DEBRECENI EGYETEM

Introduction to ABAP Programming

Anikó Vágner

Faculty of Informatics

University of Debrecen

TÁMOP 4.1.1.C-12/1/KONV-2012-0013

Integrált szervezeti és komplex felsőoktatási szolgáltatások, valamint képzések fejlesztése a versenyképes Debreceni Egyetemért

Table of contents

Preface

The readers of this material are supposed to have read and can apply the knowledge of Introduction to ERP and Navigation in SAP material, moreover, to be acquainted with a programming language such as C, Java, C# or PL/SQL, additionally to be familiar with SQL statements.

This material is not comprehensive. Somewhere it provides only simple examples for the language element. For further knowledge you must read the documentation of the ABAP Programming.

The ABAP Programming Language

With the ABAP programming language, business application programs in an SAP environment can be created. The components of an ABAP program can be organized according to their tasks in the layers of a three-tier client-server architecture, which has the presentation, the application, and the database layers.

ABAP is a 4GL language, moreover it is typed, it enables multi-language applications and SQL access, it has been enhanced as an object-oriented language, and it is platform-independent.

Log in to the System

You can write ABAP programs if you log in to the system with a special SAP user which has a developer key provided by the SAP. Certainly, companies have to pay for every new developer key. Additionally, a key can belong to only one SAP user, but more than one programmers can log into the system with the same SAP user at the same time. In this way, it is common that companies buy only a few developer keys, which are used by many programmers at the same time.

One of these special users is the BCUSER, which can be used in the most systems. If you learn ABAP programming on a central SAP server, you should ask your administrator for a username and a password. If you work on your own SAP server, you should check in the following way whether the BCUSER exists in your system.

You should log in with the DDIC or the SAP* users. Both users can perform system administration tasks. If you are not familiar with the administration tasks, please, do not use these users. But now, if you want to check whether BCUSER exists, you should log in with one of them. For these tasks, the DDIC user is better.

So, log in into your system, choose “001” for the Client and “DDIC” for the user, and then type in "SU01" into the transaction field, which executes the User Maintenance transaction. Type in “BCUSER” to the user field and press the Display button (F7).

If the system finds the user, you can change its password on the Logon Data tab.

If the system does not find it, you should create it. Type “BCUSER” into the user field and press the Create button (F8). Fill in all required entry fields on the Address tab, provide the SAP_ALL profile on the Profiles tab and your initial password (usually "initial" is used if you want it to be changed next time) on the Logon Data tab. Then, save (Ctlr+s) your work and log off. Do not forget, that the system will ask you for the access key, when you try to write a program. Ask your administrator for it.

You can find more information about the user management in the SAP Administration material.

Now, you can log in with the BCUSER. If you have set it, the system will ask you to change the password.

Documentation

You can read the ABAP documentation if you execute the ABAPDOCU transaction or you can navigate in the SAP menu: Tools, ABAP Workbench, Utilities and Example Library. You can find other useful transactions in the Utility map, such as the Keyword Documentation (ABAPHELP) and the Demos (DWDM).

In the ABAP Editor, you can get information about a statement if your cursor is on the statement and you press the F1 key or click on the “Help on …” button (Ctrl+F8).

Your First ABAP Program

The SE38 transaction opens the ABAP Editor. You can also open it if you navigate in the SAP menu: Tools, ABAP Workbench, Development, and ABAP Editor. Give

Z_FIRST_ABAP to your program name. Note that every program name should start with the Z or the Y letter (Z is more common). Choose the Source Code from the Subobjects, and click the Create button.

In the next window, you should give the Title of the program, which will be the title of the window of your program, and you should choose the type of your program, choose the Executable program. The other fields are optional, but now choose the Test program in the Status field, and the Basis in the Application field. Then, click on the Save button, click on the Local Object button, in the next window, and in the next window, you can write your code.

In the coding area, you can see comments in a few lines, the REPORT statement, the name of the program and a dot.

The REPORT statement is always the first statement in all executable programs. It is followed by the name of the program.

The dot shows that a statement finishes.

You can write comments into your program in two ways. The first solution is that a line which has an * (asterisk) in the first column is a comment. But, if you use an * anywhere else in a line, the * will have other meanings. The other opportunity is that if you use a " (quotation mark) anywhere in the line, the part of the line which is on the right side of it will be a comment.

Now, type the statement:

```
write 'Hello '.
```

in the next line after the `REPORT  Z_FIRST_ABAP.` The WRITE statement writes the string which follows the statement into the output window.

ABAP statements are not case sensitive.

You can format your code easily if you click on the Pretty Printer button (Shift+F1). You can change its settings in the Utility menu, Settings, Pretty Printer tab.

You can save your program if you press the Save button (Ctrl+S).

You can check the syntax of your program with the Check button (Ctrl+F2). If you have any mistakes, you will get error messages at the bottom of the window, otherwise you will get the following message in the status bar, namely "Program Z_FIRST_ABAP is syntactically correct".

If you click on the Direct processing button (F8), your program will be executed.

If you do not activate your program, other users can use and see it. You can activate it with the Activate button (CTRL+F3).

Variable Declaration and Predefined Elementary Types

The next statement is a simple variable declaration:

```
DATA variable_name TYPE type_name.
```

The name of the variable

- can only include English letters, digits and underscores (_),
- should begin with a letter or underscore (_),
- cannot be longer than 30 characters,
- cannot contain any names of predefined ABAP types or predefined data objects,
- not using reserved ABAP words as variable names is strongly recommended.

First, we consider the predefined elementary ABAP types. They have three categories: character-like types, numeric types, and byte-like types.

The **numeric data types** are used for numeric calculations. They are:

- i, 4-byte integer type. The division of integer numbers rounds the result number rather than truncating it. This type is typically used for counters, quantities, indexes, offsets and time periods.
- p, packed number type. Its length can be between 1 and 16. Its stores two digits in one byte, the last byte contains a digit and the plus/minus sign. There can be maximum 14 decimal places after the decimal separator. You can specify its length with the LENGTH keyword and the decimal places of its fraction part with the DECIMAL keyword. Its default length is 8. It is typically used for values such as lengths, weights, and sums of money.

- decfloat16, decimal floating point number type with 16 decimal places.
- decfloat34, decimal floating point number type with 34 decimal places.
- f, binary floating point number type in 8 bytes. It rounds big numbers, so you should use a relative difference and a predefined limit to test whether two numbers are equal or not. Use it in performance-critical algorithms where the precision is not important.

```
DATA integer01 TYPE i.
DATA packed01 TYPE p LENGTH 4.
DATA packed02 TYPE p LENGTH 4 DECIMALS 1.
```

The **predefined elementary character-like types** are:

- c, character type, you can specify its length up to 262 143 characters,
- n, numeric character type. It can store only digits, you can specify its length up to 262 143 characters. It is not used for calculations. You can store for example account numbers, article numbers, postal codes in it.
- d, date field, its format is ‘yyyymmdd’,
- t, time field, its format is ‘hhmmss’,
- string, sequence of character with variable length.

```
DATA text01 TYPE c LENGTH 30 .
DATA date01 TYPE d.
```

If you use a variable to store input data of your program, it is recommended that you should use c (character) type, because it has a fix length.

The **predefined byte-like types** are:

- x, a byte in hexadecimal. You can specify its length up to 524 287 bytes.
- xstring, sequence of bytes with variable length.

You can declare a variable that you use the type of a variable in your program with the LIKE keyword:

```
DATA integer02 LIKE integer01.
```

You can use the VALUE keyword to specify an initial value for the variable:

```
DATA integer03 TYPE i VALUE 5.
```

You can use other types in the declaration, namely reference data types, complex data types like structures and tables, the types in the ABAP Dictionary and data types which you have declared.

Declare Constant

You can declare constants with the CONSTANTS keyword. If you use it, you should specify initial value with the VALUE keyword.

```
CONSTANTS const1 TYPE p DECIMALS 1 VALUE '6.6'.
```

Literals

Numeric literals consist of digits with an optional plus or minus sign. It has no fractional part. If you need non-integer values, you should use text field literals. The following examples can be converted into correspondent numeric types: '123'; '+122'; '-045'; '-34.023'; '12.2E21'; '-45E-2'.

A **text field literal** is enclosed by apostrophes ('), its data type is the c (character) type, its maximum length is 255 characters and its minimum length is a character, which means that the literal '' is equivalent to the literal ' '. If you want to store an apostrophe inside the literal, you should use double apostrophes, for example 'It''s raining.'

A **text string literal** is similar to the text field literal, but it is enclosed by backquotes (`), its data type is string, and it can be an empty string (``).

Assign Values to Variables and Initializations

You can assign a value to a variable with the MOVE statement or the = operator. The = operator is enclosed by spaces. For example:

```
MOVE 2 TO integer01.
integer02 = 3.
packed01 = 2.
packed02 = '+23.3'.
MOVE '2015.01.01' TO date01.
```

The CLEAR statement assigns its initial value to a variable:

```
CLEAR integer03.
```

Expressions, Operators and Precedence Tables

There are four types of expressions: logical, arithmetical, string and bit expressions. You cannot mix the arithmetical, string and bit expressions in a calculation. You can use brackets (()) in the expressions.

Arithmetic expressions

Table 1 shows the precedence table of the arithmetic operators and the order in which the calculation is performed. The ** operator has the highest priority. The arithmetic operators should be enclosed by spaces.

Table 1 – Precedence table

Operator	**Calculation**	**Order**
**	Power	From right to left
*, /, DIV, MOD	Multiplication, Division, Integer part of the division, Positive remainder of the division	From left to right
+, -	Addition, Subtraction	From left to right

String expressions

The && operator makes a string which is a concatenation of two character-like operands.

Logical expressions

You can use relational operators (=, <>, <, >, <=, >=, etc.), Boolean operators (AND, OR, NOT, EQUIV) and predicates (BETWEEN, IN, IS) in the logical expressions. The relation operators should be enclosed by spaces.

Bit expression

You can use bit operands, like BIT-AND, BIT-OR, BIT-XOR, etc.

You can find further information about the logical and bit expression in the documentation.

The First Selection Screen

The PARAMETERS statement allows input parameters in your executable program. If you use this statement and execute your program, the system will automatically create a selection screen where the users can type their input values. The output of the program will be on a list, which is generated automatically if the program contains at least one WRITE statement.

The syntax of the PARAMETERS statement is similar to the DATA statement.

```
PARAMETERS pint01 TYPE i.
PARAMETERS pchar02 TYPE c LENGTH 20.
PARAMETERS pdate03 TYPE d.
PARAMETERS pchar04 LIKE pchar02.
```

The maximum length of a parameter name is 8 characters.

The PARAMETERS statement declares a variable in the program. When the program is executed, the system creates a selection screen, and places an input field with the same name and data type of each declared parameter.

You can add a default value to a parameter in the next way:

```
PARAMETERS pdate03 TYPE d DEFAULT '2015.04.01'.
```

The default value appears in the input box, and the users can change it if they want.

If you use the OBLIGATORY keyword in the PARAMETERS statement, the field will be mandatory:

```
PARAMETERS pint01 TYPE i OBLIGATORY.
```

You can create checkboxes on selection screens:

```
PARAMETERS pch01 AS CHECKBOX.
```

If the value of the parameter is ‘X’ or ‘x’, the checkbox is selected, otherwise it is not selected. The parameter type is c and its length is 1.

You can also use radio buttons on selection screens:

```
PARAMETERS ryes    RADIOBUTTON GROUP r1.
PARAMETERS rno     RADIOBUTTON GROUP r1 DEFAULT 'X'.
PARAMETERS rdontno RADIOBUTTON GROUP r1.
```

A radio button is selected if the value of the parameter is "X" or "x". Otherwise, it is not selected. The parameters with the same group name belong to a radio button group. Its name can be maximum 4 characters long. The parameter type is c and its length is 1. In a radio button group, only one parameter can have default value, which should be “X”.

The next example shows a simple program which asks two numbers as input parameters and writes their sum and their product to the screen:

```
REPORT  Z_SUM_PRODUCT.
PARAMETERS: a type i,
            b type i.
data s type i.
data p type i.
s = a + b.
p = a * b.
write: 'Sum: ' , s.
write: 'Product: ' , p.
```

When you execute it and you are ready with filling in the input fields, you should press the Execute (F8) button in order that it will write out the results.

In the example, **chained statements** are used. If you need multiple ABAP statements with the same beginning part, you can use one chained statement instead of many single statements, namely after the beginning part, use a colon (:) and separate the remaining parts with commas. For example:

```
write: 'Product: ' , p.
```

Text Elements of a Program

If you execute the previous example, you can see that the variable names are used to field names. To give more information about the expected field, you can change them in Goto menu, Text elements and Selection texts after your program has been activated. There will be the names of the parameters of your program in the name column, and you can write informative names into the text column or you can get information about the field from the Dictionary. After you have set the texts, you should activate the text elements.

The text elements can easily be changed if they need to be translated to another language, and the program code does not need to be rewritten.

Classic Lists

The classic list or the list contains text outputs defined with ABAP statements. In our examples, we use them, because they are very simple programming techniques, but the SAP documentation does not recommend using it, instead, you should use ALV techniques, wrappers of the Browser Control or Textedit Controls.

The following statements of the list are used in this material:

`WRITE 'apple'.` statement writes the 'apple' string to the classic list. You can also use it with other data types.

`WRITE 'apple' QUICKINFO 'is a kind of fruit'.` statement assigns a quick info to the output. If the mouse cursor is on the output area of the 'apple' string, the info will appear.

`ULINE` statement outputs a horizontal line in a list.

`WRITE /.` statement makes an new line.

`WRITE / 'apple tree'.` statement writes the 'apple tree' string into a new line.

`NEW-LINE` statement creates a new line.

Text Symbols

You can replace literals in a program, which support the multi-language environment. You can declare a text symbol in the Goto menu, Text elements and Text symbols. The Sym should be a three-digit number (for example 001), which will be used in the program code, and the Text (for example 'banana') will appear in the screen, which can be maximum 132 character long. In the program code, you can refer to the symbol as text-001, namely, text symbols are stored in a structure called text, and the three digit numbers are its fields.

`WRITE text-001.` statement shows the 'banana' string in the list.

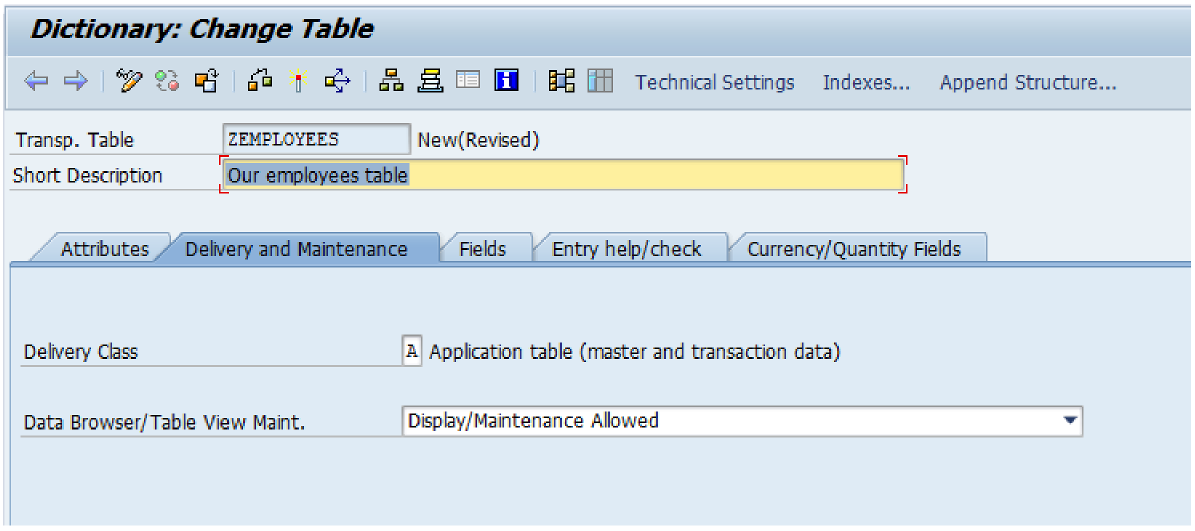
The Data Dictionary

You can search, create, alter, and drop tables, views, data types and some other objects with the data dictionary tool. You can find many objects, which you may enhance. You can open the Data Dictionary tool with the SE11 transaction or from the SAP menu: Tools, ABAP Workbench, Development and ABAP Dictionary.

Create a table

In order to create a database table, write its name into the Database table field and press create. The name of a table have to begin with Z or Y letter (Z is more common). Our first example is the ZEMPLOYEES table.

Figure 1 shows the "Dictionary: Change Table" window with the Delivery and Maintenance tab, where you have to choose a delivery class. You have to choose the Application table (master and transaction data) for our simple examples. This table type is called transparent table and you can consider it as a relational database table.



Transp. Table ZEMPLOYEES New(Revised)
Short Description Our employees table

Delivery Class A Application table (master and transaction data)
Data Browser/Table View Maint. Display/Maintenance Allowed

Figure 1 – Dictionary: Change Table window

In the Data Browser/Table View Maint. field you can choose one from three values. If you determine "Display/Maintenance Allowed" for this field, users (if they have the privilege) can display and maintain the table with the Data Browser (SE16), the Maintain Table Views (SM30 and SM31) and the Generate Table Maintenance Dialog (SE54) tools. If you choose "Display/Maintenance Allowed with Restrictions", the users will be able to display the table in the Data Browser (SE16) and can generate a maintenance dialog with SE54 transaction. If you choose the "Display/Maintenance Not Allowed", the users will not be able to perform none of the previous transactions on this table.

Choose "Display/Maintenance Allowed" for the Data Browser/Table View Maint. field for ZEMPLOYEES table.

You should also give a short description of the table.

Although you are not ready, you can save your table now. So, press the Save button (Ctrl+S). In the next window, click on the Local Object button, which means that the table will exist only in the development system, and it will not be transported.

In the Fields tab you can add fields or columns to the table. The first field should be the "Client" field, data element of which should be "MANDT" and it should be a part of the primary key, so the key box should be checked. This field will be automatically managed by the system. In this field, the system will apply the client number which the user who selects or changes this table uses. The transparent tables are always client-specific (or mandant-dependent).

The table should have a primary key (in most cases this is compound), so you should check the key boxes of the fields which are parts of the primary key. The key fields should be the first fields in the field list.

You should give a data element in the data element column or a predefined type with length if it is necessary in the Data Type, Length and Decimal Places columns. You can change between the two data types with the "Data Element"/"Predefined Type" button.

The predefined type should be one of the built-in types, which you can choose from a list if you press the F4. Similarly, you can choose data element from a list if you press F4, but you can also create a new one if you write its name into the appropriate field, and double click on it.

You can add a short description to a field.

Before you activate the table, you should maintain the technical settings with the "Technical settings" button. You should choose a data class, in our examples it could be the "Master Data, Transparent Tables", and a size category, where you estimate how many rows will be in the table.

If you are ready with the table definition, save it (Ctrl+S), check it (Ctrl+F2), and if there are not any problems, you can active it, which means that the system creates the table in the database, and after the creation, it can be used, namely you can insert rows into it, update its rows and delete rows from it, and the table can be referred by a program.

Data elements, domains and data types

A field of a table needs a data type in a way. The simplest way is if you choose a predefined type for it, but it is not suggested because there would be no more information about the field. A better solution is if you create a data element, which can have field labels, which can appear in your report where you refer to it. You can also give search information to the data elements. Moreover, if you want to use a combination of a data type and a length in your programs several times, or you need to give range values to your data type, you can create a domain. If you change the domain, every data element type which has this domain will change.

In order to create a data element, you can open the Data Dictionary tool with the SE11 transaction, choose the Data type field, write there its name, and press the Create button. The other solution is when you are creating a table, you can write the name of the new data element into the data element column, and double click on it. Choose the "Data element" option and press enter.

In the "Dictionary: Change Data Element" window, you have to give a Short description of the data element, then, you can choose a domain or a predefined type for your data element. The predefined type has to be chosen from the built-in list, whereas the domain can be built-in or you can create it. In the "Further Characteristics" tab you can give search help to your data element, and in the Field Label tab you can give field labels, which will appear in the screens using this data element.

If you are ready, save it (Ctrl+S), choose the "Local Object" button, check it (Ctrl+F2), and activate it (Ctrl+F3) in order that it can be used in the system.

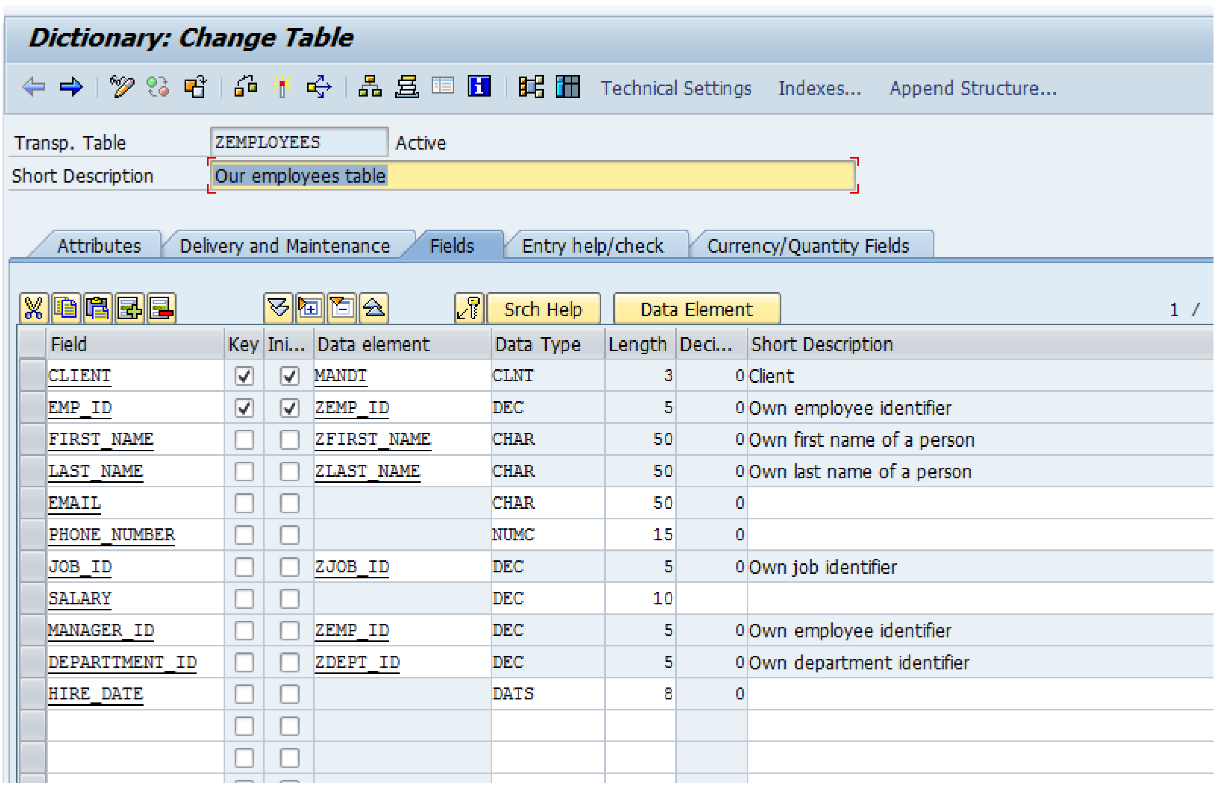
To create a domain, you can open the Data Dictionary tool with the SE11 transaction, choose the Domain field, write there its name, and press the Create button. The other solution is if you are creating a data element, you can write the new domain name into the domain field, and double click on it.

In the "Dictionary: Change Domain" window you should give a Short description of the domain, then, you should choose a data type, and give a length. In the "Value Range" tab you can give constraints for the values as single values, intervals or a value table.

If you are ready, save it (Ctrl+S), choose “Local Object” button, check it (Ctrl+F2), and activate it (Ctrl+F3) in order that it can be used in the system.

The ZEMPLOYEES table

In the material, the ZEMPLOYEES table is used. *Figure 2* shows its definition.



Transp. Table: ZEMPLOYEES Active
Short Description: Our employees table

Field	Key	Ini...	Data element	Data Type	Length	Deci...	Short Description
CLIENT	✓	✓	MANDT	CLNT	3	0	Client
EMP_ID	✓	✓	ZEMP_ID	DEC	5	0	Own employee identifier
FIRST_NAME			ZFIRST_NAME	CHAR	50	0	Own first name of a person
LAST_NAME			ZLAST_NAME	CHAR	50	0	Own last name of a person
EMAIL				CHAR	50	0	
PHONE_NUMBER				NUMC	15	0	
JOB_ID			ZJOB_ID	DEC	5	0	Own job identifier
SALARY				DEC	10		
MANAGER_ID			ZEMP_ID	DEC	5	0	Own employee identifier
DEPARTMENT_ID			ZDEPT_ID	DEC	5	0	Own department identifier
HIRE_DATE				DATS	8	0	

Figure 2 – Creating the ZEMPLOYEES table

The MANDT is a built-in data element, ZEMP_ID, ZJOB_ID, ZDEPT_ID, ZFIRST_NAME and ZLAST_NAME are your own data elements which have your own domains.

Insert records into a table

In the “Dictionary: Change Table” window, choose the Utilities menu, Table Contents and Create Entries. If you fill the fields labelled by the technical names of the table fields, and the Save button (Ctrl+S), you will insert a row into the table. If you want to see the field labels, open User Parameters in the Settings menu, and choose Field label instead of Field name for the Keyword.

Display content of a table

In the “Dictionary: Change Table” window, choose the Utilities menu, Table Contents and Display. You get a selection screen where you can use filters. Press the Execute (F8) button to see the content of the table corresponding to the filters. Of course, you do not need to give any filters; in this case, all rows of the table will be displayed.

You can double click on a row to see all data of it. Moreover, you can use the toolbars buttons for further actions.

Other opportunity can be the Data Browser (SE16 transaction).

Copy and delete a table, a data type or a domain

First, you have to find the object in the Data Dictionary tool (SE11), which means that its name has to be there in the appropriate field.

On the toolbar, you can find a Copy (Ctrl+F5) button, which will create a copy of your object. If you copy a table, the system will only create an empty table structure, so there will be no data.

You can delete only the customer-specified objects, and you cannot delete the built-in objects. Before you delete an object, press the “Where-used-list” (Ctlr+Shift+F3) button to find other objects, like programs and tables, which refer to the object you want to delete. If you really want to delete an object, press the delete (Shift+F2) button.

Add and delete fields of a table

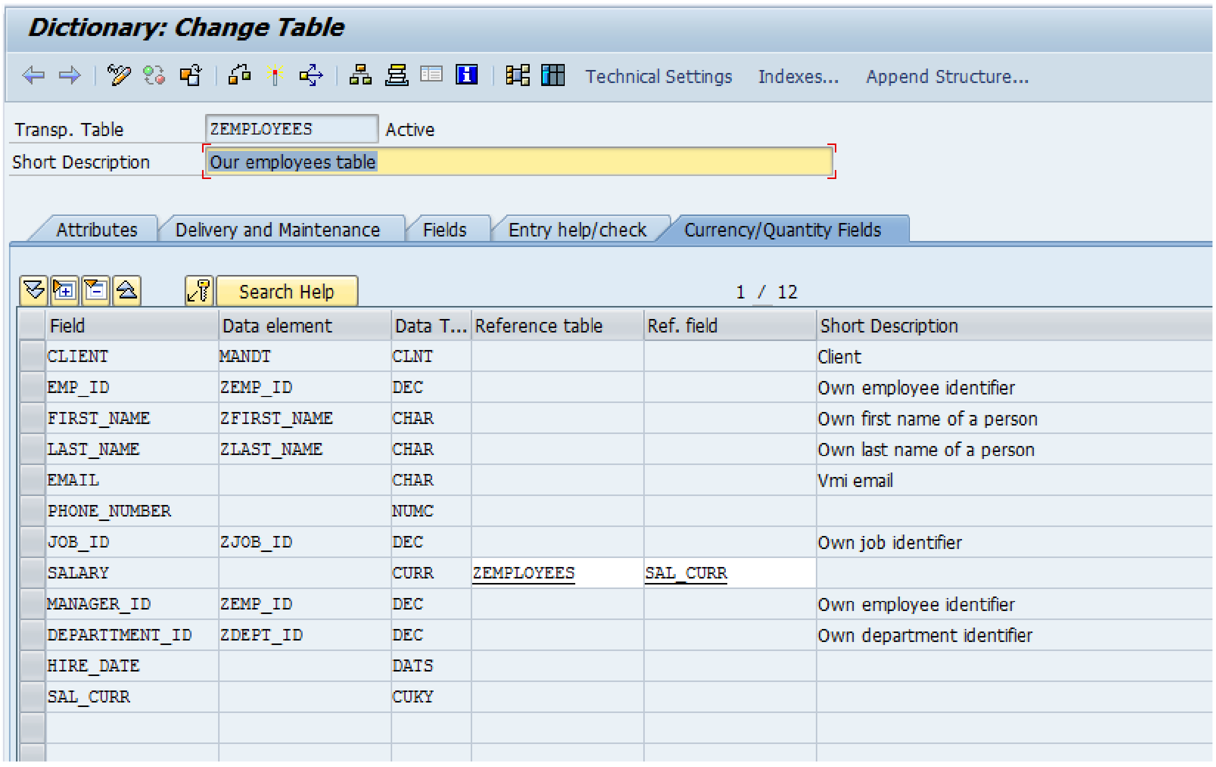
Open the Data Dictionary tool (SE11), write the table name you want to alter in the Database table field, and press the Change button.

If you want to delete a row, right click on the row and choose the delete line.

If you want to add a row, you can write it to the end of the list, or with a right click on a line you can insert a new line into the field definitions.

Currency

The SALARY field should have a currency; consequently, its type should be CURR. Change it, and define the length, too. But, it defines only the amount but not the type of the currency. To define what the currency is, you need another field named SAL_CURR with type CUKY, and the SALARY field should refer to it. Click on the "Currency/Quantity Fields" tab and set the Reference table and Ref. field columns in the row of the SALARY field. See *Figure 4*.



Transp. Table	ZEMPLOYEES Active
Short Description	Our employees table

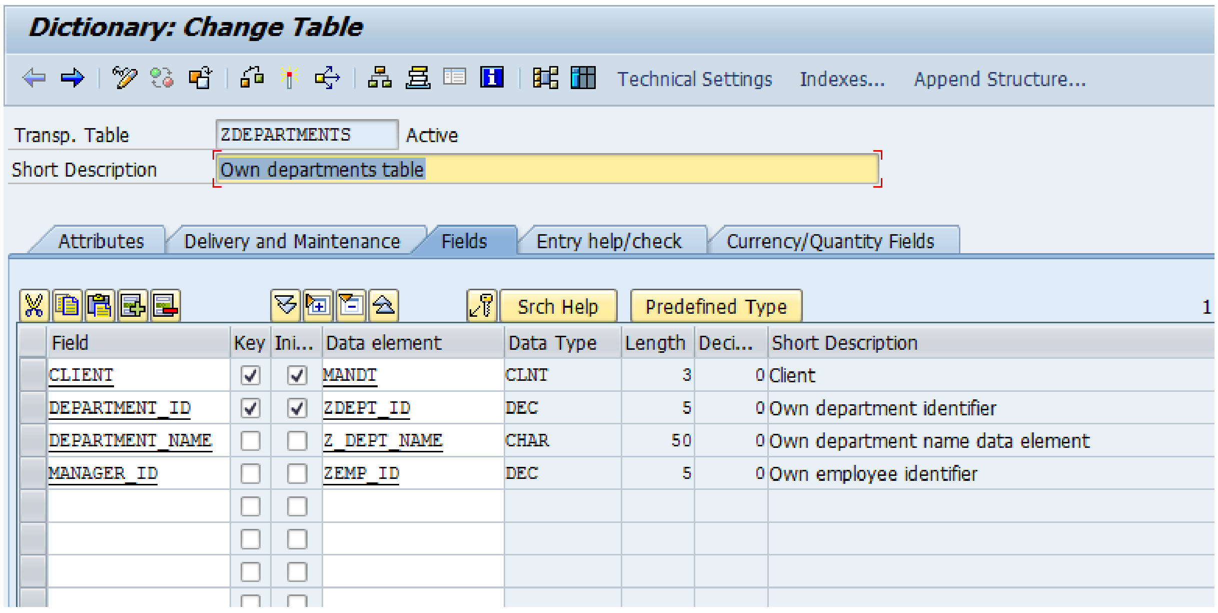
Field	Data element	Data T...	Reference table	Ref. field	Short Description
CLIENT	MANDT	CLNT			Client
EMP_ID	ZEMP_ID	DEC			Own employee identifier
FIRST_NAME	ZFIRST_NAME	CHAR			Own first name of a person
LAST_NAME	ZLAST_NAME	CHAR			Own last name of a person
EMAIL		CHAR			Vmi email
PHONE_NUMBER		NUMC			
JOB_ID	ZJOB_ID	DEC			Own job identifier
SALARY		CURR	ZEMPLOYEES	SAL_CURR	
MANAGER_ID	ZEMP_ID	DEC			Own employee identifier
DEPARTMENT_ID	ZDEPT_ID	DEC			Own department identifier
HIRE_DATE		DATS			
SAL_CURR		CUKY			

Figure 4 – Currency reference

If you activate your modification, the system may ask you to convert your table with the SE14 transaction. To do this easily, open the Utilities menu, Database Objects, Database utility (or execute the SE14 transaction). In the new window, press the button activate and adjust database.

The ZDEPARTMENTS and the ZJOBS tables

In *Figure 5*, you can see the definition of the ZDEPARTMENTS table, whereas in *Figure 6*, the ZJOBS table.

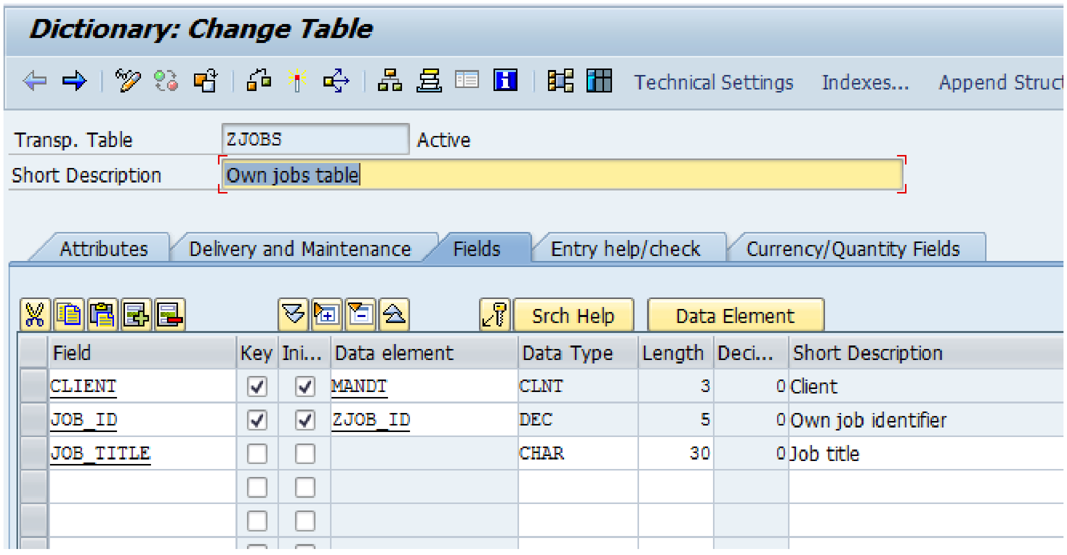


Transp. Table: ZDEPARTMENTS Active
Short Description: Own departments table

Field	Key	Ini...	Data element	Data Type	Length	Deci...	Short Description
CLIENT	✓	✓	MANDT	CLNT	3	0	Client
DEPARTMENT_ID	✓	✓	ZDEPT_ID	DEC	5	0	Own department identifier
DEPARTMENT_NAME			Z_DEPT_NAME	CHAR	50	0	Own department name data element
MANAGER_ID			ZEMP_ID	DEC	5	0	Own employee identifier

Figure 5 – Definition of the ZDEPARTMENTS table

The data element of the DEPARTMENT_ID has the same data type as the data type of the DEPARTMENT_ID of the ZEMPLOYEES table, whereas MANAGER_ID has the same data type as the data type of the EMP_ID in the ZEMPLOYEES table. The Z_DEPT_NAME is a new customer-specified data element.



Transp. Table: ZJOBS Active
Short Description: Own jobs table

Field	Key	Ini...	Data element	Data Type	Length	Deci...	Short Description
CLIENT	✓	✓	MANDT	CLNT	3	0	Client
JOB_ID	✓	✓	ZJOB_ID	DEC	5	0	Own job identifier
JOB_TITLE				CHAR	30	0	Job title

Figure 6 – Definition of ZJOBS table

The data element of the JOB_ID has the same type as the data type of the JOB_ID of the ZEMPLOYEES table.

Foreign keys

The SAP does not suggest creating foreign keys in database level, because the used database management systems can be various. Instead, click on the Entry help/check tab, click on foreign keys and fill the Check table, Origin of the input help and Srch Help fields.

Branch or Selection Control Structures

In the ABAP programming, you can use the IF and the CASE statement as a branch or selection statement.

IF statement

```
IF logical_expression_1.
  [statements_1]
[ELSEIF logical_expression_2.
  [statements_2]]…
[ELSE.
  [statements_n]]
ENDIF.
```

The first statements_i for which the logical expression is true will be executed. If none of the logical expressions are true, the statements after the ELSE keyword will be executed, if they exist; otherwise, the IF statement does nothing.

There can be several statements in a branch.

```
REPORT  Z_IF.
parameters sales type i.
data bonus  type i value 0.

IF sales > 50000.
   bonus = 1500.
 ELSEIF sales > 35000.
   bonus = 500.
 ELSE.
   bonus = 100.
ENDIF.

WRITE: 'Sales = ' , sales, ', bonus = ' , bonus.
```

CASE statement

```
CASE operand.
  [WHEN operand1 [OR operand2 [OR operand3 [...]]].
    [statements_1]]
    ...
  [WHEN OTHERS.
    [statements_n]]
ENDCASE.
```

If the operand after the CASE keyword is equal to one of the operands after the first WHEN, the statements_1 will be executed, otherwise, it examines the operands in the next WHEN. If it does not find any equal operands, the statements after the WHEN OTHERS will be executed, if it exists. Only one branch will be executed. There can be several statements in a branch. The following example asks for a grade and writes out the text of the grade.

```
REPORT  Z_CASE_STATEMENT.
parameters grade type i.
case grade.
  when 1.
    write 'Fail'.
    write /'You should learn it again'.
  when 2.
    write 'Pass'.
    write /'You can learn the next subjects'.
  when 3.
    write 'Satisfactory'.
    write /'You can learn the next subjects'.
  when 4.
    write 'Good'.
    write /'You can learn the next subjects'.
  when 5.
    write 'Excellent'.
    write /'You can learn the next subjects'.
  when others.
    write 'Grade is not known'.
endcase.
```

Loop (Iteration) Control Structures

In the ABAP programs, you can use the following loops:

- DO – ENDDO,
- WHILE – ENDWHILE,
- LOOP – ENDLOOP (reads rows from an internal table)
- PROVIDE – ENDPROVIDE (loops through internal tables)

- SELECT – ENDSELECT (loops through the result set of a database access)

DO – ENDDO loop

```
DO [n TIMES].
  [statement_block]
ENDDO.
```

The n TIMES addition can limit how many times the statements in the loop will be executed. n is a numerical expression typed i. The control structure does not take changes into account to value n within the loop. If there is no n TIMES clause, the loop will be infinite loop. You can exit the loop with the EXIT statement.

In the statements, you can use the sy-index system field, which contains the number of repetitions. In nested loops, sy-index always refers to the current loop.

```
REPORT  Z_DO.
do 3 times.
 write sy-index.
enddo.
```

WHILE – ENDWHILE loop

```
WHILE logical_expression.
  [statement_block]
ENDWHILE.
```

The statement_block is repeated as long as the logical expression is true, or until the loop is exited with an EXIT statement. You can use sy-index in the statements in the loop.

```
REPORT  Z_WHILE.
parameters sz type i.

if sz = 0.
  write 0.
  else.
  if sz < 0.
    write -1.
    sz = sz * ( -1 ).
  endif.

  data c type i value 2.
  while sz <> 1.
    if sz mod c = 0.
        sz = sz / c.
        write c.
      else. c = c + 1.
    endif.

  endwhile.
endif.
```

EXIT statement

```
EXIT.
```

The EXIT statement in a loop leaves the loop by ending the current loop process. The program execution is continued after the closing statement of the loop.

```
REPORT  Z_EXIT.
parameters x type i.
data: c type i value 1,
      sz type i value 0.

if x <= 0.
  write 'Only positive numbers'.
else.

  do.
    sz = sz + c.
    if sz + c + 1 > x.
      exit.
    endif.
    c = c + 1.
  enddo.
  write c.
endif.
```

CHECK statement

```
CHECK logical_expression.
```

If the logical expression in the CHECK statement in a loop is not true, it exits the current iteration of a loop and transfers control to the next iteration of the loop.

CONTINUE statement

```
CONTINUE.
```

The CONTINUE statement in a loop exits the current iteration of a loop and transfers control to the next iteration of the loop.

```
REPORT  Z_CONTINUE.
parameters p type i.
do p times.
  if sy-index mod 2 <> 0.
    continue.
  endif.
  check sy-index mod 3 = 0.
  write sy-index.
enddo.
```

Structures

You can declare a structure type in the following way:

```
TYPES: BEGIN OF countries,
  c_id type i,
  c_name type c length 40,
  c_continent type c length 30,
 END OF countries.
```

The field declaration is similar to a variable declaration, so you can give length information and you can also use the LIKE keyword. A structure should have at least one element. Structures can be nested.

```
TYPES: BEGIN OF address_type,
         name TYPE c LENGTH 30,
         street TYPE street_type,
         BEGIN OF city,
           zipcode TYPE n LENGTH 5,
           name TYPE c LENGTH 40,
         END OF city,
       END OF address_type.
```

You can define a structured variable directly using DATA keyword instead TYPES keyword.

You can declare a structured type variable not only with a locally declared type, but with a dictionary structure or a transparent table structure. In the following example, zemployees is a transparent table (created earlier), whereas zname is a dictionary structure (you can create it as a data type in the SE11 transaction).

```
data: vc1 type countries,
      vc2 type countries,
      ve type zemployees,
      vn type zname.
```

You can refer a field using a hyphen:

```
vc1-c_id = 1.
vc1-c_name = 'Hungary'.
vc1-c_continent = 'Europe'.
```

The MOVE-CORRESPONDING statement copies the contents of a source structure to a target structure. The two structures do not need to have the same type.

```
MOVE-CORRESPONDING struc1 TO struc2.
```

The system searches for all components with the same name in struc1 and struc2 and the contents of the components in struc1 are assigned to the components with the same name in struc2. Other components are not affected.

```
move-corresponding vc1 to vc2.

write: vc2-c_id, vc2-c_name, vc2-c_continent.
```

Internal Tables

Internal tables are dynamic objects that consist of a sequence of elements of the same data type. An element in an internal table can have any data type. The internal tables are used for storing and formatting data from database tables in a program. They exist while a program is running.

The line type is the type of the data elements in the internal table. An element has fields; consequently, you can refer to values of one field as a column.

The key of an internal table consists of key fields. It is used for sorting. The key can be unique or non-unique.

There are three kinds of internal tables: standard, sorted and hashed.

A standard table is administered internally using an index. It has a non-unique key. The elements of it can be accessed by using the index or a key. The response time for accessing the table using a key is proportional to the number of entries in the table. You can define other keys to make key access to standard tables more efficient. If you do not define any keys for a standard table, the system automatically defines a non-unique key. You can fill data into a standard table quickly, because the system does not need to check the duplicate records.

A sorted table is also administered internally using an index. It has a unique or a non-unique key. It is sorted in ascending order of the key. It can be accessed by using either the index or the key. The response time for accessing the table using the key is logarithmically proportional to the number of table entries, since a binary search is used. If you modify a sorted table, the system will automatically sort it.

A hash table is administered internally using a hash function. It has a unique key. It can be accessed by using the key. The response time for the key access is constant and independent of the number of entries in the table. If you need to search by key in a table quickly, this type of internal table can be a good solution.

The type of an internal table can be declared locally in a program or in the dictionary.

```
data st_tab1 type standard table of countries with non-unique key c_id.
data st_tab2 type standard table of countries.
data so_tab1 type sorted table of countries with non-unique key c_id.
data so_tab2 type sorted table of countries with unique key c_id.
data ht_tab1 type hashed table of countries with unique key c_id.
```

Processing Statement for Internal Tables

Filling internal tables

INSERT

INSERT statement adds one or more rows to an internal table. You can define the position of the new element in the internal table.

```
REPORT  Z_INTERNAL_INSERT.

DATA: BEGIN OF line,
            col1 TYPE i,
            col2 TYPE i,
          END OF line.

DATA: itab LIKE STANDARD TABLE OF line,
      jtab LIKE itab,

      itab1 LIKE TABLE OF line,
      jtab1 LIKE itab,
      itab2 LIKE STANDARD TABLE OF line,
      jtab2 LIKE SORTED TABLE OF line
            WITH NON-UNIQUE KEY col1 col2.
```

```
* Fill table

DO 3 TIMES.
  line-col1 = sy-index.
  line-col2 = sy-index ** 2.
  APPEND line TO itab.
  line-col1 = sy-index.
  line-col2 = sy-index ** 3.
  APPEND line TO jtab.
ENDDO.

* Insert a single line into an index table

MOVE itab TO itab1.

line-col1 = 11. line-col2 = 22.
INSERT line INTO itab1 INDEX 2.

*   INSERT INITIAL LINE INTO itab1 INDEX 1.

LOOP AT itab1 into line.
  WRITE: / line-col1, line-col2.
endloop.

* Insert lines into an index table with LOOP

MOVE itab TO itab1.

LOOP AT itab1 INTO line.
  line-col1 = 3 * sy-tabix. line-col2 = 5 * sy-tabix.
  INSERT line INTO itab1.
ENDLOOP.

LOOP AT itab1 into line.
  WRITE: / line-col1, line-col2.
endloop.

* Insert lines into an index table

MOVE itab TO itab1.
MOVE jtab TO jtab1.

INSERT LINES OF itab1 INTO jtab1 INDEX 1.

LOOP AT jtab1 into line.
  WRITE: / line-col1, line-col2.
endloop.
* Insert lines into a sorted table

MOVE itab TO itab2.
MOVE jtab TO jtab2.

INSERT LINES OF itab2 INTO TABLE jtab2.

LOOP AT jtab2 into line.
  WRITE: / line-col1, line-col2.
endloop.
```

APPEND

The statement appends one or more rows to an internal standard table or a sorted table. The new row will be the last row with regard to the index of the table. The APPEND statement sets sy-tabix to the row number of the last appended row in the index.

COLLECT

The statement inserts the content of a variable either as a single row into an internal table or adds the values of its numeric components to the corresponding values of existing rows with the same primary key.

```
REPORT  Z_INTERNAL_COLLECT.
DATA: BEGIN OF dept,
        dept_id   TYPE zdepartments-department_id,
        name   TYPE zdepartments-department_name,
        number_of_emp TYPE i,
      END OF dept.

DATA dept_tab LIKE HASHED TABLE OF dept
               WITH UNIQUE KEY dept_id name.

SELECT department_id department_name
       FROM zdepartments
       INTO dept.
  COLLECT dept INTO dept_tab.
ENDSELECT.

LOOP AT dept_tab into dept.
  WRITE: / dept-dept_id, dept-name, dept-number_of_emp.
endloop.
```

Reading internal tables

READ TABLE

The statement reads a row from the internal table. You must specify the row with a key, or an index. If there are more than one suitable rows which has to be read, the first one will be read. You should also define where and how the row is stored.

```
REPORT  Z_INTERNAL_READ.

DATA: m   TYPE i,
      itab TYPE STANDARD TABLE
           OF i
           WITH NON-UNIQUE KEY table_line.

DO 10 TIMES.
  m = 2 * sy-index.
  INSERT m INTO TABLE itab.

ENDDO.

LOOP AT ITAB into m.
  WRITE: / m.
endloop.

read table itab index 3 into m.
write: / m.
```

The elements of the itab internal table are not structures, so you can refer to the only column with table_line.

```
REPORT  Z_INTERNAL_READ_KEY.
DATA: BEGIN OF dept,
        dept_id   TYPE zdepartments-department_id,
        name   TYPE zdepartments-department_name,
        number_of_emp TYPE i,
      END OF dept.

DATA dept_tab LIKE HASHED TABLE OF dept
               WITH UNIQUE KEY dept_id name.

SELECT department_id department_name
       FROM zdepartments
       INTO dept.
  COLLECT dept INTO dept_tab.
ENDSELECT.

read table dept_tab with key name = 'IT' into dept.
write: / dept-dept_id, dept-name, dept-number_of_emp.
```

LOOP AT

The LOOP statement sequentially reads rows from an internal table. You can specify a logical condition to restrict rows to be read.

Changing internal tables

MODIFY

It changes the content of specified rows in an internal table.

```
REPORT  Z_INTERNAL_MODIFY2.

DATA: BEGIN OF line,
            col1 TYPE i ,
            col2 TYPE i ,
          END OF line.

DATA itab like standard TABLE OF line.

DO 10 TIMES.
  line-col1 = 2 * sy-index.
  line-col2 = 5 * sy-index.
  append line TO itab.
ENDDO.

LOOP AT itab into line.
  if line-col1 mod 3 = 0.
    line-col2 = line-col1 / 3.
    modify itab from line.
  endif.
ENDLOOP.

loop at itab into line.
  write: / line-col1, line-col2.
endloop.
```

DELETE

It deletes specified rows from an internal table. You can specify the rows many ways, for example by an index, by a key, by a loop key or a loop condition. The material provides an example for deleting by index and deleting with the help of the loop condition.

```
delete dept_tab index 2 .

REPORT  Z_INTERNAL_DELETE.
DATA: BEGIN OF line,
            col1 TYPE i ,
            col2 TYPE i ,
          END OF line.

DATA itab like standard TABLE OF line.

DO 10 TIMES.
  line-col1 = 2 * sy-index.
  line-col2 = 5 * sy-index.
  append line TO itab.
ENDDO.

LOOP AT itab INTO line where col1 > 15 .
  DELETE itab.
ENDLOOP.

loop at itab into line.
   write: / line-col1, line-col2.
  endloop.
```

SORT

This statement sorts an internal table. You can define ascending and descending order. After the BY keyword you can give one or more components of the line.

```
sort itab by col2 descending col1 ascending.
```

Scanning internal tables

FIND IN TABLE and REPLACE IN TABLE

These statements search an internal table row-by-row for the strings specified in a pattern. The internal table must be a standard table with string elements.

```
REPORT  Z_INTERNAL_FIND.

DATA itab TYPE TABLE OF string.
data v type string.

APPEND 'banana - kg' TO itab.
APPEND 'apple - kg' TO itab.
APPEND 'bread - pc' TO itab.
APPEND 'yoghurt - pc' TO itab.

REPLACE ALL OCCURRENCES OF 'pc'
  IN TABLE itab WITH 'piece'
  RESPECTING CASE.

loop at itab into v.
  write: / v.
endloop.

data  results TYPE match_result_tab.
data rl like line of results.

FIND ALL OCCURRENCES OF 'kg'
  IN TABLE itab
  RESPECTING CASE
  RESULTS results.

LOOP AT results INTO rl.
  READ TABLE itab INTO v index rl-line.
  IF sy-subrc = 0.
    WRITE:/ 'Line:', rl-line, 'Offset:', rl-offset.
    WRITE: 'Value is:',  v+rl-offset.
  ENDIF.
ENDLOOP.
```

Interval join of internal tables

PROVIDE – ENDPROVIDE

This loop statement works more than one internal tables together. In the internal tables, intervals are specified. The PROVIDE statement finds where the intervals are overlapped.

```
REPORT  Z_INTERNAL_PROVIDE.
DATA: BEGIN OF v1,
        lower_bound TYPE i,
        upper_bound TYPE i,
        element TYPE string,
      END OF v1.

DATA: BEGIN OF v2,
        lower_bound TYPE i,
        upper_bound TYPE i,
        element TYPE string,
      END OF v2.
      DATA: BEGIN OF r,
        lower_bound TYPE i,
        upper_bound TYPE i,
        element TYPE string,
      END OF r.

DATA: itab1 LIKE STANDARD TABLE OF v1,
      itab2 LIKE STANDARD TABLE OF v2,
      itab_r like standard table of r.

DATA: flag1 TYPE c length 1,
      flag2 TYPE c length 1.

v1-lower_bound = 1.
v1-upper_bound = 4.
v1-element = 'table1 interval1'.
APPEND v1 TO itab1.

v1-lower_bound = 10.
v1-upper_bound = 12.
v1-element = 'table1 interval2'.
APPEND v1 TO itab1.

v2-lower_bound = 3.
v2-upper_bound = 6.
v2-element = 'table2 interval1'.
APPEND v2 TO itab2.

v2-lower_bound = 9.
v2-upper_bound = 14.
v2-element = 'table2 interval2'.
APPEND v2 TO itab2.

v2-lower_bound = 16.
v2-upper_bound = 19.
v2-element = 'table2 interval3'.
APPEND v2 TO itab2.
```

```
do 20 times.
  write / sy-index.
  loop at itab1 into v1.
    if sy-index between v1-lower_bound and v1-upper_bound.
      write v1-element.
      else.
        write '                ' .
    endif.
  endloop.
  loop at itab2 into v2.
    if sy-index between v2-lower_bound and v2-upper_bound.
      write v2-element.
      else.
        write '                ' .
    endif.
  endloop.
enddo.


PROVIDE FIELDS element FROM itab1 INTO v1
                                       VALID flag1
                                       BOUNDS lower_bound AND upper_bound
        FIELDS element FROM itab2 INTO v2
                                       VALID flag2
                                       BOUNDS lower_bound AND upper_bound
        BETWEEN 2 AND 20
  including gaps.
  WRITE: / SY-INDEX.
  WRITE: / v1-lower_bound, v1-upper_bound, v1-element, flag1.
  WRITE: / v2-lower_bound, v2-upper_bound, v2-element, flag2.
  SKIP.
  r-lower_bound = v1-lower_bound.
  r-upper_bound = v1-upper_bound.
  r-element = sy-index .
  append r to itab_r.
ENDPROVIDE.

do 20 times.
  write / sy-index.
  loop at itab1 into v1.
    if sy-index between v1-lower_bound and v1-upper_bound.
      write v1-element.
      else.
        write '                ' .
    endif.
  endloop.
  loop at itab2 into v2.
    if sy-index between v2-lower_bound and v2-upper_bound.
      write v2-element.
      else.
        write '                ' .
    endif.
  endloop.
   loop at itab_r into r.
    if sy-index between r-lower_bound and r-upper_bound.
      write r-element.
    endif.
  endloop.
enddo.
```

The internal tables must be fully typed index tables and contain two special columns which have the same data type for all tables. In the BOUNDS clause, you must specify these columns. The values of the two columns in a table specify a closed interval. The intervals of a table must not be overlapped and must be in ascending order.

The INTO clause specifies a variable whose type is the same as the row type of the table, because the statement fills the row into this variable. The VALID clause specifies a variable with a character-like data type with length 1.

The BETWEEN clause specifies a closed interval with two values which have the same data type as the variables in the BOUNDS clauses.

The INCLUDING GAPS clause is optional.

CLEAR and FREE

Both statements delete all rows from the internal table.

Open SQL

The SAP system uses a relational database management system. Two SAP systems can use different types of relational database management systems. Every relational database management system has its own SQL.

The Open SQL is a SAP-defined, database-independent SQL standard for the ABAP language. Its statements are converted to the native SQL statements of the used database management system.

The Open SQL works automatically with clients (remember the client field of the database tables). You can ask for the current client ID using sy-mandt.

Data Retrieval

The SELECT statement is similar, but not the same as the SELECT statement you learnt in the subject Database Systems. The main structure is the same; there are SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY clauses with the same meaning. In the FROM clause you can use INNER JOIN and LEFT OUTER JOIN with the ON keyword to specify the join condition.

You must use the AS keyword if you want to give an alias to a column or a table. There are only spaces instead of commas between the columns and the table names. If you use an alias for a table, you can specify its columns with ~. Between the aggregate function names and their (, there is no space. In the COUNT function, you can use *, or a column name with DISTINCT keyword.

The system should store the result of the SELECT statement somewhere. The SELECT statement offers two kinds of solutions. On the one hand, you can store the result in variables, which can be either internal tables (for more than one rows) or simple variables (for only one row). If you store the results in internal tables, you can choose from the APPENDING TABLE or the INTO TABLE keyword. The first appends the results to the internal tables, whereas the second initializes first the internal tables then inserts the results into them. In both cases, you can use the CORRESPONDING FIELDS OF clause.

```
REPORT  Z_SELECT_ONE_ROW.
data  nof type i.
SELECT count( * )
  from zdepartments as dept left outer join zemployees as emp on dept~depar
tment_id = emp~departtment_id
  into nof
 where department_name = 'IT'.

write: nof.

REPORT  Z_SELECT.
DATA: BEGIN OF dept,
        department_id   TYPE zdepartments-department_id,
        department_name   TYPE zdepartments-department_name,
      END OF dept.

DATA dept_tab LIKE HASHED TABLE OF dept
               WITH UNIQUE KEY department_id department_name.

SELECT department_id department_name
       FROM zdepartments
       INTO CORRESPONDING FIELDS OF table dept_tab.

SELECT department_id department_name
     FROM zdepartments
     INTO CORRESPONDING FIELDS OF table dept_tab
  WHERE department_name = 'IT'.

SELECT department_id department_name
    FROM zdepartments
    APPENDING CORRESPONDING FIELDS OF table dept_tab
  WHERE department_name = 'SALES'.

LOOP AT dept_tab into dept.
  WRITE: / dept-department_id, dept-department_name.
endloop.
```

On the other hand, you can use the SELECT statement as a LOOP, and process the results row by row. In this case, you must specify which variables are used to store the result in the INTO clause, and the statement must be closed by the ENDSELECT clause.

```
REPORT  Z_SELECT_LOOP.
data wa type  zemployees.

select *
  from zemployees
  into wa
  where job_id = 3.
  write: / wa-first_name, wa-last_name, wa-departtment_id.
endselect.
```

Data retrieval – cursor

The ABAP language offers another solution to read data from the database; it is the cursor. The main statements of it are OPEN SELECT, FETCH NEXT CURSOR, and CLOSE CURSOR.

Modify Data in the Database

Similarly, as you learnt in database system, you can use DML statements in the ABAP language, namely INSERT, UPDATE and DELETE statements. In an ABAP program, you can also use the MODIFY statement. More other important topics belong to these statements, they are transactions, locking, data consistency and authorization.

Transaction

You learnt about the transactions in the subject Database Systems. In a database, a transaction contains one or more DML statements, and you can complete it with the COMMIT statement, or you can roll it back with the ROLLBACK statement. The SAP calls a database transaction a database LUW (Logical Unit of Work).

But, the SAP LUW and the database LUW is not the same. The programming units can call each other, and the user wants to complete their work together. Consequently, an application program can contain more work process, every change of which is linked to an implicit database commit.

When a user executes a program, it starts a new SAP LUW. This program can call other units. You should finish the SAP LUW with the COMMIT WORK or the ROLLBACK WORK statement.

INSERT

You can insert a row from variables or you can insert more rows from an internal table.

```
REPORT  Z_SELECT_DML.

data line type zjobs.

line-job_id = 10.
line-job_title = 'Administrator'.
insert into zjobs values line.

data: job_tab type table of zjobs.
line-job_id = 20.
line-job_title = 'Manager'.
append line to job_tab.
line-job_id = 30.
line-job_title = 'Cleaner'.
append line to job_tab.

insert zjobs from table job_tab.

select * from zjobs
  into line.
  write: / line-job_id, line-job_title.
endselect.
```

UPDATE

It changes one or more rows in a table.

```
select *
  from zjobs
  into line
  where job_id = 10.
  line-job_title = 'IT Administration'.
 endselect.

update zjobs from line.
```

```
select *
  from zjobs
  into table job_tab.

loop at job_tab into line.
  line-job_title = line-job_title && '!'.
  modify job_tab from line.
endloop.

update zjobs from table job_tab.
```

DELETE

You can delete one or more rows from a table. You can specify the rows with a condition or with data elements.

```
delete from zjobs where job_id = 10.

delete zjobs from line.

delete zjobs from table job_tab.
```

MODIFY

This statement inserts or overwrites one or more rows in a table.

```
line-job_id = 40.
line-job_title = 'Tester'.
modify zjobs from line.

select *
  from zjobs
  into table job_tab.

loop at job_tab into line.
  line-job_title = line-job_title && '+'.
  modify job_tab from line.
endloop.

line-job_id = 50.
line-job_title = 'Programmer'.
append line to job_tab.

modify zjobs from table job_tab.
```

Messages

The MESSAGE statement interrupts the program flow and displays a short text. The short text can be any text, or a message specified in the T100 table. The message type should be provided which determines its behaviour.

```
MESSAGE text TYPE message_type
```

The message type can be

- "A" termination message: it appears in a dialog box, and the program terminates.
- "E" error message: depending on the program context, an error dialog appears or the program terminates.
- "I" information message: it appears in a dialog box. If the user confirms the message, the program continues immediately after the MESSAGE statement.
- "S" status message: the message is displayed in the status bar and the program continues normally after it.
- "W" warning: depending on the program context, an error dialog appears or the program terminates.
- "X" exit message: no message is displayed and the program terminates.

```
REPORT  Z_MESSAGE.
parameters a type i.
if a <= 0.
  message 'Give positive number!' type 'A'.
endif.
```

Now, go to the ABAP Dictionary (SE11) and browse which messages are in the T100 table.

With SE91 transaction (Message Maintenance), you can add new messages to this table. First, create a Message Class named ZMES1. After you click on the Create button, every field should be filled in the Attributes tab. Then, in the Messages tab, write your messages and save them. Now, you can use your message from many program codes in the following way:

```
REPORT  Z_MESSAGE.
parameters a type i.
if a <= 0.
  message e000(ZMES1).
endif.
```

In the syntax, the ‘e’ refers to the error message type, whereas 000 refers to the number assigned to the message in the message class. ZMES1 is the name of the message class.

Events and Blocks

If you use the PARAMETERS keyword, the system creates a selection screen, where the user can give input values. If you use the WRITE statement, the system creates a list, which shows the output of the program. A program can contain both of them, and both of them can have event blocks, which are triggered by events.

The block has a start keyword, but you cannot define its end. Its end will be the keyword of the next event block keyword.

The material gives example of only the main event blocks.

Before the event blocks, there can be global declarations. The order of the event blocks must follow the flow of the program. Every event block is optional.

Events and event blocks for lists

There can be more events in a list, but this material introduces only the AT LINE-SELECTION event. If the user double clicks on a row, the event will be triggered.

```
REPORT  Z_SCREEN_EVENT.
write 'Click on me'.

AT LINE-SELECTION.
  WRITE: / 'You clicked on the previous list'.
```

Initialization block

In this block, you can initialize input fields of the selection screen or some other variables. It will be executed after the program is loaded. Its keyword is the INITIALIZATION.

Events and event blocks for the selection screen

AT SELECTION-SCREEN ON parameter

This event block is triggered when the parameter value is passed to the ABAP program.

AT SELECTION-SCREEN ON VALUE-REQUEST FOR parameter

The event block is triggered when the input help (F4) is called.

AT SELECTION-SCREEN ON HELP-REQUEST FOR parameter

The event block is triggered when the field help (F1) is called.

AT SELECTION-SCREEN

The event block is triggered as the last event of the selection screen processing, if all the input values were passed to the program.

START-OF-SELECTION

This event keyword defines the standard processing block of an executable program. Its event is triggered after the selection screen has been processed.

```
REPORT  Z_EVENT.
parameters: a type i,
            b type i.
data: c type i,
      d type i,
      e type i,
      f type decfloat16.

data itab type table of i.
data line like line of itab.

INITIALIZATION.

  do 10 times.
    append sy-index to itab.
  enddo.

at selection-screen on b.
  if b = 0.
    message 'The second value must not be 0' type 'E'.
  endif.

at selection-screen.
  c = a + b.
  d = a - b.
  e = a * b.
  f = a / b.

at line-selection.

  loop at itab into line.
    line = line * c.
    modify itab from line.
  endloop.

start-of-selection.
  write: c, d, e, f.
  write: / 'Click'.

at line-selection.
  loop at itab into line.
    write: / line.
  endloop.
```

Modularisation

You can split a complex ABAP code into more, smaller, simpler modules. SAP system allows a number of techniques to be used to break a program up into smaller, more manageable sections of code.

Subroutines

The subroutines helps modularize the program code inside the actual program. `Z_SUBROUTINE_1` report contains a query about zjobs table three times.

```
REPORT  Z_SUBROUTINE_1.
write 'ZJobs'.

data line type zjobs.

select * from zjobs
  into line.
  write: / line-job_id, line-job_title.
endselect.
write /.

line-job_id = 100.
line-job_title = 'IT'.
insert into zjobs values line.

select * from zjobs
  into line.
  write: / line-job_id, line-job_title.
endselect.
write /.

delete from zjobs where job_id = 100.

select * from zjobs
  into line.
  write: / line-job_id, line-job_title.
endselect.
write /.
```

You can create a subroutine which contains the query about zjobs table in order that the query will be implemented only one time, and the program will call it three times. To carry it out, write the `perform select_zjobs.` statement into the code. This statement will call the subroutine named select_zjobs. Than double click on select_zjobs. In the new window, choose that you want to create the object, and in the other new window, choose your main program, now it is z_include_1, so click on the rectangle at the beginning of the corresponding row. A new program part will appear in your code. Following the `form` select_zjobs. you can write the select statement. As a result you will get the following code:

```
REPORT  Z_SUBROUTINE_1.
write 'ZJobs'.

data line type zjobs.

perform select_zjobs.

line-job_id = 100.
line-job_title = 'IT'.
insert into zjobs values line.

perform select_zjobs.

delete from zjobs where job_id = 100.

perform select_zjobs.
*&---------------------------------------------------------------------*
*&      Form  SELECT_ZJOBS
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
*  -->  p1        text
*  <--  p2        text
*----------------------------------------------------------------------*
FORM SELECT_ZJOBS .
select * from zjobs
  into line.
  write: / line-job_id, line-job_title.
endselect.
write /.

ENDFORM.                    " SELECT_ZJOBS
```

You can pass parameters to a subroutine with using keyword. The parameters can also be tables, in which cases the tables keyword has to be used.

```
REPORT  Z_SUBROUTINE_2.

parameters gvt_max type i.
data gvt_count type i.
data gvt_st_primes type standard table of i.

perform f_prime tables gvt_st_primes using gvt_max gvt_count.

perform f_writeout tables gvt_st_primes.

*&---------------------------------------------------------------------*
*&      Form  prim
*&---------------------------------------------------------------------*
*       It generated the prime numbers between 1 and v_max
*----------------------------------------------------------------------*
*      -->T_PRIMEK   it stored the prime numbers
*      -->V_MAX      it contains the upper bound of the range
*      -->V_DB       it contains how many prime numbers are generated.
```

```
*---------------------------------------------------------------------*
form f_prime tables t_primes using v_max v_count.
  data i type i value 2.
  data prime type i.
  while i <= v_max.
    data line type i.
    prime = 1.
    LOOP AT t_primes into line.
      if i mod line = 0.
        prime = 0.
        exit.
      endif.
    endloop.

    if prime = 1.
      append i to t_primes.
    endif.
    i = i + 1.
  endwhile.

endform.                    "prim
*&--------------------------------------------------------------------*
*&      Form
*&--------------------------------------------------------------------*
*       It writes the integer typed element of the table
*         got as a parameter to the screen.
*---------------------------------------------------------------------*
form f_writeout tables t_numbers.
  data line type i.
  LOOP AT t_numbers into line.
    WRITE: / line.
  endloop.

endform.                    "f_writeout
```

You can also call subroutines of another program.

```
REPORT  Z_SUBROUTINE_3.
data i type i value 2.
data gvt_st_numbers type standard table of i.
parameters gvt_max type i.

while i <= gvt_max.
  append i to gvt_st_numbers.
  i = i + 1.
endwhile.

perform f_writeout in program z_subroutine_2 tables gvt_st_numbers.
```

Includes

A subroutine can also be placed into an include. You can create an include in two ways. In the first case, when you write the `perform select_zjobs.` statement into the code, which will call the subroutine named select_zjobs, and you double click on it, you have to choose the new include (or an existing include) instead of choosing your main program. In the second case, you can create a new include as you would do when you create a report, but now you have to choose the INCLUDE as the type of the program.

In the report, the `INCLUDE include_name.` statement will be generated in the first case. In the second case, you have to add it to your code. If you double click on the include name in a report, the system will open the include.

The include cannot directly be processed. Before you execute your code, the include has to be activated.

```
*----------------------------------------------------------------------*
***INCLUDE Z_INCLUDE_2_INC_SELECT01 .
*----------------------------------------------------------------------*
*&---------------------------------------------------------------------*
*&      Form  SELECT_ZJOBS
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
*  -->  p1          text
*  <--  p2          text
*----------------------------------------------------------------------*
FORM SELECT_ZJOBS .
  select * from zjobs
  into line.
  write: / line-job_id, line-job_title.
endselect.
write /.

ENDFORM.                    " SELECT_ZJOBS
*&---------------------------------------------------------------------*
*&      Form  DELETE_ZJOB
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
*  -->  p1          text
*  <--  p2          text
*----------------------------------------------------------------------*
FORM DELETE_ZJOB using v_job_id.
delete from zjobs where job_id = v_job_id.
ENDFORM.                    " DELETE_ZJOB
```

The Z_INLCUDE_2 report uses the Z_INCLUDE_2_INC_SELECT01 include.

```
REPORT  Z_INCLUDE_2.
write 'ZJobs'.

data line type zjobs.
data v_job_id type zjobs-job_id value 100.

perform select_zjobs.

line-job_id = 100.
line-job_title = 'IT'.
insert into zjobs values line.
```

```
perform select_zjobs.

perform delete_zjob using v_job_id.

perform select_zjobs.

INCLUDE Z_INCLUDE_2_INC_SELECT01.
```

An include program can include other include programs. You can use include programs to modularize the source code of a single ABAP program. We do not recommend reusing an include program in multiple ABAP programs.

Function modules

Function modules constitute a major part of an SAP system, because SAP has modularized them for years in order to reuse the codes. A **function group** is a kind of container for the **function modules** which logically belong together. A function module can be called from any ABAP programs. SAP systems have thousands of function modules.

In Object Navigator (SAP menu: Tools, ABAP Workbench, Overview, and Object Navigator - SE80) you can list function groups. Click on the repository browser, choose the Function Group, click on the Input Help button, in the new window click on the Information System button, and finally in the new window click on the execute button. As a result you will get the list of the function groups, however only the 200 of them. You can examine the F017 Function Group.

Function groups or function pools are ABAP programs of a special type. This program type is the only one that can contain function modules. Function modules are procedures with public interface and are designed to be used by other programs. Function groups can contain global data declarations and subroutines that are available to all function modules in the group. Function groups can contain dynpros as their components. You will use several function groups if you work with dynpros.

For each function group the system generates a main program called `SAPLfunctiongroupname`. The main program contains INCLUDE statements for the following programs:

- `LfunctiongroupnameTOP`: It is the first include program; contains the global data definitions for the function group.

- `LfunctiongroupnameUxx`: These includes contain the function modules. The numbering xx indicates the sequential order in which the function modules were created. For example, `LfunctiongroupnameU01` and `LfunctiongroupnameU02` contain the first two function modules in the function group.
- `LfunctiongroupnameF01`, `LfunctiongroupnameF02`, ... `LfunctiongroupnameFxx`: You can use these includes to write subroutines (forms) that can be called as internal forms by all function modules in the group.

These include programs are generated by the Function Builder and may not be changed.

The naming convention for includes is the following (xx indicates a two-digit number):

- Suffix “Exx”: Implementation of methods and classes
- Suffix „Oxx”: Implementation of PBO modules (PBO is the Process Before Output screen event. It is triggered before a screen is sent to the presentation layer.)
- Suffix “Ixx”: Implementation of PAI modules (PAI is the Process After Input screen event. It is triggered by a user action on the GUI.)
- Suffix “Exx”: Implementation of event blocks
- Suffix “Fxx”: Implementation of local subroutines

In the top include, you can add include programs with the suffix “Dxx” for the declaration parts of local classes.

You can search for function modules with the Function Builder (SE37), which you can find in the SAP menu: Tools, ABAP Workbench, Development, and Function Builder. Choose Utilities menu, Find, and search for *amount* function modules. Choose the Spell_amount function module of F017 function group and display it.

Parameters of function modules

Function modules use parameters to communicate with calling programs. The type of the parameters are the following:

- Import: Values of the parameters will be transferred from the calling program to the function module. You cannot overwrite the contents of import parameters at runtime. If a parameter does not have to be assigned, select the Optional checkbox on the Import tab of the Function Builder.

- Export: Values of the parameters will be transferred from the function module back to the calling program. They are always optional.
- Changing: The parameters act as import and export parameters simultaneously. The original value of a changing parameter is transferred from the calling program to the function module. The function module can change the initial value and send it back to the calling program.
- Tables: The parameter contains an internal table that can be imported and exported. The contents of an internal table are transferred from the calling program to the function module. The function module can change the contents of the internal table and then send it back to the calling program. Tables are always assigned by reference. It is obsolete.
- Exceptions: The calling program uses exceptions to find out if an error has occurred in the function module.

You can specify the data type of a formal parameter by linking it to a data type in a type pool. Type pools are ABAP Dictionary objects that allow you to define your own global types. If you want to use the types in a type pool for formal parameters, you must declare the type pool in the TOP include of the function group.

Call a function module

Call the SE38 transaction, and create a new program called `Z_FUNC_MOD_CALL`.

```
REPORT  Z_FUNC_MOD_CALL.
parameter v type i.
```

Click on the Pattern button (CRTL+F6), in the new window choose the CALL FUNCTION option, write there spell_amount, which is the name of a function module, and click the continue button. An ABAP code is generated automatically. The optional fields are commented out, whereas the mandatory fields should be filled in.

```
REPORT  Z_FUNC_MOD_CALL.
parameter v type i.
data result like spell.
CALL FUNCTION 'SPELL_AMOUNT'
                    EXPORTING
                     AMOUNT           = v
*                      CURRENCY         = ' '
*                      FILLER           = ' '
*                      LANGUAGE         = SY-LANGU
                    IMPORTING
                      IN_WORDS         = result
```

```
*                          EXCEPTIONS
*                            NOT_FOUND         = 1
*                            TOO_LARGE         = 2
*                            OTHERS            = 3
          .
IF SY-SUBRC <> 0.
  write: 'Returned value:',sy-subrc.
else.
  write: 'The amount in words is:',result-word.
ENDIF.
```

The syntax of CALL FUNCTION is the following:

```
CALL FUNCTION <function module name>
  [EXPORTING  f1 = a1.... fn = an]
  [IMPORTING  f1 = a1.... fn = an]
  [CHANGING  f1 = a1.... fn = an]
  [TABLES  t1 = itab1.... tn = itabn]
  [EXCEPTIONS e1 = r1.... en = rn]
  [ERROR_MESSAGE = rE]
  [OTHERS = ro]].
```

In the EXPORTING, IMPORTING, CHANGING, and TABLES options, you pass parameters by explicitly assigning the actual parameters to the formal parameters, namely <formal parameter> = <actual parameter>. If you assign multiple parameters within an option, insert spaces between them or start a new line.

- EXPORTING: The formal parameters must be declared as import parameters in the function module
- IMPORTING: The formal parameters must be declared as export parameters in the function module.
- CHANGING: The formal parameters must be declared as CHANGING parameters in the function module.
- TABLES: Assigns internal tables to table parameters. It is obsolete.
- EXCEPTIONS: Allows you to react to errors in the function module. If an exception occurs, the processing of the function module is terminated. If the exception is triggered, the system terminates the function module and does not pass any values back to the program. The calling program accepts the exception by assigning the value rE to the system field SY-SUBRC. You can change the error handling in the function module by specifying an ERROR_MESSAGE in the EXCEPTIONS list.
- ERROR_MESSAGE: The system treats messages that are called in the function module as follows:

- Messages of classes S, I, and W are ignored (but entered in the log if you are running the program in the background).
- Messages of classes E and A cause the function module to terminate as though the ERROR_MESSAGE exception had been triggered (SY-SUBRC is set to rE).

If you enter OTHERS in the EXCEPTION list, you can allow for all exceptions.

Create a function module

Function modules perform tasks which have general interest to other programmers. The tasks can be performing tax calculations, determining factory calendar dates, calling frequently-used dialogs and so on.

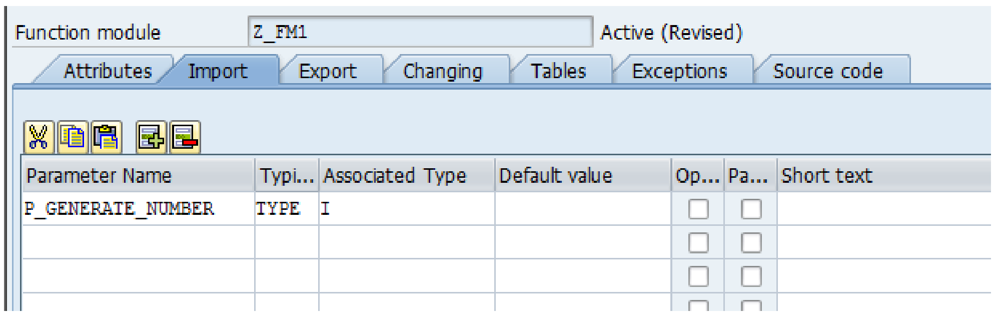
Before you create a function module, check whether an appropriate function module already exists or not. If one exists, use it. Otherwise, choose a suitable function group, if it exists. If it does not exist, create one.

After you have created the function module, activate the module, test (F8) it and document it and its parameters for other users.

You can **create** a **function group** in two ways. In the first case, in the Object Navigator (SE80) choose Function Group, enter a new name, and choose Enter. In the second case, in the Function Builder (SE37) choose the Goto menu, Function Groups, Create Group, and add a group name. In the example the Z_FG1 name is used as the new function group name.

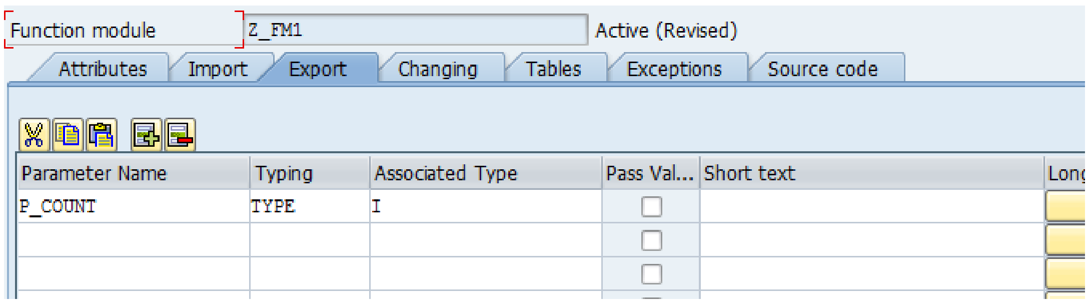
You can **create** a **function module** using the Function Builder (SE37) or the Object Navigator. In the object navigator, find the Z_FG1 function group, right click on its name in the hierarchy, and add a function module to it. Write a new name and press Create button. In the example the Z_FM1 name is used as the new function module name. In the new window give values to the function group and to the short text. In the new window, on the import, export, changing, exceptions tabs, you can add parameters to the function module. On the source code tab, you can write your code.

Figure 7, Figure 8 and Figure 9 show the tabs of Function Builder, when Z_FM1 function module is being created.


Function module Z_FM1 Active (Revised)

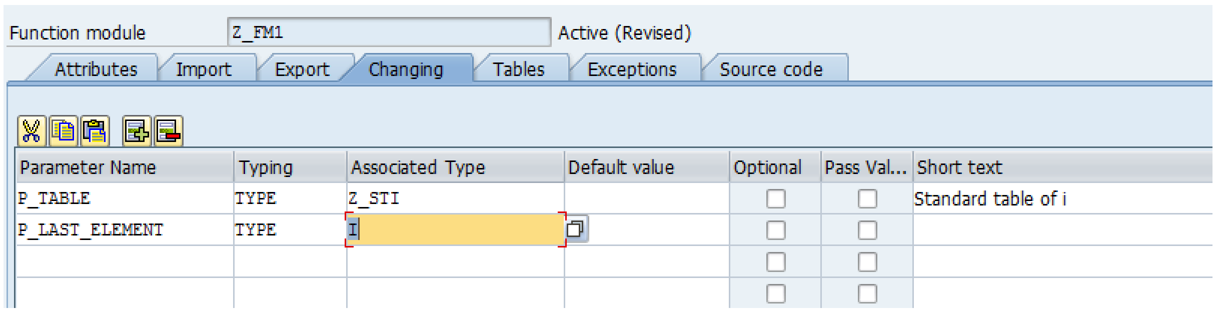
Parameter Name	Typi...	Associated Type	Default value	Op...	Pa...	Short text
P_GENERATE_NUMBER	TYPE	I		☐	☐	
				☐	☐	
				☐	☐	

Figure 7 – Import tab of Function Builder


Function module Z_FM1 Active (Revised)

Parameter Name	Typing	Associated Type	Pass Val...	Short text	Long
P_COUNT	TYPE	I	☐		
			☐		
			☐		

Figure 8 – Export tab of Function Builder


Function module Z_FM1 Active (Revised)

Parameter Name	Typing	Associated Type	Default value	Optional	Pass Val...	Short text
P_TABLE	TYPE	Z_STI		☐	☐	Standard table of i
P_LAST_ELEMENT	TYPE	I		☐	☐	
				☐	☐	
				☐	☐	

Figure 9 – Changing tab of Function Builder

Z_STI type is defined as a standard table of integers in the data dictionary.

```
FUNCTION Z_FM1.
*"----------------------------------------------------------------------
*"*"Local Interface:
*"  IMPORTING
*"     REFERENCE(P_GENERATE_NUMBER) TYPE  I
*"  EXPORTING
*"     REFERENCE(P_COUNT) TYPE  I
*"  CHANGING
*"     REFERENCE(P_TABLE) TYPE  Z_STI
*"     REFERENCE(P_LAST_ELEMENT) TYPE  I
*"----------------------------------------------------------------------
  data i type i value 1.
  data e type i .

  while i <= p_generate_number.
    e = i + p_last_element.
    append e to p_table.
    i = i + 1.
  endwhile.

  p_count = 0.
  data line type i.
  LOOP AT p_table into line.
    WRITE: / line.
    p_count = p_count + 1.
  endloop.
ENDFUNCTION.
```

In order to document your function module, add information to the parameters with the Long Text button in the Function Builder or in the Object Navigator. Moreover, you can describe the function module in the Object Navigator, if you click on Function Module Documentation button, which opens the SAPscript Editor.

You can add more function modules to Z_FG1 Function Group, such as Z_FMD.

```
FUNCTION Z_FMD.
*"----------------------------------------------------------------------
*"*"Local Interface:
*"  CHANGING
*"     REFERENCE(P_TABLE) TYPE  Z_STI
*"----------------------------------------------------------------------
data line type i.
LOOP AT p_table into line .
  DELETE p_table.
ENDLOOP.
ENDFUNCTION.
```

The function modules of a function group can call each other.

```
FUNCTION Z_FMCE.
*"----------------------------------------------------------------------
*"*"Local Interface:
*"  IMPORTING
*"     REFERENCE(P_GENERATE_NUM) TYPE  I DEFAULT 10
*"  CHANGING
*"     REFERENCE(P_TABLE_CH) TYPE  Z_STI
*"----------------------------------------------------------------------
CALL FUNCTION 'Z_FMD'
  CHANGING
    P_TABLE       = p_table_ch
          .
data l type i.
CALL FUNCTION 'Z_FM1'
  EXPORTING
    P_GENERATE_NUMBER       = p_generate_num
* IMPORTING
*   P_COUNT                 =
  CHANGING
    P_TABLE                 = p_table_ch
    P_LAST_ELEMENT          = l
          .
ENDFUNCTION.
```

User Dialogs

A dynpro is a dynamic program. Its other name is screen. It consists of a screen and the dynpro flow logic; moreover, it contains dynpro fields. Dynpros with complex screens can be created by Screen Painter tool (SE51 transaction).

The user dialogs are all based on dynpros. Special dynpros are the selection screen and the list, which are introduced in this material.

However, the user interface which has already been introduced only a fractional part of the graphical user interface of the dynpros. There can be a lot of types of screen elements in a dynpro, for example menu bars, tool bars, text fields, input fields, output fields, radio buttons, check boxes, push buttons, frames, table controls, subscreens, status bars. Furthermore, the SAP has built-in functions to display the results of select statements in tabular form, for example the ALV_GRID_DISPLAY function.

To create screen with these elements, you can use the Screen Painter (SE51), the Menu Painter (SE41), or the Object navigator (SE80).

Moreover, dynpros can be executed as a part of other dynpros, and other type of ABAP programs.

Types of ABAP Programs

The type of an ABAP program determines what parts it can contain and how it will be executed by the SAP. When you create a new ABAP program, you can choose the type of it from a list.

Executable program

It begins with the REPORT statement. This type of ABAP program is used in our examples. It has its own screen. It can be executed using one of them:

- the SUBMIT statement from another program,
- calling a dynpro with a transaction code
- calling a selection screen with a transaction code.

Class pool

It is a global class of the class library, local interfaces and classes, TYPES and CONSTANTS statements. Its visible methods can be executed by the CALL METHOD or a transaction code. It does not support its own screen.

Function group or function pool

It contains function modules. It supports its own screen.

Interface pool

It is a global interface of the class library. It cannot be executed. It does not support its own screen.

Module pool

It can be executed by calling a dynpro with a transaction code. It contains screens and dialog modules.

Subroutine pool

It contains subroutines which can be called from other ABAP programs. Its subroutines can be called externally.

Type group or type pool

It contains TYPES and CONSTANTS statements. It cannot be executed.

Transaction codes mentioned in this material

ABAPDOCU – Example Library. SAP menu: Tools, ABAP Workbench, Utilities.

ABAPHELP – Keyword Documentation. SAP menu: Tools, ABAP Workbench, Utilities.

DWDM – Demos. SAP menu: Tools, ABAP Workbench, Utilities.

SE11 – ABAP Dictionary. SAP menu: Tools, ABAP Workbench, Development.

SE14 - Utilities for Dictionary Tables

SE16 – Data Browser. SAP menu: Tools, ABAP Workbench, Overview.

SE37 – Function Builder. SAP menu: Tools, ABAP Workbench, Development.

SE38 – ABAP Editor. SAP menu: Tools, ABAP Workbench, Development.

SE41 – Menu Painter. SAP menu: Tools, ABAP Workbench, Development, User Interface.

SE51 – Screen Painter. SAP menu: Tools, ABAP Workbench, Development, User Interface.

SE54 – General Table Maintenance Dialog. SAP menu: Tools, ABAP Workbench, Development, Other Tools.

SE80 – Object Navigator. SAP menu: Tools, ABAP Workbench, Overview.

SE91 – Messages. SAP menu: Tools, ABAP Workbench, Development, Programming Environment.

SM30 - Call View Maintenance

SM31 - Call View Maintenance

SU01 – Users. SAP menu: Tools, Administration, User Maintenance.

References

1. Online ABAP documentation: http://help.sap.com/abapdocu_731/en. (May 2015.)
2. Moxon, P. (2012): Beginner's Guide to SAP ABAP. SAPPROUK, 274 p.

17 May, 2015