



**Dr.SNS RAJALAKSHMI COLLEGE OF ARTS AND SCIENCE**  
(Autonomous)

Coimbatore -641049

Accredited by NAAC (Cycle- III)with 'A+' Grade  
(Recognised by UGC, Approved by AICTE, New Delhi and  
Affiliated to Bharathiar University, Coimbatore



## **UNIT – V**

**Dr.A.DEVI**

**Associate Professor**

**Department of Computer Applications**

**DRSNSRCAS**

## Internal Tables

### Introduction

Dealing with internal tables is one of the most important parts of working with ABAP. Internal tables have been hinted at briefly before, but not examined in any great depth. This chapter will do precisely that. If one is working in ABAP in any way at all, it is crucial to understand internal tables, as almost every program will use them. You have to understand both the old method of using header lines, and the new method using separate work areas. SAP has existed a long time, and while practices change, one will still often find old methods being used. When one is creating new programs, though, the newer method is always to be used.

Internal tables only ever exist when a program is running, so when the code is written, the internal table must be structured in such a way that the program can make use of it. You will find that internal tables operate in the same way as structures. The main difference being that Structures only have one line, while an internal table can have as many as required.

Internal tables are used for many purposes in ABAP. They can be used to hold results of calculations to then use later in the program, hold records and data so that this can be accessed quickly rather than having to access this data from database tables, and a great number of other things. They are hugely versatile, as they can be defined using any number of other defined structures, allowing, for example, many tables to be grouped together and then placed into one internal table.

The basic form of these consists of a table body, which is all of the records within the table, and a header record in the case of the older-style internal table. In the case of the newer style of internal table, the header record is absent and replaced by a separate work area. The header line or work area is used when you read a record from the internal table, providing a place for this 'current' record to be placed which can then be accessed directly. The header line or work area is also used and populated if you need to add a new record to the table, which is then transferred from the structure to the table body itself.

Previously, the TABLES statement has been used to include a table which has been created in the ABAP dictionary in a program. Internal tables, on the other hand, have to be declared themselves. When this is done, you must also declare whether a header record or separate work area will be used.

When creating new programs with internal tables it is best practice to use separate work areas. Using a header record has a number of restrictions, for example, you are not able to create multi-dimensional tables. We will not be cover multi-dimensional tables at length here, but if you plan to go further with ABAP, they will become important.

There are some restrictions on the records which can be held in internal tables. The architecture of an SAP system limits the size of internal tables to around 2GB. It is also important to bear in mind how powerful one's SAP system is (the hardware and operating system). It is generally best practice to keep internal tables as small as possible, so as to avoid the system running slowly as it struggles to process enormous amounts of data.

## Types of Internal Tables

Now the difference between the older and newer style internal tables has been mentioned, from here on, assume that it is the newer kind which is being discussed - *an internal table with a work area*.

An internal table can be made up of a number of fields, corresponding to the columns of a table, just as in the ABAP dictionary a table was created using a number of fields. Key fields can also be used with in internal tables and when creating these internal tables offer slightly more flexibility. In the ABAP dictionary, using key fields is imperative to uniquely identify each record. With internal tables, one can specify a non-unique key, allowing any number of non-unique records to be stored, allowing duplicate records to be stored if required.

Different types of internal tables can also be created, so that data can be accessed in the most efficient manner possible.

### Standard Tables

First, there are *standard tables*. These give the option of accessing records using a table key or an index. When these tables are then accessed using a key, the larger the internal table is, the longer it will take to access the records. This is why the index option is also available. Standard tables do not give the option of defining a unique key, meaning the

possibility of having identical lines repeated many times throughout the table. Additionally, though, this means that standard tables can be filled with data very quickly, as the system does not have to spend time checking for duplicate records. Standard tables are the most commonly used type of internal table in SAP systems.

### Sorted Tables

Another type of internal table is the sorted table. With these, a unique key can be defined, forcing all records in the table to be unique, removing duplication. These can again be accessed via the key or index. As the records are all unique, using the table key to find records is much quicker with sorted tables, though still not the fastest in all situations. It is often preferable to use a sorted table over a standard table, given the faster access speeds and the fact that this kind of table will sort records into a specific sequence. This gives one a substantial performance increase when accessing data.

### Hashed Table

The final type of internal table to be discussed here is a hashed table. With these, an index is not used to access the data, only a unique key. When it comes to speed, these are likely to be the preferred option. These are recommended particularly when one is likely to be creating tables which will be very large, as accessing data in large table is likely to be fairly laboured when using standard or sorted tables. These tables use a special hash algorithm to ensure the fast response times to reading records are maintained no matter how many records are held.

Despite the speed of hashed tables, you will however find that standard and sorted tables are generally used significantly more in SAP programs. Because of this, the majority of focus here will be put on these.

## Internal Tables - Best Practice Guidelines

As SAP has been around a long time, many programs exist that conform to using the older style internal table. You must be aware of this without falling into bad habits and using this style. It is now considered best practice to always use the newer style of internal table in SAP, ensuring that the programs created will be continue to be usable in the future, once the older style has been completely abandoned. Both old and new styles will be discussed here, so that you gain a degree of familiarity with the old style which persists in places, but when creating programs of your own, the new style should always be used.

## Creating Standard and Sorted Tables

Create a new program in the ABAP editor called Z\_EMPLOYEES\_LIST\_03 to use for the creation of internal tables. To begin to declare an internal table, the DATA statement is used. The program must be told where the table begins and ends, so use the BEGIN OF statement, then declare the table name, here 'itab01' (itab is a commonly used shorthand when creating temporary tables in SAP). After this, the OCCURS addition is used, followed by a number, here 0. OCCURS tells SAP that an internal table is being created, and the 0 here states that it will not contain any records initially. It will then expand as it is filled with data:

```
DATA: BEGIN OF itab01 OCCURS 0,
```

On a new line, create a field called 'surname', which is declared as **LIKE zemployees-surname**. Create another field called 'dob', **LIKE zemployees-dob**. It may be useful initially to give the field names in internal tables the same names as other fields which have been created elsewhere. By doing this, later on the **MOVE-CORRESPONDING** statement can be used to move data from one table to another. Finally, declare the end of the internal table is declared with "**END OF itab01.**"

```
    surname LIKE zemployees-surname,
    dob     LIKE zemployees-dob,
END OF itab01.
```

The structure of the internal table is now created, and code can be written to fill it with records. Using the OCCURS statement above, this automatically tells the system that an *old style* internal table with a header record is being used.

As mentioned earlier, it is advisable to always create the new style of internal table, allowing ABAP objects and so on to be used. With the new style of object-oriented programming it is encouraged to keep all the objects of your code separate, so that they can be reused in other programs and so on. To create the new style of internal table, the code is slightly different, separating out the individual data objects, like building blocks, which can then be put together to create new data objects later and so on. The manner in which this is done may seem significantly more laborious, but when you are working with larger, more complicated programs, the benefits will be clear.

## Create an Internal Table with Separate Work Area

Instead of using the DATA statement, this time start by defining a **line type**, using the **TYPES** statement. Following this, the **BEGIN OF** statement is used, followed by a name, here **'line01\_typ'**. Below this, the surname and dob fields from above can be created as before. Then the **END OF** statement is used to end the line type definition:

```
TYPES: BEGIN OF line01_typ,
        surname LIKE zemployees-surname,
        dob     LIKE zemployees-dob,
        END OF line01_typ.
```

Rather than defining the entire table structure at once, here only the structure of one line is defined. The table itself has not yet been defined. As a result of this, the OCCURS statement has not been used.

Once the line has been defined, next you define the **table type**. Again, use the **TYPES** statement, followed this time by the table, here **'itab02\_typ'** (*note the **\_typ** addition to the end as it is only the table type being defined, not the table itself*). Follow this with “**TYPE STANDARD TABLE OF line01\_typ.**”; telling the system it will be a standard table containing the structure of the line-type defined above:

```
TYPES itab02_typ TYPE STANDARD TABLE OF line01_typ.
```

In place of the OCCURS clause used for the old style of table, you can optionally add to the end of the line “**INITIAL SIZE (n)**” where (n) would be a number corresponding to the size you initially want the table to be. However, this is completely optional and is not frequently used.

If you want to create a *sorted table*, the ‘STANDARD’ in the above line is replaced with ‘SORTED’. You then have to specify the table key, with the addition “**WITH UNIQUE KEY (field name)**” where (field name) would be one of the fields set up in the line type definition, in this example ‘surname’. If you want more than one key field, these are simply then separated by commas:

```
TYPES itab02_typ TYPE SORTED TABLE OF line01_typ
                        WITH UNIQUE KEY surname.
```

Next, the table itself must be declared. As the table type defined was based on the line type previously defined, the table itself will be based on the table type. Here, the `DATA` statement returns, followed by the name of the table, `'itab02'`, and the `TYPE` of table to be used - `'itab02_typ'`:

```
|| DATA itab02 TYPE itab02_typ.
```

You still have the option to use a header line, but this must be explicitly stated when creating an internal table in this way. To do this, you simply add **WITH HEADER LINE** to the code above. This is however, as stated several times already, generally not advisable.

The final thing to do when creating an internal table this way is declare the work area which will be used in conjunction with the table. Remember that the work area is completely separate from the table, which has now already been created, allowing one to work with the data from the table in a way which is removed from it. This also allows for, if one wants, the same work area to be used for multiple tables, as long as they have the same structures, an example of reusing the code.

To declare *Work Area*, again use the `DATA` statement followed by the work area name, here `'wa_itab02'`. After this, the `TYPE` statement is used to specify the line type, here we can use the one already defined as `'line01_typ'`:

```
|| DATA wa_itab02 TYPE line01_typ.
```

While the manner in which the old style table is created may certainly seem easier, the newer method is much better and much more flexible. For example, having written all of the above code, if one then wanted to create a new table with the same structure, only one new line of code would have to be written, since the line and table types have already defined. The table `'itab03'`, for example, could be created simply by adding one line of code:

```
|| DATA itab02 TYPE itab02_typ.
|| DATA itab03 TYPE itab02_typ.
```

## Filling an Internal Table with Header Line

When you are reading a record from an internal table with a header line, that record is moved from the table itself into the header line. It is then the header line that you program works with. The same applies when creating a new record. It is the header line with which you work with and from which the new record is sent to the table body itself.

Below appears some slightly more extensive code for an old-style internal table, which can then be populated:

```
TABLES: zemployees.

*Internal Table with Header line
DATA: BEGIN OF itab01 OCCURS 0,
      employee LIKE zemployees-employee,
      surname  LIKE zemployees-surname,
      forename LIKE zemployees-forename,
      title    LIKE zemployees-title,
      dob      LIKE zemployees-dob,
      los      TYPE i VALUE 3,
END OF itab01.
```

The fields should broadly be familiar. The only new one here is 'los', representing 'length of service', an integer type with a default value of 3.

To start to fill this table, you can use a SELECT statement to select all of the records from the **zemployees** table and then use "INTO CORRESPONDING FIELDS OF TABLE **itab01**.", which will move the records from the original table into the new internal table into the fields where the names correspond. This type of select statement is called an *array fetch*, as it fetches all of the records at once, and places them in a new location. Notice that there is no ENDSELECT statement here - it is not a loop that is created:

```
SELECT * FROM zemployees
      INTO CORRESPONDING FIELDS OF TABLE itab01.
```

As the new los field does not have a corresponding field in the zemployees tables, every record will have this field populated with the los' default value of 3. Add a WRITE statement for itab01-surname below just to assist in the debug session coming up. Set a breakpoint on the SELECT statement, and execute the code to enter debug mode and observe the code as it works.



If you view the internal table before executing the next line of code here, you can see that it is currently empty. The line with the hat icon represents the current contents of the header line and below this, the lines of the internal table will be filled in. As you execute the array fetch, all of the lines of the internal table are filled at once:

```

➔ STOP SELECT * FROM zemployees
      INTO CORRESPONDING FIELDS OF TABLE itab01.

      WRITE itab01->surname.

```

Internal table **itab01** Type STANDARD Format E

1	EMPLOYEE SURNAME	FORENAME
	00000000	

Internal table **itab01** Type STANDARD Format E

1	EMPLOYEE SURNAME	FORENAME
	00000000	
1	10000002 JONES	AMY
2	10000003 MICHAELS	ANDREW
3	10000004 NICHOLS	BRENDAN
4	10000005 MILLS	ALICE

A different way of filling the table would be with the code below, this time with a select loop filling each field one at a time, using the MOVE statement to move the data from one table's field to the other. Note that los is not present here since it does not have a field in the zemployees table.

```

SELECT * FROM zemployees.
  MOVE zemployees-employee TO itab01-employee.
  MOVE zemployees-surname  TO itab01-surname.
  MOVE zemployees-forename TO itab01-forename.
  MOVE zemployees-title    TO itab01-title.
  MOVE zemployees-dob      TO itab01-dob.

ENDSELECT.

WRITE itab01-surname.

```

If you debug this code, you can see how it operates line-by-line as opposed to the array fetch which did all of the records at once. As you execute the first MOVE statement, it is visible that the first employee number appears in the header record of the internal table:

The screenshot shows a debugger window with the following code:

```

SELECT * FROM zemployees.
  MOVE zemployees-employee TO itab01-employee.
  MOVE zemployees-surname  TO itab01-surname.
  MOVE zemployees-forename TO itab01-forename.
  MOVE zemployees-title    TO itab01-title.

```

The debugger is paused at the second line. Below the code, the 'Internal table' window is visible, showing the table 'itab01' with the following structure:

1	EMPLOYEE SURNAME	FORENAME
100000021		

Stepping through the code you will see the other fields gradually appear in the header line until the end of the SELECT loop is reached. However, once this happens, since no code has been included telling the program to append the data in the header record to the internal table, this will simply be overwritten by the next iteration of the loop. This is a common mistake when using header lines and can be avoided by using the **APPEND** statement.

Before the ENDSELECT statement add another line of code reading “**APPEND itab01.**”, telling the system to add the contents of the header line to the internal table.

```

SELECT * FROM zemployees.
  MOVE zemployees-employee TO itab01-employee.
  MOVE zemployees-surname  TO itab01-surname.
  MOVE zemployees-forename TO itab01-forename.
  MOVE zemployees-title    TO itab01-title.
  MOVE zemployees-dob      TO itab01-dob.

  APPEND itab01.
ENDSELECT.

```

Internal table **itab01** Type STANDARD Format E

1	EMPLOYEE SURNAME	FORENAME
1	10000002 JONES	AMY
1	10000002 JONES	AMY

## Move-Corresponding

In the example, the MOVE statement was used several times to move the contents of the zemployees table to the corresponding fields in the internal table. It is possible however to accomplish this action with just one line of code. You can use the **MOVE-CORRESPONDING** statement. The syntax for this is simply “**MOVE-CORRESPONDING zemployees TO itab01.**”, telling the system to move the data from the fields of zemployees to their corresponding fields in itab01. This is made possible by the fact that both have matching field names. When making use of this statement you need to make sure that both fields have matching data types and lengths. This has been done here with the LIKE statement previously, but if it is not, the results could be unpredictable:

```

SELECT * FROM zemployees.
  MOVE-CORRESPONDING zemployees TO itab01.

  APPEND itab01.
ENDSELECT.

```

Next, copy the code with which the itab01 table was created to create another internal table called itab02. This time, the fields will be populated with an INCLUDE statement, so remove the fields between the BEGIN OF and END OF statements and replace them with the code “**INCLUDE STRUCTURE itab01.**” This will create a new table with the same structure:

```
DATA: BEGIN OF itab02 OCCURS 0.
      INCLUDE STRUCTURE itab01.
DATA END OF itab02.
```

You are not limited to using the structure of another internal table, another table created in the ABAP dictionary’s structure could be used with the same statement:

```
DATA: BEGIN OF itab03 OCCURS 0.
      INCLUDE STRUCTURE zemployees.
DATA END OF itab03.
```

Using this method can save a lot of time coding, and can be enhanced further allowing you to include multiple structures within one internal table, as below (though this example would, in fact, just include two of each column as zemployees and itab01 have effectively the same structures):

```
DATA: BEGIN OF itab04 OCCURS 0.
      INCLUDE STRUCTURE zemployees.
      INCLUDE STRUCTURE itab01.
DATA END OF itab04.
```

As long as the structures used have previously been defined in the system, this statement can be used to include many structures within newly created internal tables. You can also add new data statements as were previously used to declare internal table structures, extending the structures which have been included with new fields.

Let’s return to the array fetch method of populating internal tables. You will note that when using this method, all of fields were filled simultaneously, without using the header record. This is a very effective and quick method to use, given that there is no loop, so records do not have to be written to the table one at a time:

```
SELECT * FROM zemployees INTO CORRESPONDING FIELDS OF TABLE itab01.
```

Additionally, you do not have to use the \* which selects all of the fields of zemployees, but can specify the individual fields you want to move in this way. See the example below:

```
SELECT surname forename dob FROM zemployees INTO CORRESPONDING FIELDS
OF TABLE itab01.
```

## Filling Internal Tables with a Work Area

Now, if you are, following the newer method of using internal tables, the header record is to be bypassed entirely and the table filled from a separate work area.

Return to the code which was shown above for creating a table with the new method, shown below:

```
*Declare a Line Type
TYPES: BEGIN OF line01_typ,
        surname LIKE zemployees-surname,
        dob     LIKE zemployees-dob,
      END OF line01_typ.

*Declare the 'Table Type' based on the 'Line Type'
TYPES itab02_typ TYPE STANDARD TABLE OF line01_typ.

*Declare the table based on the 'Table Type'
DATA itab02 TYPE itab02_typ.

*Declare the Work Area to use with our Internal Table
DATA wa_itab02 TYPE line01_typ.
```

Here, the *SELECT* statement is used again. Since the line type only includes two fields, only those two fields should be selected. Once they're selected, *INTO* is used with the work area specified as the area to populate. An *APPEND* statement is added to move the data from the work area into the table itself. Finally, *ENDSELECT* is used:

```
SELECT surname dob FROM zemployees
      INTO wa_itab02.
      APPEND wa_itab02 TO itab02.
ENDSELECT.
```

An array fetch can also be used to populate the internal table. Note that here you can still use the \* to select all of the records in zemployees, but as the internal table has only two of these corresponding fields, the rest will just be ignored:

```
SELECT * FROM zemployees
INTO CORRESPONDING FIELDS OF TABLE itab02.
```

## Using Internal Tables One Line at a Time

Now you know how to fill internal tables with data, a look will be taken at how to use the data in them line-by-line.

Internal tables are just stored in memory, so cannot be directly accessed, their contents can only be read via the work area, using a loop. The way this is done is slightly different from database tables and, rather than using `SELECT` and `ENDSELECT`, `LOOP` and `ENDLOOP` are used instead.

*First, tables using a header line.* Add some new code to your program as follows. Begin the `LOOP` and specify the internal table by adding “`AT itab01`”. Code is then added to achieve the desired outcome and the loop is closed with `ENDLOOP`. For example:

```
LOOP AT itab01.
  WRITE: / itab01-surname, itab01-forename.
ENDLOOP.
```

If you execute code in debug mode, you will see that for each loop pass, the header line (represented by the hat icon) is filled with data before being written to the output screen:

The screenshot shows the SAP ABAP debugger interface. The top pane displays the following code:

```

LOOP AT itab01.
  WRITE: / itab01-surname, itab01-forename.
ENDLOOP.

```

The bottom pane shows the internal table `itab01` with the following data:

Internal table	itab01	Type	STANDARD	Format	E
1	EMPLOYEE SURNAME				
	FORENAME				
1	10000002 JONES		ANY		
1	10000002 JONES		ANY		
2	10000003 MICHAELS		ANDREW		

Internal Tables 1	
JONES	AMY
MICHAELS	ANDREW
NICHOLS	BRENDAN
MILLS	ALICE
NORTHMORE	PETER

## Modify

Now a look will be taken at how records in the table can be changed with the **MODIFY** statement. Using the code below, the IF statement will check whether an entry's surname matches the set value of 'JONES'. Where it does match, this will be updated to the new value of 'SMITH' in the header line. The MODIFY statement will then update the internal table itself with the new value. Note that the MODIFY statement here will not create a brand new record, but will replace the existing JONES record in the table. If a MODIFY statement is used in a loop, it is always the current line which is changed. This should not be done if you are trying to modify key fields of an internal table that uses a unique key. If the MODIFY statement is used outside of a loop, the record index number must be specified. The way in which the statement is used here can only be used in tables with index tables or header lines:

```

LOOP AT itab01.
  IF itab01-surname = 'JONES'.
    itab01-surname = 'SMITH'.
    MODIFY itab01.
  ENDIF.
ENDLOOP.

```

## Describe and Insert

In the same loop, the DESCRIBE TABLE statement will be used. This statement can be used to find out information about the content of an internal table, including the number of records the table holds, the reserve memory space used, and the type of table it is. *In practice you normally only ever really see this being used to find out the first of these three pieces of information though.*

Beneath the ENDIF, add the line of code “**DESCRIBE TABLE itab01 LINES line\_cnt.**” The **LINES** part of this statement is used to request the value of the number of lines contained in the internal table, and ‘**line\_cnt**’ is a new variable (of type i) set up to hold this value.

Up until now, the APPEND statement has been used to add records to the table. This automatically inserts the new record at the end of the table. If you want to add a record somewhere in the middle, the **INSERT** statement should be used, along with the table *index number*, to specify the position where a new record is to be inserted. For example, if you used the index number 10, the new record would appear between the 9<sup>th</sup> and 10<sup>th</sup> records in the table.

The syntax used here is “**INSERT itab01 INDEX (n)**” where (n) is the index number where you want to insert the new record. In the example below, (n) is represented by line\_cnt, so the new record will be inserted at the line matching the index number which corresponds to the value of line\_cnt. The new record will be inserted on the line before the last line of the table:

```

LOOP AT itab01.
  IF itab01-surname = 'JONES'.
    itab01-surname = 'SMITH'.
    MODIFY itab01.
  ENDIF.
ENDLOOP.

DESCRIBE TABLE itab01 LINES line_cnt.

INSERT itab01 INDEX line_cnt.

```

If you execute the code in debug mode, you will see the surname JONES is modified to become SMITH. The DESCRIBE statement is then triggered and *line\_cnt* given a value of 5. Now, the last record in the table is that with the surname NORTHMORE, employee number 10000006, so once the loop completes, this is the record held in the header line. The INSERT statement, then will add a copy of this record at the 5<sup>th</sup> line of the table. *Remember that, as this is a standard type table, duplicate records are allowed.* Because you are in debug mode you can alter the header record’s values can be manually altered in debug mode, so a new, non-duplicate record can in fact be created, with the surname BLOGS and employee number 10000007. The image below shows the header record and internal table just before and after the INSERT statement is executed:



Internal table		itab01	Type	STANDARD	Format	E
2	EMPLOYEE SURNAME		FORENAME			
10000007	BLOGS		PETER			
2	10000003 MICHAELS		ANDREW			
3	10000004 NICHOLS		BRENDAN			
4	10000005 MILLS		ALICE			
5	10000006 NORTHMORE		PETER			

Internal table		itab01	Type	STANDARD	Format	E
3	EMPLOYEE SURNAME		FORENAME			
10000007	BLOGS		PETER			
3	10000004 NICHOLS		BRENDAN			
4	10000005 MILLS		ALICE			
5	10000007 BLOGS		PETER			
6	10000006 NORTHMORE		PETER			

## Read

The READ statement is another way in which you can access the records of an internal table, allowing you to read specific individual records from the table. Given that these examples are using the old style method and as such using a header line, this record will be sent to the header line and accessed from there.

The way that the internal table has been declared will affect the way in which a READ statement's code is written, bear this in mind. Depending on whether the table has a unique key or not will also change how the READ statement is specified. For a standard table without a unique key, the record's index number is used:

```
|| READ TABLE itab01 INDEX 5.
```

The READ statement is generally the fastest way you can access the records of an internal table, and using the index number is the fastest way to use this statement. It can be up to 14 times faster than a table key. However, you do not always know the index number of the record which is to be read. If you are using a table key, the syntax would be as follows:

```
READ TABLE itab01 WITH KEY
      employee = 10000007.
```

This can also be done with non-unique keys, but this can become problematic. For example, if you used ‘surname’ as your table key and the table contained 3 surnames which were the same, the system sequentially reads the records resulting in the first occurrence be read.

This type of code, particularly with key fields, can also be used with sorted and hashed tables, which contain unique key fields.

## Delete Records

To delete records from an internal table, you simply use the **DELETE** statement. This can be used to delete either individual records or groups of records from a table. The fastest way of achieving this is by specifying a table index. Note this only applies to standard and sorted tables as only these two types of tables have an index. The syntax is as follows:

```
DELETE itab01 INDEX 5.
```

The header line is not used at all. The record to be deleted is directly accessed via its index number.

This statement can also be used inside a loop:

```
LOOP AT itab01.
  IF itab01-surname = 'SMITH'.
    DELETE itab01 INDEX sy-index.
  ENDIF.
ENDLOOP.
```

The code here will identify any record with the surname SMITH and delete it. As you do not know the index number of SMITH beforehand, the system variable **sy-index** is used, which is always set to the index number of the current loop, so when the SMITH record appears, **sy-index** will match its index number and the record will be deleted.

The DELETE statement should not be used without the INDEX addition. If used outside of a loop result in a runtime error, causing the program to crash. Inside a loop, it must be present to adhere to future releases of the ABAP syntax.

Another addition to the DELETE statement is the **WHERE** clause. There are times where when you will not know the index number of the record you want to delete, so more code will have to be added. The WHERE addition is useful here, and can be combined with other logic to locate the record(s) which should be deleted. Using this, you must always be as specific as possible, otherwise data which should not be deleted can be. The syntax should look like this:

```
DELETE itab01 WHERE surname = 'SMITH'.
```

Note that if there are multiple records which match the logical expression, they will all be deleted.

## Sort Records

The statement used to sort records in an internal table is, unsurprisingly, SORT. The basic syntax is very simple:

```
SORT itab01.
```

Without any additions, this will sort the records in ascending order by the table's unique key. This works for sorted and hashed tables. For a standard table, you must use the **BY** addition to specify which fields to sort by:

```
SORT itab01 BY surname.
```

This would sort the table alphabetically in ascending order by the field SURNAME. Bear in mind that SAP systems work with a wide variety of languages all at the same time, so if you are sorting by language-specific criteria, **AS TEXT** should be added between the table name and BY addition.

You are not limited to sorting just by one field; you can list up to 250 fields if desired. In this example, FORENAME is added. Note that it is not necessary to separate these with commas:

```
|| SORT itab01 AS TEXT BY surname forename.
```

Given the position of AS TEXT in the statement, this will be applied to all fields which are specified. If you only wanted AS TEXT to apply to forename, it would be placed after the forename:

```
|| SORT itab01 BY surname forename AS TEXT.
```

By default, the system will sort records in ascending order. This can be changed to descending as shown:

```
|| SORT itab01 DESCENDING AS TEXT BY surname forename.
```

## Work Area Differences

Having been through the statements with which one can work with internal tables with a header record, the old style, now the differences in using these methods with the new, encouraged style of operating with a separate work area will be looked at

### Loops

First, let's look at the differences in reading data in a loop. Here, the loop will read each record from the internal table and place each record into the work area instead of the header line. Because the work area is completely separate from the internal table, the work area you want to use within the loop must be specified. The **INTO** addition is used to specify the work area the record is to be read into:

```
|| LOOP AT itab02 INTO wa_itab02.  
||     WRITE wa_itab02-surname.  
|| ENDLLOOP.
```

In this example the records will be read one record at a time into the work area **wa\_itab02**, then the contents of the surname field will be written to the output screen.

## Modify

Using the **MODIFY** statement with this kind of internal table the statement must specifically reference the work area. The example below shows our previous **MODIFY** statement example altered to work with a work area:

```

LOOP AT itab02 INTO wa_itab02.
  IF wa_itab02-surname = 'JONES'.
    wa_itab02-surname = 'SMITH'.
    MODIFY itab02 FROM wa_itab02.
  ENDIF.
ENDLOOP.

```

## Insert

When working with the **INSERT** statement with this type of internal table, nothing needs to change to the **DESCRIBE** statement. The only change is to the **INSERT** statement. Here the new record held in `wa_itab02` is to be inserted **INTO** the internal table `itab02`:

```

DESCRIBE TABLE itab02 LINES line_cnt.
INSERT wa_itab02 INTO itab02 INDEX line_cnt.

```

## Read

The **READ** statement again follows a similar logic, insisting that the work area is also referenced in the code:

```

READ TABLE itab02 INDEX 5 INTO wa_itab02.

```

```

READ TABLE itab02 INTO wa_itab02
  WITH KEY surname = 'SMITH'.

```

## Delete

Just as the **DELETE** statement does not require any reference to the header record to work, nor does it require any reference to the work area. The statement deletes records from the table directly by their index number or other key, so operates no differently at all here.

## Delete a Table with a Header Line

When working with internal tables, you will often come upon situations where it is necessary to delete all of the records in a table in one go, depending upon the specific task you trying to complete. For example, if you fill an internal table in a high level loop, you will want the table to be empty when it comes to the next iteration. This section will explain how to delete internal tables and their contents, first for those with header lines, then for those with work areas.

There is a certain sequence of tasks you should adhere to when deleting the contents of an internal table with a header line. First, you should ensure the header line is clear, then that the body of the table is clear.

### CLEAR

To do the first of these tasks, use the **CLEAR** statement, followed by the table name. This will clear out the header line only, and set the header-line fields to their initial value. To clear the body of the table, the statement is used again, only this time followed by [], deleting all of the records in the table itself:

```
CLEAR itab01.  
CLEAR itab01[].
```

### REFRESH

Alternatively, the **REFRESH** statement can be used. This will clear all records from the table, but you must bear in mind that it does not clear the header record, which will still contain values:

```
REFRESH itab01.
```

### FREE

You could also use the **FREE** statement, with the same syntax as REFRESH. This statement not only clears out the internal table, but also frees up the memory which it was using. It does not mean the table ceases to exist entirely, but no longer is operating in memory. With this statement, like REFRESH, the header line is unaffected, so the first CLEAR statement must always be used in conjunction with both of these:

```
FREE itab01.
```

## Delete a Table with a Work Area

To delete internal tables which are using work areas, similar methods are used. However, as the work area is an entirely different structure, any code written which will affect the internal table will not affect the work area, and vice versa.

The CLEAR statement above, when used on a table without a header line, will clear the whole contents of the table without needing to add the []. Remember that another CLEAR statement must be used to empty the work area. The same applies to the REFRESH and FREE statements. The syntax above will work, and a further CLEAR statement must be used to empty the work area. In the examples below, assume itab01 and wa\_itab01 refer to the newer style internal table and its work area:

```
CLEAR itab01.  
CLEAR wa_itab01.  
  
REFRESH itab01.  
CLEAR wa_itab01.  
  
FREE itab01.  
CLEAR wa_itab01.
```

## Modularizing Programs

### Introduction

As has been discussed before, it is good practice when using SAP to keep your programs as self-contained and easy to read as possible. Try to split large, complicated tasks up into smaller, simpler ones by placing each task in its own separate, individual module which the developer can concentrate on without other distractions. Modularizing your code allows single tasks to be focussed upon one at a time, without the distraction and confusion which can be caused if the code you are working with is in the middle of a large, complicated structure. Doing this makes the program much easier to work with and debug. Once a small, modularized section of code is complete, debugged and so on, it does not subsequently have to be returned to, meaning the developer can then move on and focus on other issues.

Creating individual source code modules also prevents one from having to repeatedly write the same statements again and again, which in turn makes the code easier to read and understand for anyone coming to it for the first time. This is also useful when it comes to support. Anyone later having to support the program will again find the code much more comprehensible if it is written this way.

It is important to concentrate on the design of a program. Rather than starting to code a solution straight away, a solution should be mapped out, using pseudo-code or flow-charts for example. Only when the design makes sense should the coding exercise begin. Having a solution design also helps when modularizing a program, because this allows you to see how the program can be split up into separate pieces, allowing you to then focus on the individual pieces of development one piece at a time.

In the chapter covering selection screens, modularization was hinted at with the use of processing blocks. However, modularization in your own programs is not just limited to processing blocks. The SAP system allows for a number of techniques to be used to break a program up into smaller, more manageable sections of code.

This chapter will look at the tools SAP provides for achieving this.



## Includes

When talking about modularization, what we are really talking about is taking a sequence of ABAP statements and placing them in their own, separate module. We can then 'call' this code module from our program.

Here, some code which has been used previously will be modularized. Below is the code for the second internal table which was created, the one with a work area, followed by some logic which will perform tasks involving the internal table:

```
REPORT z_mod_1 .

TABLES: zemployees.

*Declare a Line Type
TYPES: BEGIN OF line01_typ,
        surname LIKE zemployees-surname,
        dob     LIKE zemployees-dob,
      END OF line01_typ.

*Declare the 'Table Type' based on the 'Line Type'
TYPES itab02_typ TYPE STANDARD TABLE OF line01_typ.

*Declare the table based on the 'Table Type'
DATA itab02 TYPE itab02_typ.

*Declare the Work Area to use with our Internal Table
DATA wa_itab02 TYPE line01_typ.
```

```
DATA line_cnt TYPE i.
*****

SELECT * FROM zemployees
      INTO CORRESPONDING FIELDS OF TABLE itab02.

LOOP AT itab02 INTO wa_itab02.
  WRITE wa_itab02-surname.
ENDLOOP.

CLEAR: itab02, wa_itab02.

LOOP AT itab02 INTO wa_itab02.
  IF wa_itab02-surname = 'JONES'.
    wa_itab02-surname = 'SMITH'.
    MODIFY itab02 FROM wa_itab02.
  ENDIF.
ENDLOOP.

DESCRIBE TABLE itab02 LINES line_cnt.
INSERT wa_itab02 INTO itab02 INDEX line_cnt.

READ TABLE itab02 INDEX 5 INTO wa_itab02. [

READ TABLE itab02 INTO wa_itab02
      WITH KEY surname = 'SMITH'.
```

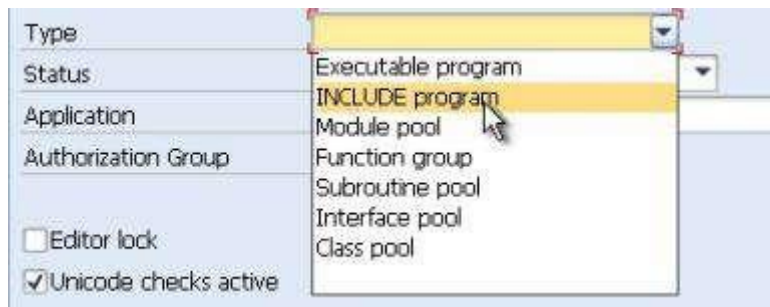
First, we will look at INCLUDE programs. INCLUDE's are made available globally within an SAP system and their sole purpose is modularizing code. They are simple to define and accept no parameters. Below the REPORT statement, fill in the statement for declaring an include. Type **INCLUDE** and then define a name, here "**Z\_EMPLOYEE\_DEFINITIONS**":

```
REPORT z_mod_1

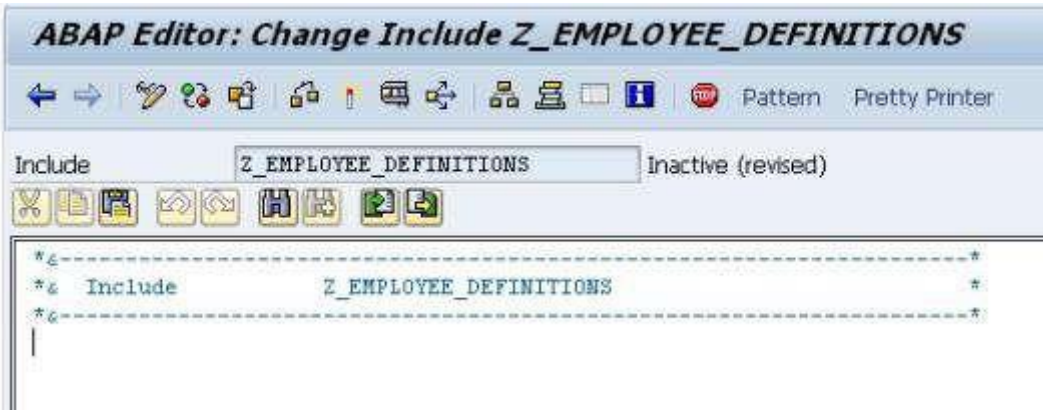
INCLUDE Z_EMPLOYEE_DEFINITIONS.

TABLES: zemployees.
```

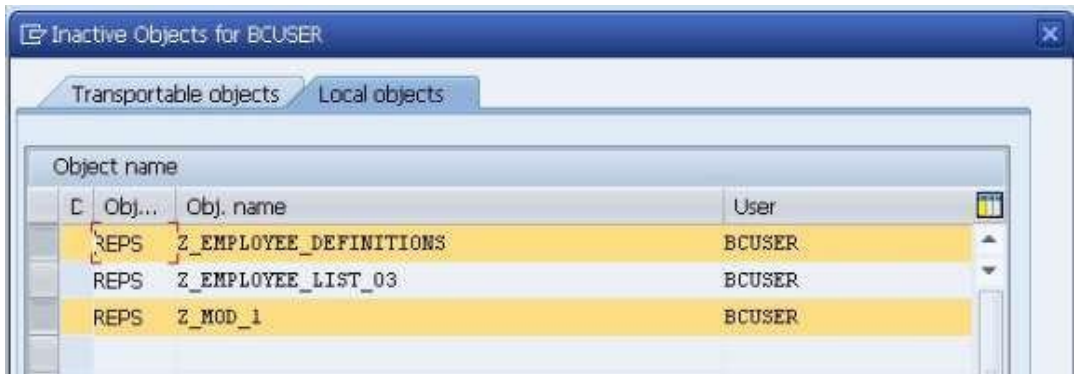
This statement is telling the program to *include* the INCLUDE program within our original program. There are two ways of creating this new INCLUDE program. You can type the name into the ABAP editor's initial screen and select the 'Attributes' radio button, followed by 'Create'. Then, when the window appears asking what kind of program this is, select 'INCLUDE program':



The second method is by using forward navigation. In the code window, double-click **Z\_EMPLOYEE\_DEFINITIONS** and select 'Yes' to create the new object. Save as 'Local object' as before, and then you will be presented with a new, blank coding screen where the INCLUDE program code can be typed/inserted:



Remember, the INCLUDE program is a separate file on the SAP system so can be included in any other program. The INCLUDE program must be activated itself, and when you activate any program that includes it, it will always check to see if the INCLUDE program is active too. If not, error messages will appear. A simple way to activate both at once is to select both in the menu offered when activating the main program:



In the main program, comment out the section where the *line type* is defined, and copy & paste it into the INCLUDE program:

```

INCLUDE Z_EMPLOYEE_DEFINITIONS.

TABLES: zemployees.

**Declare a Line Type
*TYPES: BEGIN OF line01_typ,
*      surname LIKE zemployees-surname,
*      dob     LIKE zemployees-dob,
*      END OF line01_typ.

```

```

*-----
* Include           Z_EMPLOYEE_DEFINITIONS
*-----

*Declare a Line Type
TYPES: BEGIN OF line01_typ,
      surname LIKE zemployees-surname,
      dob     LIKE zemployees-dob,
      END OF line01_typ.

```

Because the INCLUDE program has been declared in the main program above, the program will continue to work as normal. This is an example of a way in which code can be effectively outsourced to an INCLUDE program, removing that code from service in the main program and hence making that program less densely populated with code. This does not have to be used only for data declarations as in this case. It is commonly used for sections of programs which involve program logic too.

## Procedures

If you want to split programs into separate functional modules, procedures can be used. These are processing blocks which are called from the main ABAP program, and come in the form of **sub-routines**, **sub-programs**, and **function modules**.

Sub-routines and sub-programs are mainly used for local modularization of code, meaning small, modular, self-contained units of code called from the program in which they are defined. These can then, if necessary, be used many times in the program without having to be typed out repeatedly. Function modules, on the other hand, allow you to create modular blocks of code which are held separately from an ABAP program and can be called from any other program.

Sub-routines are *local*, and function modules are *global*, and both types of procedure are commonly used in SAP systems. The latter, though, are probably the more widely used of

the two. Function modules can be used to encapsulate all of the processing logic used within the business system, and SAP has ensured that they can be used both by their own developers and SAP's customers.

INCLUDE programs cannot accept any parameters; procedures differ here, and have an interface for transferring data from the calling program to the procedure itself. Because data can be passed into a procedure, this means that you can define data definitions within the procedure itself which are only available to that procedure.

## Sub-Routines

One of the great benefits of using sub-routines is that it helps to modularize program code inside the actual program, giving the program structure.

To create a sub-routine, forward navigation is used. Copy, and then comment out, the array fetch SELECT statement from the internal table code above:

```
SELECT * FROM zemployees
      INTO CORRESPONDING FIELDS OF TABLE itab02.
```

Above the commented-out section, use the statement **PERFORM**. This statement is used to perform a sub-routine. Then a name for the sub-routine is added. Here, since this code fills the itab02 internal table, call the sub-routine “**itab02\_fill**” as shown:

```
PERFORM itab02_fill.

*SELECT * FROM zemployees
* INTO CORRESPONDING FIELDS OF TABLE itab02.
```

Double-click the statement then to use forward navigation and create the sub-routine. Answer ‘Yes’ to the dialog box and a window appears asking where the sub-routine is to be created. A choice is offered between the main program, the INCLUDE program and a new INCLUDE program which can be created. Select the main program here. Once this is done, code block starting with ‘form’ and ending with ‘endform.’ Is created located at the end of your program, where the code for the sub-routine can be filled in. Paste in the code for the array fetch, and the sub-routine is created:

```

*
*-----*
* Form itab02_fill
*-----*
*   text
*-----*
* --> p1      text
* <-- p2      text
*-----*
form itab02_fill .

SELECT * FROM zemployees
      INTO CORRESPONDING FIELDS OF TABLE itab02.

endform.                " itab02_fill

```

When the `PERFORM` statement is reached as the program executes, the sub-routine created will be triggered, meaning that the array fetch is performed in exactly the same way as previously. Once 'endform.' is reached, processing returns to the next statement following `PERFORM` and continues as normal, terminating at the end. Though the sub-routine does appear at the bottom of the code, the system can identify it as a sub-routine and hence it will not be executed again.

Up until now, only global variables have been discussed. These are variables which are defined as in the program itself, usually at the top of the program and, in this instance, the `INCLUDE` program. These variables, including internal tables and so on, can be accessed throughout the program. If variables are declared only in sub-routines, however, these are considered local variables. These can only be accessed within the single sub-routine where they are declared. Once control passes back to the main body of the program, local variables can no longer be referenced.

Given that these variables only have to be declared within sub-routines, rather than the whole program, memory usage is kept to a minimum. Additionally, these can be useful in helping keep everything self-contained and modularized. As mentioned previously, sub-routines have an interface, and these local variables can be used in the interface.

To declare a local variable, one simply uses the `DATA` statement as normal within the sub-routine. Declare one of these named "**zempl**", which is **LIKE zemployees-surname**. This new variable can now only be referenced by other code which appears in the sub-routine, between `form` and `endform`. You can also declare a variable to be used in the interface. In

doing this, the system is being told that data will be transferred to the sub-routine data interface.

Create code for a second sub-routine, called “**itab\_02\_fill\_again**” and above this create 2 new DATA fields, as shown in the example below, telling the new sub-routine to use the new data fields. Then use forward navigation to create this sub-routine:

```
DATA z_field1 like zemployees-surname.
DATA z_field2 like zemployees-forename.

*****

PERFORM itab02_fill.

perform itab02_fill_again USING z_field1 z_field2.
```

```
*-----*
* Form itab02_fill_again
*-----*
* text
*-----*
* -->P_Z_FIELD1 text
* -->P_Z_FIELD2 text
*-----*
form itab02_fill_again using p_z_field1
                           p_z_field2.

endform.                  " itab02_fill_again
```

Note the difference in how the new sub-routine appears. This form has now been generated including two fields which will then be used in the interface. It is advisable here to rename the fields in the sub-routine so you know what they refer to:

```
form itab02_fill_again using p_zsurname
                           p_zforename.
```

Notice that there is no data type for these fields, since they are taken from the fields referenced in the PERFORM statement, however, they will take on the same properties as those fields. Add some new code to the form as shown below. The values of p\_zsurname and p\_zforename will be written, then the value of p\_zsurname changed to ‘abcde’:

```

form itab02_fill_again using    p_zsurname
                               p_zforename.

    write / p_zsurname.
    write / p_zforename.

    p_zsurname = 'abcde'.

endform.                        " itab02_fill_again
    
```

Ensure these fields hold some data by giving z\_field1 and z\_field2 values in the main program:





```

z_field1 = 'ANDREWS'.
z_field2 = 'SUSAN'.

perform itab02_fill_again USING z_field1 z_field2.
    
```

When the PERFORM statement is executed, these values will be passed through to the fields in the sub-routine. Add a breakpoint above this and run the program can be run in debug mode.

You can see z\_field1 and z\_field2 are filled with their initial values:

z_field1	ANDREWS	 
z_field2	SUSAN	 

Next, the sub-routine is entered and the values of these fields are passed in via the interface, so that the local variables here take on the same values as those in the main program:

Field names	1 - 4	Field contents
z_field1	ANDREWS	 
z_field2	SUSAN	 
p_zsurname	ANDREWS	 
p_zforename	SUSAN	 

The two WRITE statements are then executed, followed by the change in value for p\_zsurname. Because the field is used in the interface, the global variable, z\_field1's value also changes:



Field names		Field contents
z_field1	abcde	
z_field2	SUSAN	
p_zsurname	abcde	
p_zforename	SUSAN	

When using fields in the interface, it is important to keep this in mind. Any fields attached to the **USING** addition that are changed in the sub-routine will also be changed in the program.

### Passing Tables

Sub-routines are not limited to only passing individual fields. Internal tables can also be passed, as well as a combination of both fields and tables. When passing fields though, one must always get the sequence of field names correct, as it is the sequence which will determine which field is passed to the interface variable of the form.

Create a new sub-routine called **itab02\_write**. Then, use the **TABLES** addition to specify the table to be passed, here **itab02**:

```
PERFORM itab02_write TABLES itab02.
```

Removing any unnecessary code, the form will look like this:

```

*&-----*
*&   Form itab02_write
*&-----*
*   text
*-----*
*   -->P_ITAB02 text
*-----*
form itab02_write tables p_itab02 structure.
endform.           " itab02_write

```

Using the **TABLES** addition, the program ensures that the contents of the internal table are transferred to the subroutine and stored in the internal table **p\_itab\_02**. Once this subroutine is processed, the contents of the local internal table are then passed back to the global internal table.

Note that this method is for a table without a header line. If this code was used with an old-style internal table, only the header line would be passed to the table. To pass the full table, you need to add [] at the end of the statement.

When an internal table is passed into a sub-routine, the local internal table is always declared with a header line. Write some code and then debug the program to see this. The code below will loop through the records of the internal table, sending the contents to a temporary work area and then writing the contents of the surname field to the output screen:

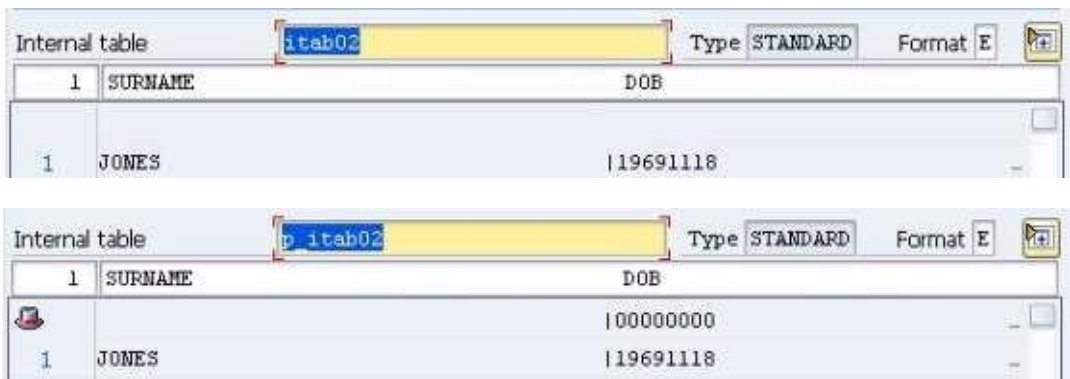
```
FORM itab02_write TABLES p_itab02.

DATA wa_tmp TYPE line01_typ.

LOOP AT p_itab02 INTO wa_tmp.
  WRITE wa_tmp-surname.
ENDLOOP.

ENDFORM.                " itab02 write
```

When analysed in debug mode, the itab02 table does not have a header record, but p\_itab02 does:



Still, since a new work area was created for the LOOP statement to follow, the header record becomes irrelevant.

### Passing Tables and Fields Together

Now, a combination of fields and tables will be passed into a subroutine at the same time. Create another PERFORM statement, called **itab02\_multi**. Retain the TABLES statement, but then add the USING statement afterwards:

```
PERFORM itab02_multi TABLES itab02 USING z_field1 z_field2.
```

Use forward navigation to generate the form.

```
form itab02_multi tables p_itab02 structure < itab02 #local# >
                        "Insert correct name for <...>
using p_z_field1
      p_z_field2.
```

You can then use write code to interact with both fields and the table.

## Sub-Routines - External Programs

Sub-routines were initially designed for modularizing and structuring a program, but they can be extended so that they can be called externally from other programs. Generally to do this, though, one should create function modules instead.

If you do want to create external sub-routines, however, this is possible. There are two ways in which a sub-routine can be called from an external program. The first of these is the one which should really always be used if doing this, as this is compatible with the use of ABAP objects.

If you want to call a sub-routine called ‘**sub\_1**’, held in a program called ‘zemployee\_hire’, the code would look like this. Note that additions can still be used with this method:

```
PERFORM sub_1 IN PROGRAM zemployee_hire USING z_field1 z_field2.
```

The difference here is that the sub-routine is being called from a separate program in the SAP system.

The second form is very similar, and works the same with additions and so on, the program is just included in brackets. Keep in mind though this form of the code cannot be used with ABAP objects:

```
PERFORM sub_1(zemployee_hire) TABLES itab02 USING z_field1 z_field2.
```

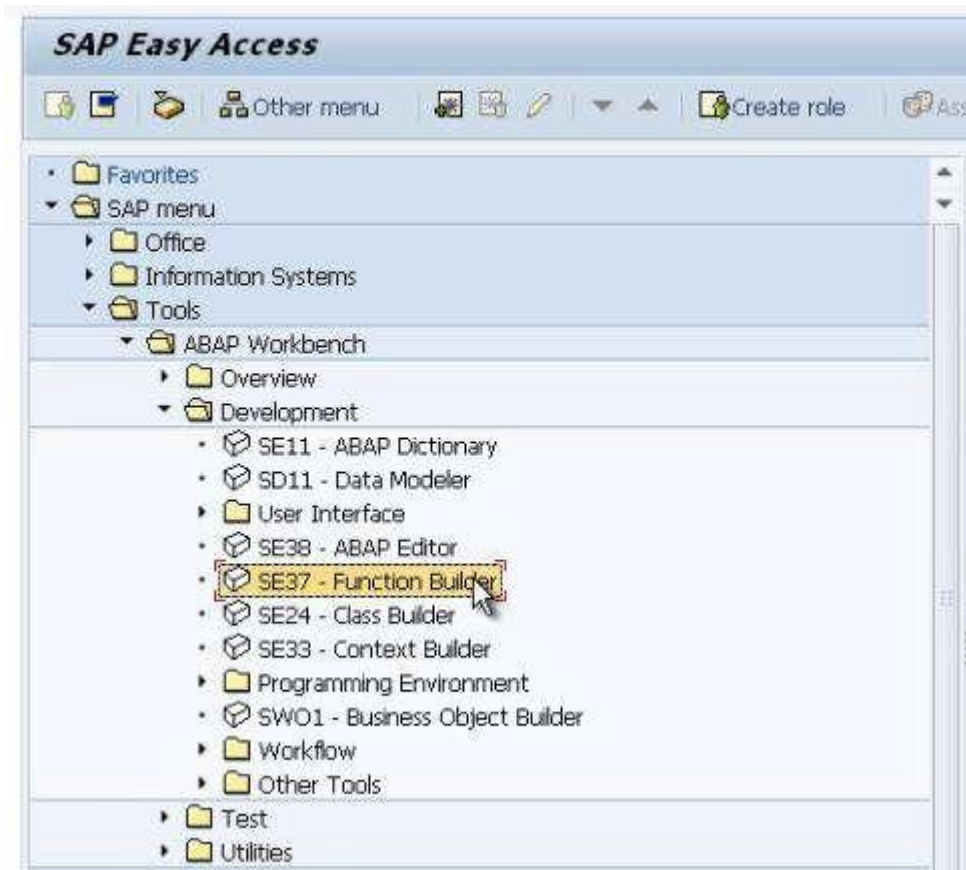
Calling external sub-routines is not common practice, sub-routines tend to stay internal to the program and where you want to call sub-routines in external programs, this is usually done via function modules.

## Function Modules

Function modules make up a major part of an SAP system, because for years SAP have modularized code using them, allowing for code re-use, first by themselves and their developers, then by customers.

Function modules refer to specific procedures which are defined in function groups, and can be called from any other ABAP program. The function group acts as a kind of container for a number of function modules which would logically belong together, for example, the function modules for an HR payroll system would be put together into a function group. SAP systems have thousands of function modules available for use in programs, so if you search around the system it will often be possible to find pre-existing modules for the tasks you may be asked to code.

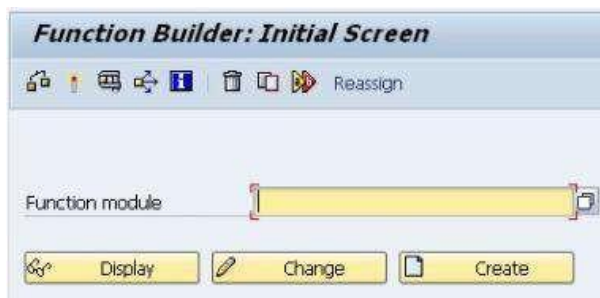
To look at how to create function modules, the function builder must be looked at. This is found via the menu at the very beginning of the system, via the SAP menu  Tools  ABAP Workbench  Development. There one will find the function builder, with transaction code SE37:



Before diving into an example of how to use a function module we need look at how function modules are put together, so as to understand how to use them in a program.

## Function Modules – Components

The initial screen of the function builder appears like this:



Rather than typing the full name here, part of a function module name will be typed with a wild card character to demonstrate the way function modules can be searched for. Type `*amount*` and then press the F4 key. The results of the search will then be displayed in a new window:

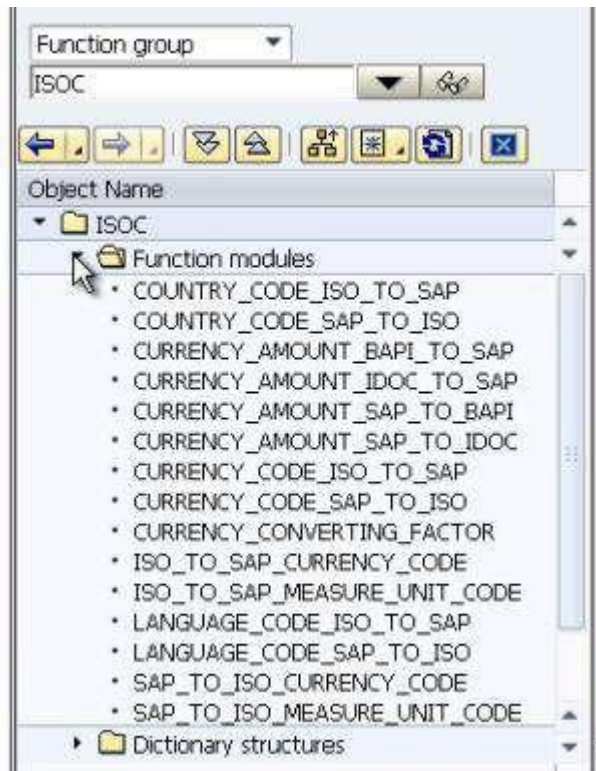
Function module:

Display Change Create

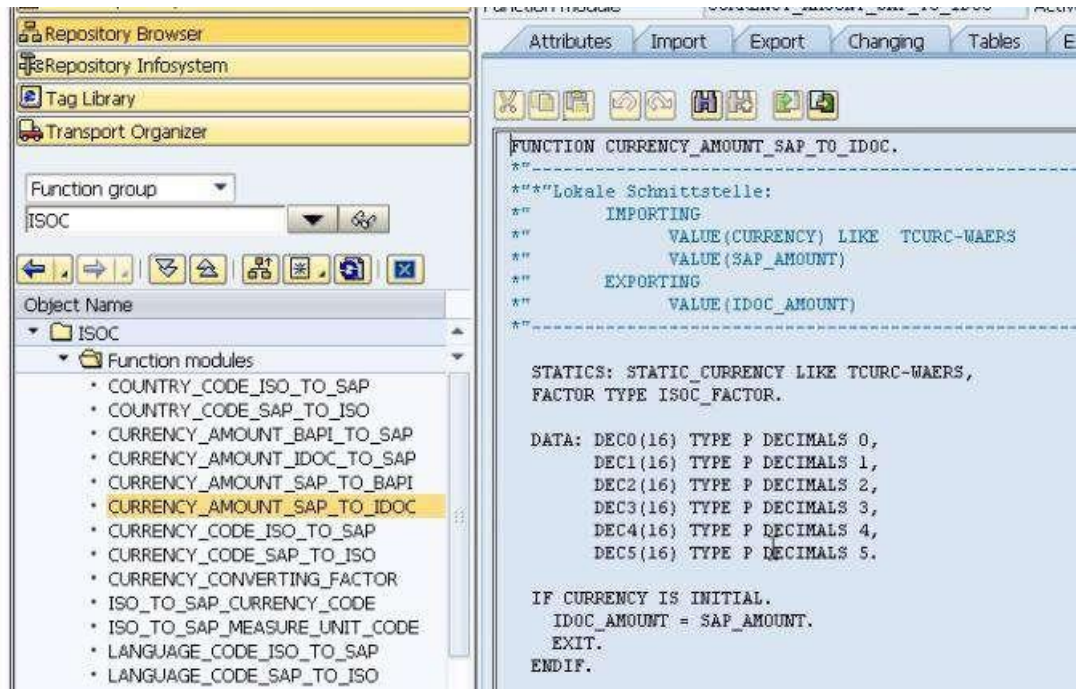
Repository Info System: Function modules Find (12 Hits)

Function group	Function group short text
Name of function module	Short text for function module
FF017	Conversion of amounts to words utility
SPELL_AMOUNT	Convert numbers and figures in words
FF01	
FIMA_COND_DETAIL_AMOUNT_CHECK	
FIMA_COND_DETAIL_AMOUNT_PA	
FIMA_COND_DETAIL_AMOUNT_PBO	
FF05	
FIMA_AMOUNT_DISCOUNT	
ISOC	
CURRENCY_AMOUNT_BAPI_TO_SAP	
CURRENCY_AMOUNT_IDOC_TO_SAP	
CURRENCY_AMOUNT_SAP_TO_BAPI	
CURRENCY_AMOUNT_SAP_TO_IDOC	
RHALE_CONVERT	
RH_ALE_CURR_AMOUNT_IDOC_TO_SAP	
RH_ALE_CURR_AMOUNT_SAP_TO_IDOC	
SFCC	
ALV_CORRECT_CURR_AMOUNTS	

The function modules are displayed in the lines with a blue background and their function groups in the pink lines above. If you would like to look further at the function group **ISOC**, the Object Navigator screen (se80) can be used. This screen can in fact be used to navigate many objects held in the SAP system, not only function modules but programs and so on, using the menus on the left hand side of the screen. Here, we can see a list of function modules (and other objects) held in the function group ISOC:



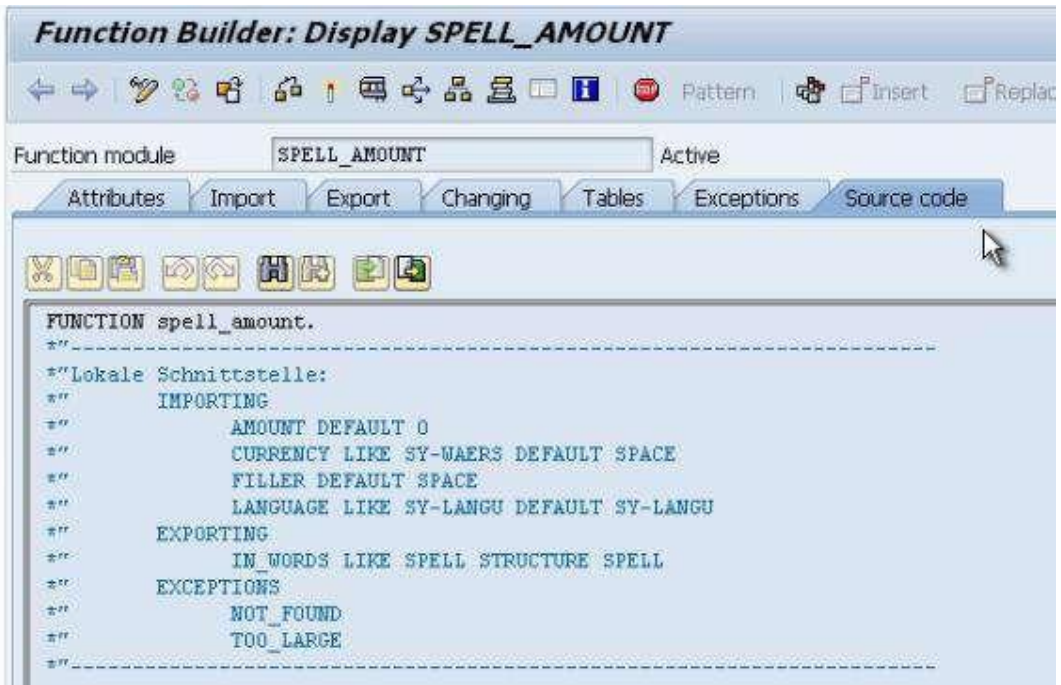
The four which showed up in the \*amount\* search are present, along with a number of others. If double-click any of these function modules, the code for that function module will appear on screen to the right of the menu:



Return back to the function builder screen, do the \*amount\* search again and this time select the function module **SPELL\_AMOUNT**. Double-click it and choose Display.

The code will then appear in a screen similar to that of the ABAP editor. There are, however, a series of tabs along the top which will now be looked at.





### Attributes Tab

This shows the function group and some descriptive text for the function module, as well as some options for the function module’s processing type, plus some general data.

### Import Tab

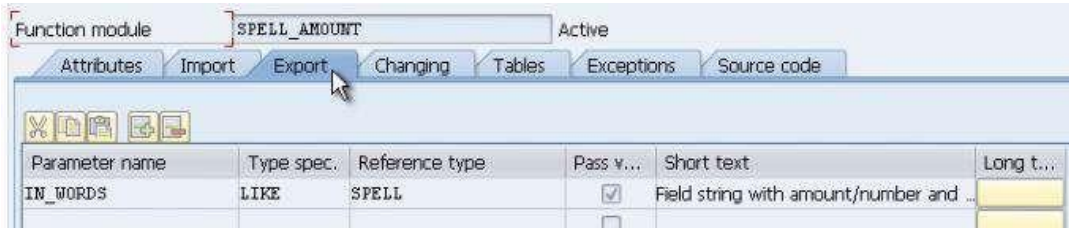
This lists the fields which will be used in the data interface which are passed into the function module from the calling program. These fields are then used by the function module code:

Parameter name	Ty...	Reference type	Default value	O...	P...	Short text
AMOUNT			0	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Amount/number that is/are to be covered
CURRENCY	LIKE	SY-WAERS	SPACE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Currency for amounts, for number SPACE
FILLER			SPACE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Filler with which the output field is entered
LANGUAGE	LIKE	SY-LANGU	SY-LANGU	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Language indicator for conversion in words

Take note of the different column labels. The fifth column, with a checkbox, is labelled ‘**Optional**’, meaning that these fields do not have to be passed into the function module by the calling program. More often than not though, there will be at least one mandatory field.

### Export Tab

This specifies the fields which are sent back to the calling program once the function module’s code has been processed:



### Changing Tab

This lists fields which can be *changed* by the function module.

### Tables Tab

Like sub-routines, with function modules you are not restricted to only passing in fields, but can also pass in internal tables.

### Exceptions Tab

This tab lists exception information which can be passed back to the calling program, which indicate whether the function module was executed successfully or not. This is where specific error messages for can be defined to identify any specific errors or warn- ings that occur during code execution that need to be passed back to the calling program to allow the programmer take the necessary course of action.



## Source Code Tab

The final tab is the source code itself for the function module, which appears automatically when one opens it from the function builder screen. Here, you can examine the code in depth so as to determine what exactly the function module is doing.

With pre-existing function modules you generally do not even have to look at this, as you should know what data the function module is supposed to send back.

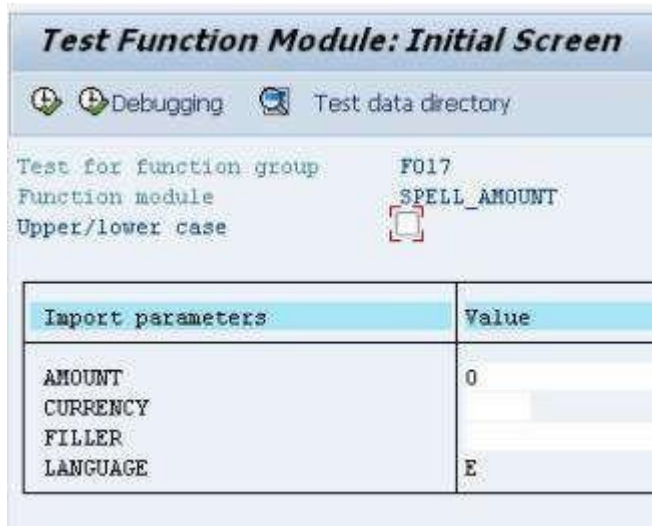
*The function module in this example converts numeric figures into words, so there is little need to examine the code in depth if one already knows what the output is to be.*

## Function Module Testing

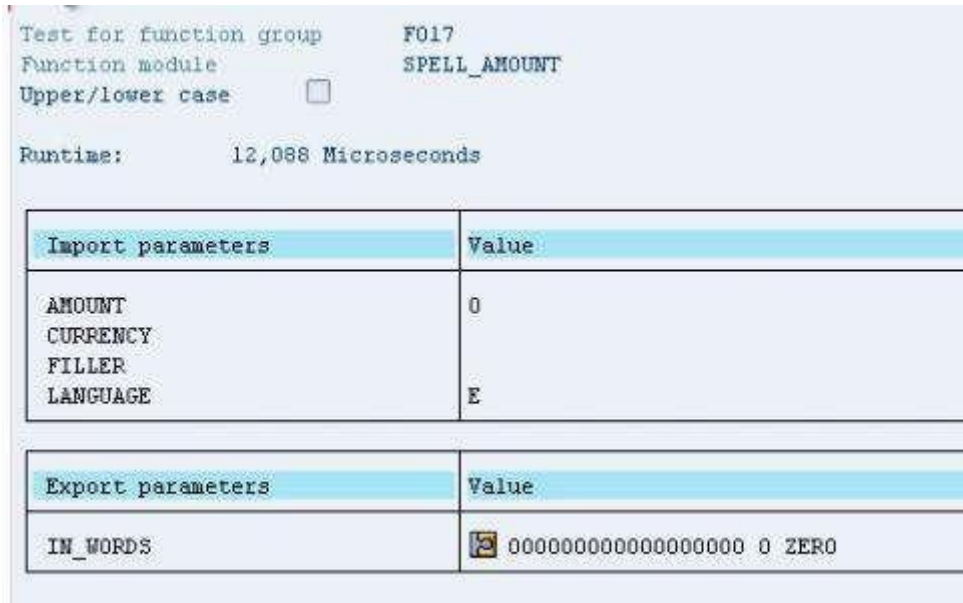
As Function Modules are created as separate objects, there are tools you can use to test function modules without having to write the code to call them. Just as programs can be tested and their output checked, you can do exactly the same with function modules. This is done with the F8 key or the same Test/Execute icon found in your own programs. In fact, you don't even have to be within the function module to do test it out. It can be done from the initial SE37 screen once the module's name appears in the appropriate field:



Test out the function module using the Test button as shown above.



As all fields are optional, this can then be executed without inputting any data.



Since the amount in the import parameters was 0, the export parameters then read ZERO. If you click the small button in the *Value* column of the export parameters, the results are broken into their individual export fields.

**Structure Editor: Display IN\_WORDS from Entry**

Column Metadata

NUMBER	DEC	CUR	WORD
000000000000000000	000	0	ZERO

The number input was 0, the decimal value was 0 and a currency was not specified, so the WORD output is simply ZERO.

Let's run the test again but this time enter some data into the AMOUNT field and CURRENCY field. Then execute the test again.

Import parameters	Value
AMOUNT	123456
CURRENCY	
FILLER	
LANGUAGE	E

Import parameters	Value
AMOUNT	123456
CURRENCY	
FILLER	
LANGUAGE	E

Export parameters	Value
IN_WORDS	000000000123456000 0 ONE HUNDRED TWENTY-THREE THOUSAND FOUR HUNDRED FIFTY-SIX

Import parameters	Value
AMOUNT	123456
CURRENCY	GBP
FILLER	
LANGUAGE	E

Export parameters	Value
IN_WORDS	0000000000001234560 2 ONE THOUSAND TWO HUNDRED THIRTY-FOUR

This output may look odd, but when the button is pressed you will see that, as **GBP** has **2 decimals**, the value **56** has been included in the **decimals** column rather than the number column:

NUMBER	DEC	CUR	WORD
0000000000001234	560	2	ONE THOUSAND TWO HUNDRED THIRTY-FOUR

If you were to select a currency which does not use decimals, the full number would appear.

The ability to test function modules in this way is a great time saver for the programmer, as it allows you to confirm whether a function module will complete the tasks you want before generating the code to use it in your program.

