

## **SAP-ABAP**

### **ENTERPRISE RESOURCE PLANNING (ERP):**

ERP is a package which provides solution for departmental functionalities of an organization.

### **SAP ADVANTAGES:**

- SAP can support any database at the back end.
- Platform independent.
- Supports multiple languages.
- Faster b/w networks.
- Can support any database in the backend.
- Transaction failure is very less comparing 2 other erp packages (0.03%).

### **SAP life cycle implementation.**

- Package evaluation.
- Project planning and designing.
- Proto type.
- Gap analysis.
- Business process re-engineering.
- s/w installation and testing.
- End user training.
- Application transportation.
- Go live.
- Support and maintenance.

### **SAP landscape:**

- Development server (functional).
- Quality server( testing).
- Production server (end-user uses the system).

### **Role of ABAPer:**

- Screen creation.
- Table creation.
- Data migration.
- Reporting.
- Redirecting SAP data from d-base to o/p devices.(scripts & forms).

**R/3** – real time data processing.

**ABAP/4** – advanced business application programming. If a language provides a d-base by default is called 4<sup>th</sup> generation language.

## **R/3 ARCHITECTURE COMPONENTS:**

**1. DISPATCHER** - This component acts as an interface between the PL and AS. It receives user's request from PL and allocates a work area for each request from the user.

**2. WORK PROCESS** - This component allocates memory area for each request received from dispatcher on a roll-in and roll-out basis. Once a request is processed, the memory area is rolled out to allocate for next request from dispatcher.

### **3. DATABASE CONNECTIVITY**

- OPEN SQL - It receives each user's request in open SQL format (INSERT, UPDATE, DELETE, MODIFY) and converts them in native SQL format (Database used at the back end).

- NATIVE SQL - It checks for syntax errors in requests received and passes the request to database through gateway service.

**4. MESSAGE SERVICE** - If there are any syntax errors in requests received, the Native SQL component uses message service to raise error messages for the end user from the message pool. Message pool is a container of user-defined and pre-defined messages.

**5. GATEWAY SERVICE** - If there is a distributed database at the back end, user's queries are redirected to appropriate database using gateway service. This component acts as an interface between AS and DL.

## **WHAT IS CLIENT IN SAP?**

- CLIENT is the topmost hierarchy in SAP database.
- Each functional module in SAP is assigned a client.
- The objects developed by ABAPer for each module is stored in a package and request number and finally saved under a client number assigned for the particular module.

The development objects in SAP is divided into two types:

### **REPOSITORY OBJECTS**

- EXECUTABLE PROGRAMS
- INCLUDE PROGRAMS
- MODULE POOL PROGRAMS
- MESSAGE CLASSES
- SUBROUTINE POOLS
- FUNCTION GROUPS
- CLASS POOLS

### **DATA DICTIONARY OBJECTS**

- TABLES
- VIEWS
- DATA TYPES
- DOMAINS
- TYPE GROUPS
- SEARCH HELPS
- LOCK OBJECTS

### **LOGGING INTO SAP:**

CLIENT : 800

USERNAME : SAPUSER

PASSWORD : VIT

LANG : EN (OPTIONAL)

### **CREATING A PACKAGE IN SAP:**

PACKAGE is a folder created by ABAPer to store his own development objects in SAP.

### **Navigations to create a package:**

Enter 'SE21' Tcode in Command Prompt area of SAP EAsy Access Screen -> Press Enter -> Opens Package Builder Screen -> Select Package Radiobutton -> Specify Package Name starting with Z or Y -> Click on Create pushbutton -> Opens an interface -> Enter Short Description for your package -> Select Software Component as HOME -> Click on Save -> OPens another interface to assign request number -> Click on CREATE REQUEST -> Opens interface -> Enter short description -> Click on Continue -> A Request number is

created and assigned for your package -> Click on Continue -> A package is created -> Save -> Come back.

### **CREATING PROGRAMS IN SAP EDITOR:**

SE38 is the Tcode used to open ABAP editor, where we create source code.

### **Navigations to create simple program in SE38 Editor:**

SE38 -> Opens ABAP Editor Initial screen -> Specify program name starting with Z or Y -> Select Source Code radio button -> Click on Create pushbutton -> Opens another interface -> Enter short description for program -> Select Program type as Executable Program -> Click on Save -> Specify Package Name -> Assign request number -> Click on continue -> Opens ABAP Editor page.

### **Characteristics of ABAP Program:**

- ABAP is not case sensitive.
- ABAP is case sensitive only during comparison.
- Every ABAP statement should end with a period (full stop).
- CTRL+S shortcut key is used to save ABAP codes in editor.
- CTRL+F1 shortcut key is used to transform the screen into CHANGE/DISPLAY mode.
- CTRL+F2 shortcut key is used to check for syntax errors in source code.
- CTRL+F3 shortcut key is used to activate current program.

### **ACTIVATION:**

This concept is used to make the current object to be accessible by other objects within the same client or different client or vice versa.

- F8 function key is used to execute the ABAP program (Direct Processing).
- CTRL+< is used to comment the ABAP statement.
- CTRL+> is used to de comment the ABAP statement.

### **VARIABLE DECLARATION IN ABAP:**

DATA is the statement used to declare variables in ABAP.

### **Eg. code:**

```
DATA A TYPE I.
A = 100.
WRITE A.
```

In the above code, A is a variable of integer type. A is assigned a value '100'. WRITE statement is used to print out the value of A.

- The output of a program is printed in LPS (list-processing screen).
- The default value of an integer data type is '0'.
- The default value of a character data type is 'NULL'.
- An integer variable takes 11 character spaces in LPS screen to print out its value and the value is printed in right-justified manner.
- The first 10 character space is used to print the value and the 11th character space is for sign (positive or negative).

**eg. code:**

```
DATA A TYPE I VALUE '-100'.  
WRITE A LEFT-JUSTIFIED.
```

TYPE statement is used to assign data type for the variable declared.  
VALUE statement is used to assign initial value for the variable.

# ABAP GUI

ABAP screens can be created using the following two statements:

1. PARAMETERS
2. SELECTION-SCREEN

The following statements are used in common:

**1. PARAMETERS** to create input boxes, checkboxes and radio buttons.

**2. SELECTION-SCREEN BEGIN OF LINE - END OF LINE.** - This statement is used to create input fields in a same row.

**3. SELECTION-SCREEN COMMENT.** - This statement is used to specify the positions and label names for each input field declared using begin of line - end of line.

## eg. code

Selection-screen begin of line.

selection-screen comment 10(15) lb2 for field a.  
Parameters: a (10).

selection-screen comment 50(15) lb3 for field b.

parameters : b(10).

selection-screen comment (15) lb4.  
parameters : c(10).

selection-screen end of line.

selection-screen pushbutton 10(10) LB1 user-command PB1.

Initialization.

lb1 = 'PRINT'.

lb2 = 'enter A value'.

lb3 = 'enter B value'.

at SELECTION-SCREEN.

```

case sy-ucomm.
when 'PB1'.
LEAVE TO LIST-PROCESSING.
write :/ a, b .
endcase.

```

#### **4. SELECTION-SCREEN POSITION.**

This statement is used to specify the position of input fields. There are two attributes for specifying position using this statement. They are:

```

POS_LOW
POS_HIGH

```

EG.

```

SELECTION-SCREEN BEGIN OF LINE.
SELECTION-SCREEN POSITION POS_LOW.
PARAMETERS A(10).
SELECTION-SCREEN POSITION POS_HIGH.
PARAMETERS B(10).
SELECTION-SCREEN END OF LINE.

```

#### **5. SELECTION-SCREEN BEGIN OF SCREEN - END OF SCREEN.**

- This statement is used to create a screen in GUI. A screen number should be specified for each user-defined screen. This screen can be called within the program using the following statement:

**CALL selection-SCREEN <screen\_number>.**

eg.

```

SELECTION-SCREEN BEGIN OF SCREEN 100 TITLE T1.
SELECTION-SCREEN BEGIN OF BLOCK B1.
PARAMETERS : CH1 AS CHECKBOX,
              CH2 AS CHECKBOX,
              CH3 AS CHECKBOX.
SELECTION-SCREEN END OF BLOCK B1.
SELECTION-SCREEN BEGIN OF BLOCK B2.
PARAMETERS : RB1 RADIOBUTTON GROUP A,
              RB2 RADIOBUTTON GROUP A,
              RB3 RADIOBUTTON GROUP A.
SELECTION-SCREEN END OF BLOCK B2.

```

SELECTION-SCREEN PUSHBUTTON /10(10) LB1 USER-COMMAND PB1.  
 SELECTION-SCREEN END OF SCREEN 100.  
 SELECTION-SCREEN BEGIN OF SCREEN 200 AS WINDOW TITLE T2.

PARAMETERS : D(10), E(10), F(10).

SELECTION-SCREEN END OF SCREEN 200.

initialization.

t1 = 'SELECT CHECKBOXES AND RADIOBUTTONS'.

T2 = 'PARAMETERS'.

at selection-screen.

case sy-ucomm.

WHEN 'PB1'.

CALL SCREEN 200.

ENDCASE.

## **6. SELECTION-SCREEN BEGIN OF BLOCK - END OF BLOCK.**

This statement is used to create blocks within the screen area.

**7. SELECTION-SCREEN PUSHBUTTON.** - This statement is used to create pushbuttons in GUI.

SYNTAX:

SELECTION-SCREEN PUSHBUTTON <starting position>(pushbutton length)  
 <label\_Variable> USER-COMMAND <pushbutton name>.

Whenever a pushbutton is created and executed, a system variable SY-UCOMM holds the arguments for the pushbutton.

INITIALIZATION and AT SELECTION-SCREEN are the selection-screen events. Using these type of events, ABAP is called as event-driven programming. Whenever the selection-screen programs are executed, the order of execution of events will be:

INITIALIZATION.

AT SELECTION-SCREEN.

START-OF-SELECTION.

TOP-OF-PAGE.  
 END-OF-PAGE.  
 END-OF-SELECTION.

EG. CODE

PARAMETERS : A(10), B(10).

SELECTION-SCREEN PUSHBUTTON /10(10) LB1 USER-COMMAND PB1.

INITIALIZATION.  
 LB1 = 'PRINT A'.

AT SELECTION-SCREEN.  
 CASE SY-UCOMM.  
 WHEN 'PB1'.

LEAVE TO LIST-PROCESSING.  
 WRITE :/ A, B.  
 ENDCASE.

In the above code,

**AT SELECTION-SCREEN** event is used to hold the event functionalities of the pushbuttons declared within the program. Whenever a pushbutton is created and executed, AT SELECTION-SCREEN event get triggered.

**INITIALIZATION** event is used to declare initial display values for the pushbutton component in the selection-screen.

**SY-UCOMM** system variable holds the arguments of pushbutton and passes these arguments dynamically to the screen.

**LEAVE TO LIST-PROCESSING** statement is used to print the statements in LPS, because these statements cannot be printed inside selection-screen.

After executing the program, use 'BACK' statement in the command prompt of LPS to come back to the selection-screen and then come back to the program.

EG. CODE 2

PARAMETERS : A(10), B(10).

```
SELECTION-SCREEN PUSHBUTTON /10(10) LB1 USER-COMMAND PB1.
selection-screen skip.
SELECTION-SCREEN PUSHBUTTON /10(10) LB2 USER-COMMAND PB2.
```

```
INITIALIZATION.
LB1 = 'PRINT A'.
lb2 = 'EXIT'.
```

```
AT SELECTION-SCREEN.
CASE SY-UCOMM.
WHEN 'PB1'.
LEAVE TO LIST-PROCESSING.
WRITE :/ A, B.
WHEN 'PB2'.
LEAVE PROGRAM.
ENDCASE.
```

## **8. SELECTION-SCREEN SKIP.**

This statement is used to create a blank line within the selection-screen components.

## **9. SELECTION-SCREEN ULINE.**

This statement is used to draw a horizontal line within the selection-screen component.

## **10. SELECT-OPTIONS.**

EG. CODE:

```
TABLES MARA.
SELECT-OPTIONS STALIN FOR MARA-MTART.
SELECT * FROM MARA WHERE MTART IN STALIN.
WRITE :/ MARA-MTART.
ENDSELECT.
```

```
WRITE :/ STALIN-LOW, STALIN-HIGH, STALIN-SIGN, STALIN-OPTION.
```

Whenever a range is created using SELECT-OPTIONS statement, SAP creates an internal table with the same variable name (STALIN in this case) in the background with the following fields:

LOW  
HIGH  
SIGN  
OPTION

## **11. SELECTION-SCREEN FUNCTION KEYS.**

The pushbuttons declared in the header part of the screen are referred to as FUNCTION KEYS. A user can be allowed to create only 5 function keys.

SYNTAX:

SELECTION-SCREEN FUNCTION KEY <function\_key\_number>.

SSCRFIELDS is a predefined structure used to assign display values for function keys.

TABLES SSCRFIELDS.

PARAMETERS : A(10), B(10).

SELECTION-SCREEN FUNCTION KEY 1.  
SELECTION-SCREEN FUNCTION KEY 2.  
SELECTION-SCREEN FUNCTION KEY 3.  
SELECTION-SCREEN FUNCTION KEY 4.  
SELECTION-SCREEN FUNCTION KEY 5.

INITIALIZATION.

SSCRFIELDS-FUNCTXT\_01 = 'STALIN'.  
SSCRFIELDS-FUNCTXT\_02 = 'JEGAN'.  
SSCRFIELDS-FUNCTXT\_03 = 'SYED'.  
SSCRFIELDS-FUNCTXT\_04 = 'DEEPA'.  
SSCRFIELDS-FUNCTXT\_05 = 'KARTHIK'.

AT SELECTION-SCREEN.

CASE SY-UCOMM.

WHEN 'FC02'.  
CALL TRANSACTION 'SE80'.

WHEN 'FC01'.  
CALL TRANSACTION 'SE11'.

WHEN 'FC03'.  
CALL TRANSACTION 'SE37'.

WHEN 'FC04'.  
CALL TRANSACTION 'SE38'.

WHEN 'FC05'.  
LEAVE PROGRAM.

ENDCASE.

When we create function keys, SAP automatically generates name for each function key. The name of first function key is 'FC01' , the second one is 'FC02', etc.

So the name range for function keys start from FC01 - FC05.

EG. PROGRAM TO CALL SELECTION-SCREEN WITH PUSHBUTTONS

SELECTION-SCREEN BEGIN OF LINE.

SELECTION-SCREEN COMMENT 10(15) LB1.  
PARAMETERS : A(10).

SELECTION-SCREEN COMMENT 40(15) LB2.  
PARAMETERS : B(10).

SELECTION-SCREEN END OF LINE.

SELECTION-SCREEN PUSHBUTTON 10(10) LB3 USER-COMMAND PB1.  
SELECTION-SCREEN SKIP.  
SELECTION-SCREEN PUSHBUTTON /10(10) LB5 USER-COMMAND PB3.  
SELECTION-SCREEN SKIP.  
SELECTION-SCREEN PUSHBUTTON /10(10) LB4 USER-COMMAND PB2.

\*SELECTION-SCREEN BEGIN OF SCREEN 200 AS WINDOW TITLE T2.

SELECTION-SCREEN BEGIN OF BLOCK B1.

PARAMETERS : CH1 AS CHECKBOX, CH2 AS CHECKBOX, CH3 AS CHECKBOX.

SELECTION-SCREEN END OF BLOCK B1.

SELECTION-SCREEN BEGIN OF BLOCK B2.

PARAMETERS : RB1 RADIOBUTTON GROUP A,  
RB2 RADIOBUTTON GROUP A,  
RB3 RADIOBUTTON GROUP A.

SELECTION-SCREEN END OF BLOCK B2.

\*SELECTION-SCREEN END OF SCREEN 200.

INITIALIZATION.

LB1 = 'ENTER A VALUE'.

LB2 = 'ENTER B VALUE'.

LB3 = 'DISPLAY'.

LB4 = 'EXIT'.

LB5 = 'CALL NEXT'.

AT SELECTION-SCREEN.

CASE SY-UCOMM.

WHEN 'PB1'.

LEAVE TO LIST-PROCESSING.

WRITE : A, B.

IF CH1 = 'X'.

LEAVE TO LIST-PROCESSING.

WRITE 'FIRST CHECKBOX IS SELECTED'.

ENDIF.

IF CH2 = 'X'.

WRITE :/ 'SECOND CHECKBOX IS SELECTED'.

ENDIF.

IF CH3 = 'X'.

WRITE :/ 'THIRD CHECKBOX IS SELECTED'.

ENDIF.

IF RB1 = 'X'.

WRITE :/ 'FIRST RADIOBUTTON'.

```
ELSEIF RB2 = 'X'.  
WRITE :/ 'SECOND RADIOBUTTON'.  
ELSEIF RB3 = 'X'.  
WRITE :/ 'THIRD RADIOBUTTON'.  
ENDIF.
```

```
WHEN 'PB2'.  
LEAVE PROGRAM.
```

```
WHEN 'PB3'.  
CALL SELECTION-SCREEN 200.  
ENDCASE.
```

```
EN - LIBRARY HELP
```

## DATA DICTIONARY

Data dictionary objects are:

- TABLES
- VIEWS
- DATA ELEMENTS
- DOMAINS
- SEARCH HELP
- LOCK OBJECT
- TYPE GROUP

SE11 is the Tcode used to access and create data dictionary objects.

Some of the predefined tables to be used in common are:

MARA - GENERAL MATERIAL DATA.  
MARC - PLANT DATA FOR MATERIAL.  
VBAK - SALES DOCUMENT : HEADER DATA.  
KNA1 - GENERAL DATA IN CUSTOMER MASTER.  
KNB1 -  
KNC1 -  
T001 -  
LFA1 -  
LFB1 -  
LFC1 -

### CREATING A TABLE:

There are two ways to create a table:

1. built-in type.
2. using data elements and domains.

SE11 -> select DATABASE TABLE radiobutton -> Specify table name starting with Z or Y -> Enter short description.

Table creation involves specifying delivery class, maintenance, data class and size.

DELIVERY CLASS specifies whether the table is Application table, customizing table, etc.

Application table stands for master and transaction data table.

In master table, reads are more, writes are less.

In transaction table, reads are less, writes are more.

### **Maintenance:**

Specifies whether a table can be modified in the future or not. It is suggested to create a built-in type table with 'MAINTENANCE ALLOWED'.

### **Technical Settings:**

This area has two attributes namely:

Data Class - Specifies whether the table is application table or customizing table.

Size Category - Specifies initial size for the table inside the database.

### **TABLES CLASSIFICATION BASED ON BUFFERING:**

SINGLE BUFFERING - Whenever a table is created with this option and when the user tries to access records from the database table, a buffer is created in AS for only a single record.

GENERIC BUFFERING - By default, a buffer is created in AS for a single record. But depending on selection criteria, buffer size can be increased or decreased.

select \* from <table\_name> up to 10 rows.

FULLY BUFFERING - Based on the size category specified, a buffer is created in AS for the table.

### **TABLES CLASSIFICATION BASED ON CLIENT:**

CLIENT-DEPENDENT TABLE - If a first field of a table is 'MANDT', this table is called client-dependent table. This type of table datas can be accessed from only within the same client.

CLIENT-INDEPENDENT TABLE - If the first field is not 'MANDT', then this table is called as CLIENT-INDEPENDENT table.

Use the following code to access records from the existing database table:

```
tables zstudtable.
select * from zstudtable.
write :/ zstudtable-studid, zstudtable-studname, zstudtable-studmarks.
endselect.
```

Here, TABLES statement will create a temporary table memory area in AS.

### **CREATING TABLE USING DATA ELEMENTS AND DOMAINS:**

**DOMAIN :** This data dictionary object is used to assign data type and length for a field to be created in a table.

**DATA TYPE OR DATA ELEMENT :** This data dictionary object acts as a label for the domain.

#### **Navigations for creating a domain:**

SE11 -> Select Domain radiobutton -> Specify name of the domain starting with Z or Y -> Create -> Enter short description -> Select data type -> Enter length -> Save -> Activate.

#### **Navigations for creating a data element:**

SE11 -> Select Data Type radiobutton -> Specify name of the data element starting with Z or Y -> Create -> Enter short description -> Select 'Domain' radiobutton -> Specify name of the domain you created previously -> Press Enter -> Save -> Activate -> Come back.

Create a table with the same navigations. Select 'DATA ELEMENT' PUSHBUTTON from the Fields tab button.

Enter field name -> Select the checkbox of initial field to declare it as a primary key -> Specify data element name -> Press Enter.

**VIEWS:**

- View is a data dictionary object.
- It exists logically inside the data dictionary.
- It is used to fetch datas from more than one database table.
- In views, we specify the tables and SAP will automatically generate a join condition code in the background.

**Navigations to create a view:**

SE11 -> Select 'VIEW' radiobutton -> Specify view name starting with Z or Y  
-> Create -> Select as 'DATABASE VIEW' -> Click on Copy -> Enter short description -> Specify name of the tables from which you want to fetch records -> Click on Relationships pushbutton -> Opens an interface -> Select the condition based on primary key relationship -> Click on Copy -> A join condition is created -> Click on 'View Fields' tab button -> Click on Table fields pushbutton -> Opens an interface with specified table names -> Click on first table -> Click on Choose -> Select required fields -> Click on Copy -> Do the same thing for choosing fields from second table -> Save -> Activate -> Discard the warning message.

To implement the view created, use SE38 editor and write the following code:

```
TABLES ZMYVIEW1.
```

```
SELECT * FROM ZMYVIEW1.  
WRITE :/ ZMYVIEW1-MATNR, ZMYVIEW1-MTART, ZMYVIEW1-MBRSH,  
ZMYVIEW1-MEINS,  
ZMYVIEW1-WERKS, ZMYVIEW1-LVORM, ZMYVIEW1-MMSTA.  
ENDSELECT.
```

Save -> Activate -> Execute.

In the above code, TABLES statement is used to create a temporary table space area to hold the records of view.

**TABLE TYPES:****TRANSPARENT TABLES:**

These type of tables have one to one relationship with the underlying database table. The structure of this table will be in the form of rows and columns.

MASTER AND TRANSACTION DATAS.

**POOLED TABLE:**

Several pooled tables in data dictionary are stored under one table pool in the database.

**CLUSTER TABLE:**

Several cluster tables in data dictionary are stored under one table cluster in the database.

The difference between pooled table and cluster table is in the form of storage of datas.

Several pooled table records are stored in one table pool in the database in continuous format using a row separator.

Whereas in the case of table cluster, the records are stored in continuous format without any row separator.

**TABLES CLASSIFICATION BASED ON CLIENT:**

- CLIENT-DEPENDENT TABLE
- CLIENT-INDEPENDENT TABLE.

**TABLES CLASSIFICATION BASED ON BUFFERING:**

- SINGLE RECORD BUFFERING
- GENERIC BUFFERING

- FULLY BUFFERED.

### **ADDING SEARCH HELP FUNCTIONALITY (F4) TO THE PARAMETERS STATEMENT:**

SE11 -> Select SEARCH HELP RADIOBUTTON -> Specify search help name starting with Z or Y -> Click on Create -> Select 'Elementary Search Help' radiobutton -> Click on Continue -> Opens an interface -> Enter Short description -> Specify table name in 'SELECTION TEXT' field -> Enter field name in the first tabular column (Under search help parameter) -> Save -> Activate.

To implement F4 functionality for parameters statement:

SE38 -> Create a program -> Enter the following code:

```
PARAMETERS MATNUM(20) MATCHCODE OBJECT ZMYSEARCH2.
WRITE MATNUM.
```

In the above code, 'MATCHCODE OBJECT' addition for parameters statement is used to assign search help functionality for the parameter variable.

-> Save -> Activate -> Execute.

### **TYPE GROUP:**

Type group is a data dictionary object used to store user-defined data types and constants.

DATA statement is used to declare variables for pre-defined data types.

TYPES statement is used to declare user-defined data types.

### **Navigations to create TYPE GROUP:**

SE11 -> Select Type group radiobutton -> Specify type group name starting with Z or Y -> Click on Create -> Enter short description -> Continue -> Opens Type Group interface -> Write the following code inside editor:

```
TYPE-POOL YMYTY .
```

```
TYPES : BEGIN OF YMYTY_STALIN,
        NAME(20),
        COURSE(10),
        AGE TYPE I,
```

END OF YMYTY\_STALIN.

Save -> Activate.

In the SE38 editor, write the following code:  
TYPE-POOLS YMYTY.

```
DATA A TYPE YMYTY_STALIN.
A-NAME = 'JEGAN'.
A-COURSE = 'ABAP'.
A-AGE = '24'.
WRITE :/ A-NAME, A-COURSE, A-AGE.
```

Save -> Activate -> Execute.

In the above example, TYPE-POOLS statement is used to access user-defined data types created in a type group.

User-defined data types can be made globally accessed if saved under a type group.

### **MESSAGE CLASSES:**

Message Classes are container of user-defined messages. We can create our own messages to be displayed in the screen and we can create these messages in a message pool inside the message class.

SE91 is the Tcode to create a message class.

The created messages can be called from programs using the statement 'MESSAGE'.

SYNTAX:

*MESSAGE <type\_of\_message> <message id>(message\_class).*

### **TYPES OF MESSAGES:**

- S - Status message.
- I - Information message.
- E - Error message.
- W - Warning message.
- A - Abort message.
- T - Terminate message.

You can create up to 1000 messages in a single message class. The message id ranges from 0 - 999.

### **Navigations to create a MESSAGE CLASS:**

SE91 -> Opens Message Maintenance Initial screen -> Specify Name of message class starting with Z or Y -> Click on Create -> Enter short description -> Click on MESSAGES Tab button -> Save message class under a package and assign request number -> Define your own messages -> Save Again -> Come back.

From SE38 editor, use the following code:

```
selection-screen pushbutton 10(15) lb1 user-command pb1.
initialization.
lb1 = 'press this'.
```

```
at selection-screen.
case sy-ucomm.
when 'PB1'.
MESSAGE T000(ZMYMESSAGE).
ENDCASE.
```

### **VARIANTS:**

This concept is used to create default values for the input fields. Variants are helpful for the SAP end users and data entry operators.

In SE38 editor, create the following code:

```
parameters : a(10), b(10), c(10), d(10).
```

Save -> Activate -> Come back -> Click on Variants radiobutton -> Click on Change -> Opens Variants Initial Screen -> Specify name of the variant starting with Z or Y -> Click on Create -> Opens an interface with the input fields -> Click on Attributes pushbutton from Application toolbar -> Enter short description for variant -> Save -> Enter variant values in the input fields -> Save again -> Come back -> Come back -> Click on Source code radiobutton -> Click on change -> Execute the program -> You will see 'Get Variant' pushbutton option in the application toolbar -> Click on pushbutton to get variant values.

## **INTERNAL TABLES**

Internal table is a user-defined temporary work area to store the database table records.

If you want to make any changes to the table records, make the changes in internal table and then update the records back into the database table.

syntax:

data <internal\_table\_name> like <database\_table\_name> occurs <size> with header line.

DATA JEGAN LIKE MARA OCCURS 0 WITH HEADER LINE.

data stalin like kna1 occurs 0 with header line.

Here, OCCURS 0 specifies that internal table should initially take 8 KB of memory area in AS.

1 kb = 1024 bytes

Occurs 0 = 8 \* 1024 bytes

### **Internal tables are used in:**

BATCH DATA COMMUNICATIONS - Migrate data from legacy system to SAP database.

SAPSCRIPTS AND SMARTFORMS - To redirect SAP data into external devices like printer, fax, e-mail, etc.

MODULE POOL PROGRAMS - Dialog programming to make use of screens.

SUBROUTINES AND FUNCTION MODULES. - Modularization technique (module programming).

### **PURPOSE OF INTERNAL TABLES:**

\* To avoid unnecessary traffic in the network while retrieving records from database.

**SYNTAX OF internal table:**

```
DATA : BEGIN OF <internal table name> occurs 0,
      .
      .
      attributes ..
      .
      END OF <internal table name>.
```

**DEFINING A STRUCTURE:**

eg.

```
DATA : BEGIN OF WA,
      NAME(10),
      COURSE(10),
      END OF WA.
```

```
WA-NAME = 'PRAVIN'.
WA-COURSE = 'SQLSERVER'.
```

```
WA-NAME = 'DEEPA'.
WA-COURSE = 'ORACLE'.
```

```
WA-NAME = 'SYED'.
WA-COURSE = 'ABAP'.
```

```
WRITE :/ WA-NAME, WA-COURSE.
```

A structure can hold only one record. To make the structure hold more than one record, declare an internal table.

```
DATA : BEGIN OF <internal_table> occurs 0,
      .
      .
      attributes
      .
      end of <internal_table>.
```

eg. code

```
data : begin of itab occurs 0,
```

```

    name(10),
    course(10),
    end of itab.

itab-name = 'karthik'.
itab-course = 'abap'.
append itab.

itab-name = 'pravin'.
itab-course = 'sql'.
append itab.

loop at itab.
write :/ itab-name, itab-course.
endloop.

```

In the above code, when executed, an internal table in the name of ITAB is created without header line.

**APPEND** statement is used to insert the values into body area of the internal table even if the record already exists.

**COLLECT** statement is also used to insert the values into body area of the internal table, but will check for index value. If the record already exists, it wont insert the same value.

**LOOP-ENDLOOP** statement is used to view the contents of internal table body area. Using this statement, we can view the entire contents of internal table.

To view a single line from body area of internal table, use **READ** statement.

eg. code using READ STATEMENT:

```

data itab like mara occurs 0 with header line.
select * from mara into table itab.
loop at itab.
write :/ sy-tabix, itab-matnr, itab-mbrsh, itab-mtart, itab-meins.
endloop.
read table itab index 70.
write :/ sy-tabix color 5, itab-matnr, itab-mbrsh, itab-mtart, itab-meins.

```

A structure can hold only record by default. To allocate extra memory space for a structure to hold more values, use the statement OCCURS 0. Using this statement, an internal table structure is created with 8 KB memory of body area without header line.

eg.

```
DATA : Begin of syed occurs 0,
      name(10),
      course(10),
      End of syed.
```

To create an internal table with HEADER LINE, use 'WITH HEADER LINE' statement.

eg.

```
DATA DEEPA LIKE <database table name> OCCURS 0 WITH HEADER LINE.
```

Now, an internal table is created with header line and body area.

### **HEADER LINE :**

This is a structure to hold field attributes of a table. The size is by default created to hold 1 record.

### **BODY AREA :**

This is a storage area for holding records fetched from database table.

### **STATEMENTS TO WRITE OUT THE CONTENTS OF INTERNAL TABLE:**

LOOP-ENDLOOP - Used to view or write out the entire contents of internal table body area.

READ - Used to view or write out a single record we specify using index value from the internal table body area.

If we use READ statement, SAP performs binary search by default to fetch the index value.

Eg. code to insert user-defined values into database table using internal table structure:

DATA ITAB1 LIKE MARA OCCURS 0 WITH HEADER LINE.

```
ITAB1-MATNR = 'MATNUMBER01'.  
ITAB1-MBRSH = 'P'.  
ITAB1-MTART = 'COUP'.  
ITAB1-MEINS = 'KG'.  
APPEND ITAB1.
```

```
ITAB1-MATNR = 'MATNUMBER02'.  
ITAB1-MBRSH = 'P'.  
ITAB1-MTART = 'COUP'.  
ITAB1-MEINS = 'KG'.  
APPEND ITAB1.
```

```
ITAB1-MATNR = 'MATNUMBER03'.  
ITAB1-MBRSH = 'P'.  
ITAB1-MTART = 'COUP'.  
ITAB1-MEINS = 'KG'.  
APPEND ITAB1.
```

LOOP AT ITAB1.

INSERT INTO MARA VALUES ITAB1.

```
IF SY-SUBRC = 0.  
MESSAGE S005(ZMSG).
```

```
*WRITE :/ 'VALUES ARE INSERTED'.  
ELSEIF SY-SUBRC = 4.  
MESSAGE E006(ZMSG).
```

```
*WRITE :/ 'VALUES ALREADY EXIST'.  
ENDIF.
```

ENDLOOP.

In the above example, SY-SUBRC system variable is used to check whether the action has been successfully completed or not. This system variable is predefined by SAP with some default values such as 0 and 4.

IF SY-SUBRC = 0, the specified action has been completed successfully,  
IF SY-SUBRC = 4, the specified action has not taken place.

**TYPES OF INTERNAL TABLE:**

Internal table is basically divided into two types:

## 1. INDEXED INTERNAL TABLES

- STRUCTURED INTERNAL TABLE
- STANDARD INTERNAL TABLE
- SORTED INTERNAL TABLE

## 2. NON-INDEXED INTERNAL TABLES

- HASHED INTERNAL TABLE

**STRUCTURED INTERNAL TABLE:**

Any internal table declared or created using BEGIN OF - END OF statement is called as structured internal table.

eg.

```
DATA : BEGIN OF <internal table name>,
      ..attributes...
      END OF <internal table name>.
```

**STANDARD INTERNAL TABLE:**

This type of internal table performs the search for a record in the body area using LINEAR SEARCH TECHNIQUE.

**SYNTAX:**

```
DATA <internal_table_name> LIKE STANDARD TABLE OF
<database_table_name>
```

```
.
WITH DEFAULT/NON-UNIQUE KEY
      WITH HEADER LINE
      INITIAL SIZE <size>
```

eg. code

```
DATA ITAB LIKE STANDARD TABLE OF MARA WITH NON-UNIQUE KEY MTART
WITH HEADER LINE INITIAL SIZE 0.
SELECT * FROM MARA INTO TABLE ITAB.
LOOP AT ITAB.
WRITE :/ ITAB-MATNR, ITAB-MBRSH, ITAB-MTART, ITAB-MEINS.
ENDLOOP.
```

If we try to read a single value from the above internal table body area, a linear search will be performed. Whereas, for normal internal table, the default search is binary search.

### **SORTED INTERNAL TABLE:**

This internal table will hold the records from the database table in a sorted manner.

SYNTAX:

```
DATA <internal_table_name> LIKE SORTED TABLE OF
<database_table_name>
    WITH DEFAULT/NON-UNIQUE KEY
    WITH HEADER LINE
    INITIAL SIZE <size>.
```

Eg.

```
DATA ITAB LIKE SORTED TABLE OF MARA WITH NON-UNIQUE KEY MBRSH
WITH HEADER LINE INITIAL SIZE 0.
```

```
SELECT * FROM MARA INTO TABLE ITAB.
LOOP AT ITAB.
WRITE :/ ITAB-MATNR, ITAB-MBRSH, ITAB-MTART, ITAB-MEINS.
ENDLOOP.
```

### **NON-INDEXED HASHED INTERNAL TABLE:**

SYNTAX:

```
DATA <internal_table_name> LIKE HASHED TABLE OF
<database_table_name>
    WITH DEFAULT/NON-UNIQUE KEY
    WITH HEADER LINE
    INITIAL SIZE <size>.
```

This internal table is not assigned any index value. The retrieval of records from internal body area is based on key field or a row.

Eg. code

```
DATA ITAB LIKE HASHED TABLE OF MARA WITH UNIQUE KEY MTART WITH
HEADER
LINE INITIAL SIZE 0.
SELECT * FROM MARA INTO TABLE ITAB.
LOOP AT ITAB.
WRITE :/ ITAB-MATNR, ITAB-MBRSH, ITAB-MTART, ITAB-MEINS.
ENDLOOP.
```

For this type of internal table, we have to specify only the primary key for a retrieval function.

### **SYSTEM VARIABLES USED IN INTERNAL TABLE:**

**SY-TFILL** - This system variable is used to describe number of records fetched and inserted into body area of internal table. Before using this statement, we have to use 'DESCRIBE TABLE' statement.

**SY-TABIX** - This system variable is used to create a loop iteration counter for internal table body area.

**SY-DBCNT** - This system variable is used to describe number of records fetched from database table.

Eg. code

```
DATA ITAB LIKE KNA1 OCCURS 0 WITH HEADER LINE.
SELECT * FROM KNA1 INTO TABLE ITAB.
WRITE :/ SY-DBCNT COLOR 4.
```

```
DESCRIBE TABLE ITAB.
WRITE :/ SY-TFILL COLOR 6.
```

```
LOOP AT ITAB.
WRITE :/ SY-TABIX, ITAB-KUNNR, ITAB-LAND1, ITAB-NAME1, ITAB-ORT01.
ENDLOOP.
```

**SORTING THE INTERNAL TABLE AFTER FETCHING VALUES:**

SYNTAX:

`SORT <internal_table_name> BY <field_names>.`

example code:

```
DATA ITAB LIKE KNA1 OCCURS 0 WITH HEADER LINE.
```

```
SELECT * FROM KNA1 INTO TABLE ITAB.
```

```
WRITE :/ SY-DBCNT COLOR 4.
```

```
DESCRIBE TABLE ITAB.
```

```
WRITE :/ SY-TFILL COLOR 6.
```

```
SORT ITAB BY LAND1 ORT01 NAME1 KUNNR.
```

```
LOOP AT ITAB.
```

```
WRITE :/ sy-tabix, ITAB-KUNNR, ITAB-LAND1, ITAB-NAME1, ITAB-ORT01.
```

```
ENDLOOP.
```

A SORTED INTERNAL TABLE cannot be resorted. i.e, we cannot use SORT statement to the sorted internal table.

## SUBROUTINES

**FORM - ENDFORM** statement is used to create subroutines.

**PERFORM** statement is used to invoke the subroutine created.

Subroutines are used to call a piece of code frequently within the program or externally from other program.

### **LOCAL SUBROUTINES:**

The subroutine is called locally from the same program using PERFORM statement.

eg. code of local subroutine without any value:

```
PERFORM STALIN.
```

```
FORM STALIN.
```

```
DO 5 TIMES.  
WRITE :/ 'WELCOME TO ABAP'.  
ENDDO.
```

```
ENDFORM.
```

eg. code of external subroutine without any value:

create an executable program and write the following code:

```
REPORT ZSUBROUTINES2          .
```

```
PERFORM STALIN.
```

```
FORM STALIN.  
DO 5 TIMES.  
WRITE :/ 'WELCOME TO ABAP'.  
ENDDO.  
ENDFORM.
```

Save -> Activate.

Create another executable program and write the following code:

```
REPORT ZSUBROUTINES1.
```

```
PERFORM STALIN(ZSUBROUTINES2).
```

Save -> Activate -> Execute.

We have to specify name of the program where subroutine is created within the bracket whenever we try to invoke the subroutine externally.

### **PASS BY REFERENCE:**

```
DATA : A(10) VALUE 'INDIA', B TYPE I VALUE '20'.
```

```
PERFORM STALIN USING A B.
```

```
FORM STALIN USING X Y.
```

```
WRITE :/ X , Y.
```

```
ENDFORM.
```

### **PASS BY VALUE:**

```
DATA : A(10) VALUE 'INDIA', B TYPE I VALUE '20'.
```

```
PERFORM STALIN USING A B.
```

```
FORM STALIN USING X Y.
```

```
X = 'AMERICA'.
```

```
Y = '100'.
```

```
WRITE :/ X , Y.
```

```
ENDFORM.
```

### **PASSING INTERNAL TABLE AS AN ARGUMENT:**

```
DATA ITAB LIKE KNA1 OCCURS 0 WITH HEADER LINE.
```

```
SELECT * FROM KNA1 INTO TABLE ITAB.
```

PERFORM DISPLAY TABLES ITAB .

FORM DISPLAY TABLES ITAB STRUCTURE KNA1.

LOOP AT ITAB.

WRITE :/ ITAB-KUNNR, ITAB-NAME1, ITAB-LAND1, ITAB-ORT01.

ENDLOOP.

ENDFORM.

## Reports

Fetching required data from the database and redirecting to the output devices or displaying the data in LPS screen as per client's requirements is the concept behind report generation.

### **TYPES OF REPORTS:**

1. GRAPHICAL REPORTS
2. ALV REPORTS (ABAP LIST VIEWER).
3. GROUP REPORTS
4. INTERACTIVE REPORTS
5. CLASSICAL REPORTS (FORMATTED & UN FORMATTED).

order of execution of classical report:

INITIALIZATION.  
 AT SELECTION-SCREEN.  
 START-OF-SELECTION.  
 TOP-OF-PAGE.  
 END-OF-PAGE.  
 END OF SELECTION.

ORDER OF EXECUTION OF INTERACTIVE REPORT:

INITIALIZATION.  
 AT USER-COMMAND.  
 START-OF-SELECTION.  
 TOP-OF-PAGE.  
 TOP-OF-PAGE DURING LINE SELECTION.  
 END OF PAGE.  
 END OF SELECTION.

### **Classical report.**

REPORT YCLASSRPT no standard page heading line-count 10(2)

.

tables: spfli.  
 include <list>.

DATA: BEGIN OF it occurs 0,  
       carrid like spfli-carrid,  
       connid like spfli-connid,

cityfrom like spfli-cityfrom,  
 cityto like spfli-cityto,  
 fltime like spfli-fltime,  
 END OF it.

select-options: carrid for spfli-carrid,  
 connid for spfli-connid,  
 cityfrom for spfli-cityfrom.

start-of-selection.

select \* from spfli into corresponding fields of table it where carrid  
 in carrid and connid in connid and cityfrom in cityfrom.

loop at it.  
 write:/ it-carrid, it-connid, it-cityfrom, it-cityto , it-fltime input on.  
 endloop.

top-of-page.

write:/15 'flight details' color COL\_HEADING, icon\_green\_light AS ICON .  
 .  
 write:/15 '-----'.

end-of-page.

### **GRAPHICAL REPORTS:**

These type of reports are used display the database table in a graphical format either in two-dimensional or three-dimensional format.

### **FUNCTION MODULES USED IN GRAPHICAL REPORTS:**

***GRAPH\_2D***

***GRAPH\_3D***

eg. code to generate graphical report for annual company sales:

DATA : BEGIN OF ITAB OCCURS 0,

```

COMPNAME(20),
COMPSALES TYPE I,
END OF ITAB.

```

```

ITAB-COMPNAME = 'MRF'.
ITAB-COMPSALES = '2000000'.
APPEND ITAB.

```

```

ITAB-COMPNAME = 'FENNER INDIA LTD'.
ITAB-COMPSALES = '4000000'.
APPEND ITAB.

```

```

ITAB-COMPNAME = 'SANMAR'.
ITAB-COMPSALES = '7000000'.
APPEND ITAB.

```

```

ITAB-COMPNAME = 'EICHER'.
ITAB-COMPSALES = '3500000'.
APPEND ITAB.

```

```

*CALL FUNCTION 'GRAPH_2D'
CALL FUNCTION 'GRAPH_3D' (CTRL+F6)
EXPORTING
  TITL           = 'COMPANY SALES REPORT'
  TABLES
    data         = ITAB.

```

### **ALV REPORTS (ABAP LIST VIEWER):**

The function modules used in this kind of report are:

***REUSE\_ALV\_GRID\_DISPLAY***  
***REUSE\_ALV\_LIST\_DISPLAY***

eg. code:

DATA ITAB LIKE MARA OCCURS 0 WITH HEADER LINE.

SELECT \* FROM MARA INTO TABLE ITAB.

\*CALL FUNCTION 'REUSE\_ALV\_LIST\_DISPLAY'

```
CALL FUNCTION 'REUSE_ALV_GRID_DISPLAY'
EXPORTING
  I_STRUCTURE_NAME          = 'MARA'
  TABLES
    t_outtab                = ITAB.
```

### **GROUP REPORTS:**

These reports are also called as CLASSICAL REPORTS which is divided into:

CLASSICAL UNFORMATTED REPORTS

### **CLASSICAL FORMATTED REPORTS.**

TOP-OF-PAGE - Used to create header area for the report.

END-OF-PAGE - Used to create footer area for the report.

eg. code:

DATA BASKAR LIKE MARA OCCURS 0 WITH HEADER LINE.

```
SELECT * FROM MARA INTO TABLE BASKAR.
```

```
WRITE :/5(70) SY-ULINE.
```

```
LOOP AT BASKAR.
```

```
WRITE :/ SY-VLINE, 5(20) BASKAR-MATNR, SY-VLINE, 27(15) BASKAR-
MBRSH, SY-VLINE, 45(15) BASKAR-MTART, SY-VLINE, 62(10) BASKAR-
MEINS, SY-VLINE.
```

```
WRITE:/(70) SY-ULINE.
```

```
ENDLOOP.
```

```
WRITE :/5(70) SY-ULINE.
```

top-of-page.

```
*WRITE :/40(50) 'THIS IS HEADER AREA FOR THE REPORT' COLOR 4
CENTERED.
```

```
WRITE :/5(20) 'MATERIAL NUMBER', 27(15) 'INDUSTRY SECTOR', 45(15)
'MATERIAL TYPE', 62(10) 'MEASUREMENT'.
SKIP 2.
```

END-OF-PAGE.

```
WRITE :/40(50) 'THIS IS FOOTER AREA FOR THE REPORT' COLOR 7
CENTERED.
```

```
WRITE :/ 'CURRENT PAGE NUMBER IS', SY-PAGNO COLOR 6.
```

In the above example, TOP-OF-PAGE event is used to create header area and END-OF-PAGE event is used to create footer area. The header area can be used to create subheadings or report headings and the footer area can be used to specify page numbers, totals, etc.

### **ALV REPORTS:**

ALV stands for ABAP List Viewer. Using ALV reports, we can generate the list in LIST and GRID formats. The function modules used for ALV reports are:

1. REUSE\_ALV\_LIST\_DISPLAY
2. REUSE\_ALV\_GRID\_DISPLAY

Eg. Code:

```
DATA ITAB LIKE KNC1 OCCURS 0 WITH HEADER LINE.
SELECT * FROM KNC1 INTO TABLE ITAB.
CALL FUNCTION 'REUSE_ALV_GRID_DISPLAY'
*CALL FUNCTION 'REUSE_ALV_LIST_DISPLAY'
EXPORTING
  I_STRUCTURE_NAME      = 'KNC1'
  TABLES
    T_OUTTAB            = ITAB.
```

### **GROUP REPORTS:**

CONTROL BREAK STATEMENTS - These statements are used to control the flow of execution of program within the loop. The following statements are used within LOOP-ENDLOOP statement as control break statements:

1. ON CHANGE OF  
ENDON.

2. AT FIRST.  
ENDAT.

3. AT LAST.  
ENDAT.

4. AT NEW.  
ENDAT.

5. AT END OF.  
ENDAT.

### **ON CHANGE OF-ENDON:**

This processing block gets executed whenever there is a change in the field name of internal table.

eg. code:

```
DATA ITAB LIKE KNA1 OCCURS 0 WITH HEADER LINE.
SELECT * FROM KNA1 INTO TABLE ITAB.
SORT ITAB BY LAND1.
```

```
LOOP AT ITAB.
ON CHANGE OF ITAB-LAND1.
SKIP 1.
WRITE :/ 'CUSTOMER MASTER DETAILS FOR' COLOR 4, ITAB-LAND1 COLOR
4.
SKIP 1.
ENDON.
WRITE :/ ITAB-KUNNR, ITAB-NAME1, ITAB-LAND1, ITAB-ORT01.
ENDLOOP.
```

```
TOP-OF-PAGE.
WRITE :/40(40) 'THIS IS CUSTOMER MASTER DATA' COLOR 5.
```

### **AT FIRST-ENDAT:**

This processing block gets triggered at the first pass of internal table loop.

eg. code:

```
DATA ITAB LIKE KNA1 OCCURS 0 WITH HEADER LINE.
SELECT * FROM KNA1 INTO TABLE ITAB.
```

SORT ITAB BY LAND1.

LOOP AT ITAB.

AT FIRST.

WRITE :/ 'THIS IS CUSTOMER MASTER DATA'.

SKIP 1.

ENDAT.

WRITE :/ ITAB-KUNNR, ITAB-NAME1, ITAB-LAND1, ITAB-ORT01.

ENDLOOP.

TOP-OF-PAGE.

WRITE :/40(40) 'THIS IS CUSTOMER MASTER DATA' COLOR 5.

### **AT LAST - ENDAT:**

This processing block gets executed at the last pass of the internal table loop.

eg. code:

DATA ITAB LIKE KNC1 OCCURS 0 WITH HEADER LINE.

SELECT \* FROM KNC1 INTO TABLE ITAB.

SORT ITAB BY GJAHR.

LOOP AT ITAB.

AT LAST.

SUM. \*aggregate function

WRITE :/ 'THE GRANDTOTAL OF SALES IS', ITAB-UM01U.

ENDAT.

WRITE :/ ITAB-KUNNR, ITAB-GJAHR, ITAB-UM01U.

ENDLOOP.

### **AT NEW-ENDAT:**

Whenever we use this control break statement, make use of only the required fields in the internal table.

DATA : BEGIN OF ITAB OCCURS 0,

    GJAHR LIKE KNC1-GJAHR,

    UM01U LIKE KNC1-UM01U,

    END OF ITAB.

SELECT GJAHR UM01U FROM KNC1 INTO TABLE ITAB.

SORT ITAB BY GJAHR.

LOOP AT ITAB.

```

AT NEW GJAHR.
SKIP 1.
WRITE :/ 'SALES DONE IN THE FISCAL YEAR', ITAB-GJAHR.
SKIP 1.
ENDAT.
WRITE :/ ITAB-GJAHR, ITAB-UM01U.
ENDLOOP.

```

### **AT END OF-ENDAT.**

This processing block gets executed at the end of each field inside the internal table.

eg. code:

```

DATA : BEGIN OF ITAB OCCURS 0,
        GJAHR LIKE KNC1-GJAHR,
        UM01U LIKE KNC1-UM01U,
        END OF ITAB.

```

```

SELECT GJAHR UM01U FROM KNC1 INTO TABLE ITAB.
SORT ITAB BY GJAHR.

```

```

LOOP AT ITAB.
AT END OF GJAHR.
SUM.
WRITE :/ 'TOTAL SALES DONE IN THE FISCAL YEAR', ITAB-UM01U.
SKIP 1.
ENDAT.

```

```

WRITE :/ ITAB-GJAHR, ITAB-UM01U.
ENDLOOP.

```

### **INTERACTIVE REPORTS:**

Making the lists to communicate with each other is called as interactive report. When we execute the reporting program, the initial list we are getting as output is called as PRIMARY LIST.

Using interactive report concept, we can create secondary lists. Totally, we can create 21 lists using interactive reports. Each list is assigned an index value ranging from 0-20.

The list index assigned for the primary list is 0. The list index assigned for the first secondary list is 1. The secondary list index ranges from 1-20.

The system variables used in interactive reports:

**SY-LSIND** - Holds the current list index.

**SY-LILLI** - Holds the line number in the list.

**SY-LISEL** - Holds the contents of the line selected.

Eg. code:

```
DATA FLDNAME(30).
DATA ITAB LIKE KNA1 OCCURS 0 WITH HEADER LINE.
SELECT * FROM KNA1 INTO TABLE ITAB.
```

```
SORT ITAB BY LAND1.
```

```
LOOP AT ITAB.
WRITE :/ ITAB-KUNNR, ITAB-NAME1, ITAB-LAND1, ITAB-ORT01.
ENDLOOP.
```

```
AT LINE-SELECTION.
GET CURSOR FIELD FLDNAME.
IF FLDNAME = 'ITAB-KUNNR'.
WRITE :/ 'THE CURRENT LIST INDEX IS', SY-LSIND.
SKIP 1.
WRITE :/ SY-LISEL.
ENDIF.
```

GET CURSOR FIELD statement is used to specify the cursor position on the list.

'HOTSPOT ON' statement is used to provide single click functionality on any field value in the list.

### **STRING FUNCTIONS:**

SPLIT - This function is used to split the given string value based on any separator and save in separate string variables.

eg. code:

```
DATA STR(30) VALUE 'SAP IS AN ERP'.
DATA: S1(5), S2(5), S3(5), S4(5).
SPLIT STR AT ' ' INTO S1 S2 S3 S4.
```

WRITE :/ S1, / S2, / S3, / S4.

**SEARCH** - This string function is used to search for the specified value in a given string.

eg. code:

```
DATA STR(30) VALUE 'SAP IS AN ERP'.
DATA S1(3) VALUE 'AN'.
SEARCH STR FOR S1.
WRITE :/ SY-FDPOS.
```

**sy-fdpos** is the system variable used to hold the position of an alphabet in the string.

**CONCATENATE** - This string function is used to add the different character type variables into one string.

eg. code:

```
DATA STR(30).
DATA: S1(5) VALUE 'INDIA', S2(5) VALUE 'IS', S3(5) VALUE 'GREAT'.
CONCATENATE S1 S2 S3 INTO STR SEPARATED BY ' '.
WRITE :/ STR.
```

**SHIFT** - This string function is used to move the string to the specified position (LEFT, RIGHT, CIRCULAR) based on the number of places specified.

eg. code:

```
*DATA STR(20) VALUE 'ABAP IS IN SAP'.
*SHIFT STR BY 5 PLACES.
*WRITE STR COLOR 5.
```

```
*DATA STR(20) VALUE 'ABAP IS IN SAP'.
*SHIFT STR RIGHT BY 5 PLACES.
*WRITE STR COLOR 5.
```

```
*DATA STR(20) VALUE 'ABAP IS IN SAP'.
*SHIFT STR CIRCULAR BY 5 PLACES.
*WRITE STR COLOR 5.
```

**TRANSLATE** - This string function is used to change the case of the given string value.

eg. code:

```
DATA STR(10) VALUE 'CHENNAI'.
TRANSLATE STR TO LOWER CASE.
WRITE STR.
```

**CONDENSE** - This string function is used to remove the blank spaces in the given string.

eg. code:

```
data str(30) value '      INDIA IS GREAT'.
CONDENSE STR.
WRITE STR.
```

**REPLACE** - This string function is used to replace the given string with specified string value.

eg. code:

```
data str(30) value 'INDIA IS GREAT'.
DATA S1(10) VALUE 'WRITE'.
REPLACE 'IS' IN STR WITH S1.
WRITE STR.
```

**STRLEN** - This string function is used to identify the length of the given string.

Eg. code:

```
data str(30) value 'INDIA IS GREAT'.
data len type i.
len = STRLEN( STR ).
write len.
```

**OVERLAY**  
**FIND**

### **Logical Data bases:**

- It is a tool used to increase the report efficiency.
- It is a collection of nodes, where each node is created for one table from the database.
- The first node created in the logical database is called as root node.
- Under the root node, we create any number of child nodes for the table.
- Logical database is not an Data Dictionary object. It is a repository object.
- SLDB is the t-code to create the logical database or SE36.

### **Navigations:**

SE36 -> specify logical database name starting with Z or y -> create -> enter a short description -> save under a pack -> specify root node name -> enter short description for root node -> enter the database table name -> click on create -> opens an interface with root node -> select node -> click on selection push button from application toolbar -> click on yes to generate from structure -> click on no to search help -> select two checkboxes for field selections and free selections -> click on transfer -> opens an include file -> specify range variable as follows....

Select-options : ? for kna1-kunnr

Replace ? with cusnum.

save -> activate -> come back -> select node -> click on source code pushbutton from application toolbar -> click on yes to generate program from structure and selections -> opens two include file -> double click on the header file -> declare an internal table as follows...

tables kna1.

Data itab like kna1 occurs 0 with header line.

Save -> activate -> comeback -> double click on Nxxx (sys routine file) -> double click on the node kna1 include file -> write the following code between the FORM and ENDFORM .

Select \* from kan1 into table itab where kunnr in cusnum.  
Put kna1. (\* autogen code).

Loop at itab.

```
Write :/ itab-kunnr,itab-name1,itab-land1,itab-ort01.  
Endloop.
```

Save -> activate.

Put statement is used to fetch records from the specified database table and insert into node created from that table in the logical database.

To invoke the logical database from SE38,create an executable program -> specify logical database name in the attribute selection while creating the program -> write the following code.

Nodes kna1. (Creates link b/w node and LDB and invoke pgm)

Get kna1. (to get the record from node in LDB).

Save -> activate -> execute.

## **MODULE POOL PROGRAMMING:**

- \* These are type M programs in SAP.
- \* Screen painter is a tool used to create GUI in ABAP.
- \* MPP provides screen painter to create a screen for the program.
- \* These programs cannot be executed directly.
- \* Customer-specified Transaction code (starting with Z or Y) should be created for each MPP program to execute.
- \* We can create any number of screens for a program and we can call these screens very easily.
- \* MPP programs should start with the naming convention SAPMZ or SAPMY.
- \* SE80 is the Tcode to create MPP programs.

### **EVENTS IN MPP:**

PROCESS BEFORE OUTPUT (PBO)

PROCESS AFTER INPUT (PAI)

PROCESS ON VALUE REQUEST - To provide F4 functionality to MPP program screen i/p fields.

PROCESS ON HELP REQUEST - To provide F1 functionality to MPP program screen components.

### **Navigations to create a simple MPP program:**

SE80 -> Select Program from drop-down list -> Specify program name starting with SAPMZ or SAPMY -> Press Enter -> Click on Yes to create object -> Create Top Include File by clicking on Continue icon in pop-up screens -> Save under a package -> Assign a request number -> MPP program with specified name is created with Top include File.

To create screen, Right click on program name -> Select Create -> Screen -> Opens Screen Description page -> Enter short description for screen -> Select screen type as NORMAL -> Click on LAYOUT pushbutton from application toolbar -> Opens Screen Painter -> Drag and drop two input fields from toolbar -> Likewise, create two pushbuttons -> Double click on each component -> Opens Attributes box -> Specify attributes for each screen component -> For pushbutton, specify FCT code -> Save the screen -> Click on Flowlogic pushbutton from application toolbar -> Opens Flow logic editor to create event functionalities for screen components -> Decoment PAI module -> Double click on PAI module name -> Click on Yes to create PAI module object -> Opens PAI module -> Specify the following code within module-endmodule statements:

```
CASE SY-UCOMM.
```

```
WHEN 'DISPLAY'.
LEAVE TO LIST-PROCESSING.
WRITE :/ IO1, IO2.
```

```
WHEN 'EXIT'.
LEAVE PROGRAM.
```

```
ENDCASE.
```

-> Save.

Now double click on 'Includes' Folder (TOP INCLUDE FILE) -> Declare variables for input fields as follows:

```
DATA : IO1(10), IO2(10).
```

Save -> Activate.

Now To create Transaction Code, right click on Main Program Name -> Create -> Transaction -> Opens an interface -> Enter Tcode name starting with Z or Y -> Enter short description -> Continue -> Opens interface -> Enter Main program name and Screen number to be called first -> Save under a package -> Assign a request number.

Activate all the objects of MPP program by right clicking on Main program Name -> Click on Activate -> Raises 'Objects Activated' message.

To execute the MPP program, specify Tcode name in the Command Prompt area -> Press Enter.

### **EVENTS IN MPP:**

PROCESS BEFORE OUTPUT - This event gets triggered whenever the program is executed using Tcode. This event is used to assign initial default values to the screen components.

PROCESS AFTER INPUT - This event gets triggered after some user's action in the screen (for eg, after clicking pushbutton, subsequent event functionalities).

PROCESS ON VALUE REQUEST - This event is used to assign F1 functionality for the screen components.

PROCESS ON HELP REQUEST - This event is used to assign F4 functionality for the input field in the screen.

Eg. code to make field validations in MPP program:

Using screen painter, design a screen consisting of four input fields for client, username, password and language as we have in login screen of SAP.

Assign the first two input fields to one group called GR1, the third input field to a group GR2.

Create two pushbuttons and assign FCT codes.

In the TOP INCLUDE FILE, declare following variables:

```
PROGRAM SAPMYSCREENVALID.
DATA : IO1(3), IO2(8), IO3(8), IO4(2).
DATA A TYPE I.
```

Save -> Activate.

In Flow logic editor, decomment PAI MODULE, double click on module name, and inside the module, write the following code:

```
module USER_COMMAND_0200 input.

case sy-ucomm.
WHEN 'LOGIN'.
CALL TRANSACTION 'SE38'.
WHEN 'EXIT'.
LEAVE PROGRAM.
ENDCASE.

endmodule.           " USER_COMMAND_0200 INPUT
```

Save -> Activate.

In PBO module, write the following code to assign default input field attributes:

```
module STATUS_0200 output.
```

```
* SET PF-STATUS 'xxxxxxxx'.
* SET TITLEBAR 'xxx'.
```

```
IF A = 0.
MESSAGE S010(ZMSG).
LOOP AT SCREEN.
IF SCREEN-GROUP1 = 'GR1'.
SCREEN-REQUIRED = '1'.
ENDIF.
IF SCREEN-GROUP1 = 'GR2'.
SCREEN-INVISIBLE = '1'.
ENDIF.
MODIFY SCREEN.
ENDLOOP.
A = 1.
ENDIF.
endmodule.          " STATUS_0200 OUTPUT
```

Save -> Activate.

Create a Transaction Code -> Execute the program.

### **MPP SCREEN VALIDATIONS:**

Eg. code to make field validations in MPP program:

Using screen painter, design a screen consisting of four input fields for client, username, password and language as we have in login screen of SAP.

Assign the first two input fields to one group called GR1, the third input field to a group GR2.

Create two pushbuttons and assign FCT codes.

In the TOP INCLUDE FILE, declare following variables:

```
PROGRAM SAPMYSCREENVALID.
DATA : IO1(3), IO2(8), IO3(8), IO4(2).
DATA A TYPE I.
```

Save -> Activate.

In Flow logic editor, decomment PAI MODULE, double click on module name, and inside the module, write the following code:

```
module USER_COMMAND_0200 input.
```

```
case sy-ucomm.
WHEN 'LOGIN'.
CALL TRANSACTION 'SE38'.
WHEN 'EXIT'.
LEAVE PROGRAM.
ENDCASE.
```

```
endmodule.          " USER_COMMAND_0200 INPUT
```

Save -> Activate.

In PBO module, write the following code to assign default input field attributes:

```
module STATUS_0200 output.
* SET PF-STATUS 'xxxxxxxx'.
* SET TITLEBAR 'xxx'.
```

```
IF A = 0.
MESSAGE S010(ZMSG).
LOOP AT SCREEN.
IF SCREEN-GROUP1 = 'GR1'.
SCREEN-REQUIRED = '1'.
ENDIF.
IF SCREEN-GROUP1 = 'GR2'.
SCREEN-INVISIBLE = '1'.
ENDIF.
MODIFY SCREEN.
ENDLOOP.
A = 1.
ENDIF.
endmodule.          " STATUS_0200 OUTPUT
```

Save -> Activate.

Create a Transaction Code -> Execute the program.

## Menu Painter

### **ADDING USER-DEFIND MENUS TO THE INTERACTIVE REPORTS:**

Menu Painter is a tool used to create user-defined menus, application toolbar and function keys.

SET PF-STATUS '<menu\_name>' is the syntax used in reports to access Menu Painter directly to create user-defined menus.

A menu bar consists of set of menus.  
Menu contains set of menu items.  
Each menu item may have its own sub menu items.

Another way of accessing Menu Painter is using SE41 Tcode.

Eg. code:

```
WRITE :/ 'SELECT ONE FROM THE MENU'.
```

```
SET PF-STATUS 'MYMENU'.
```

```
AT USER-COMMAND.
```

```
CASE SY-UCOMM.
```

```
WHEN ' SAMPATH 1'.  
MESSAGE S000(ZSHABMSG).
```

```
WHEN ' SAMPATH 2'.  
MESSAGE S001(ZSHABMSG).
```

```
WHEN 'SAMPATH3'.  
LEAVE PROGRAM.
```

```
WHEN 'SUBMENU11'.  
CALL TRANSACTION 'SE38'.
```

```
WHEN 'SUBMENU12'.  
CALL TRANSACTION 'SE37'.
```

```
ENDCASE.
```

-> Save -> Activate -> Execute.

## **INSERTING RECORDS INTO DATABASE TABLE USING MPP SCREEN FIELDS:**

Using Dictionary/Program Fields (F6) pushbutton from the application toolbar of Graphical Screen Painter, we can create input fields for the database table field structures.

### **Navigations:**

1. Create MPP program.
2. In Top Include File, declare the following structure:

```
DATA SHABANA LIKE KNA1.
```

Save -> Activate.

3. Create Screen -> Click on 'Dictionary/Program Fields (F6)' pushbutton from the application toolbar -> Opens an interface -> Specify structure name (SHABANA) -> Click on Get From Program pushbutton -> Opens interface -> Select required fields -> Place on Screen painter -> specify labels -> Create two pushbuttons (INSERT, EXIT) -> Save -> Flowlogic.

4. In PAI module, specify following code:

```
CASE SY-UCOMM.
```

```
  WHEN 'INSERT'.
    INSERT INTO KNA1 VALUES SHABANA.
```

```
  IF SY-SUBRC = 0.
    MESSAGE S003(ZSHABMSG).
  ELSEIF SY-SUBRC = 4.
    MESSAGE E004(ZSHABMSG).
  ENDIF.
```

```
  WHEN 'EXIT'.
    LEAVE PROGRAM.
```

```
ENDCASE.
```

5. Create a Tcode -> Activate all -> Execute.

## **TABLE CONTROL COMPONENTS:**

Table Control component is used to view the table records and if needed, we can directly modify table records and update the database table using table control.

Here, the records can be viewed in rows and columns format separated by horizontal and vertical lines.

### **SYNTAX:**

***CONTROLS <table\_Control\_name> TYPE TABLEVIEW USING SCREEN <MPP\_screen\_number>***

CONTROLS statement is used to create a memory space area for table control component in AS.

TABLEVIEW is a data type for table control component.

SCREEN NUMBER should be specified to make the system know where the table control was physically created.

Navigations to create TABLE CONTROL COMPONENT:

Create MPP program -> In TOP INCLUDE FILE, write the following code:

```
DATA ITAB LIKE KNA1 OCCURS 0 WITH HEADER LINE.
CONTROLS TABCTRL TYPE TABLEVIEW USING SCREEN '123'.
DATA CUR TYPE I.
```

Save -> Activate.

Create a Normal screen (123) -> Drag and drop TABLE CONTROL component from application toolbar -> Specify its name in attributes box -> Specify title if necessary -> Select HORIZONTAL and VERTICAL SEPARATORS checkbox -> If needed, select COLUMN and ROW selection radiobuttons -> Click on Dictionary/Program Fields from Appn. Toolbar -> Specify internal table name specified in top include file -> Click on 'GET FROM PROGRAM' pushbutton -> Choose required fields -> Click on continue -> Place the fields in table control component -> Add labels for each fields -> Create two pushbuttons (FETCH, EXIT) -> Save -> Flow Logic.

In PAI module, write following code:

```

CASE SY-UCOMM.

WHEN 'FETCH'.
SELECT * FROM KNA1 INTO TABLE ITAB.
TABCTRL-LINES = SY-DBCNT.

WHEN 'EXIT'.
LEAVE PROGRAM.

ENDCASE.

```

In Flow Logic editor, write following code:

```

PROCESS BEFORE OUTPUT.
MODULE STATUS_0123.
LOOP AT ITAB CURSOR CUR WITH CONTROL TABCTRL.
ENDLOOP.

```

```

PROCESS AFTER INPUT.
MODULE USER_COMMAND_0123.
LOOP AT ITAB.
ENDLOOP.

```

Here, LOOP AT ITAB-ENDLOOP statement in PBO event is used to fetch the records and insert into table control component. CURSOR statement is used to make use of the cursor in table control component whenever we try to select a particular field and modify it.

LOOP AT ITAB-ENDLOOP statement in PAI event is used to make necessary modifications to the database table from table control component.

Create Tcode -> Execute.

### **TABSTRIP CONTROLS:**

Using normal screen, we can add only 40 components to the screen. To add more than 40 components, make use of tabstrip control. You can specify any number of tab fields for a tabstrip control and create subscreen for each tab field created.

### **Navigations to create tabstrip control:**

Create an MPP program -> Create a screen -> Drag and drop tabstrip control from toolbar -> Specify name for the tabstrip created (KARTHIK) -> Specify required number of tab fields (2) -> Drag and drop subscreen area for each tab field -> Name the subscreen areas (SUB1, SUB2) -> Specify attributes for each tab field (NAME, TEXT, FCTCODE, REF\_FIELD) -> Create two pushbuttons (DISPLAY, EXIT) Save the screen painter -> Click on Flow logic editor.

Now create two subscreens (10, 20) for each tab field subscreen areas. Create required screen components for each subscreen (input fields namely IO1, IO2, IO3, IO4) -> Save -> Come back to Flow logic editor.

In TOP INCLUDE FILE, specify following code:

```
DATA : IO1(10), IO2(10), IO3(10), IO4(10).
CONTROLS KARTHIK TYPE TABSTRIP.
DATA SCREEN LIKE SY-DYNNR .
```

CONTROLS statement is used to create a memory for tabstrip component in AS.

SY-DYNNR is a system variable to hold screen number.

Save -> Activate.

In the FLOW LOGIC EDITOR, Specify following code for PBO and PAI modules:

```
PROCESS BEFORE OUTPUT.
MODULE STATUS_0100.
  CALL SUBSCREEN SUB2 INCLUDING 'SAPMYONCEMORE' '20'.
  CALL SUBSCREEN SUB1 INCLUDING 'SAPMYONCEMORE' '10'.
```

```
PROCESS AFTER INPUT.
MODULE USER_COMMAND_0100.
  CALL SUBSCREEN SUB1.
  CALL SUBSCREEN SUB2.
```

Save -> Activate.

Specify following code in PAI event between module-endmodule statements:

```

CASE SY-UCOMM.

WHEN 'DISPLAY'.
LEAVE TO LIST-PROCESSING.
WRITE :/ IO1, IO2, IO3, IO4, IO5.

WHEN 'EXIT'.
LEAVE PROGRAM.

WHEN 'TAB1'.
SCREEN = '10'.
KARTHIK-ACTIVETAB = 'TAB1'.

WHEN 'TAB2'.
SCREEN = '20'.
KARTHIK-ACTIVETAB = 'TAB2'.

ENDCASE.

```

Create Tcode -> Activate all components -> Execute the program.

### **LIST OF VALUES:**

This concept is used to provide drop-down facility for the input fields created using screen painter. Here, a type group called VRM is used in which we have following structure and internal table:

**VRM\_VALUE** is a structure with following fields,

KEY - Code for the display value  
 TEXT - Content of the display value

VRM\_VALUES is an internal table created for the above structure without header line.

Declare an internal table (ITAB) of type VRM\_VALUES and also declare an explicit structure (WA) to append values into internal table using LIKE LINE OF statement.

VRM\_SET\_VALUES is the function module used to fetch records from internal table ITAB to input field IO1.

**TABLE CONTROL:**

This component is used to view the internal table records in MPP screen.

Table control modification:

LOOP AT ITAB-ENDLOOP statement in PBO is used to fetch the records into internal table from db table.

LOOP AT ITAB-ENDLOOP statement in PAI is used to view the current data in internal table through table control.

To make modification, create one more internal table for the same dbtable structure used in top include file. To move first internal table records to the newly created one, create a separate module to specify a statement for this purpose b/w LOOP-ENDLOOP in PAI.

eg.

```
PROCESS AFTER INPUT.
  MODULE USER_COMMAND_0800.
  LOOP AT ITAB.
    MODULE ITABTOITAB1. * New module
  ENDLOOP.
```

Double click on module name to specify following move statement.

eg.

```
MODULE ITABTOITAB1 INPUT.
  APPEND ITAB TO ITAB1.
ENDMODULE.
```

In Screen painter, create 'MODIFY' pushbutton and specify event functionality in PAI as follows:

```
WHEN 'MODIFY'.
```

```
  LOOP AT ITAB1.
  MODIFY KNA1 FROM ITAB1.
```

```
  IF SY-SUBRC = 0.
  MESSAGE S006(ZSHABMSG).
  ELSEIF SY-SUBRC = 4.
  MESSAGE E004(ZSHABMSG).
```

ENDIF.

ENDLOOP.

SELECT \* FROM KNA1 INTO TABLE ITAB.  
TBCL-LINES = SY-DBCNT.

ENDCASE.

## BATCH DATA COMMUNICATION

To perform data migration from legacy database to SAP database, we use batch data communications (BDC).

To perform data migration, following methods are available in BDC:

- 1. DIRECT INPUT METHOD**
- 2. CALL TRANSACTION METHOD**
- 3. SESSION METHOD.**

The above three methods require source code to perform data migration.

There are predefined tools available to perform data migration with only minimal source code. They are:

- 1. LEGACY SYSTEM MIGRATION WORKBENCH (LSMW).**
- 2. RECORDING METHOD**

### **DIRECT INPUT METHOD:**

#### **Advantages:**

- Using this method, we can perform bulk data transfer.
- No manual intervention is required for direct input method.
- Data migration time is very less.

#### **Disadvantages:**

- Since no manual intervention is required, we cannot correct the error record during runtime.
- This method can be used only during SAP implementation, not for support projects.

Eg. code:

```
DATA : BEGIN OF ITAB OCCURS 0,
      STR(255),
      END OF ITAB.
```

```
DATA ITAB1 LIKE KNA1 OCCURS 0 WITH HEADER LINE.
```

```
CALL FUNCTION 'UPLOAD'
  EXPORTING
    FILENAME           = 'C:\DIR.TXT'
```

```

FILETYPE          = 'ASC'
TABLES
  DATA_TAB       = ITAB.

```

```

LOOP AT ITAB.
SPLIT ITAB-STR AT ' ' INTO ITAB1-KUNNR ITAB1-NAME1 ITAB1-ORT01
ITAB1-LAND1.
APPEND ITAB1.
ENDLOOP.

```

```

LOOP AT ITAB1.
INSERT INTO KNA1 VALUES ITAB1.

```

```

IF SY-SUBRC = 0.
WRITE :/ 'RECORDS ARE INSERTED INTO KNA1'.
ELSEIF SY-SUBRC = 4.
WRITE :/ 'RECORDS ALREADY EXISTS'.
ENDIF.

```

```

ENDLOOP.

```

### **BATCH DATA COMMUNICATIONS:**

This concept deals with data migration from legacy system database into SAP database. Whenever a company moves into SAP from legacy system, BDC is used to populate the new SAP database with their old required records.

### **Methods in BDC:**

1. **DIRECT INPUT METHOD** - Using this method, records from the flat file are uploaded first into an internal table created for the flat file structure. Using a string function (SPLIT functionality), the records are splitted based on the separators and then inserted into a new internal table which is created for a table where the datas are to be inserted.

eg. code:

```

DATA : BEGIN OF ITAB OCCURS 0,
      STR(255),
      END OF ITAB.

```

```

DATA ITAB1 LIKE KNA1 OCCURS 0 WITH HEADER LINE.

```

```
CALL FUNCTION 'UPLOAD'
EXPORTING
  FILENAME           = 'C:\BASK.TXT'
  FILETYPE           = 'ASC'
  TABLES
    data_tab         = ITAB.
```

```
LOOP AT ITAB.
SPLIT ITAB-STR AT ' ' INTO ITAB1-KUNNR ITAB1-NAME1 ITAB1-LAND1
ITAB1-ORT01.
APPEND ITAB1.
ENDLOOP.
```

```
LOOP AT ITAB1.
INSERT INTO KNA1 VALUES ITAB1.
IF SY-SUBRC = 0.
WRITE :/ 'RECORDS ARE INSERTED'.
ELSEIF SY-SUBRC = 4.
WRITE :/ 'RECORDS ALREADY EXIST'.
ENDIF.
ENDLOOP.
```

In the above program, 'UPLOAD' function module is used to fetch the flat file records and then insert into internal table created for flat file structure (ITAB).

## **2. CALL TRANSACTION METHOD.**

CODE:

```
DATA : BEGIN OF ITAB OCCURS 0,
       STR(255),
       END OF ITAB.
```

```
CALL FUNCTION 'UPLOAD'
EXPORTING
  FILENAME           = 'C:\STAL.TXT'
  FILETYPE           = 'ASC'
  TABLES
    data_tab         = ITAB.
```

```
DATA ITAB1 LIKE KNA1 OCCURS 0 WITH HEADER LINE.
```

```
LOOP AT ITAB.  
SPLIT ITAB-STR AT ',' INTO ITAB1-KUNNR ITAB1-NAME1 ITAB1-ORT01  
ITAB1-LAND1.  
APPEND ITAB1.  
ENDLOOP.
```

```
DATA JTAB LIKE BDCDATA OCCURS 0 WITH HEADER LINE.
```

```
DATA KTAB LIKE BDCMSGCOLL OCCURS 0 WITH HEADER LINE.
```

```
LOOP AT ITAB1.
```

```
PERFORM PROGINFO USING 'SAPMZSTALINCT' '100'.
```

```
PERFORM FLDINFO USING 'WA-KUNNR' ITAB1-KUNNR.  
PERFORM FLDINFO USING 'WA-NAME1' ITAB1-NAME1.  
PERFORM FLDINFO USING 'WA-LAND1' ITAB1-LAND1.  
PERFORM FLDINFO USING 'WA-ORT01' ITAB1-ORT01.
```

```
CALL TRANSACTION 'ZSTALINCT' USING JTAB MODE 'N' MESSAGES INTO  
KTAB.
```

```
ENDLOOP.
```

```
LOOP AT KTAB.  
WRITE :/ KTAB-TCODE, KTAB-DYNAME, KTAB-DYNUMB, KTAB-MSGTYP,  
KTAB-MSGNR, SY-SUBRC.  
ENDLOOP.
```

```
FORM PROGINFO USING PROGNAME SCRNUM.
```

```
CLEAR JTAB.  
REFRESH JTAB.
```

```
JTAB-PROGRAM = PROGNAME.  
JTAB-DYNPRO = SCRNUM.  
JTAB-DYNBEGIN = 'X'.
```

```
APPEND JTAB.
```

```
ENDFORM.
```

FORM FLDINFO USING FLDNAME FLDVAL.

CLEAR JTAB.

JTAB-FNAM = FLDNAME.

JTAB-FVAL = FLDVAL.

APPEND JTAB.

3. BDC SESSION METHOD.

### **TOOLS USED IN BDC:**

LEGACY SYSTEM MIGRATION WORKBENCH (LSMW)

BDC RECORDING METHOD

### **BDC CALL TRANSACTION METHOD:**

Since we cannot modify the error record using direct input method, we go for call transaction method. Here, we create a screen to populate error records in input fields and from the screen, we can modify the error records and then insert into database table.

### **Steps to be followed in Call Transaction Method:**

1. Analyze the flat file.
2. Create a screen for database table fields using MPP. Create a Transaction code for the screen.
3. Write the following code in SE38 editor:

```
DATA : BEGIN OF ITAB OCCURS 0,
      STR(255),
      END OF ITAB.
```

```
CALL FUNCTION 'UPLOAD'
  EXPORTING
    FILENAME           = 'C:\KNA.TXT'
    FILETYPE           = 'ASC'
    TABLES
      DATA_TAB        = ITAB.
```

```
DATA ITAB1 LIKE KNA1 OCCURS 0 WITH HEADER LINE.
```

DATA JTAB LIKE BDCDATA OCCURS 0 WITH HEADER LINE.

LOOP AT ITAB.

SPLIT ITAB-STR AT ',' INTO ITAB1-KUNNR ITAB1-NAME1 ITAB1-ORT01  
ITAB1-LAND1.  
APPEND ITAB1.

ENDLOOP.

LOOP AT ITAB1.

PERFORM PROGINFO USING 'SAPMZCALLTRANSACTION' '100'.

PERFORM FLDINFO USING 'KARTHIK-KUNNR' ITAB1-KUNNR.  
PERFORM FLDINFO USING 'KARTHIK-NAME1' ITAB1-NAME1.  
PERFORM FLDINFO USING 'KARTHIK-ORT01' ITAB1-ORT01.  
PERFORM FLDINFO USING 'KARTHIK-LAND1' ITAB1-LAND1.

CALL TRANSACTION 'ZCALLTRANS' USING JTAB.

ENDLOOP.

FORM PROGINFO USING PROGNAME SCRNUM.

CLEAR JTAB.  
REFRESH JTAB.

JTAB-PROGRAM = PROGNAME.  
JTAB-DYNPRO = SCRNUM.  
JTAB-DYNBEGIN = 'X'.

APPEND JTAB.

ENDFORM.

FORM FLDINFO USING FLDNAME FLDVALUE.

CLEAR JTAB.

JTAB-FNAM = FLDNAME.  
JTAB-FVAL = FLDVALUE.

APPEND JTAB.

ENDFORM.

Save -> Activate -> Execute.

In the above code,

**BDCDATA** is a structure used to populate the internal table records into the screen fields. The BDCDATA structure has following components:

**PROGRAM** - Holds the name of MPP program where the screen is created.

**DYNPRO** - Holds the screen number where the internal fields to be populated.

**DYNBEGIN** - Used to initiate the screen when the program is executed. The default value to be specified is 'X'.

**FNAM** - Specifies input field name in the screen where the data is to be populated.

**FVAL** - Specifies from which internal table field, the data should be passed to the screen field.

'**CALL TRANSACTION**' statement is used to call the screen created to populate error records.

SYNTAX:

***CALL TRANSACTION <Tcode> USING <BDCDATA\_itab> MODE <mode> UPDATE <update>.***

MODE: This is used to specify which mode to be followed when calling transaction. The types of mode are:

A - Display the screen.

E - Display only error records from the flat file.

N - Background processing.

UPDATE: This is used to specify the update task of records in the database table. The types of update tasks are:

A - Asynchronous update

S - Synchronous update

L - Local update

**Advantages of CALL TRANSACTION:**

\* Error records can be modified.

\* This method can be used in support projects.

**BDC CALL TRANSACTION:**

```
DATA : BEGIN OF ITAB OCCURS 0,  
      STR(255),  
      END OF ITAB.
```

```
DATA ITAB1 LIKE KNA1 OCCURS 0 WITH HEADER LINE.
```

```
DATA JTAB LIKE BDCDATA OCCURS 0 WITH HEADER LINE.
```

```
CALL FUNCTION 'UPLOAD'  
EXPORTING  
  FILENAME           = 'C:\KARTHIK.TXT'  
  FILETYPE           = 'ASC'  
TABLES  
  DATA_TAB          = ITAB.
```

```
LOOP AT ITAB.  
SPLIT ITAB-STR AT ',' INTO ITAB1-KUNNR ITAB1-NAME1 ITAB1-ORT01  
ITAB1-LAND1.  
APPEND ITAB1.  
ENDLOOP.
```

```
LOOP AT ITAB1.  
PERFORM PROGINFO USING 'SAPMYCALLTRANSACTION' '400'.
```

```
PERFORM FLDINFO USING 'WA-KUNNR' ITAB1-KUNNR.  
PERFORM FLDINFO USING 'WA-NAME1' ITAB1-NAME1.  
PERFORM FLDINFO USING 'WA-ORT01' ITAB1-ORT01.  
PERFORM FLDINFO USING 'WA-LAND1' ITAB1-LAND1.
```

```
CALL TRANSACTION 'YCALLTRANS' USING JTAB.
```

```
ENDLOOP.
```

```
FORM PROGINFO USING PROGNAME SCRNUM.  
CLEAR JTAB.  
REFRESH JTAB.  
JTAB-PROGRAM = PROGNAME.  
JTAB-DYNPRO = SCRNUM.  
JTAB-DYNBEGIN = 'X'.  
APPEND JTAB.
```

ENDFORM.

FORM FLDINFO USING FLDNAME FLDVALUE.  
 CLEAR JTAB.  
 JTAB-FNAM = FLDNAME.  
 JTAB-FVAL = FLDVALUE.  
 APPEND JTAB.

ENDFORM.

**CLEAR** statement is used to delete the contents of the header line of internal table.

**REFRESH** statement is used to delete the contents of the body area of internal table.

**FREE** statement is used to delete the internal table after the program is closed.

**DELETE** statement is used to delete some particular contents of the internal table.

#### SYNTAX:

```
CALL TRANSACTION <Tcode> USING <bdcdata_itab> MODE <mode>
UPDATE <task>.
```

Tcode is the name of the transaction code which contains screen to display error records.

bdcdata\_itab is an internal table which is created for BDCDATA structure. mode specifies the type of mode for processing. There are three types as follows:

- A - Foreground processing
- E - Errors only
- N - Background processing

When 'N' is used, a predefined structure called BDCMSGCOLL should be used to collect the messages triggered during background processing.

#### Syntax:

```
DATA KTAB LIKE BDCMSGCOLL OCCURS 0 WITH HEADER LINE.
```

```
CALL TRANSACTION 'YCALLTRANS' USING JTAB MODE 'N' MESSAGES INTO
KTAB.
```

Update specifies the task of updating records into database table.

A - Asynchronous update - A return value is generated only after updating all the records in related tables.

S - Synchronous update - A return value is generated for each record updation in related tables inside the database.

### **BDC SESSION METHOD:**

Since Direct Input and Call Transaction methods cannot be used for support projects due to the database availability and networking constraints, SAP suggests to use Session Method for data migration in support projects.

In this method, a session is created in the Application Server. A session has a session memory used to hold the internal table records. We can process the session later whenever database is available with free network traffic.

### **FUNCTION MODULES USED IN BDC SESSION METHOD:**

1. BDC\_OPEN\_GROUP - This FM is used to create a session in Appn. Server.
2. BDC\_INSERT - This FM is used to insert the internal table records into session memory.
3. BDC\_CLOSE\_GROUP - This FM is used to save the records in session memory and close it to process later.

Eg. code:

```
DATA : BEGIN OF ITAB OCCURS 0,
      STR(255),
      END OF ITAB.
```

```
DATA ITAB1 LIKE KNA1 OCCURS 0 WITH HEADER LINE.
```

```
DATA JTAB LIKE BDCDATA OCCURS 0 WITH HEADER LINE.
```

```
CALL FUNCTION 'UPLOAD'
  EXPORTING
    FILENAME           = 'C:\KARTHIK.TXT'
    FILETYPE           = 'ASC'
  TABLES
    DATA_TAB          = ITAB.
```

```
LOOP AT ITAB.
```

```
SPLIT ITAB-STR AT ',' INTO ITAB1-KUNNR ITAB1-NAME1 ITAB1-ORT01
```

```
ITAB1-LAND1.  
APPEND ITAB1.
```

```
ENDLOOP.
```

```
CALL FUNCTION 'BDC_OPEN_GROUP'  
EXPORTING  
  CLIENT          = SY-MANDT  
  GROUP          = 'SHABANA'  
  KEEP           = 'X'  
  USER           = SY-UNAME.
```

```
LOOP AT ITAB1.
```

```
PERFORM PROGINFO USING 'SAPMYCALLTRANSACTION' '400'.
```

```
PERFORM FLDINFO USING 'WA-KUNNR' ITAB1-KUNNR.  
PERFORM FLDINFO USING 'WA-NAME1' ITAB1-NAME1.  
PERFORM FLDINFO USING 'WA-ORT01' ITAB1-ORT01.  
PERFORM FLDINFO USING 'WA-LAND1' ITAB1-LAND1.
```

```
CALL FUNCTION 'BDC_INSERT'  
EXPORTING  
  TCODE          = 'YCALLTRANS'  
  TABLES  
  DYNPROTAB      = JTAB.
```

```
ENDLOOP.
```

```
CALL FUNCTION 'BDC_CLOSE_GROUP'.
```

```
FORM PROGINFO USING PROGNAME SCRNUM.
```

```
CLEAR JTAB.  
REFRESH JTAB.
```

```
JTAB-PROGRAM = PROGNAME.  
JTAB-DYNPRO = SCRNUM.  
JTAB-DYNBEGIN = 'X'.  
APPEND JTAB.
```

```
ENDFORM.
```

```
FORM FLDINFO USING FLDNAME FLDVALUE.
```

CLEAR JTAB.

JTAB-FNAM = FLDNAME.

JTAB-FVAL = FLDVALUE.

APPEND JTAB.

ENDFORM.

Save -> Activate -> Execute -> A session is created.

To process the session, GOTO SM35 Tcode.

### **Navigations to process the session:**

SM35 -> Select the session -> click on PROCESS pushbutton from appn. toolbar -> Opens an interface -> Select DISPLAY ERRORS ONLY radiobutton -> Process -> Opens the user-defined screen -> Modify and Insert the records -> Finally opens a message -> Click on Session Overview pushbutton -> Comes back to SM35 screen -> Select the session -> Click on Log Pushbutton from appn. toolbar -> To view the detailed log, select session from here -> Click Analyze Log pushbutton from appn. toolbar -> Opens an interface -> Click on Log Created on...date... tab button -> We can view the detailed log of transaction.

If Background mode is selected, GOTO SM36 Tcode.

SM36 -> Click on Own Jobs pushbutton from appn. toolbar -> Select session from the list -> Click on Job Log pushbutton from appn. toolbar -> Gives you detailed log of background processing.

## **LSMW ( Legacy System Migration Workbench)**

Is a 14 step Navigation to Perform BDC, here ABAPer is not writing a single Code to transfer Flat file data.

LSMW is a collection of Projects.

Project is a collection of Sub Projects.

Sub project is a collection of Objects, here Object represents transactions.

LSMW -> Name the Project ( Z730LSMW ) -> Name the Sub Project ( MM ) -> Name the Object ( MM01 ) -> Click on Create From application toolbar -> Opens an interface -> Enter the Description for project , sub project and Object -> Click on Execute from Application toolbar .

Opens an interface with list of LSMW navigations.

### **Step #1. Maintain Object Attributes**

In this step we need to specify a recording object to be assigned for LSMW to process the BDC data.

select the radiobutton -> Click on Execute from Application toolbar -> Opens Another Interface -> Click on Pushbutton Recording Overviews , to create a new recording Object -> Opens LPS -> Select the data from screen -> Click on Create Recording from application toolbar -> Name the recording Object ( zrec1 ) -> Enter description ( any ) -> Click on continue -> Specify Tcode ( mm01 ) -> Accept the SAMPLE data -> Save the Record -> by default sap displays list of fields information where data is accepted in the form "JTAB" Internal table data -> Move this fields into Recording Object fields by clicking on "Default all" pushbutton from application toolbar -> save the Entries -> Come back -> come back -> Change mode of the screen into Change mode -> Select the radiobutton called as Batch Input Recording -> Name the recording Object ( ZREC1 ) -> Save the Entries -> Come back .

### **Step #2 Maintain Source Structures**

In this step we have to define Name of the Internal table where data to be populated from Flat files.

Select -> Execute from Application -> Opens an interface -> Change the mode into Change mode -> Click on Create Structure From Application toolbar -> Name the Internal table ( ITAB1 ) -> Enter Description ( Database format internal table ) -> Click on Continue -> save the Entries -> Come back.

### **Step #3. Maintain Source Fields**

In this step we are adding fields to the Internal table created in Second step.

Select the radiobutton -> Click on Execute -> Opens an interface -> Change Mode of the screen -> select the Internal table from the List -> Click on Create field from application toolbar -> Name the Field ( MATNR ) -> Enter Description (any) -> Set the Size ( 18 ) -> Specify Data type ( C ) -> Click on Continue.

Using the same step add Other Fields

MBRSH	( Industry )	1	(C)
MTART	(Mat type)	4	(C)
MAKTX	(Mat Desc)	40	(C)
MEINS	(Mat measure)	3	(C)

-> save the Entries -> Come back.

#### **Step #4.** Maintain Structure Relations

In this step we need to Generate Relation between ITAB1 and Recording Object.

select -> Execute -> Opens an interface -> Relation ship , by default generated by SAP -> Change mode of the screen -> save the Entries -> Come back.

#### **step #5.** Maintain Field Mapping and Conversion Rules

Mapping the Fields of Itab1 with Recording Object is the step to be processed.

select -> Execute -> Change Mode of Screen -> Select the Field From List of recording object -> Click on Source Field From Application toolbar -> Displays the List of Internal table fields -> Double click on Proper Field. In the same way map all Fields of Recording Object with All Fields of Internal table -> Save the Entries -> Come back.

#### **Step #6.** Maintain Fixed Values, Translations, User-Defined Routines

This is optional Step In BDC of LSMW

Execute -> Come back.

#### **Step #7.** Specify Files

Select -> Execute -> Opens an interface -> Change Mode of Screen -> select legacy data on PC -> Click on Create From Application toolbar -> Name the File ( D:\730AM\matdata.txt ) -> Enter Description ( any ) ->

Specify Separator as "Comma" -> Click on Continue -> Save the Entries -> Come back.

### **Step #8.** Assign Files

In this step we need to assign the Flat file Into Internal table created in Second step.

select -> Execute -> Change Mode of the screen -> Save the Entries -> Come back.

### **Step #9.** Import Data

In this step SAP Reads the Flat File data and stores in the form of ".lsmw.read" file.

Select -> Execute -> Opens an interface -> Click on Execute -> Sap Generates a report as Flat file is converted into ".read" file with Number of transactions -> Come back -> Come back.

### **Step #10.** Display Imported Data

Select ->Execute -> Open an Interface -> Click on Continue -> displays the data in Internal table (itab1) -> Come back.

### **Step #11.** Convert Data

In this step Data will be converted into ".lsmw.conv" file to make data is ready for populating into Session object.

select -> execute -> Execute -> Sap Generates a report as file is converted into Conv -> Come back -> come back.

### **Step #12.** Display Converted Data

select -> Execute ->Opens an interface -> Click on continue-> Generates a report with Green Color -> Come back.

### **Step #13.** Create Batch Input Session

Select ->Execute -> Opens an interface -> Select the checkbox -> Click on Execute -> Prompts a message.

### **Step #14.** Run Batch Input Session

Opens SM35 tcode to process BDC data.

select -> Execute -> Select session object name -> Click on Process -> Click on continue -> SAP transfers all file data into R/3 applications.

### **RECORDING METHOD: (SHDB).**

BDC recording method is a SAP tool used to perform data migration. Using this method, a recording object is created for any one of the predefined screens (MM01, XK01, and XD01) the sample values entered in the screen input fields are taken to create a BDCDATA structure. Based on the structure created, the source code is automatically generated by the system.

In this method we cannot use user-defined screens to populate the error records.

### **Navigations to perform data migration using recording method.**

SHDB-> click on new recording push button from the application tool bar-> opens an interface->name the recording object->specify TCODE( mm01) for which the recording is done(mm01) ->continue->opens specified screen (mm01) ->enter sample values [ Material no, Material type, industry sector(matnr,mtart,mbrsh) ->click on select view pushbutton from the application tool bar->select basic data1 from interface->continue->opens second screen-> enter description(maktx) and measurement(meins) -> click on save -> opens an interface with BDCDATA structure filled with the sample values -> save the recording -> comeback -> select the recording object -> click on program pushbutton from the application tool bar -> opens an interface -> enter the program name -> select transfer from recording radio button -> continue -> enter a short description -> click on source code push button -> opens SE38 editor with auto generated code.

After the include statement, specify the following code.

\* Include bdcrcx1.(include stmt).

```
Data : begin of itab occurs 0,
      Str(255),
      End of itab.
```

Data itab1 like mara occurs 0 with header line.

After the start of selection event, specify the following code.

Start-of-selection.

Call function 'upload',

Exporting  
File name = 'c:\mara.txt'  
File type = 'asc'  
Tables  
Data\_tab = itab.

Loop at itab.

Split itab at ',' into itab1-matnr itab1-mtart itab1-mbrsh itab1-meins.  
Append itab1.

Endloop.

After the perform open-group

Loop at itab1.

Replace all the sample values for the fields with the internal table name.

itab1-matnr  
itab1-mtart  
itab1-mbrsh  
itab1-meins  
endloop.

Perform close\_group.

Save it -> activate -> execute.

Using recording method, we can perform data migration in the following 2 ways.

1. Session method.
2. Call transaction method.

When session method is selected, specify session name -> checkbox,keep batch input session -> execute.

To process the session, go to sm35 and process.

## SAPSCRIPTS

This is a tool used to redirect SAP data to output devices. SE71 is the Tcode to create SAPScript.

Components of a SAPScript tool are:

### 1. BASIC SETTINGS.

Paragraph format, character format.

### 2. ADMINISTRATIVE SETTINGS.

Name of the form, short description.

Layout is used to create a form in SAPScript. Layout is a collection of pages. Page is a collection of Windows.

#### **Types of Windows:**

1. Main Window - This is a common window for all pages. This is a default window.
2. Constant Window - This window is used to create footer space, header space for a particular page.
3. Variable Window - This is a subwindow.
4. Graphical Window - This is an optional window, which is used to create logos or some other graphics for the page.

#### **NAVIGATIONS FOR CREATING A SAPSCRIPT:**

SE71 -> Specify Form name starting with Z or Y (ZSHABFORM) -> Click on Create -> Opens an interface -> Enter short description -> Click on 'Paragraph Format' from Appn. toolbar -> Specify Paragraph Name (P1)-> Press Enter -> Enter short description -> Click on 'Definitions' pushbutton from application toolbar -> Specify Default Paragraph (P1) created -> Click on Layout pushbutton from appn. toolbar -> Opens a layout with a default window 'MAIN' -> Right click on Main Window -> Select 'Edit Text' -> Opens a Line Editor -> Specify a statement -> Come back -> Save -> Activate the form -> A SAPscript is created.

To invoke the form created, we have to create a print program. Create an Executable Program and specify the following:

```
CALL FUNCTION 'OPEN_FORM'
  EXPORTING
    FORM           = 'ZSHABFORM'
    LANGUAGE       = SY-LANGU.
```

```
CALL FUNCTION 'WRITE_FORM'
  EXPORTING
    ELEMENT        = 'ELEMENT'
    WINDOW         = 'MAIN'.
```

```
CALL FUNCTION 'CLOSE_FORM'.
```

-> Save -> Activate -> Execute -> Opens an interface -> Specify output device as LP01 -> Click on Print Preview (F8) pushbutton -> Executes the form.

The FM 'OPEN\_FORM' is used to call the sapscrip form. Here, we have to specify the name of the form as an argument.

'WRITE\_FORM' is used to specify the name of the text elements and window types.

'CLOSE\_FORM' is used to save and close the form attributes.

The function modules OPEN\_FORM and CLOSE\_FORM are mandatory ones.

### **PASSING ARGUMENTS TO THE FORM:**

In Line editor, specify an argument enclosed by an ampersand symbol (&).

eg. &KARTHIK&.

Save -> Activate the form.

To pass a value from the print program to the form, declare the variable as follows in Print PProgram:

```
DATA KARTHIK(10) VALUE 'CHENNAI'.
```

```
....OPEN_FORM
...
....CLOSE_FORM
```

Save -> Activate -> Execute.

**PASSING TABLE VALUES AS AN ARGUMENT TO SAPSCRIPT:**

In the line editor, specify the table field arguments enclosed by '&' symbol as follows:

```
/E ELEMENT
      &KNA1-KUNNR& ,, &KNA1-NAME1& ,, &KNA1-LAND1&
```

Save -> Activate.

In the Print Program, specify following code:

```
TABLES KNA1.

CALL FUNCTION 'OPEN_FORM'
  EXPORTING
    FORM           = 'ZSHABFORM1'
    LANGUAGE       = SY-LANGU.

SELECT * FROM KNA1.
CALL FUNCTION 'WRITE_FORM'
  EXPORTING
    ELEMENT        = 'ELEMENT'
    WINDOW         = 'MAIN'.
ENDSELECT.

CALL FUNCTION 'CLOSE_FORM'.
```

Save -> Activate -> Execute.

**PASSING INTERNAL TABLE AS AN ARGUMENT TO THE FORM:**

In line editor, specify following arguments:

```
/E ELEMENT
      &ITAB-KUNNR&      &ITAB-NAME1&      &ITAB-LAND1&
```

Save -> Activate.

In Print Program, specify following code:

```
DATA ITAB LIKE KNA1 OCCURS 0 WITH HEADER LINE.
```

```
SELECT * FROM KNA1 INTO TABLE ITAB.
```

```
CALL FUNCTION 'OPEN_FORM'
  EXPORTING
    FORM          = 'ZSHABFORM1'
    LANGUAGE      = SY-LANGU.
```

```
LOOP AT ITAB.
CALL FUNCTION 'WRITE_FORM'
  EXPORTING
    ELEMENT       = 'ELEMENT'
    WINDOW        = 'MAIN'.
ENDLOOP.
```

```
CALL FUNCTION 'CLOSE_FORM'.
```

-> Save -> Activate -> Execute.

### **ADDING IMAGES TO THE FORM:**

Create a .bmp file and save it in a directory -> Goto SE78 Tcode -> Double click BMAP Bitmap images -> Click on Import icon from appn. toolbar -> Opens an interface -> Specify the path of .bmp file from the OS -> Specify the name for the image -> Select Color bitmap image radiobutton -> Click on Continue -> Image is imported.

To add the imported image into the form, right click on the form layout -> Select Create Graphic -> Opens an interface -> Select image from the form directory -> Select Color bitmap image radiobutton -> Specify resolution as 100 -> Continue -> An image is added to the script.

Simply call the form from the print program.

To upload .TIFF files into the SAPscript directory, make use of a predefined executable program called as RSTXLDMC.

In SE38 Tcode, specify the above name, click on execute pushbutton from application toolbar -> Opens an interface -> Specify the file path -> Execute.

Text Elements in the line editor are used to avoid data duplication.

## **SMARTFORMS**

### **ADVANTAGES OF SMARTFORMS:**

1. Smartforms does not require paragraph formats as a mandatory one.
2. Smartforms does not duplicate the data.
3. Smartforms are Client-independent and language-independent.
4. We can apply styles for the texts using Smartforms.
5. Print programs does not contain any function modules to call the form.

SMARTFORMS is the Tcode to create smartforms in ABAP.

### **Smartforms has three components:**

#### **Global Settings:**

1. Form Attributes - Holds the description about the smartform.
2. Form Interface - Holds the import and export parameters for the smartforms.
3. Global Definitions - Used to declare variables for the smartform that can be accessed by any component within the smartform.

#### **Pages and Windows:**

This area is used to create new pages, windows (header, footer, constant window, main window, variable window, graphic window).

#### **Navigations to create Smartforms:**

SMARTFORMS -> Specify form name starting with Z or Y -> Click on Create -> Opens an interface -> Enter short description -> Double click on Form Interface -> Specify following parameter in IMPORT tab button:  
STR TYPE C

-> Expand Pages and Windows -> By default, a Main Window is present -> Right click on Main Window -> Create -> Text -> Opens Form Editor -> To go to line editor, Click on 'Txt Editor' pushbutton on the upper left corner of the form editor -> Specify the variable (&STR&) in the first line of line editor -> Come back -> Save -> Activate -> Execute -> Opens Form Builder with autogenerated function module for the smartform ('/1BCDWB/SF00000042') -> Copy the function module generated.

To invoke the smartform from SE38 editor, Call the Function module and paste the FM copied from smart form function builder screen as follows:

```
DATA NAME(10) VALUE 'INDIA'.
```

```
CALL FUNCTION '/1BCDWB/SF00000042'  
  EXPORTING
```

STR = NAME.  
 -> Save -> Activate -> Execute in the same way as Sapscript.

**Navigations to pass internal table as an argument to Smartforms:**

Create a smartform -> In the Form interface -> Click on Tables tab button -> Specify the following internal table:

ITAB LIKE MARA.

-> Double click on Global Definitions -> Specify the global variable as follows:

ITAB1 LIKE MARA.

-> Expand Pages and Windows -> Right click on Main Window -> Create -> Table -> Click on Data tab button -> In the LOOP section, for the operand fields, specify the following:

ITAB INTO ITAB1.

-> Right click on Main Area -> Create -> Table Line -> In the Line Type input field, a line type called %LTYPE1 is automatically generated -> Select it.

To create internal table fields, right click on Text cell -> Create -> Text -> Select General Attributes tab button -> Opens Form Editor -> Goto Line editor by clicking on 'Txt Editor' pushbutton -> Specify the following in the first line:

```
&ITAB1-MATNR&,,&ITAB1-MTART&,,&ITAB1-MBRSH&,,&ITAB1-MEINS&
```

-> Come back -> Save -> Activate -> Execute -> Copy the autogenerated FM.

In SE38 program, specify the following:

```
DATA ITAB1 LIKE MARA OCCURS 0 WITH HEADER LINE.
```

```
SELECT * FROM MARA INTO TABLE ITAB1.
CALL FUNCTION '/1BCDWB/SF00000043'
TABLES
  ITAB      =      ITAB1.
```

Save -> Activate -> Execute in the same way as above.