard reports 542
 with integrated data access 576
 ALV grid control 483, 566, 567
 ALV object model 231, 568, 582
 class 568
 default status 574
 hierarchical display 570, 572
 set GUI status 570
 table display 569
 tree display 575
 tree object model 574
 AMDP Debugger 611, 612, 627
 breakpoint 627
 capabilities 627
 prerequisites 627
 American Standard Code for Information
 Interchange (ASCII) 107
 Analysis 847
 Anomaly 172
 Anonymous data object 597
 access component 598
 Any field 152
 Any table 152
 API 283, 699
 APPEND 156
 APPENDING TABLE 412

Appending type 429
 Append structure 143, 366
 add 366
 Application flow 487
 Application layer 45, 46, 49, 70, 217
 Application Link Enabling (ALE) 727
 Application server 49, 405
 components 50
 file 428
 Application toolbar 444
 Architecture 45
 Arithmetic operations 416
 Array fetch 177
 ASSIGNING addition 592
 Assignment operator 100
 asXML 792
 Asynchronous data update 707
 AT LINE-SELECTION 229
 AT LINE-SELECTION event 505
 AT SELECTION-SCREEN 521
 ON BLOCK 522
 ON END OF \|u003cfield\|> 522
 ON EXIT-COMMAND 523
 ON HELP-REQUEST FOR \|
 u003cfield\|> 523
 ON RADIO BUTTON GROUP 522
 ON \|u003cfield\|> 522
 OUTPUT 521
 AT SELECTION-SCREEN block 225
 Attributes 271
 AT USER-COMMAND event 506
 Authorization group 358
 Authorization object 435, 436
 Automatic type conversion 112
 Automation Controller 483
 Average 410
 AVG 410
B
 Background mode 707
 Background processing 519
 executing programs 534
 BAdI 811, 822
 call 833
 create 825
 create definition 824
 create interface 829
 define 823
 definition part 822
 enhancement spot 824
 fallback class 832
 BAdI (Cont.)
 filter values 827
 IF_FILTER_CHECK_AND_F4 interface 828
 implementation 830
 implementation part 822
 BAdI Builder 824
 BAPI 231, 713, 714
 address parameters 718
 BAPI Explorer 714
 business component 714
 business object type 714, 722
 change parameters 718
 class method 716
 conventions 720
 development phases 715
 documentation 726
 extension parameters 718
 implementation 719
 instance method 716
 releasing 727
 return parameters 718
 standardized parameter 718
 test run parameters 718
 tool 719
 use case 713
 BAPI Explorer 714
 tabs 714
 Basic list 491, 510, 514, 517
 Basis 70
 BASIS Developer View 887
 Batch Data Communication (BDC) 699, 700
 batch input 703
 direct input 701
 Batch input 700, 703
 BDCDATA 704
 CALL TRANSACTION 707
 create a material 703
 create program 704
 create program with Transaction
 Recorder 708
 SESSION 707
 update mode 708
 BDCDATA structure 706
 Billing document report 512
 Binary search 154
 Block 522
 Block display 542
 example 554
 function module 551
 Boxed
 structure 383
 Boxed data type 382, 383

- Box screen element 454
 - Breakpoint 183, 613
 - AMDP Debugger 627
 - in a call 621
 - setup 613, 628
 - view 617
 - BTE 811
 - event 842
 - implement 841
 - interface 840
 - test custom function module 844
 - Buffer 57
 - Buffering 352, 869
 - permissions 352
 - Buffer trace 871, 878
 - Bundling technique 419
 - Business Add-In (BAdI) 37
 - Business Address Services (BAS) 662
 - Business Application Programming
 - Interface 699
 - Business component 714
 - Business Object Repository (BOR)
 - BAPI 720
 - Business object type 714
 - BOR 722
- C**
- Calendar control 483
 - CALL FUNCTION 215, 605
 - statement 256
 - Calling program 544
 - CALL METHOD statement 215, 262
 - CALL SCREEN 446
 - CALL SCREEN statement 480
 - Call sequence 500
 - Call stack 618
 - Call statement 127
 - CALL SUBSCREEN 470
 - CALL TRANSACTION 232, 605, 707, 711
 - Cancel() 717
 - Candidate key 169
 - Cardinality 364
 - CASE statement 416, 823
 - Casting 593, 594
 - implicit or explicit 594
 - narrowing cast 299
 - widening cast 299, 301
 - CATCH 325
 - statement 328
 - Catchable exception 326
 - CDS view 410, 777
 - Center of Excellence (CoE) 35
 - CHAIN 473
 - Chained statement 101
 - Change() 717
 - CHANGING 245, 247, 254
 - output parameters 246
 - Character
 - data type 107
 - field 107
 - Character literal 124
 - CHECK 221, 222
 - statement 220
 - check_count 285
 - check_status 285
 - Checkbox 454, 457
 - Check table 361–363
 - cl_notification_api 284
 - Class 258, 271
 - abstract 290
 - final 293
 - friend 282
 - global 293
 - pool 259
 - properties 259
 - Class-based exception 324
 - Class Builder 71, 83, 258, 292, 293, 295, 309, 829
 - class pool 494
 - components 806
 - exception class 331
 - interface pools 494
 - Methods tab 259
 - modification 805
 - persistent class 424
 - test class 853
 - visibility 260
 - Classical list 491
 - Classical report 511
 - Classic BAdIs 822
 - Classic Debugger 183, 611, 612, 614
 - activate 612
 - additional features 620
 - breakpoint 614, 617
 - calls view 618
 - execute code 615
 - field view 616
 - jumping to statements 620
 - object view 620
 - overview view 619
 - program status 622
 - settings 613
 - settings view 619

- Classic Debugger (Cont.)
 - shortcut 623
 - table view 616
 - view 615
 - watchpoints view 617
 - Classic view 368
 - Class pool 494
 - Client 60, 61
 - data 61
 - specific data 61
 - Client proxy 771
 - CLOSE_FORM 647
 - CLOSE DATASET 430
 - CLUSTD 434
 - CLUSTR 434
 - CMOD enhancement 813
 - Code Inspector 848, 860
 - results 860
 - use case 860
 - Code pushdown 175, 368
 - COLLECT 158
 - Collective search help 393, 396, 397
 - create 397
 - Collision check 402
 - Command node 669
 - Comment 102
 - COMMIT WORK 418, 726
 - Comparison operators 412
 - Complex data type 114
 - Component 135
 - Component type 142
 - Composer 633
 - Composition 294
 - relationship 294
 - Constant 126
 - Constant window 636
 - Constructor 278, 335
 - instance 278
 - static 279
 - Consumer program 798
 - Container control 483
 - Context menu 451
 - Control break statement 162, 163
 - rules 166
 - Control break statements
 - AT END OF 162
 - AT FIRST 162
 - AT LAST 162
 - AT NEW comp 162
 - Control Framework 440, 483, 542, 546
 - ALV 566
 - grid display 545
 - Control Framework (Cont.)
 - server and client side 483
 - Controlling (CO) 33
 - Control record 730
 - Control statement 127
 - Conversion
 - routine 121, 122
 - rule 110, 112
 - Conversion exit 390
 - function module 390
 - Conversion logic 701
 - Conversion program 40
 - Conversion routine 390
 - Core data services (CDS) 175
 - Count 411
 - CREATE OBJECT statement 258
 - Cross-client 61
 - data 61
 - CRUD 784
 - Currency field 349
 - Custom code
 - execute 359
 - Custom container 455, 483, 484
 - Custom control
 - create 484
 - Custom development 802
 - Customer development 36, 38
 - Customer enhancement 36, 37
 - Customer exit 811, 813, 816
 - create 816
 - function module 813
 - function module exit 818
 - menu exit 820
 - screen exit 818
 - type 814
 - Customization 36, 38
 - Customizing and development client
 - (CUST) 63
 - Customizing Include (CI) 143
 - CX_DYNAMIC_CHECK 328, 329
 - CX_NO_CHECK 329
 - CX_ROOT 328
 - CX_STATIC_CHECK 329
 - CX_SY_ 327
- D**
- DATA 125
 - DATA(.) 264
 - Database
 - relationship 171
 - Database access statement 127

Database interface 441
Database kernel 354
Database layer 45, 47, 55, 70
Database lock 419
Database LUW 345, 417, 436
 database lock 419
 dialog step 418
Databases
 fetching data 408
 persistent data 406
Database structure 168
Database table 138, 176, 342, 405
 append structure 366
 components 344
 create 344
 Currency/Quantity Fields tab 349
 enhancement category 350
 fields tab 348
 hint 354
 include structure 364, 365
 index 353
 persistent data 405
 SELECT...ENDSELECT 177
 technical setting 344
 unique and nonunique indexes 356
Database tables
 Display/Maintenance tab 346
 SELECT SINGLE 176
Database view 368
 create 368
Data browser 346
Data class 351
Data cluster 405, 432
 administration section 432
 data section 432
 export 434
 exporting to databases 434
 import 434
 media 432
 persistent data 405
Data Control Language (DCL) 407
Data definition 341
Data Definition Language (DDL) 87, 168, 406
Data element 117, 118, 348, 378
 activate 119
 change 118
 domain 120
 global user-defined elementary type 117
 modify 808
 relationship with domains and fields 120
 search help 399
Data format 114
Data inconsistency 112
Data Manipulation Language (DML) 86, 168, 407
Data model definition 778
Data modeling 258
Data node 690
Data object 98, 123
 anonymous 597
 constant 126
 DATA 125
 declaration 106
 field symbol 589
 inline declaration 125
 literal 124
 named 597
 PARAMETERS 125
 predefined type 113
 reference 596
 text symbol 126
 user-defined type 114
 variable 125
Data record 730
Data reference 594
 debug mode 595
 dereference 596
 get 595
 initial 595
 variable 594
Data references 580
Data security 435
Data structure 60
Data transfer 700
 frequency 700
Data type 88, 104, 107, 117, 348, 378
 data element 378
 data format 114
 documentation 379
 further characteristics 380
 output length 116
 structure 382
 tab 118, 378
 table type 384
 value list 119
Debugging 136, 183, 611, 628
 breakpoint 183
 Classic Debugger 183
 exit 185
 New Debugger 183
 troubleshooting 627
Debug session 627
Declarative statement 127
Deep structure 137

Default key 152
Delegation of tasks 270
DELETE 168
Delete() 717
Delete anomaly 172
DELETE DATASET 430
Delivery classes 346
Dependency 851
DEQUEUE 400
DESCRIBE LIST statement 516
Desktop 623
Destination 422
Detail list 491, 507, 514
Dialog box container 483
Dialog module 215, 217, 226, 230, 441, 444
 selection screen 238
Dialog programming
 component 444
 practice application 486
Dialog step 217, 418
 database LUW 418
Dialog transaction 443
Diamond problem 305
Direct input 700, 701
 manage 702
 program 702
Dispatch control 733
Distributed environment 732
Distribution model 732
Docking container 484
Domain 88, 170, 389
 attach to data element 122
 bottom-up approach 121
 create 121
 format 389
 output length 122
 relationship with data elements and fields 120
 top-down approach 121
Domains 120
Downcasting 598, 599
Drilldown report 491, 507
Dropdown list box 454
Dual-stack system 50
Dump analysis 849, 886
 views 886
Dynamic binding
 CALL method 302
Dynamic date 532
 calculation 532
 selection 533
Dynamic element 209
Dynamic enhancement point 835
Dynamic message 209
Dynamic procedure call 606
Dynamic program 187
Dynamic program generation 608
 persistent program 609
 transient program 609
Dynamic programming 152, 579, 580
Dynamic RFC destination 746
Dynamic subroutine pool 609
Dynamic text 664
Dynamic time 533
 calculation 533
 selection 533
Dynamic token 580, 604, 606
Dynamic token specification 604
Dynamic type 298
Dynpro 187, 447, 448

E

Eclipse 71, 89
 installation 90
 installation wizard 90
 Project Explorer 93
Eclipse IDE 374
EDI 727
 benefits 728
 inbound process 731
 outbound and inbound processing 730
 process 728
 system configuration 744
 using IDocs 729
Element 648
 address 662
 call 648
 graphics 661
 maintain 659
 program line 668
 table 666
 text 664
Elementary data type 104, 191, 378, 382
Elementary search help 393
 create 393
 options 395
Element bar 456
Element list 451
Element palette 456
Encapsulation 267, 270, 279
END-OF-PAGE 516
END-OF-PAGE event 504
END-OF-SELECTION 236, 237

- End user 34
 - Enhancement 36, 801, 811, 845
 - assignment 816
 - hook 812
 - Enhancement category 350
 - Enhancement Framework 822, 834
 - Enhancement package 29
 - ENHANCEMENT-POINT 835
 - Enhancement point 834
 - explicit 835
 - Enhancement spot 824
 - create 824
 - Enjoy transactions 76
 - ENQUEUE 400
 - Enqueue server 50, 53
 - Enqueue trace 849, 871, 877
 - Enterprise resource planning 27
 - ERP system 30
 - advantages 32
 - departments 30
 - layout 32
 - vs. non-ERP systems 30
 - Error message 205
 - Event 231, 359
 - event handler 311
 - instance event 311
 - list event 238
 - program constructor 231
 - reporting event 232
 - screen event 239
 - selection screen 238
 - sender 311
 - static event 311
 - Event block 214, 217, 225, 226, 229
 - Event controlling 270
 - Events 311
 - sequence 496
 - Exception
 - ASSIGN 326
 - catching 321, 326
 - local class 323
 - maintain 321
 - managing via function modules 320
 - MESSAGE addition 339
 - passing messages 335
 - raise 321, 322, 325, 332
 - Exception class 330
 - define globally 330
 - define locally 333
 - function module 332
 - message 333
 - Exception handling 250, 319
 - local class 323
 - using methods 322
 - Exceptions
 - overview 319
 - EXCEPTIONS statement 323
 - Exclude
 - ranges 200
 - single values 200
 - Executable program 129, 187, 232, 492
 - background processing 534
 - flow 496
 - Exit message 205
 - EXIT statement 220
 - Explicit enhancement 834
 - Explicit enhancement point 835
 - Extended Binary Coded Decimal Interchange Code (EBCDI) 108
 - Extends 351
 - Extensible Stylesheet Language (XSL) 788
 - Extensible Stylesheet Language Transformations (XSLT) 788
 - Extension 41
 - External breakpoint 613
 - External data 97
 - External developer 699
 - External program 280
- ## F
- Fallback class 826, 832
 - implement 832
 - FIELD 472
 - Field
 - relationship with domains and data
 - elements 120
 - Field attributes 399
 - Field catalog 542, 547
 - component 547
 - usage 549
 - Field conversion 733, 734
 - Field exit 815
 - Field help 478, 479
 - Field label 120, 381
 - Field symbol 580, 587
 - assign data object 589
 - assign internal table record 592
 - assignment check 592
 - define 588
 - dynamic field assignment 590
 - field position 591
 - filter functionality 588

- Field symbol (Cont.)
 - generic type 589
 - making programs dynamic 582
 - modifying an internal table record 581
 - static assignment 590
 - structure component 589
 - structure field 591
 - unassign 593
- File interface 428
- Filtering logic 285
- FINAL 293
- Financial Accounting (FI) 33
- First normal form (1NF) 173
- Fixed point arithmetic 130
- Fixture 853
- Flat structure 137
- Flow logic 444, 466
 - tab 451
- Foreground mode 707
- Foreground on error mode 707
- Foreign key 169, 170, 178, 344, 361, 406
 - create relationship 362
 - field types 363
 - relationship 361
 - table 361
- Form 631, 636
 - address 662
 - address node 688
 - alternative node 691
 - attributes 657
 - command 669
 - create 684
 - data node 690
 - driver program 674
 - element 648
 - global definition 659
 - graphic node 686
 - graphics 661
 - interface 658
 - invoice 631
 - loop 669, 689
 - maintain elements 659
 - print 648
 - program line 668
 - SAP scripts 633, 651
 - single-record node 691
 - structure node 689
 - style 670
 - table 666
 - tab positions 650
 - template 663
 - text 664
- Form (Cont.)
 - text node 690
 - windows 660
- Formal parameter 242, 246
 - typed and untyped 243
- Form Builder 656
 - draft page 656
 - form style 670
 - SAP Interactive Forms by Adobe 678, 684
 - smart forms 655
 - text module 664
- Form Painter 635, 636
 - create window 639
 - graphical 638
 - page layout 638
 - paragraph and character formatting 640
 - SAP scripts 633
- Forms 42
- FORM statement 243
- FOR TESTING 853
- Free key 159
- FROM clause 411
- Functional consultant 34
- Function Builder 71, 80, 82, 129, 215, 250, 494, 543, 842
 - Attributes tab 253
 - BAPI 719, 722
 - Changing tab 254
 - create web service 767
 - Exceptions tab 255
 - Export tab 254
 - function module 250, 251, 722
 - Import tab 254
 - Source Code tab 256
 - Tables tab 255
 - update function module 420
- Function group 187, 250, 494, 521
 - create 251
- Function module 215, 226, 242, 250, 499
 - calling 256
 - CLOSE_FORM 648
 - CONVERSION_EXIT_MATN1_OUTPUT 669
 - create 251, 252
 - enhance interface 837
 - F4IF_SHLP_EXIT_EXAMPLE 400
 - FP_JOB_CLOSE 696
 - function group 250
 - modify 808
 - normal 254
 - OPEN_FORM 648
 - Pattern button 256
 - remote-enabled 254

- Function module (Cont.)
 - REUSE_ALV_HIERSEQ_LIST_DISPLAY* 559
 - test in BTE* 844
 - update module* 254
 - Function module exit 815, 818
 - Function modules
 - REUSE_ALV_FIELDCATALOG_MERGE* 550
 - Function pool 215
- G**
- Garbage collector 298
 - Gateway 50, 53
 - General dynpro 187, 188
 - General screen 239, 440
 - Generic type 243
 - GET
 - BADI* 833
 - CURSOR* 505, 515
 - DATASET* 430
 - SBOOK* 236
 - table* 234
 - \\u003ctable\\|> LATE* 235
 - GET_SOURCE_position 329
 - GetDetail() 717
 - GetList() 717
 - Getter method 282
 - Global class 258
 - Global declaration 98, 218
 - Global table type 248
 - Graphic
 - element* 661
 - node* 686
 - Graphical layout editor 452, 467
 - Grid display 542
 - GROUP BY clause 409
 - GTT 353
 - GUI_DOWNLOAD 431
 - GUI_UPLOAD 431
 - GUI status 443, 444, 474, 506, 561
 - activate and load* 475
 - create* 474
 - maintain* 474
 - GUI title 476
- H**
- Hash algorithm 151
 - Hashed table 151
 - secondary key* 154
 - Hash key 153, 155
 - HAVING clause 409
 - Help documentation 120
 - Help view 368, 374
 - Hexadecimal data type 107
 - Hide area 511
 - HIDE statement 511
 - Hierarchical display 543
 - Hierarchical sequential display 555
 - field catalog* 555
 - Hierarchical sequential list
 - example* 558
 - Hierarchical sequential report 559
 - Hierarchical structure 860
 - High-priority update 420
 - Hint 354
 - Hold data 450
 - Hook 812
 - Host variable 416
 - HTTP trace 872
- I**
- I/O field 454, 460
 - create/add* 460
 - I/O fields 448
 - IDoc
 - assign basic types* 743
 - attributes* 750
 - create basic type* 740
 - create logical type* 743
 - create segment* 738
 - development and tools* 738
 - EDI* 729
 - inbound program* 751, 752
 - master IDoc* 733
 - outbound program* 753, 754
 - record* 730
 - status code* 752
 - structure* 736
 - system configuration* 744
 - IDocs 734
 - IF_MESSAGE 327
 - GET_LONGTEXT* 327
 - GET_TEXT* 327
 - Implementation 47
 - Implementation hiding 283
 - Implicit enhancement 837
 - IMPORT/EXPORT statement 500
 - Inbound interface 40, 429
 - Inbound process code 749
 - Inbound program 751
 - Include program 129, 215, 216, 495
 - Include structure 364

- Include text 664
 - Index 344, 355
 - unique and nonunique* 356
 - Index access 148
 - Index table 152
 - INDX structure 433
 - Information message 205
 - Inheritance 267, 270, 286, 833
 - relationship* 287
 - Inheritance relationship 295
 - Inheritance tree 293
 - INITIALIZATION 497
 - INITIALIZATION block 225
 - INITIALIZATION event 234
 - INITIALIZATION reporting event 232
 - Injection 853
 - Inline declaration 125, 264, 415
 - assign value to data object* 264
 - avoid helper variables* 265
 - declare actual parameters* 265
 - table work area* 265
 - Inner join 178, 179
 - Input field 524
 - Input help 478, 479
 - Insertion anomaly 172
 - INSERT statement 156, 157, 168
 - inserting a single row* 413
 - inserting multiple rows* 413
 - SY-DBCNT* 412
 - SY-SUBRC* 412
 - Instance component 274
 - Instantiation operation 597
 - int8 104, 105
 - Integrated development environment
 - (IDE) 71
 - Interactive report 511, 559
 - custom GUI status* 560
 - TOP-OF-PAGE* 565
 - user action* 564
 - Interface 699
 - INTERFACE* 305
 - PUBLIC SECTION* 306
 - Interface pool 494
 - Interface program 40
 - Interface work area 234
 - Intermediate Document (IDoc) 727
 - Internal table 135, 146, 149, 248
 - APPEND* 156
 - COLLECT statement* 158
 - define* 146
 - hashed table* 151
 - INSERT* 156
 - Internal table (Cont.)
 - modifying records* 161
 - processing data* 181
 - reading data* 159
 - small* 155
 - sorted table* 150
 - usage* 156
 - work area* 147
 - Internet Communication Framework
 - (ICF) 766
 - Internet Communication Manager
 - (ICM) 48, 50, 766
 - Internet Communication Manager (ICM) 50
 - Internet Demonstration and Evaluation
 - System (IDES) 42
 - Internet Transaction Server (ITS) 48
 - INTO clause 412, 416
 - Invoice 631
 - iXML 313
 - iXML library 315, 317
- J**
- JavaBeans 545
 - JavaScript Object Notation (JSON) 790
 - Join 178, 367, 369
 - conditions* 369
 - JSON
 - transformation* 792
- K**
- Key access 148
 - Keyword 101, 102
 - access* 103
 - TYPES* 139
- L**
- Languages 55
 - Lazy update 155
 - LEAVE SCREEN statement 480
 - LEAVE TO LIST-PROCESSING statement 510
 - LEAVE TO SCREEN statement 480
 - Legacy System Migration Workbench
 - (LSMW) 41, 700
 - LFGRPF01 252
 - LIKE
 - vs. TYPE* 192
 - LIKE addition 248
 - Line type 146
 - List display 542

- List dynpro 188
 - List event 238, 239, 502
 - type 502
 - List screen 132, 238, 440, 491
 - list event 502
 - practice 516
 - program type 491
 - List system 509
 - Literals 124
 - Local declaration 98, 218
 - Local exception class 333
 - Local object 65
 - Local structure 136
 - Lock object 88, 400, 434
 - code example 403
 - create 400
 - function module 402
 - maintain 400
 - Logical database 234
 - Logical expression 412
 - Logical message type 738, 743
 - Logical unit of work (LUW) 250, 406, 419
 - LOOP 147, 159, 473
 - LOOP AT SCREEN 537
 - Loop node 669
 - LOOP statement 161
 - Low-priority update 420
 - LSMW 754
 - field mapping 761
 - getting started 755
 - object attribute 756
 - process steps 756
 - recording 757
 - reusable rules 761
 - source structure 759
 - LZCB_FGTOP 252
 - LZCB_FGUXX 252
- M**
- Macro 216
 - Main program group 498
 - Maintenance view 357, 359, 368, 372
 - create 372
 - maintenance status 373
 - options 373
 - view key 372
 - Main window 636, 656
 - Many-to-many relationship 172
 - MAX 411
 - Maximum 411
 - Memory 498, 500
 - allocation 383
 - analysis 866, 867
 - storage 501
 - unit 123
 - Memory Inspector 848, 866
 - compare memory snapshots 868
 - memory snapshot 867
 - Memory snapshot
 - compare 868
 - steps 867
 - Memory snapshots 867
 - Menu bar 444
 - Menu exit 815, 820
 - Menu Painter 71, 74, 85, 86, 439
 - custom GUI status 561
 - GUI status 444
 - load custom GUI status 561
 - modification 808
 - screen elements 443
 - Message 204, 285
 - assigning attributes 338
 - dynamic 209
 - maintenance 207
 - message class 207
 - placeholder 210
 - statement 204
 - tab 207
 - text symbol 206
 - translation 210
 - type 205
 - Message class 207
 - IF_T100_DYN_MSG 339
 - IF_T100_MESSAGE 337
 - maintaining messages 208
 - MSGTY 339
 - TYPE addition 339
 - using messages 337
 - WITH addition 339
 - Message server 49, 50, 53
 - Metadata 341
 - Method 215, 242, 258, 274
 - abstract 291
 - calling 262
 - cl_notification_api 284
 - class 258
 - create 258
 - create global classes 258
 - functional method 276
 - me 276
 - set_message 284
 - static 262

- Methods 226, 271
 - maintain code 262
 - MIN 411
 - Minimum 411
 - Mock object 852, 853
 - Modification 36, 37, 801, 803
 - program 804
 - registration 804
 - Modification Assistant 803
 - ABAP Data Dictionary 808
 - Class Builder 805
 - function module 808
 - insert 805
 - modification 808
 - program 804
 - replace 805
 - reset original 809
 - Screen Painter 807
 - Modification Browser 803, 810
 - reset to original 809
 - MODIFY 161
 - Modularization 213
 - benefits 214
 - include programs and macros 215
 - local declaration 98
 - processing block 213
 - Modularization statement 127
 - MODULE 472
 - statement 451
 - Module pool 129, 187, 215, 239, 493, 521
 - flow 497
 - Modules 33
 - Multiline comment 102
 - Multiple selection window 197
 - tabs 200
- N**
- Named data object 597
 - Narrow cast
 - child->meth3 299
 - parent->meth1 299
 - parent->meth2 299
 - parent->meth3 299
 - Native SQL 87, 167, 406, 875
 - GTT 346
 - Nested structure 137
 - type 141
 - New Debugger 183, 611, 623, 866
 - layout 625
 - Memory Inspector 867
 - session 625
 - New Debugger (Cont.)
 - tools 623
 - UI 623
 - New project 36
 - NODE keyword 235
 - Noncatchable exception 326
 - Nonnumeric data types 109
 - Nontransportable package 65
 - Nonunique index 356
 - Normal function module 254
 - Normalization 172
 - Numeric data type 107, 109
 - Numeric literal 124
- O**
- Object 277
 - Object Navigator 65, 71, 80, 88, 443, 775, 824
 - ABAP messaging channel 794
 - create screen 448
 - create transaction 480
 - form interface 678
 - module pool 493
 - navigation and tool areas 88
 - web service consumer 771
 - Object-oriented programming 215, 267
 - basics 267
 - Object palette 692
 - Objects 271
 - Object view 620
 - OData 700, 777
 - data model definition 779
 - READ 786
 - service creation 778
 - service implementation 784
 - service maintenance 782
 - ON COMMIT 422
 - One-to-many relationship 171
 - One-to-one relationship 171
 - Online Text Repository (OTR) 334
 - OPEN_FORM 647
 - Open Data Protocol (OData) 777
 - OPEN DATASET 429
 - Open Specification Promise (OSP) 777
 - Open SQL 86, 136, 167, 436, 874
 - database 168
 - database relationship 171
 - data in a database 407
 - DDL 168
 - DELETE statement 414
 - DML 168, 407
 - GTT 346

Open SQL (Cont.)
INSERT statement 412
logical database 234
MODIFY statement 414, 421
persistent data 405, 407
selecting data from database tables 176
selecting data from multiple tables 178
SELECT statement 408
statements 408
UPDATE statement 413

Operand 100

Operational statement 127

Optimistic lock 402

Oracle 355

ORDER BY clause 409

Outbound interface 40, 429

Outbound process code 748

Outbound program 753

Outer join 179

Output length 116

Output table 542

P

Package 64
assign 131
create 68
naming conventions 66
nontransportable 65
transportable 64

Package Builder 68

Packages
encapsulation 69

Parallelism 175

PARAMETERS 125, 132, 189, 211
SCREEN_OPTIONS 192
VALUE_OPTIONS 195

PARAMETERS keyword 189
TYPE_OPTIONS 190

PARAMETERS statement
effect 190

Parameter table 607, 608

Partner profile 747

PC editor 664

PDF 563

PERFORM 234, 812
keyword 226
statement 215, 225

Performance trace 848, 871
with filter 872

Persistence mapping 424

Persistence service 423

Persistent attribute 423

Persistent class 423
Class Builder 426
create 424
mapping tool 426

Persistent data 97, 405
security 434
storage 405

Persistent object 423, 427
working with 427

Persistent program 609

Picture control 483, 484

Placeholder 210

Polymorphism 267, 270, 296
casting 298
dynamic type 296
static type 296

Pooled table 345

Port definition 746

Predefined elementary ABAP type 108

Predefined elementary data type 104
use case 109

Predefined nonnumeric elementary
 data type 105

Predefined nonnumeric elementary data type
category 107

Predefined numeric elementary data
 type 105

Predefined type 385

Presentation layer 45, 46, 70, 217, 447
file 431
GUI_DOWNLOAD 431
GUI_UPLOAD 431
SAP GUI 47

Presentation server 405

Primary index 353

Primary key 170, 385, 406

Procedural area 99

Procedural exception 320

Procedural programming 97

Procedure 214, 217, 226, 231, 240

Process after input (PAI) 240, 442,
 445, 459, 466, 472

Process analysis 848, 864

Process before output (PBO) 240, 442,
 446, 477, 485
selection screen 521

Processing block 99, 127, 213,
 217, 225, 492, 493
calling 219
CHECK 221
dynpro 187

Processing block (Cont.)
ending 220
EXIT 220
procedure 214
RETURN 222
sequence 219, 228
type 226
types 214
usage 223
virtual (global data declaration) 223

Process interfaces 840

Process on help request (POH) 240, 442, 478

Process on value request (POV) 240, 442,
 478, 497

Production client (PROD) 63

Production support 36

Program
RMVKON00 527
zdemo_prg 538

Program attributes 128, 491
maintain 130

Program constructor 231

Program execution 495

Program group 498

Program line element 668, 669

Programming concepts 97

Program type 491

Projection view 368, 370
create 370

Promote optimistic lock 402

Pseudocomment 862

Publish and subscribe interfaces 840

Pure Java system 49

Push button 454, 459

Q

Quality assurance 847

Quality assurance client (QTST) 63

Quantity field 349

R

R_ER 328

Radio button 194, 454, 457
group 194

RAISE 255
EVENT 313
EXCEPTION 325
statement 321

Range 198
operator 199

Range field 197

Range table 195
HIGH 196
LOW 196
OPTION 196
SIGN 196
values 199

READ 147, 154, 159
DATASET 430
statement 160

READ CURRENT LINE statement 515

READ LINE statement 515

Receiver 732

Receiver determination 732

Recording routine 358

Reference data type 114, 378, 385

Referenced data type 382

Reference variable 277, 594
upcast and downcast 598

Reference variables
assignment 598

Relational database 406

Relational database management system
 (RDBMS) 47, 55, 167, 168
example 55
foreign key 47, 170
primary key 169
relational database design 169
table 169

Remote-enabled function module 254

Remove\|u003csubobject\|>() 717

Replacement object 377

Report 496
background execution 535
form 631
program 39
run in background 534
scheduling 535

Reporting event 232
END-OF-SELECTION 236
GET table 234
GET \|u003ctable\|> LATE 235
INITIALIZATION 232
START-OF-SELECTION 233

Reporting transaction 481

Report output
type 542

Report transaction 493

Repository 64
object 36, 70
package 64

Repository Browser 82

Representational State Transfer (REST) 777
 Request 65
 REQUEST_LOC 403
 Request-response cycle 447
 RESTful APIs 777
 RETCODE 324
 RETURN 222
 statement 220
 Returning parameter 261
 REUSE_ALV_BLOCK_LIST_APPEND 552
 REUSE_ALV_BLOCK_LIST_INIT 552
 REUSE_ALV* 542
 RFC 53, 231, 250
 destination 745
 trace 849, 871, 876
 RFC/BOR interface 780
 ROLLBACK WORK 418, 420
 Row type 149
 Runtime Analysis 849, 879
 Runtime analysis 407
 Runtime error 111
 Runtime Type Creation (RTTC) 599, 601
 dynamic 603
 Runtime Type Information (RTTI) 599, 600
 query 601
 Runtime Type Services (RTTS) 580, 599

S

Sales and Distribution (SD) 813
 SAP
 data structure 36
 functional module 33
 introduction 33
 module 33
 systems 36
 technical module 34
 users 34
 SAP Business Technology Platform 94
 SAP Easy Access 76
 Favorites 73
 SAP Menu 73
 screen 73
 User Menu 73
 SAP ERP 27, 822
 SAP Gateway 777
 activate/register an OData service 782
 READ 786
 Service Builder 778
 SAP GUI 46, 47, 71, 217
 architecture 49
 for HTML (Web GUI) 48
 SAP GUI (Cont.)
 for the Java environment 48
 for the Windows environment 48
 set status 562
 SAP HANA 47, 55, 169, 175, 368, 627
 ABAP CDS view 175
 aggregate functions 410
 code pushdown 175
 parallelism 175
 SAP Interactive Forms by Adobe 631, 677
 address node 688
 alternative node 691
 context and layout 684
 Context tab 685
 currency/quantity fields 683
 data node 690
 development 678
 download as PDF 696
 driver program 695, 696
 form interface 678
 global definitions 682
 graphic node 686
 import form data 697
 layout 692
 Layout tab 691
 loop 689
 object palette 692
 single-record node 691
 structure node 689
 SAP List Viewer (ALV) 541
 SAP liveCache 55
 SAP LUW 417, 419, 436
 bundle with function modules 420
 bundle with RFC 422
 bundle with subroutines 422
 bundling technique 419
 dialog step 419
 SAP NetWeaver 49, 276
 SAP NetWeaver 7.5 33, 746
 debugging 611
 global temporary table 345
 new SQL 415
 SAP Notes 802
 SAP S/4HANA Cloud 94
 SAP S/4HANA Cloud, ABAP Environment 94
 SAP script editor 642, 645, 664
 printing 645
 SAP scripts 631, 633
 align and print 649
 billing document 633
 change header 637
 composer 633

SAP scripts (Cont.)
 create form layout 636
 create standard text 646
 create window 639
 disadvantages 654
 driver program 635, 652, 675
 element 648
 field entries 643
 formatting 636
 graphics administration 642
 insert graphic 644
 layout 633
 layout designer 638
 maintain tab positions 649
 maintain window details 642
 paragraph and character formatting 640
 printing 648
 print logo 642
 process forms with function modules 647
 return_code 652
 subroutine 635
 us_screen 652
 window 636
 SAP Software Change Registration (SSCR) 803
 SAP start service 50, 53
 SAP Support Portal 803
 SAP system
 architecture 45
 enqueue server 53
 environment 71
 gateway 53
 instance 50
 layers 56
 logon 72
 message server 53
 SAP Web Dispatcher 53
 session 76
 user context 54
 SAPUI5 872
 SAP Web Dispatcher 50, 53
 Schema 168
 Screen 444, 448
 application toolbar 75
 attributes 449, 450
 command field 75
 create 448
 event 239, 440, 441
 exit 815, 818
 group 451
 menu bar 74
 number 448
 processor 441
 Screen (Cont.)
 sequence 479
 standard toolbar 74
 status bar 75
 title bar 74
 transaction code 75
 type 450
 SCREEN_OPTIONS 192
 AS CHECKBOX 193
 AS LISTBOX VISIBLE LENGTH vlen 194
 RADIOBUTTON GROUP 194
 VISIBLE LENGTH vlen 193
 Screen element 439, 443, 444, 452, 454
 basic 455
 Screen field 448
 modifying dynamically 476
 Screen flow logic ... 441–443, 445, 447, 472, 807
 SCREEN-OPTIONS
 NO-DISPLAY 193
 OBLIGATORY 193
 Screen Painter 71, 83, 204, 439, 450, 483, 489
 alphanumeric layout editor 452
 dynpro 187
 graphical layout editor 85, 452, 456
 modification 807
 module pool 494
 screen elements 443
 subscreens 470
 tab 84, 452
 Screen structure 476
 components 477, 536
 Search help 120, 380, 392
 assign 399
 change 397
 exit 400
 parameters 396
 Secondary index 354, 357
 create 355
 Secondary key 153, 169, 385, 387, 406
 Secondary list 509
 Secondary window 656
 Second normal form (2NF) 173
 Security 434
 Segment 736
 add 741
 create 738
 definition 739
 Segment Editor 738, 740
 Segment filtering 734
 SELECT 157, 168

Select
 range 198
 single values 198
 SELECT clause 409
 SELECTION-SCREEN 188, 202
 BEGIN OF BLOCK 203
 SKIP 202
 ULINE 203
 Selection screen 132, 187, 440, 519
 calling programs 538
 create variant 526
 define 519, 520
 dynamically display/hide screen
 element 536
 dynamic date 532
 dynamic time 533
 event 519, 521
 field 188
 PARAMETERS 189
 radio button 524
 standard 520
 task 188
 user-defined 520
 user-specific variable 534
 variant 525
 variant attributes 529
 Selection screen event 238
 Selection screens
 standard 520
 Selection text 203
 Selectivity analysis 848, 862
 SELECT-OPTIONS 165, 195, 201, 211
 multiple selection window 197
 SELECT statement 408
 FROM clause 411
 INTO clause 412
 SELECT clause 409
 WHERE clause 412
 Semantic attribute 378, 381
 Separation of concerns 852
 Sequential data 138
 Serialization 734
 Service implementation 778, 784
 Service maintenance 778, 782
 SESSION 707, 711
 create manually 712
 Session 76, 625
 component 626
 Session breakpoint 613
 SET DATASET 430
 SET HANDLER 311
 SET SCREEN statement 479
 Setter method 282
 set_message 284
 Shared lock (read lock) 401
 Simple Object Access Protocol (SOAP) 766
 Simple report 543
 field catalog 548
 Single inheritance 304
 Single-record node 691
 Singleton design pattern 427
 Single-transaction analysis 849, 883
 Size category 351
 SKIP 239
 Smart form 631, 679, 686
 address 662
 advantages over SAP scripts 654
 character formatting 673
 command 669
 create template 663
 create text element 665
 draft page 656
 driver program 674, 676
 form attributes 656, 657
 form interface 656, 658
 Form Painter 656
 global definition 656, 659
 graphics 661
 interface 680
 layout 654
 line type 666
 loop 669
 maintain elements 659
 maintain global settings 657
 maintenance area 655
 navigation area 655
 paragraph formatting 671
 program line 668
 row type 666
 style 670, 671, 674
 table 666
 text 664
 windows 656, 660
 Smart forms 654, 678
 SOAP 766, 771
 Sorted key 153
 Sorted table 150, 151
 secondary key 154
 Special dynpro 187
 Special screen 239
 Splitter containers 484
 SQL 167
 SQL optimizer 354
 SQLScript 627

SQL trace 407, 848, 871, 874
 display 875
 use case 875
 views 876
 Standardized BAPI 716, 717
 Standard report 542
 Standard selection screen 188
 Standard table 149, 151
 secondary key 153
 Standard toolbar 444
 START-OF-SELECTION 223
 START-OF-SELECTION block 225
 START-OF-SELECTION reporting event 233
 Statement
 UNION 176
 Static boxes 383
 Static component 274
 Static enhancement point 835
 Status message 205
 Status record 730
 Status type 474
 Strict mode 416
 STRING 108, 110
 String literal 125
 Structure 135, 382
 create global structure 142
 define 138
 global 137
 global structure 141
 local structure 138
 processing data 181
 type 137
 usage 144
 use case 145
 Structured Query Language (SQL) 167
 Structure node 689
 Structures 136
 Structure type 135, 140
 Subclass 280, 286
 SUBMIT (dobj) 605
 Subproject 755
 Subroutine 215, 226, 241, 242, 359
 error handling 324
 input parameters that pass values 246
 local declaration 137
 output parameters that pass values 247
 parameters passed by reference 245
 passing internal tables 247
 passing parameters 244
 SAP LUW 422
 USING and CHANGING 242
 Subroutine pool 495, 498
 Subscreen 470, 521
 Subscreen area 455
 SUM 411
 Superclass 286, 331
 Switch Framework 29, 813, 835
 SY-DBCNT 412
 Synchronous data update 707
 Syntax 99
 chained statement 101
 comment line 102
 rules 100
 System field 157
 SY-SUBRC 324, 412

T

Table 87
 cells 667
 EDID4 736
 EDIDC 736
 EDIDS 737
 field 371
 ICON 75
 INDX 433
 IT_SFLIGHT 162
 IT_VBRP 165
 line 667
 MAKT 56, 343
 MARA 56, 62, 362, 397
 MARC 56, 182
 NAST 653
 PTAB 608
 SAPLANE 361
 SBOOK 236
 SFLIGHT 138, 140, 361, 507, 577
 SPFLI 354, 409, 507
 TOOIW 183
 TIOO 207
 TVARVC 530
 VBLOG 422
 VBRK 165, 376, 514, 633
 VBRP 144, 165, 633
 table_line 154
 Table buffer trace 849
 Table call statistics 848, 869
 screen 870
 Table category 149
 Table control 454, 462
 attributes 464
 create 463
 create without wizard 463
 create with wizard 467

Table display 569
Table element 666
 areas 666
 table line 667
Table field 344
Table key 152, 159
 default key 152
 hash key 153
 secondary key 153, 155
 sorted key 153
Table keyword 235
Table maintenance generator 357, 358
Table Painter 655
Table type 384
 global 248
 line type 384
Table types
 primary key 386
Table view maintenance 346
Tabstrip control 454, 468
 create 469
 wizard 469
Tag 313
Task 65
Technical consultant 34
Template 663
 create 663
Termination message 205
Test class 853
 define 853, 855
 fixture 853
 implementation 857
 properties 853
Testing 847
 results 859
Text
 element 664
 module 664
 node 690
Text field 454
 create 456
Text field literal 125
Text literal 206
Text symbol 126, 206
 change 206
Third normal form (3NF) 174
Three-tier architecture 27, 45
 application layer 46
 buffer 57
 database layer 47
 presentation layer 46
TOP-OF-PAGE 565
TOP-OF-PAGE event 503
TOP-OF-PAGE list event 502
Total 411
Transaction 75
 /*xxxx 78
 /h 79
 /i 78
 /IWFND/GW_CLIENT 784, 787
 /IWFND/MAINT_SERVICE 782
 /N 78
 /n 78
 /nend 78
 /nex 79
 /ns000 78
 /nxxxx 78
 /O 78
 /o 78
 /oxxxx 78
 ABAPDOCU 330
 assign 480
 BAPI 714
 BD51 750
 BMVO 702
 CMOD 816, 818
 code 446
 create 480
 custom namespace 482
 DB05 848, 862
 FIBF 841
 FILE 430
 FKO2 844
 LSMW 755
 ME21N 76, 819
 ME21n 497
 ME22N 76
 ME23N 76
 MMO1 76, 701, 703, 758
 MMO2 76
 MMO3 76
 MRKO 496, 527, 528
 NACE 652, 696
 navigating and opening 76
 opening 78
 RZ10 53
 S_MEMORY_INSPECT 866
 S_MEMORY_INSPECTOR 848, 868
 SA38 496, 534
 SAMC 794
 SAP Easy Access screen 78
 SAT 407, 849, 879
 SAUNIT_CLIENT_SETUP 853
 SCC4 64
 SCII 860
 SEO1 66

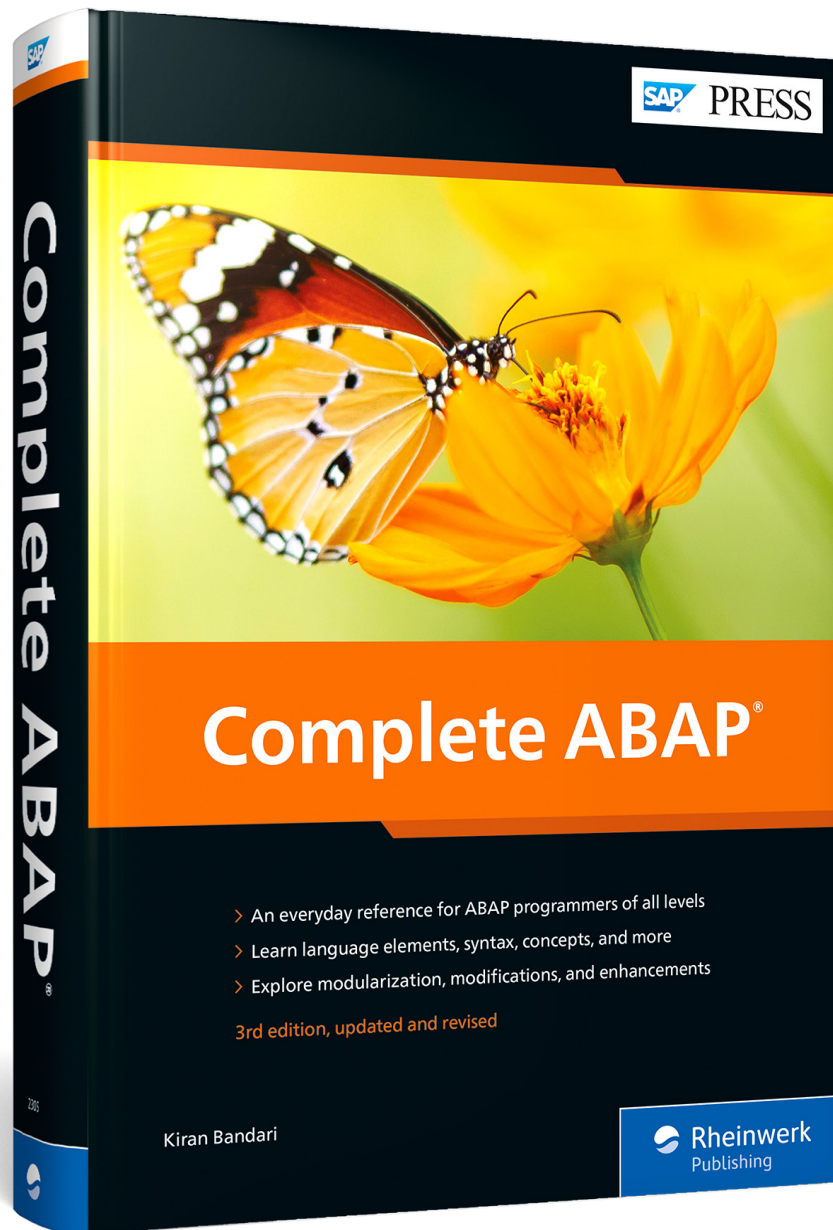
Transaction (Cont.)
 SEO3 66
 SEO9 66
 SE10 66
 SE11 78, 117, 138, 142, 249, 341, 495
 SE13 351
 SE18 823, 824
 SE19 823, 830
 SE21 68
 SE24 83, 258, 309, 331, 853
 SE37 82, 215, 250, 494, 543
 SE38 80, 128, 493, 612
 SE41 74, 443, 561
 SE51 443
 SE71 633, 635, 636
 SE78 642
 SE80 65, 80, 82, 83, 443, 824
 SE91 207
 SE93 65, 75, 443, 447
 SE95 809, 810
 SEGW 778, 779
 SHDB 708
 SICF 767
 SLDB 130
 SM35 712
 SM37 863
 SM50 52, 848, 864, 866
 SM59 745
 SM66 848, 864, 866
 SMARTFORMS 654, 657, 670
 SMOD 815, 819
 SNOTE 802
 SO10 646
 SOAMANAGER 769, 773
 SPACKAGE 68
 SPAU 803
 SPDD 803
 ST05 407, 848, 872, 883
 ST10 848, 869
 ST12 883, 885
 ST22 886
 STMS 66
 STVARV 530, 532
 SU3 534
 SWO1 719, 726
 VA01 497
 VFO1 76, 446
 VFO2 76
 VFO3 76, 440
 WE20 747
 WE21 746
 WE30 740
 WE31 738

Transaction (Cont.)
 WE41 748
 WE42 749, 751
 WE81 738, 743
 WE82 738, 743
Transactional RFC (tRFC) 746
Transaction Recorder 708
 create program from recording 710
 update mode 708
TRANSFER 429
Transient data 97
Transient program 609
Translation 210
Transportable package 64
TRANSPORTING addition 161
Transport Organizer 65, 66
trigger_event 313
Troubleshooting 627, 628
TRUNCATE DATASET 430
TRY block 325
Two-dimensional array 146
TYPE 103, 191
 c 112
 concept 97, 103
 d 113
 f 109
 i 109
 n 110
 p 109
 vs. LIKE 192
TYPE_OPTIONS 190
 LIKE (*name*) 192
 LIKE (*obj*) 191
 TYPE *data_type* [DECIMALS *dec*] 191
Type class 600
Type conversion 110, 112
 with invalid content 111
 with valid content 111
Typed formal parameter 243
Type group 388
 data type 388
Type object 600
Type pool 495
TYPE-POOLS 389
TYPE RANGE OF 196

U

ULINE 239, 502
UNASSIGN 593
Uncomment 102
Undelete() 717
Unique index 356

-
- Unit test 850
 - write and implement* 853
 - Universal Description, Discovery, and
 - Integration (UDDI) 766
 - Unnamed data object 124
 - Upcasting 598, 599
 - UPDATE 168
 - Update anomaly 172
 - Update function module 420
 - attributes* 420
 - Update module 250, 254
 - Update work process 421
 - User action 564
 - User context 54
 - User-defined elementary data type 104, 113
 - User-defined selection screen 188, 520
 - define* 520
 - User exit 811, 812
 - SD* 813
 - User interaction 187
 - User interface 45
 - User-specific value 534
 - USING 245
 - input parameters* 246
- V**
-
- VALUE 246
 - VALUE_OPTIONS 195
 - Value range 389, 391
 - Value table 362
 - Variable 125
 - Variable window 636
 - Variant 525
 - attributes* 529
 - create* 526
 - dynamic date* 532
 - dynamic time* 533
 - for selection screen field* 531
 - system variant* 527
 - user-specific variable* 534
 - Variants
 - attributes* 527
 - View 88, 342, 367
 - ABAP CDS views* 374
 - database* 368
 - help* 374
 - maintenance view* 372
 - projection* 370
 - replacement object* 378
 - View (Cont.)
 - type* 368
 - Virtual processing block 223
 - Visibility 136
 - Visibility section 280
 - PRIVATE* 280
 - PROTECTED* 280
 - PUBLIC* 280
- W**
-
- Warning message 205
 - Watchpoint 615, 628
 - create* 617
 - Watchpoints
 - view* 618
 - Web Dynpro ABAP 71, 680
 - Web Dynpro for ABAP 51
 - Web service
 - consume* 771, 776
 - create* 767
 - create ABAP program to consume* 775
 - maintain port information* 773
 - Web services 700, 765
 - Web Services Description Language
 - (WSDL) 765
 - WebSockets 792
 - WHERE clause 408, 412
 - Windows 660
 - WITH 538
 - Work area 146, 456
 - Work process 49, 51, 441, 447
 - WRITE 112, 239
 - WRITE_FORM 647
 - Write lock) 401
 - WRITE statement 517
- X**
-
- XI Message interface 766
 - XML 790
 - array* 791
 - if_ixml_element* 315
 - XML Path Language (XPath) 788
 - XML schema-based interface 680
 - XSLT 700
 - XSL transformation
 - deserialization* 789
 - serialization* 789
 - XSTRING 108, 110



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

We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.