SAP ABAP Modularization

What is Modularization?

In some scenarios, the application programs are too big and it's not easy to maintain or find out issues or debug the issues in the program. The application program in ABAP can be divided into smaller units using the technique called Modularization.

Modularization is a technique used to divide the application program into smaller units to maintain easily and reduce the code redundancy. The identical logic coded in many places (either in the same program or in multiple programs) called as a redundandant code.

To understand the concept clearly, let us discuss with an example of multiplying two numbers. Let us assume the multiplication of two numbers logic coded in many places of the application program.

Modularization supports to reduce the code redundancy by separating the multiplication logic from multiple places into a single modulization unit and calling the separated modulization unit from all the place where it is required.

In the same way if any variables are using in multiple programs, those are coded in modularization unit and use it in the multiple programs where those are required. This technique reduces the code redundancy and increases the readability, reliability and re-usability.

Modularizing source code is set of statements coded in a separate module and the module can be called by any program as required.

Advantages -

- Easy to read and understood
- Easy to maintain
- Easy to debug
- Eliminates code redundancy
- Increases reusability of code

Modularization usage -

Modularization concept implemented in ABAP programs in the following ways -

Modularization Usage	Description
Macros	If same set of statements are repeated more than once in the program, those statements can be used to create a macro. The macro only used in the program where it is defined.
Subroutines	Subroutines are procedures can define in any ABAP program and call from any ABAP program. Subroutines are two types - **Internal** and **External**. Subroutines normally contains section code or algorithms code.
Include programs	If same set of statements (source code) are used more than once in program, those statements can be used to create an include program. Include programs can use in any other ABAP program. Include programs are only used to modularize the source code but have no parameter interface.
Function modules	Function modules are stored in central library. Function modules are available to the entire system. RFC (Remote Function Call) enables function modules that can be called from non-SAP system.
Methods/Classes	Methods/classes used in ABAP object oriented programming. Methods/Classes implements parameter interface.

SAP ABAP Macros

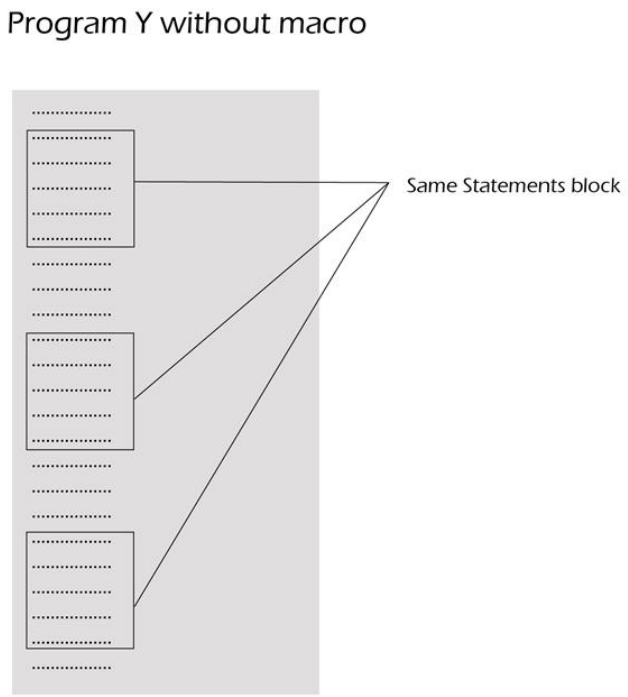
If same statements block is repeated more than once in various places of the program, it causes code redundancy. One of the repeated statements block can place in the macro and the macro call replaces the statements block where ever it repeatedly existed.

Macro is a statements block created with set of statements that calls multiple times within the same program.

The macro only used in the program where it is defined. The macro should be defined before the macro call coded. All the statements in the macro should code in between DEFINE and END-OF-DEFINITION.

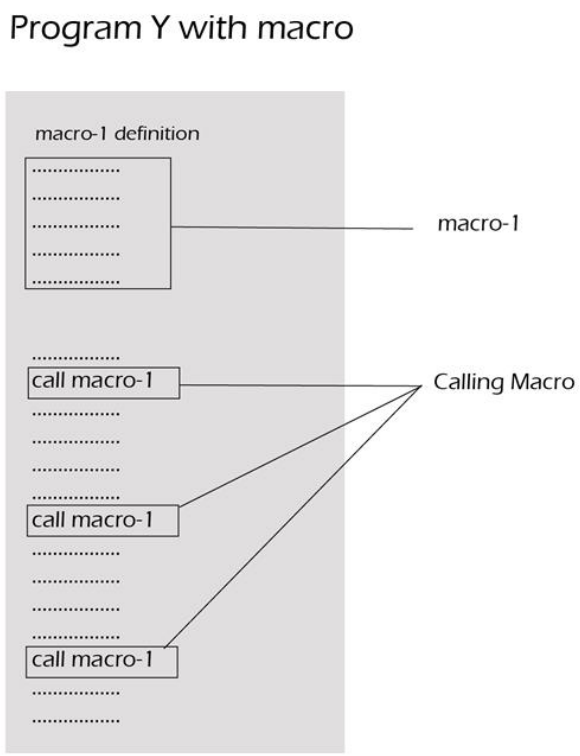
MACROS are not operational statements. MACRO cannot call itself. Macros are nested, another MACRO can be called within a MACRO.

Lets take an example program Y and below diagram explains how the program Y look like before and after macro implementation -



Program Y has the same statements block in three places which is a redundant code. It also increases the complexity of the program.

The below diagram shows the same program after macro implementation -



In the above diagram, macro-1 is the macro definition and the repeated code block is replaced with calling macro-1 in all the places.

Syntax -

Definition -

```
DEFINE <macro-name>..
        Statements-block
END-OF-DEFINITION.
```

Calling -

```
<macro-name> [<param1> <param2>....<param9>].
```

- **Macro-name -** Specifies the macro name.

- **Param1, param2,… -** Specifies the macro parameters. The parameters replace with place holders in the macro definition. The maximum number of parameters/place holders used in macro are nine. <param1> <param2>....<param9> replaces the comma separated place holders &1, &2, …., &9.

Example -

Below example displays the selected option from two radio buttons without using macros.

Code without Macro -

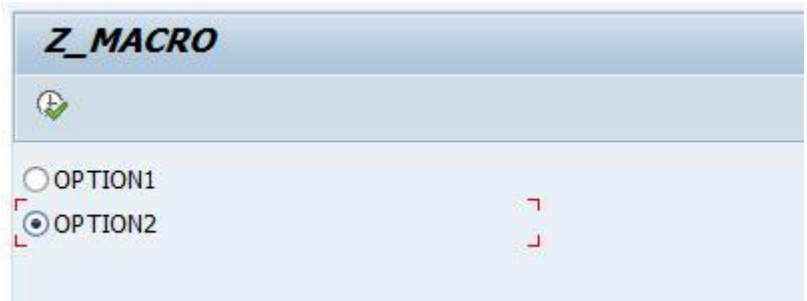
```
*&---------------------------------------------------------------------*
*& Report  Z_MACRO
*&---------------------------------------------------------------------*
REPORT  Z_MACRO.
PARAMETERS: OPTION1 type C RADIOBUTTON  group RG,
          OPTION2 type C RADIOBUTTON  group RG.
START-OF-SELECTION.
IF OPTION1 = 'X'.
  WRITE: 'Macro from option:  1'.
  SKIP 1.
  WRITE: 'Radio button selected:  1'.
ENDIF.
IF OPTION2 = 'X'.
  WRITE: 'Macro from option:  2'.
  SKIP 1.
  WRITE: 'Radio button selected:  2'.
ENDIF.
```

Code with Macro -

```
*&---------------------------------------------------------------------*\
*& Report  Z_MACRO*
*&---------------------------------------------------------------------*
REPORT  Z_MACRO.
DEFINE disp_macro_info.
  WRITE: 'Macro call : &1'.
  SKIP 1.
  WRITE: 'Radio button selected: &1'.
```

```
END-OF-DEFINITION.
PARAMETERS: OPTION1 type C RADIOBUTTON  group RG,
          OPTION2 type C RADIOBUTTON  group RG.
START-OF-SELECTION.
IF OPTION1 = 'X'.
  disp_macro_info 1.
ENDIF.
IF OPTION2 = 'X'.
  disp_macro_info 2.
ENDIF.
```

Output -



Select OPTION2 radio button and the output is –

Explaining Example -

In the above example, each and every statement is preceeded with a comment to explain about the statement. Go through them to get clear understanding of example code.

In **code without macro**, the below peice of code repeated 2 times with very minimal changes for every radio button selected.

```
WRITE: 'Macro from option:  1'.
SKIP 1.
WRITE: 'Radio button selected:  1'.
```

The peice of code can added to macro and macro can be called two times to reduce redundancy. So the macro can be coded like below.

```
DEFINE disp_macro_info.
  WRITE: 'Macro call : &1'.
  SKIP 1.
  WRITE: 'Radio button selected: &1'.END-OF-DEFINITION.
```

SAP ABAP Include Programs

If same set of statements (source code) used in more than one program, those statements can add to the include program. Include programs is available to all the programs and used in any program.

Include programs are only used to modularize the source code but have no parameter interface. Include programs are not standalone programs and cannot be executed independently.

Include programs available globally and can use in any ABAP program. Include program contains small piece of source code that can be included in a program with an INCLUDE statement.

INCLUDE statement is responsible for copying the include program source code to the main program during the runtime. Include programs can't call themselves. Include program must be syntax error free and contain complete statements.

Syntax -

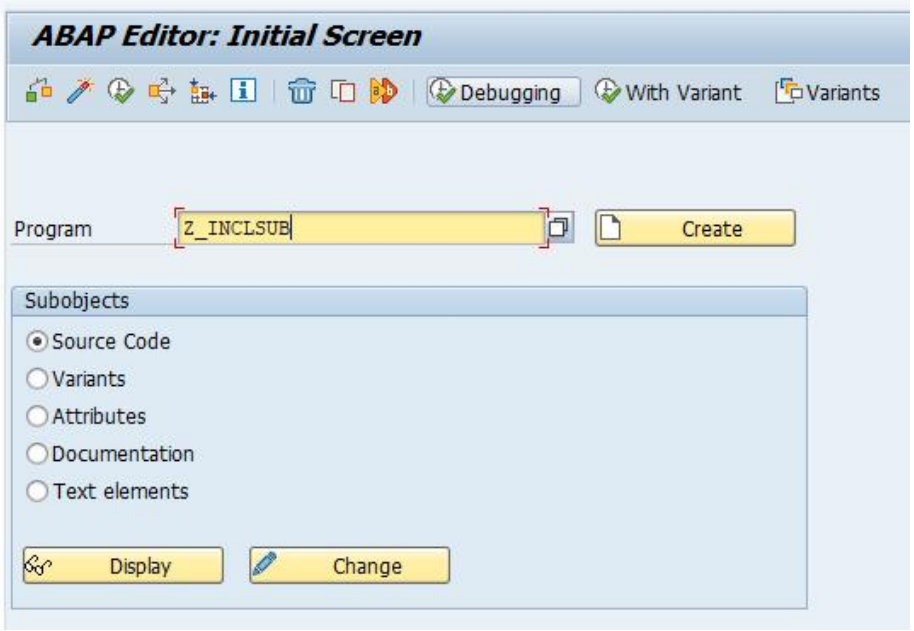
INCLUDE <include-program>

- **Include-program** - Specifies the include program name. INCLUDE programs can be created in the ABAP Editor.

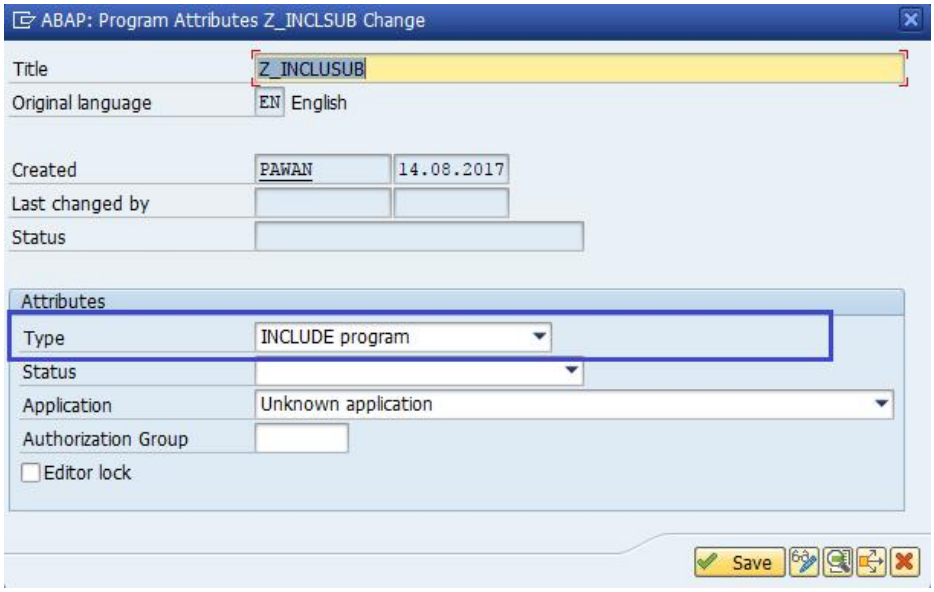
Example -

Below example explains how to create an include program and how it is included in the program.

Step-1: Go to **SE38** transaction. Enter the include name **(Z_INCLSUB)** and click on create.



Step-2: Re-enter the program name(Z_INCLSUB), select the Type of the program as **INCLUDE program** and click on **Save**.



Step-3: Enter the **package details** and click on **Local object** to open ABAP editor.

Step-4: Add the code to the **Z_INCLSUB** in ABAP editor. In this case, we added the below code to the include **Z_INCLSUB**.

```
WRITE / 'Inside the include..'.
```

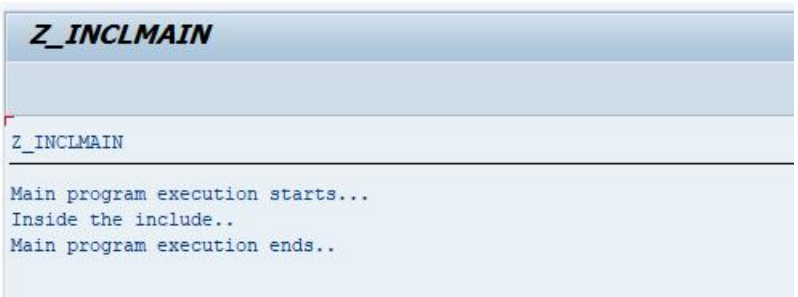
Step-5: Save, **Activate** the include and close the ABAP Editor.

Step-6: Open a new program to add **Z_INCLSUB** to the program. In this case we are creating **Z_INCLMAIN** program as a main program. Add the include **Z_INCLSUB** to the program **Z_INCLMAIN** like below –

```
REPORT  Z_INCLMAIN.
Write 'Main program execution starts...'.
INCLUDE Z_INCLSUB.
Write / 'Main program execution ends..'.
```

Step-7: Save, **Activate** and **Execute** the main program Z_INCLMAIN.

Output -



Explaining Example -

In the above example, each and every statement is preceeded with a comment to explain about the statement. Go through them to get clear understanding of example code.

The code from **Z_INCLSUB** includes in **Z_INCLMAIN** during the run time and produces the above result.

SAP ABAP Subroutines

What is Subroutines -

Subroutines are procedures that can define in any program and call from any ABAP program. Subroutines normally contains sections code or algorithms.

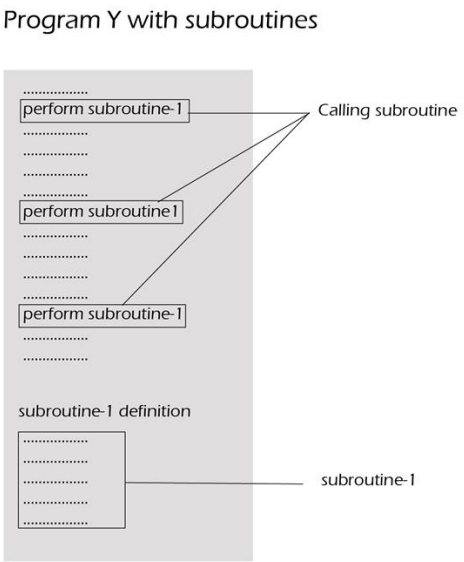
Subroutines can be defined anywhere in the program and no restriction on where to define. Subroutines can be defined using FORM and ENDFORM statements.

PERFORM statement used to call the subroutine. PERFORM and FORM must contain the same number of parameters.

Let us take an example program Y and below diagram explains how the program Y look like before and after subroutine implementation -

Program Y has the same statements block in three places which is redundant code. It also increases the complexity of the program.

The below diagram shows the same program after subroutines implementation -



In the above diagram, subroutine-1 is the subroutine definition and the code block is replaced with calling subroutine-1 in all the places.

Subroutine Types -

Subroutines are classified into two types based on how they are used. Those are -

- Internal Subroutine
- External Subroutine

Internal Subroutines -

Internal subroutines used in the same program where it is defined. These subroutines can't available to the other programs. Internal subroutine can't be nested.

Syntax -

Definition subroutine -

```
FORM <subroutine-name> [USING   ... [VALUE(]<parm1>[)]
                                    [TYPE <data-type>|LIKE <field>]... ]
        [CHANGING... [VALUE(]<parm1>[)]
        [TYPE <data-type>|LIKE <field>]... ].
            .
            Statements
            .ENDFORM.
```

Calling subroutine -

```
PERFORM <subroutine-name> [USING   ... <parm1>... ]
        [CHANGING... <parm1>... ].
```

- **Subroutine-name** - Specifies the subroutine name.
- **<parm1>…** - Specifies the parameters being passed. Once the PERFORM statement executes, control transfers to the subroutine. Control returns to the next statement immediate to PERFORM once subroutine execution completed.

External Subroutines -

External subroutines defined globally. This type of subroutines can be used by other programs. External subroutines can be nested.

Syntax -

Definition subroutine -

```
FORM <subroutine-name> [USING   ... [VALUE(]<parm1>[)]
                                    [TYPE <data-type>|LIKE <field>]... ]
        [CHANGING... [VALUE(]<parm1>[)]
        [TYPE <data-type>|LIKE <field>]... ].
            .
            Statements
            .ENDFORM.
```

Calling subroutine -

```
PERFORM (<fsubr>)[IN PROGRAM (<program-name>)][USING   ... <pi>... ]
                [CHANGING... <pi>... ]
                [IF FOUND].
```

- **Subroutine-name** - Specifies the subroutine name. IF FOUND option can prevent a runtime error from being triggered if <program-name> does not contain a subroutine with the name <subroutine-name>. Once the PERFORM statement executes, control transfers to the subroutine. Control returns to the next statement immediate to PERFORM once subroutine execution completed.

Variables in Subroutine -

There are two types of variables used in subroutines. Those are -

Local variables	**Global variables**
A variable that define within the subroutine.	A variable that define outside of the subroutine.
It is said to be local to the subroutine.	It is said to be global to the subroutine.

These variables can't be used outside the subroutine.	These variables can be used outside the subroutine and within the program.
Local variable life time is until the end of subroutine execution.	Global variable life time is until the end of program execution.

Parameters in Subroutine -

There are two types of parameters in subroutines. Those are -

Actual parameters	**Formal parameters**
Parameters that appear on the PERFORM statement are called actual parameters.	Parameters that appear on the FORM statement are called formal parameters.
For Example - PERFORM s1 p1, p2, p3	**For Example -** FORM s1 p1, p2, p3
p1, p2 and p3 are the actual parameters.	p1, p2 and p3 are the formal parameters.

Passing parameters to subroutine -

There are three ways of passing parameters to a subroutine. Those are -

- Pass by reference
- Pass by value
- Pass by value and result

Method	**Description**
By reference	Passes the pointer of the original memory location where the value stored. Most efficient method. Value changes in the subroutine are reflected in the calling program.
By value	Only value passed to the subroutine. Allocates new temporary memory location within the subroutine for use. The memory is freed when the subroutine ends. Value changes are not reflected in the calling program.
By value and	Like pass by value, the contents of the new temporary memory copied

result	back into the original memory before returning. Allows changes and allows rollback.

Subroutines statements addition -

Below are the list of additions and the corresponding methods refer to -

Addition	**Method**
using v1	Pass by reference
changing v1	Pass by reference
using value(v1)	Pass by value
changing value(v1)	Pass by value and result

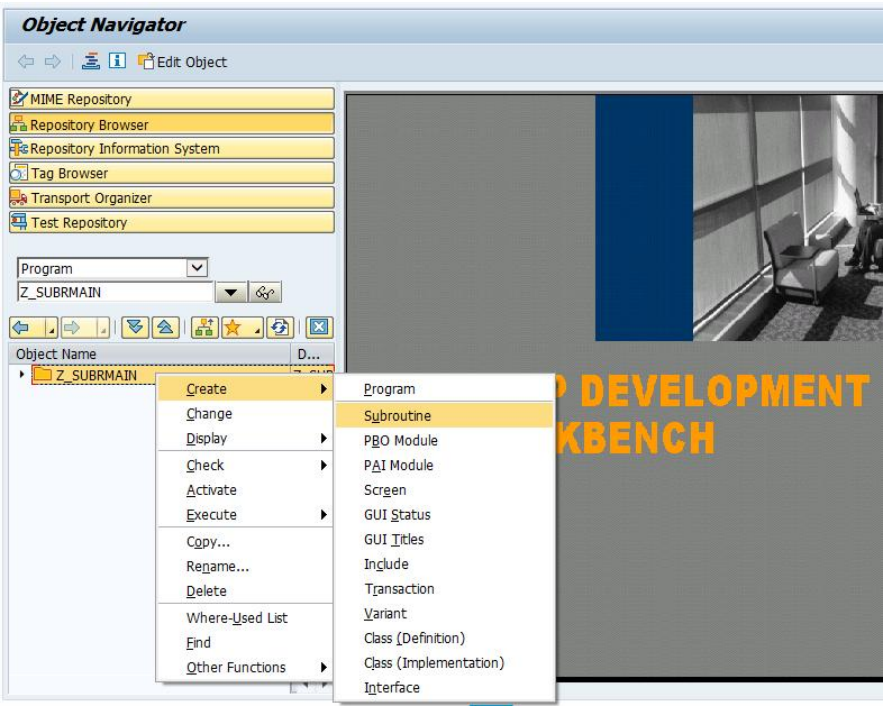
Example -

Simple example to create one program with subroutine.

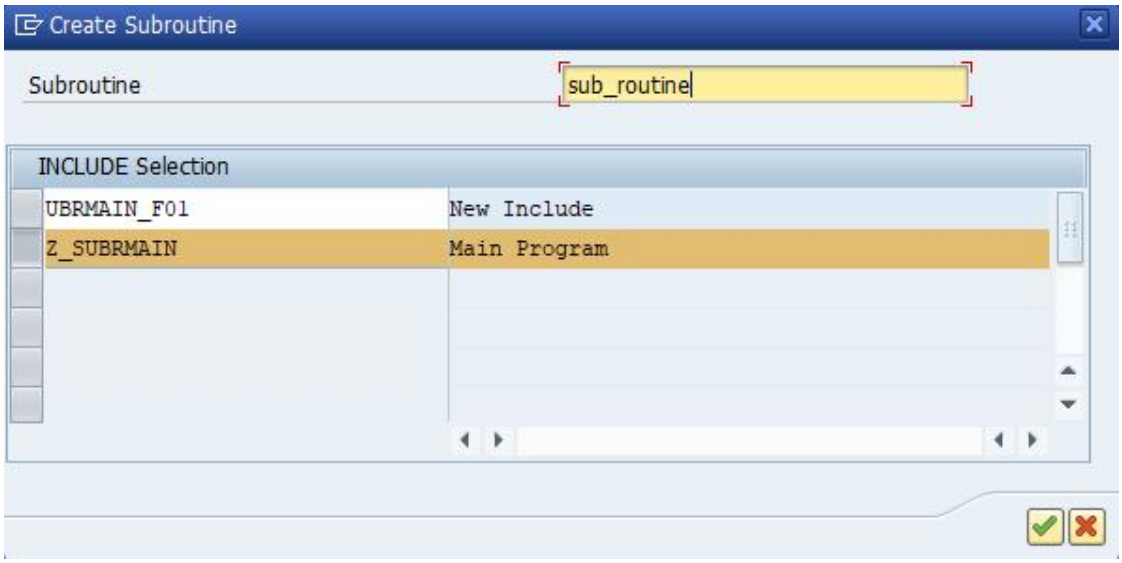
Step-1: Create the main program Z_SUBRMAIN with the below code to perform subroutine named sub_routine.

```
REPORT  Z_SUBRMAIN.
Write 'Before calling subroutine..'.
PERFORM sub_routine.
Write /'After subroutine called....'.
```

Step-2: In this case, Z_SUBRMAIN is the main program. Go to SE80 transaction, select the program and right click on the program to create subroutine (Create Subroutine).



Step-3: Enter the **subroutine name** in the below screen, select the **New Include Z_SUBRMAIN** and click on **Continue** icon.



Step-4: Now, new subroutine **sub_routine** code included in the **Z_SUBMAIN program** like below.

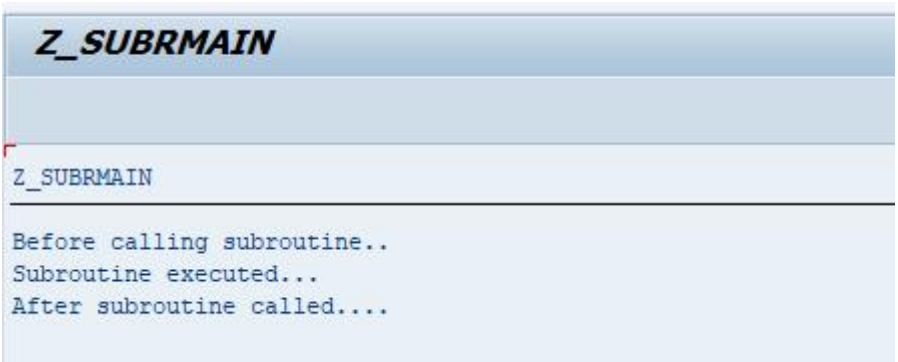
```
REPORT  Z_SUBRMAIN.
Write 'Before calling subroutine..'.
PERFORM sub_routine.
Write /'After subroutine called....'.
*&---------------------------------------------------------------------*
```

```
*&      Form  sub_routine\
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
*  -->  p1        text
*  <--  p2        text
*----------------------------------------------------------------------*
FORM sub_routine .
…
ENDFORM.
```

Step-5: Add the subroutine code in between FORM and ENDFORM. Complete the coding and activate the program to execute.

```
REPORT  Z_SUBRMAIN.
Write 'Before calling subroutine..'.
PERFORM sub_routine.
Write /'After subroutine called....'.
*&---------------------------------------------------------------------*
*&      Form  sub_routine
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
*  -->  p1        text
*  <--  p2        text
*----------------------------------------------------------------------*
FORM sub_routine .
    Write /'Subroutine executed...'.
ENDFORM.
```

Output -



Explaining Example -

In the above example, each and every statement is preceeded with a comment to explain about the statement. Go through them to get clear understanding of example code.

In the example, we have added a subroutine **sub_routine** into the program **Z_SUBRMAIN**.

SAP ABAP Function Modules

What is Function Module?

Function modules are sub programs that contains set of reusable source code statements with importing, exporting parameters and exceptions. Function modules are stored in central library.

Function modules available to the entire system. Function modules can execute independently. If the source code is only used within the same program, then use a Subroutine. Otherwise a function module is preferred.

SAP R/3 system contains wide range of predefined function modules that can be called from any ABAP program.

Every function module is a part of function group. The function group acts as a container for function modules that would logically belong together. Function modules plays significant role in updating the databases.

Function modules plays a key role in remote function calls (RFC) in between SAP R/3 Systems or between an SAP R/3 System and a non SAP system. Function modules support exception handling to handle any errors when the function module is running.

Components of Function Module -

- **Import** - Input parameters of a Function Module.
- **Export** - Output parameters of a Function Module.
- **Changing** - Specifies the parameters act as importing and exporting parameters to a Function Module.
- **Tables** - Specifies internal tables acts as importing and exporting parameters.
- **Exceptions** - Specifies exceptions in Function Modules.

Function Module Creation Process -

The function module creation process involves the below steps -

- Create Function Group. If exist, use function group.
- Create Function Module
- Execute Function Module

Let us discuss function module creation process in ABAP programming with a simple example. In the example, we are triggering the exception when the input2 is ZERO in the function module.

Create Function Group -

To create the function module, the function group should already exist first. If function group not exist, create new function group.

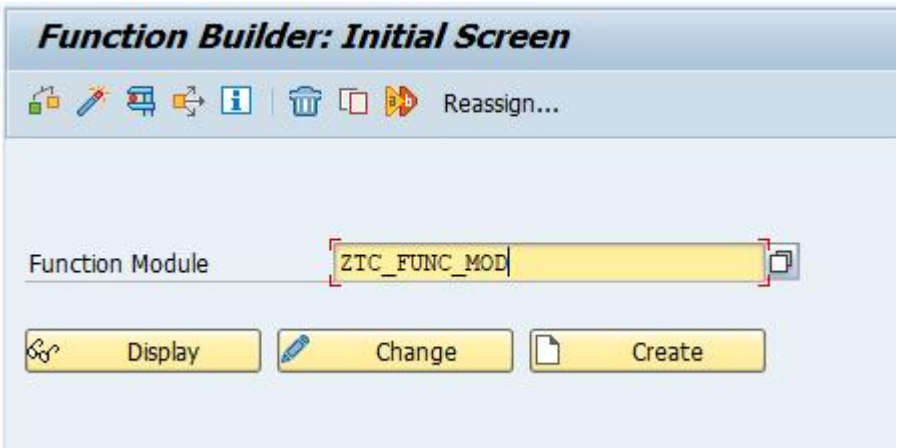
Step-1: Go to **SE37** transaction. From the menu, select **Goto Function Groups Create Group**.

Step-2: Enter the **function group name**, **Short text** and click on "**Save**"

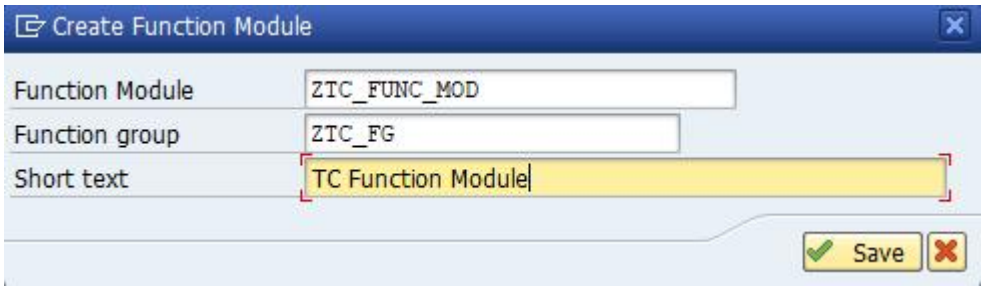
Step-3: Enter the **package details** and click on **Local Object** icon to continue. The status bar displays the "**Function group ZTC_FG created**" message if it successfully created.

Create Function Module -

Step-1: Go to **SE37** to open **Function builder: initial** Screen. Enter the **new function module name** and click on **Create** icon to proceed.

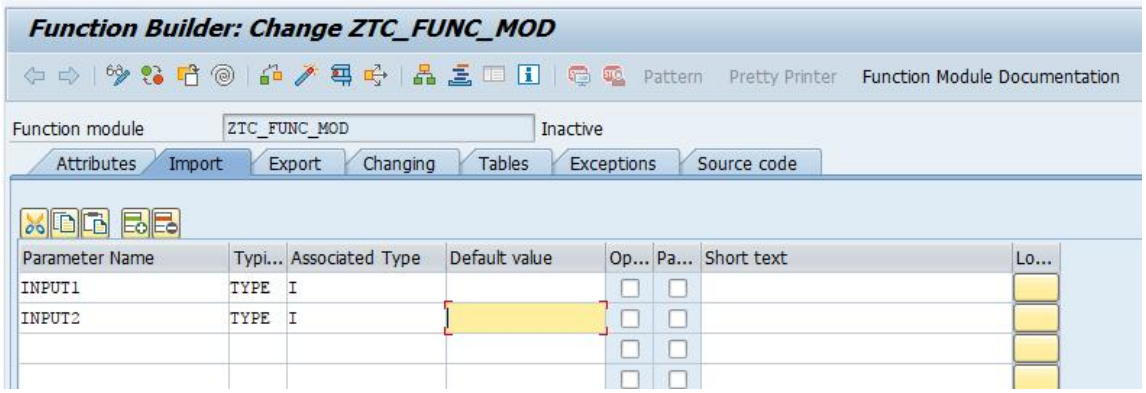


Step-2: Enter the **function group**, **short text** and click on "**Save**" icon.

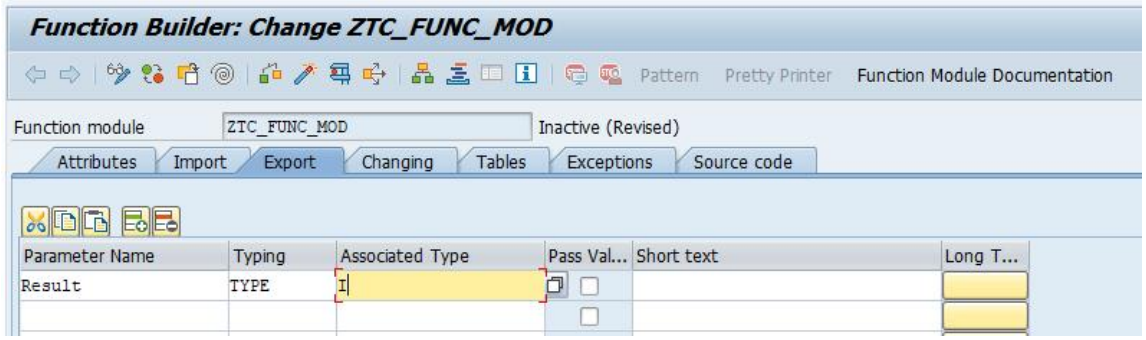


Below informational dialog displays with following message. Click on "**Continue**" icon to proceed.

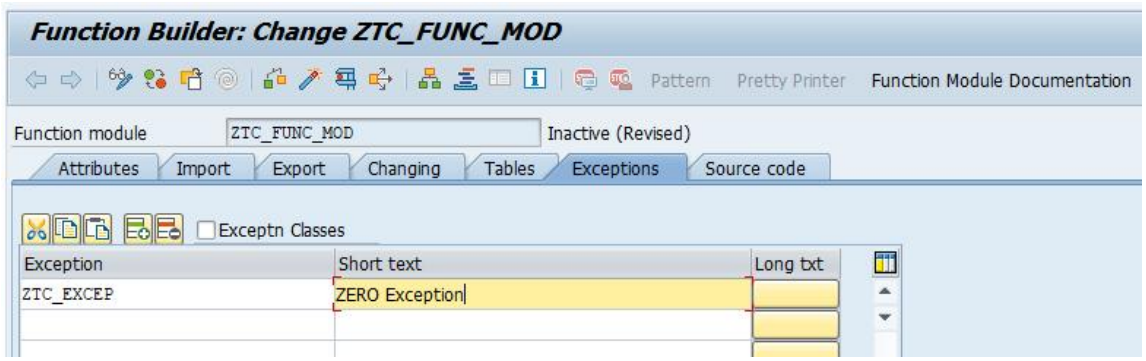
Step-3: Enter the input **Parameters Name** and their **Associated Type** in **import tab**.



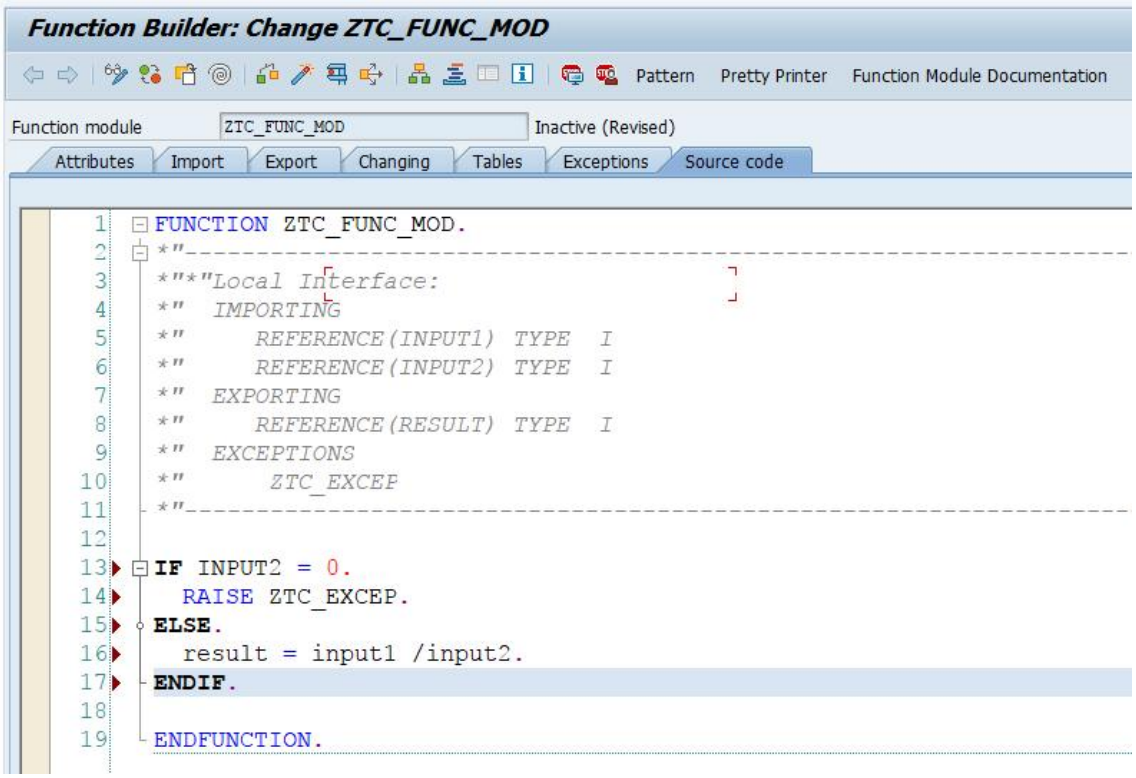
Step-4: Go to **Export tab**, enter output **Parameters Name** and their **Associated Type**.



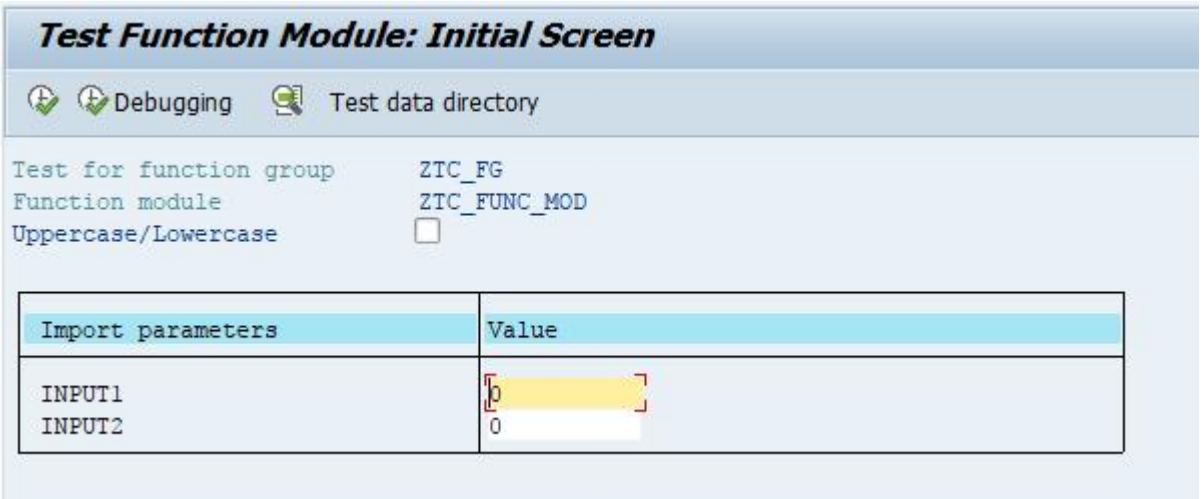
Step-5: Go to **Exceptions tab**, enter the **Exceptions name** and **Short Text** like shown below –



Step-6: Go to **Source code tab**, add **Source Code** in the gap in between **FUNCTION** and **ENDFUNCTION** shown below.

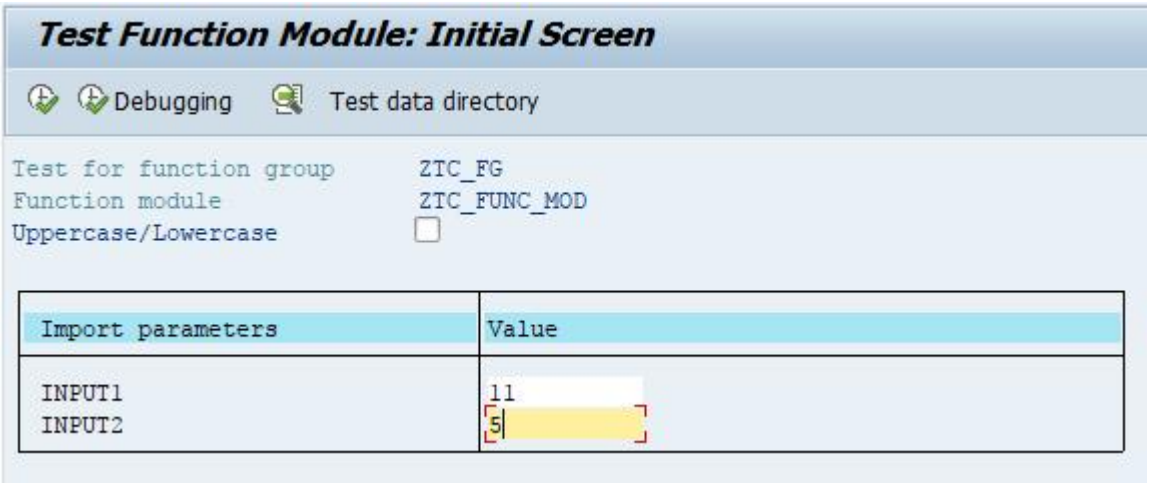


Step-7: Save and **activate** the function module. Once the function module executed, it displays the below screen.

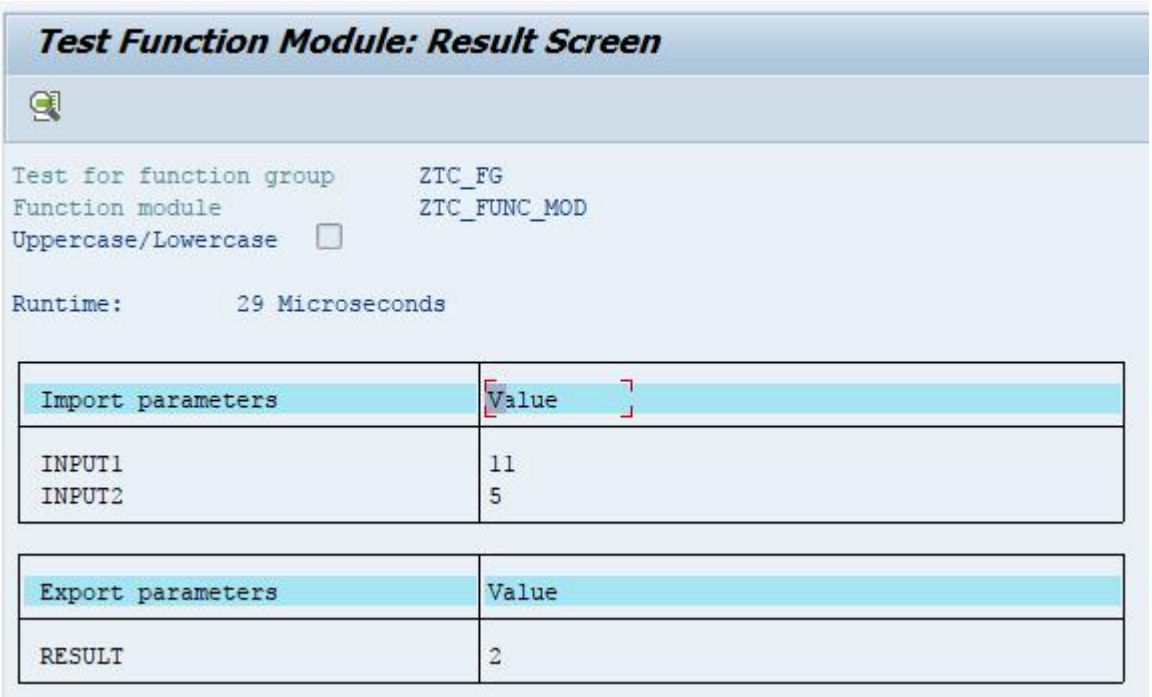


Execute Function Module -

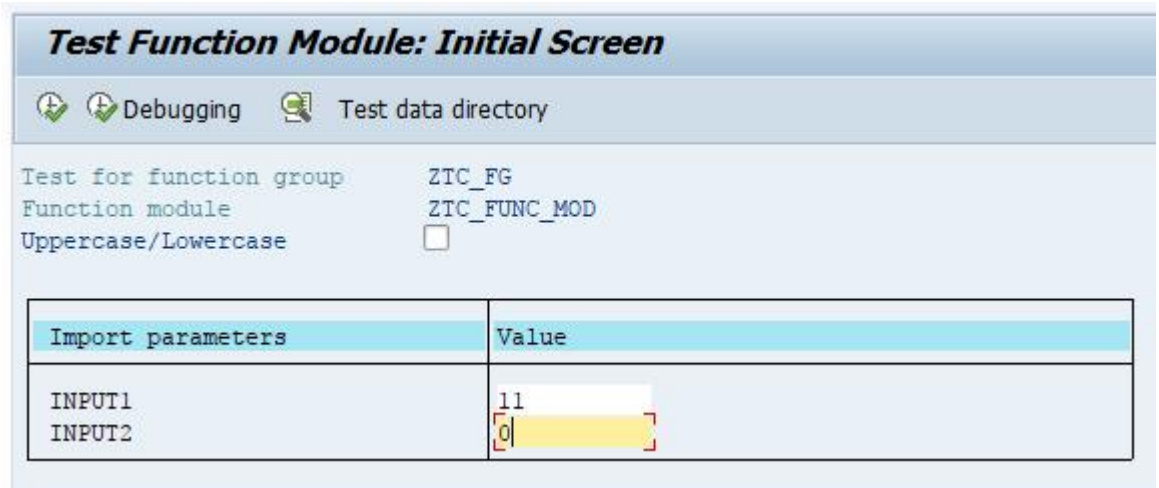
Case-1: Enter the valid **input1** and **input2** to perform the division operation and click on **Execute (F8)** to get the output.



The **output** gets displayed like below -



Case-2: Enter the valid **input1**, **input2** as **ZERO** and click on **Execute (F8)** to trigger the exception.



The **exception** gets displayed like below -