Dr.SNS RAJALAKSHMI COLLEGE OF ARTS AND SCIENCE
(Autonomous)
Coimbatore -641049
Accredited by NAAC (Cycle- III)with 'Á+' Grade
(Recognised by UGC, Approved by AICTE, New Delhi and
Affiliated to Bharathiar University , Coimbatore

UNIT – IV

Dr.A.DEVI

Associate Professor

Department of Computer Applications

DRSNSRCAS

Modifying Data in a Database Table

Authorisations

When writing programs using open SQL, one has to bear in mind the concepts of authorisation in an SAP system. An SAP system has its own security tools to ensure that users can only access data which they are authorised to see. This includes individual fields as well as individual records. The way authorisations are set up can also limit how data is used, whether a user can only display data or whether they can modify it. All the rules pertaining to this are stored as authorisation objects. These will not be examined in great detail here, but ordinarily users are assigned their own authorisation profile (or composite profile) against their user record, which for informational purposes is managed through transaction code SU01.

This profile then gives the user the correct rights within the program to then carry out their job and SAP delivers many predetermined user profiles with the base system. The system administrators can then use and enhance these to be applied to users. Once a user has one of these profiles, the system will tell them whether or not they can execute a transaction when they try to do this. For example, transaction SE38, the ABAP editor, could be tweaked so that while some users may be able to access it, perhaps they can only do so in display mode, or perhaps they can display and debug the code, but not change it themselves.

Where specific authorisations have not been implemented, programs can be made to carry out an authority check, using the statement AUTHORITY-CHECK. This must be used if a program or transaction is not sufficiently protected by the standard authorisation pro- files already set up in the system.

While, this will not be examined in great detail here (the topic is huge in itself), it is important to bear authorisations in mind when working in SAP.

Fundamentals

So far, reading data from database tables has been looked at, now modifying and deleting this data will be examined. There are some important concepts to keep in mind here, for example, the architecture of the system. If one has a three-tier architecture (with a presentation layer, an application server and an underlying database), you must bear in mind that there may be a very large number of users accessing the data at any one time. It is important to ensure that programs created do not cause any problems in the rest of the system and that the most recent version of the data held on the database is accessed when a program runs. If records are constantly being updated, programs must be able to read and work with data which is current in the system. Fortunately, most of this work is done automatically by the SAP system, and one doesn't have to worry too much about the underlying technologies related to how data is locked and so on.

One of the key tools which can be used is Open SQL. This acts as an interface between the programs created and the database. By using Open SQL, one can read and modify data, and also buffer data on the application server, which reduces the number of database ac- cesses the system has to perform. It is the database interface which is also responsible for synchronising the buffers with the database tables at predetermined intervals.

When one is creating programs it is important to keep in mind that if data is buffered, and this buffered data is subsequently read, it may not always be up to date. So, when tables are created, they must be created in such a way that the system is told that buffering can or cannot be used, or that it can only be used in certain situations. When the example ta- bles were created earlier, the system was told not to use buffering. Using this setting means that every time data is read from a table, it will always use the most up to date re- cords.

Buffering can be useful for tables which hold master data and configuration settings, because this kind of data does not get updated regularly. When one is working with transactional data however, one wants this data to be as up to date as possible. If transactional data is being used in a context where tables are using buffering, it is important to ensure that programs related to this can take this into account, and make sure that the buffer is updated with new data when this is needed.

When one uses Open SQL statements in a program, tables can only be accessed through the ABAP dictionary. This acts as an interface, one does not access the tables directly through programs. This is not a problem however, as when one uses Open SQL state- ments, it works just the same as if one was accessing the database directly. Open SQL manages its interface with the database by itself, without the need for the user to do any-thing here. Statements can be coded just as though they had direct access to the tables, though with the underlying knowledge that by using Open SQL, the data is in fact being accessed through the ABAP dictionary with a built-in level of safety to ensure the ABAP code does not have a direct effect on the SAP database system itself.

Database Lock Objects

Now, locking concepts will be considered. This refers to locking data in database tables and there are two basic types of locking which must be kept in mind. First of all, database locks. These lock data in a physical database. When a record is updated, a lock is set on this, then when it is updated the lock is released. It is there to ensure that, once set, the data can only be accessed and updated by those authorised to do so. When released, it can be accessed more widely.

These locks, though, are not sufficient in an SAP system, and are generally only used when a record is being modified in a single step dialogue process. This process refers to any time that the data in a database can be updated in a single step, on a single screen. In this case, the data can be locked, updated and released very quickly.

As you work more with SAP, the insufficiency of database locks will become clearer, because transactions in an SAP system often occur over multiple steps. If, for example, an employee record is added to the system, one may have to fill in many screens of data. The user in this case will only want the record to be added to the system at the end of the last screen, once all of the data in all of the screens has been input. If just the first screen's data was saved into the database, then the second's, and so on, one by one, if the user were to quit halfway through the process, an invalid and unfinished record would be in the database.

This demonstrates the hazard of using database locks with multi-step dialogue processes. For these instances, SAP has introduced a new kind of lock, independent of the database system. These are called lock objects, and allow data records to be locked in multiple database tables for the whole duration of the SAP transaction, provided that these are linked in the ABAP dictionary by foreign key relationships.

SAP lock objects form the basis of the lock concept, and are fully independent of database locks. A lock object allows one to lock a record for multiple tables for the entire duration of an SAP transaction. For this to work, the tables must be linked together using foreign keys. The ABAP dictionary is used to create lock objects, which contain the tables and key fields which make up a shared lock.

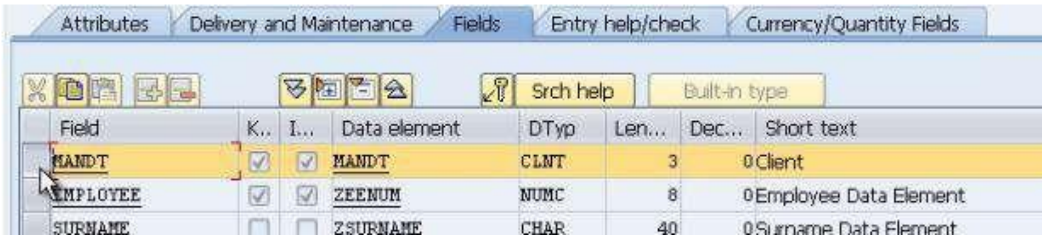
When the lock object is created, the system automatically creates two function modules, which will be discussed later. These function modules are simply modularised ABAP programs that can be called from other programs. The first of these has the action of setting a lock, and the second releases this lock. It is the programmer's responsibility to ensure that these function modules are called at the correct place in the program. When a lock is set, a lock record is created in the central lock table for the entire SAP system. All programs must adhere to using the SAP lock concept to ensure that they set, delete and query the lock table that stores the lock records for the relevant entries.

Lock objects will not be discussed much further, however subsequent programs created, tables accessed and so on here will be done on the assumption that they are not to be used outside of one's own system.

Using Open SQL Statements

Now, some of the Open SQL statements which can be used in programs will be looked at. As mentioned before, Open SQL statements allow one to indirectly access and modify data held in the underlying database tables. The SELECT statement, which has been used several times previously, is very similar to the standard SQL SELECT statement used by many other programming languages. With Open SQL, these kinds of statements can be used in ABAP programs regardless of what the underlying database is. The system could be running, for example, an Oracle database, a Microsoft SQL database, or any other, and by using Open SQL in programs in conjunction with the ABAP dictionary to create and modify database tables, one can be certain that the ABAP code will not have any issues accessing the data held by the specific type of database the SAP system uses.

When the first database table was created previously, the field MANDT was used, representing the client number and forming part of the database table key, highlighted below:



Field	K..	I...	Data element	DTyp	Len...	Dec...	Short text
MANDT	✓	✓	MANDT	CLNT	3	0	Client
EMPLOYEE	✓	✓	ZEENUM	NUMC	8	0	Employee Data Element
SURNAME	☐	☐	ZSURNAME	CHAR	40	0	Surname Data Element

One may think that, given the importance of this field, it would have to be used in ABAP programs when using Open SQL statements, however, it does not. Almost all tables will include this ‘hidden’ field within them, and the SAP system is built in such a way that a filter is automatically applied to this field, based on the client ID being used. If one is logged in, for example, to client 100, the system will automatically filter all records in the database on this client key and only return those for client 100. When Open SQL is used in the programs one creates, the system manages this field itself, meaning it never has to be included in any selections or update statements used in programs. Also, this carries the benefit of security in the knowledge that any Open SQL statement executed in a program will only affect the records held in the current client.

Using Open SQL Statements – 5 Statements

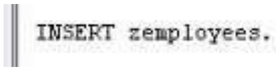
There are 5 basic Open SQL statements which will be used regularly in programs from here forward. These are SELECT, INSERT, UPDATE, MODIFY and DELETE.

- The SELECT statement has, of course, already been used. This statement allows one to select records from database tables which will then be used in a program.
- INSERT allows new records to be inserted into a database table.
- UPDATE allows records which already exist in the table to be modified.
- MODIFY performs a similar task to update, with slight differences which we will discuss shortly.
- DELETE, of course, allows records to be deleted from a table.

Whenever any of these statements are used in an ABAP program, it is important to check whether the action executed has been successful. If one tries to insert a record into a database table, and it is not inserted correctly or at all, it is important to know, so that the appropriate action can be taken in the program. This is done using a system field which has already been used: SY-SUBRC. When a statement is executed successfully, the SY-SUBRC field will contain a value of 0, so this can be checked for and, if it appears, one can continue with the program. If it is not successful, however, this field will contain a different value, and depending on the statement, this value can have different meanings. It is therefore important to know what the different return codes are for the different ABAP statements, so as to recognise problems and take the correct course of action to solve them. This may sound difficult, but with practice will become second-nature.

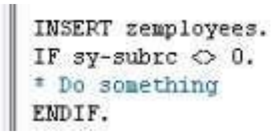
Insert Statement

The SELECT statement has already been used, so here it will be skipped for now to focus on the INSERT statement. In this example then, a new record will be inserted into the ZEMPLOYEES table. Firstly, type INSERT, followed by the table name, and then a period:



```
INSERT zemployees.
```

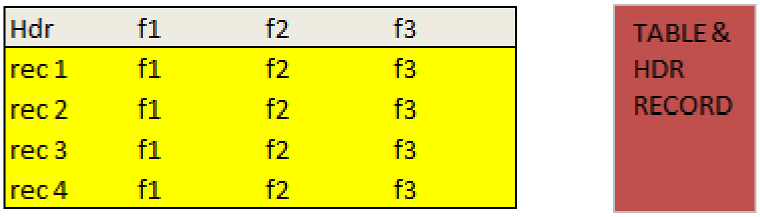
Doing this, one must always type the table name, a variable's name cannot be used instead. Use the check statement (IF) to include an SY-SUBRC check, telling the system to do if this does not equal 0:



```
INSERT zemployees.
IF sy-subrc <> 0.
* Do something
ENDIF.
```

This is the simplest form of the INSERT statement, and not necessarily the one which is encouraged. Using this form is no longer standard practice, though one may come across it if working with older ABAP programs.

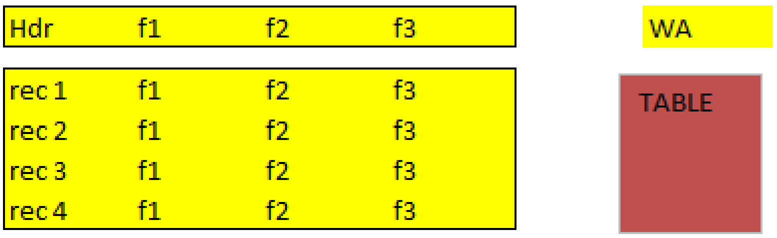
In the above statement, nothing is specified to be inserted. This is where the concept of the work area enters. The statement here expects a work area to exist which has been created when an internal table was declared. This type of work area is often referred to as a header record:



Hdr	f1	f2	f3
rec 1	f1	f2	f3
rec 2	f1	f2	f3
rec 3	f1	f2	f3
rec 4	f1	f2	f3

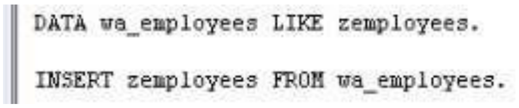
The table above shows the yellow area as a standard table containing four records and their respective fields, the area above in grey is the header record, which is stored in memory and is the area which is accessed when the table is referenced from a program only by its table name. If an INSERT statement is executed, whatever is contained in the header record will be inserted into the table itself. The header record does not exist in the table, it is just an area stored in memory where a current record can be worked with, hence the term work area. When someone refers to the table only by its table name, it is the header record which is referred to, and this can become confusing. One thinks that one is referencing the table itself, but in fact it is the header record which is being worked with, a record held in memory with the same structure as the table. ABAP objects, which are important when one gets to a more advanced stage in ABAP, will not allow a header record to be referred to, so it is important not to do this. Header records were used com- monly for this in the past, but as noted previously, this is no longer the way things are done.

To avoid confusion when working with internal tables should programs must work with separate work areas, which are perhaps similar in structure to a header record, but not attached to the table, with a separate name. These are separate structures from the initial table, which are created in a program.



To declare a work area the DATA statement is used. Give this the name “wa_employees”. Now, rather than declaring one data type for this, several fields which make up the table will be declared. The easiest way to do this is to use the LIKE statement.

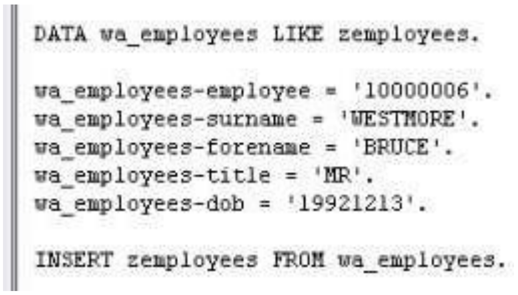
So here, the wa_employees work area is declared LIKE the zemployees table, taking on the same structure without becoming a table itself. This work area will only store one record. Once this is declared, the INSERT statement can be used to insert the work area and the record it holds into the table. The code here will read “INSERT zemployees FROM wa_employees”:



```
DATA wa_employees LIKE zemployees.

INSERT zemployees FROM wa_employees.
```

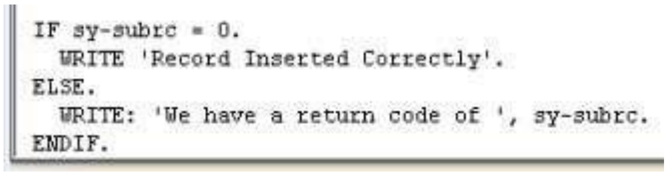
Additionally, using this form of the INSERT statement allows you to specify the table name using a variable instead. It is important to note here that if one is doing this, the variable must be surrounded by brackets.

Now, the work area must be filled with some data. Use the field names from the zemployees table. This can be done by forward navigation, double-clicking the table name in the code, or by opening a new session and using SE11. The fields of the table can then be copy & pasted into the ABAP editor and the work area's fields populated as in the image below:



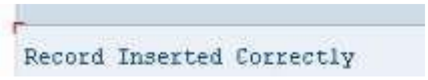
```
DATA wa_employees LIKE zemployees.

wa_employees-employee = '10000006'.
wa_employees-surname = 'WESTMORE'.
wa_employees-forename = 'BRUCE'.
wa_employees-title = 'MR'.
wa_employees-dob = '19921213'.

INSERT zemployees FROM wa_employees.
```

The check statement can then be formulated as follows, meaning that if the record is inserted correctly, the system will state this, if not then the SY-SUBRC code which will not equal zero is will be displayed:



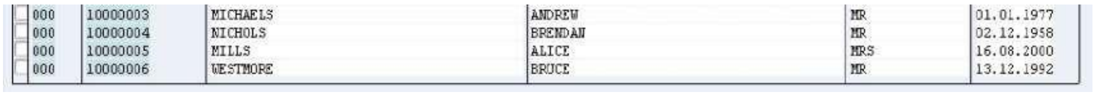
```
IF sy-subrc = 0.
  WRITE 'Record Inserted Correctly'.
ELSE.
  WRITE: 'We have a return code of ', sy-subrc.
ENDIF.
```

Check the program, save, and activate the code, then test it. The output window will display:



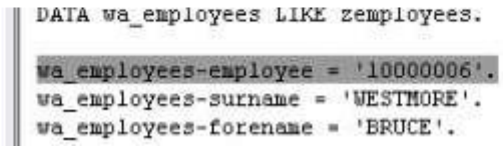
```
Record Inserted Correctly
```

If you check the records in your table via the 'Data Browser' screen in the ABAP dictionary, a new record will be visible:



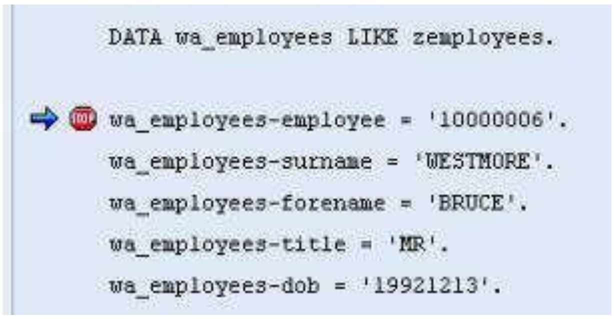
000	10000003	MICHAELS	ANDREW	MR	01.01.1977
000	10000004	NICHOLS	BRENDAN	MR	02.12.1958
000	10000005	MILLS	ALICE	MRS	16.08.2000
000	10000006	WESTMORE	BRUCE	MR	13.12.1992

For practice use the ABAP debugger to execute the code step-by-step. First, delete the record from the table in the ABAP dictionary and put a breakpoint in the code at the beginning of the record entry to the work area:



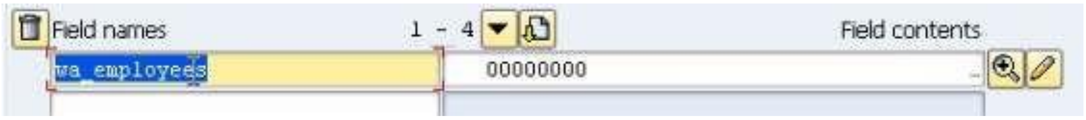
```
DATA wa_employees LIKE zemployees.

wa_employees-employee = '10000006'.
wa_employees-surname = 'WESTMORE'.
wa_employees-forename = 'BRUCE'.
```

Now execute the program. The breakpoint will cause program execution to pause at your breakpoint and the debugger will open:

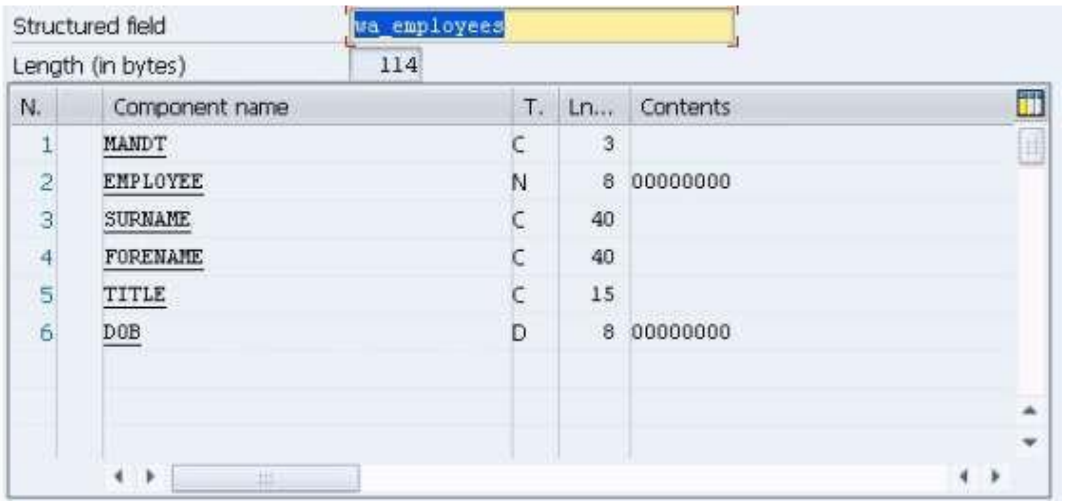


```
DATA wa_employees LIKE zemployees.

wa_employees-employee = '10000006'.
wa_employees-surname = 'WESTMORE'.
wa_employees-forename = 'BRUCE'.
wa_employees-title = 'MR'.
wa_employees-dob = '19921213'.
```

Firstly, use the Fields mode to view the work area structure. Double click the wa_employees after the DATA statement and it will appear in the 'Field names' box at the bottom. At this point the work area is completely empty, evidenced by the zeros in the adjacent box. To display the full structure, double click the wa_employees in the left box:



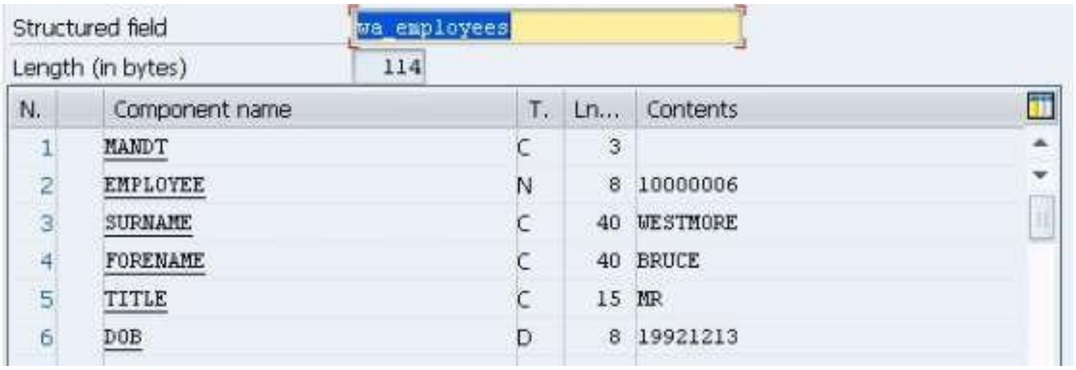
Field names	Field contents
wa_employees	00000000



Structured field: wa_employees
Length (in bytes): 114

N.	Component name	T.	Ln...	Contents
1	MANDT	C	3	
2	EMPLOYEE	N	8	00000000
3	SURNAME	C	40	
4	FORENAME	C	40	
5	TITLE	C	15	
6	DOB	D	8	00000000

Then, execute each line of code starting from the breakpoint using the F5 key, the fields within this structure view are filled one by one:



Structured field: wa_employees
Length (in bytes): 114

N.	Component name	T.	Ln...	Contents
1	MANDT	C	3	
2	EMPLOYEE	N	8	10000006
3	SURNAME	C	40	WESTMORE
4	FORENAME	C	40	BRUCE
5	TITLE	C	15	MR
6	DOB	D	8	19921213

Return to the Fields view before executing the INSERT statement, and observe the SY-SUBRC field at the bottom of the window. It displays a value of 0. If there are any problems in the execution, this will then change (4 for a warning, 8 for an error). Given that this code has already been successful, you already know that it will remain 0. Once the program has been executed in the debugger, refresh the table in the Data Browser screen again, and the record will be visible.

Clear Statement

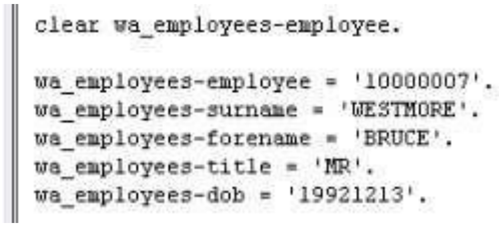
At this point, the CLEAR statement will be introduced. In ABAP programs, one will not always simply see the program start at the top, insert one data record and continue on.

Loops and the like will be set up, allowing, for example, many records to be inserted at once. To do this, variables and structures are re-used repeatedly. The CLEAR statement allows a field or variable to be cleared out for the insertion of new data in its place, allowing it to be re-used. The CLEAR statement is certainly one which is used commonly in programs, as it allows existing fields to be used multiple times.

In the previous example, the work area structure was filled with data to create a new record to be inserted into the zemployees table, then a validation check performed. If one then wants to insert a new record, the work area code can then be copy & pasted below this. However, since the work area structure is already full, the CLEAR statement must be used so that it can then be filled again with the new data.

To do this, the new line of code would read "CLEAR wa_employees."

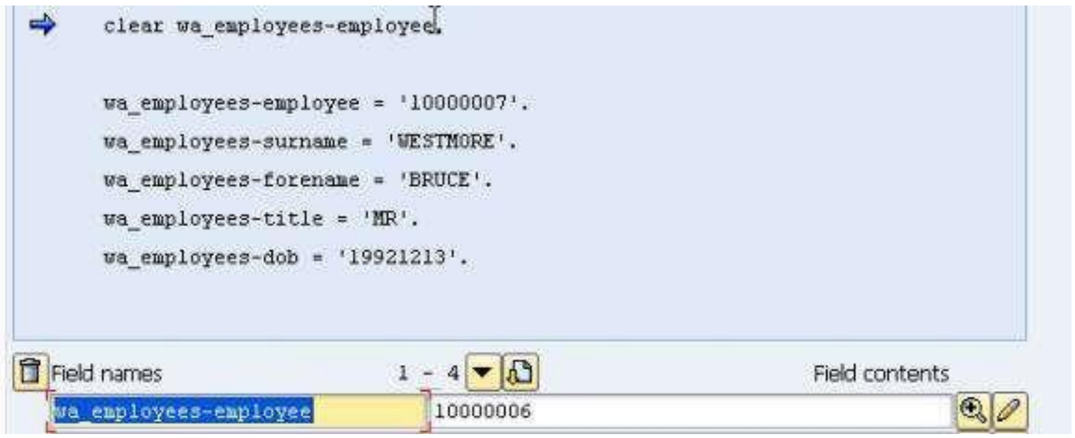
If you just wanted to clear specific fields within your structure you just need to specify the individual fields to be cleared, as in the example below, clear the employee number field. New data can then be entered into the work area again:



```
clear wa_employees-employee.

wa_employees-employee = '10000007'.
wa_employees-surname = 'WESTMORE'.
wa_employees-forename = 'BRUCE'.
wa_employees-title = 'MR'.
wa_employees-dob = '19921213'.
```

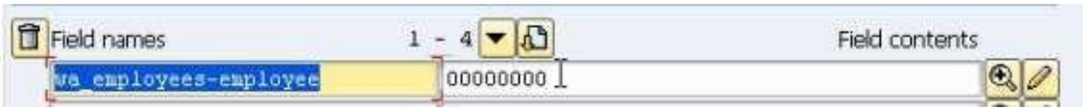
Remember that the employee number is a key field for the zemployees table, so as long as this is unique, duplicate information could be entered into the other fields. If one tries to enter the same employee number again though, the sy-subrc field will display a warning with the number 4.

You can see the operation of the CLEAR statement in debug mode. The three images below display the three stages of its operation on the field contents as the code is executed:

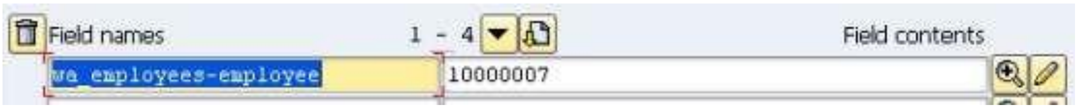


```
clear wa_employees-employee.

wa_employees-employee = '10000007'.
wa_employees-surname = 'WESTMORE'.
wa_employees-forename = 'BRUCE'.
wa_employees-title = 'MR'.
wa_employees-dob = '19921213'.
```

```
wa_employees-employee    10000006
```



```
wa_employees-employee    00000000
```



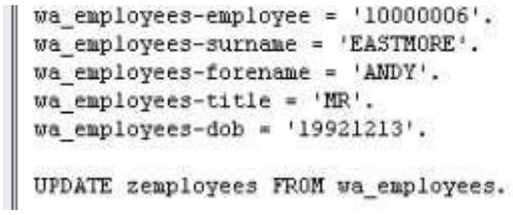
```
wa_employees-employee    10000007
```

Update Statement

The UPDATE statement allows one or more existing records in a table to be modified at the same time. In this example it will just be applied to one, but for more the same princi- ples generally apply.

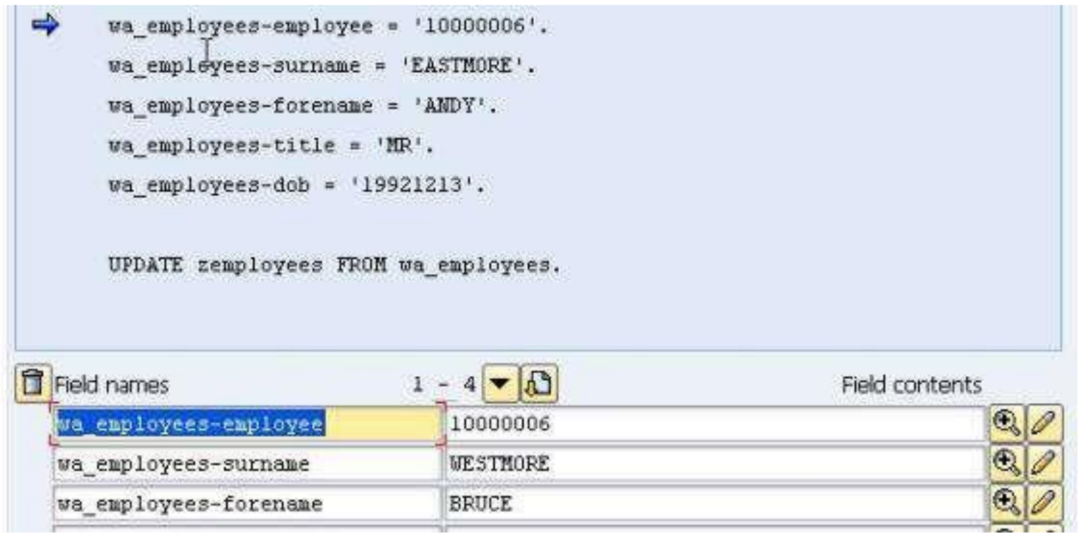
Just as with the INSERT statement, a work area is declared, filled with the new data which is then put into the record as the program is executed.

Delete the record created with the CLEAR statement as before. Here, the record previ- ously created with the INSERT statement will be updated. Copy & paste the work area and then alter, the text stored in the SURNAME and FORENAME fields. Then on a new line, the same structure as for the INSERT statement is used, but this time using UPDATE:



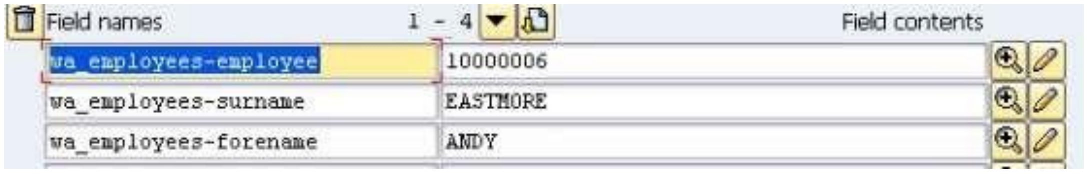
```
wa_employees-employee = '10000006'.
wa_employees-surname = 'EASTMORE'.
wa_employees-forename = 'ANDY'.
wa_employees-title = 'MR'.
wa_employees-dob = '19921213'.

UPDATE zemployees FROM wa_employees.
```

As this is run line-by-line in debug mode, you can see the Field contents change as it is executed:



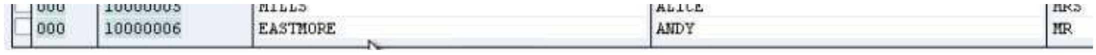
```
wa_employees-employee = '10000006'.
wa_employees-surname = 'EASTMORE'.
wa_employees-forename = 'ANDY'.
wa_employees-title = 'MR'.
wa_employees-dob = '19921213'.

UPDATE zemployees FROM wa_employees.
```

Field names 1 - 4	Field contents
wa_employees-employee	10000006
wa_employees-surname	WESTMORE
wa_employees-forename	BRUCE



Field names 1 - 4	Field contents
wa_employees-employee	10000006
wa_employees-surname	EASTMORE
wa_employees-forename	ANDY

Once the UPDATE statement has been executed you can view the Data Browser in the ABAP Dictionary to see that the record has been changed successfully:



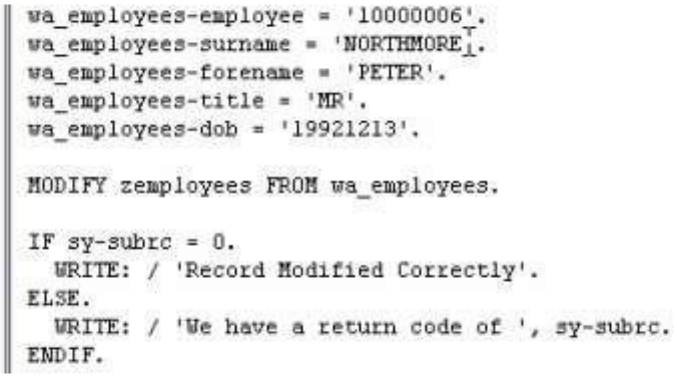
000	10000005	MILLS	ALICE	MRS
000	10000006	EASTMORE	ANDY	MR

Modify Statement

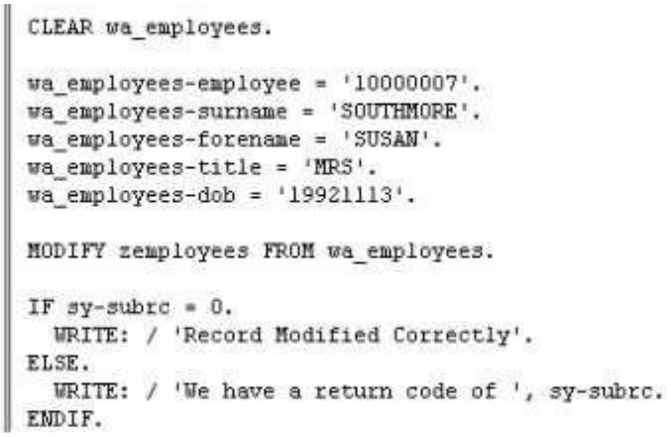
The MODIFY statement could be said to be like a combination of the INSERT and UPDATE statements. It can be used to either insert a new record or modify an existing one. Gener- ally, though the INSERT and UPDATE statements are more widely used for these purposes, since these offer greater clarity. Using the MODIFY statement regularly for these purposes is generally considered bad practice. However, times will arise where its use is appropri- ate, for example of one is writing code where a record must be inserted or updated de- pending on a certain situation.

Unsurprisingly, the MODIFY statement follows similar syntax to the previous two statements, modifying the record from the data entered into a work area. When this statement is executed, the key fields involved will be checked against those in the table. If a record with these key field values already exists, it will be updated, if not then a new record will be created.

In the first section of code in the image below, since employee number is the key field, and '10000006' already exists, the record for that employee number will be updated with the new name in the code. A validation check is performed next. The CLEAR statement is then used so a new entry can be put into the work area, and then employee 10000007 is added. Since this is a new, unique key field value, a new record will be inserted, and an- other validation check executed:

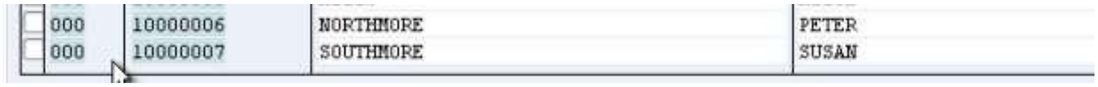


```
wa_employees-employee = '10000006'.
wa_employees-surname = 'NORTHMORE'.
wa_employees-forename = 'PETER'.
wa_employees-title = 'MR'.
wa_employees-dob = '19921213'.

MODIFY zemployees FROM wa_employees.

IF sy-subrc = 0.
  WRITE: / 'Record Modified Correctly'.
ELSE.
  WRITE: / 'We have a return code of ', sy-subrc.
ENDIF.
```



```
CLEAR wa_employees.

wa_employees-employee = '10000007'.
wa_employees-surname = 'SOUTHMORE'.
wa_employees-forename = 'SUSAN'.
wa_employees-title = 'MRS'.
wa_employees-dob = '19921113'.

MODIFY zemployees FROM wa_employees.

IF sy-subrc = 0.
  WRITE: / 'Record Modified Correctly'.
ELSE.
  WRITE: / 'We have a return code of ', sy-subrc.
ENDIF.
```

When this is executed, and the data then viewed in the Data Browser, employee number 10000006 will have been updated with the new name, Peter Northmore, and a new re- cord will have been created for number 10000007, Susan Southmore:



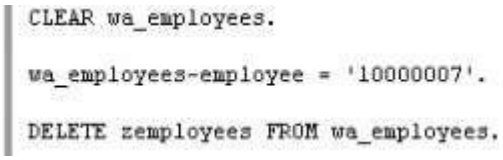
000	10000006	NORTHMORE	PETER
000	10000007	SOUTHMORE	SUSAN

Delete Statement

The last statement to be looked at in this section is the DELETE statement. One must be careful using this, because if used incorrectly, there is the possibility of wiping the entire contents of the table, however, as long as it is used correctly, there should be no problem of this sort.

Unlike the previous SQL statements, the DELETE statement does not take into account most fields, only the primary key field. When you want to delete a record from a table, the system only needs to be told what the primary key field value for that record is.

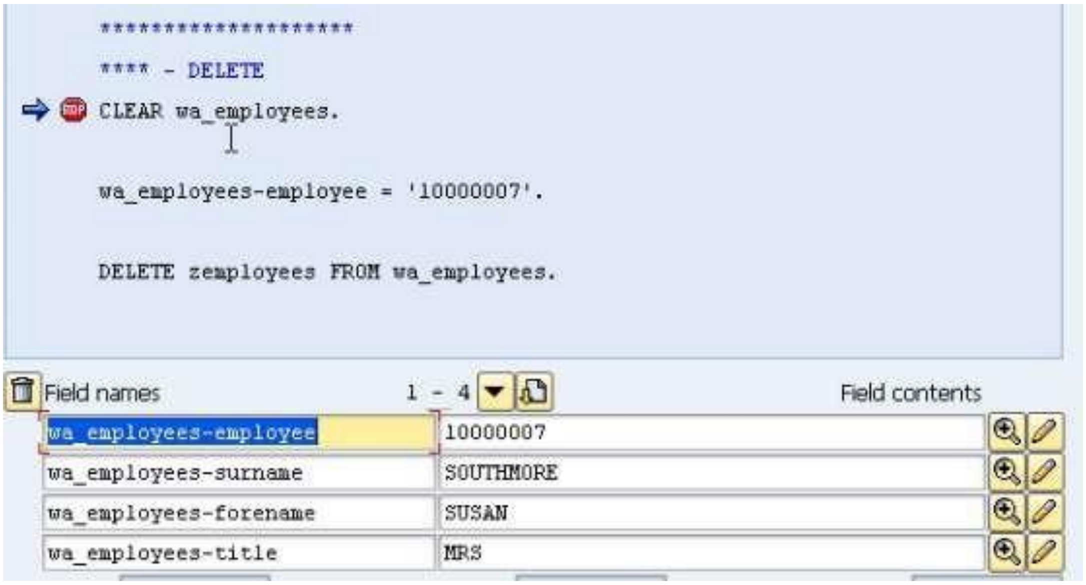
In this example, the last record created, for the employee Susan Southmore will be de- leted. For the zemployees table, there are two key fields, the client field and the employee number. The client field is dealt with automatically by the system, and this never has to be included in programs, so the important field here is the employee number field. The syn- tax to delete the last record created in the previous section would be this:



```
CLEAR wa_employees.

wa_employees-employee = '10000007'.

DELETE zemployees FROM wa_employees.
```

The FROM addition in the last line ensures only the record referred to by its key field in the work area will be deleted. Again, a validation check is performed to ensure the record is deleted successfully. When this is run in debug mode you can see the fields which are filled with the creation of the record are cleared as the CLEAR statement executes.

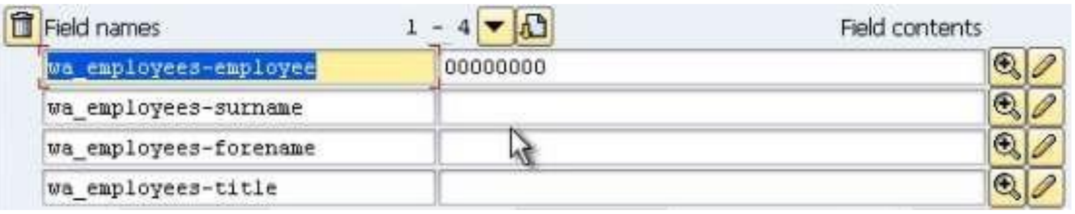
After the employee number is filled again the DELETE statement is executed. The code’s output window will indicate the success of the deletion and the record will no longer ap- pear in the Browser view of the table:



```
    ********************
    **** - DELETE
    CLEAR wa_employees.

    wa_employees-employee = '10000007'.

    DELETE zemployees FROM wa_employees.
```

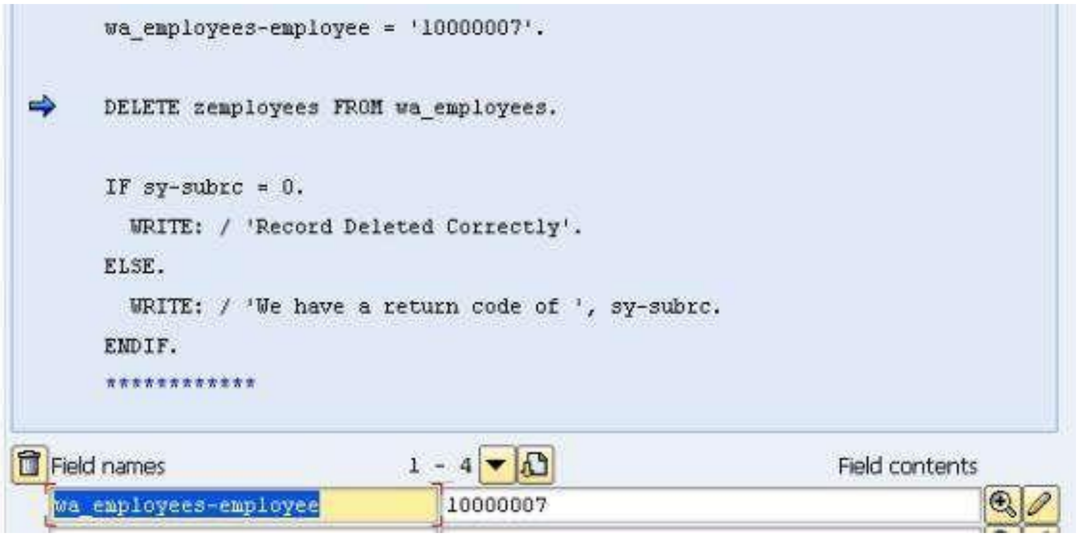
Field names 1 - 4 Field contents

wa_employees-employee	10000007
wa_employees-surname	SOUTHMORE
wa_employees-forename	SUSAN
wa_employees-title	MRS



Field names 1 - 4 Field contents

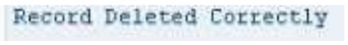
wa_employees-employee	00000000
wa_employees-surname	
wa_employees-forename	
wa_employees-title	



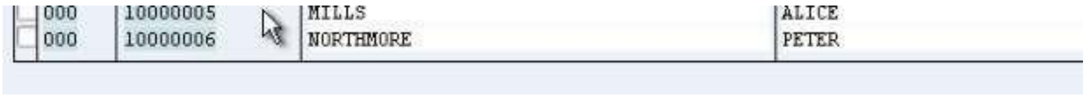
```
    wa_employees-employee = '10000007'.

    DELETE zemployees FROM wa_employees.

    IF sy-subrc = 0.
      WRITE: / 'Record Deleted Correctly'.
    ELSE.
      WRITE: / 'We have a return code of ', sy-subrc.
    ENDIF.
    ************
```

Field names 1 - 4 Field contents

wa_employees-employee	10000007



```
Record Deleted Correctly
```

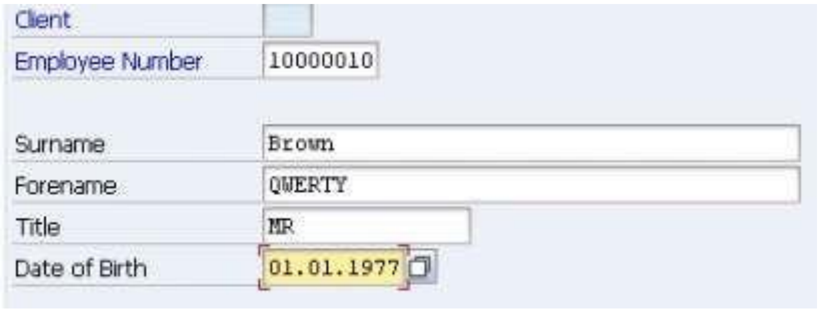


000	10000005	MILLS	ALICE
000	10000006	NORTHMORE	PETER

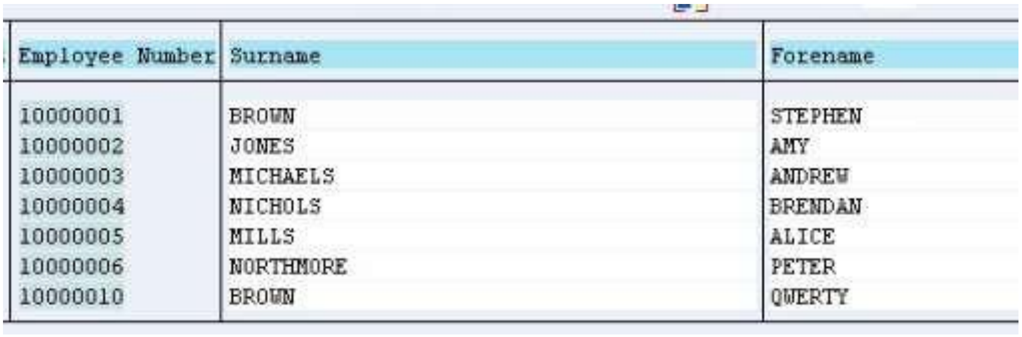
The record is now gone from the table.

There is another form of the DELETE statement which can be used. You are not just restricted to using the table key to delete records, logic can also be used. So, rather than using the work area to specify a key field, and using the FROM addition to the DELETE statement, one can use the WHERE addition to tell the program to delete all records where a certain field matches a certain value, meaning that if one has several records which match this value, all of them will be deleted.

The next example will demonstrate this. All records with the surname *Brown* will be deleted. To be able to demonstrate this, create a second record containing a surname of Brown, save this and view the data:



Client	
Employee Number	10000010
Surname	Brown
Forename	QWERTY
Title	MR
Date of Birth	01.01.1977

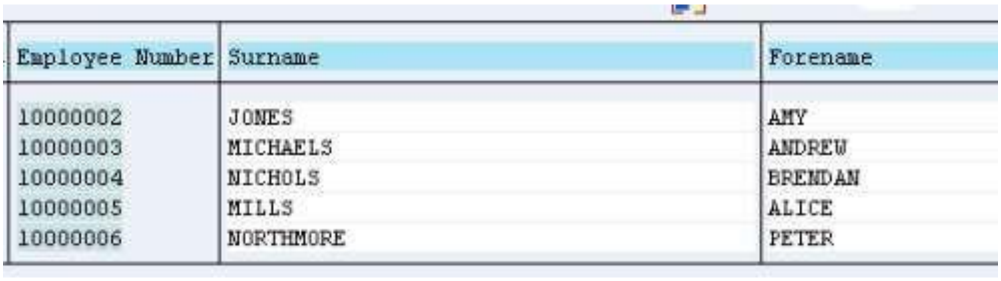


Employee Number	Surname	Forename
10000001	BROWN	STEPHEN
10000002	JONES	AMY
10000003	MICHAELS	ANDREW
10000004	NICHOLS	BRENDAN
10000005	MILLS	ALICE
10000006	NORTHMORE	PETER
10000010	BROWN	QWERTY

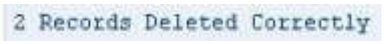
The code for the new DELETE statement should then look like this. Note the additional FROM which must be used in this instance:

```
CLEAR wa_employees.

DELETE FROM zemployees WHERE surname = 'BROWN'.

IF sy-subrc = 0.
  WRITE: / '2 Records Deleted Correctly'.
ELSE.
  WRITE: / 'We have a return code of ', sy-subrc.
ENDIF.
```

When this code is executed, both records containing a Surname of Brown will be deleted.



Employee Number	Surname	Forename
10000002	JONES	AMY
10000003	MICHAELS	ANDREW
10000004	NICHOLS	BRENDAN
10000005	MILLS	ALICE
10000006	NORTHMORE	PETER



```
2 Records Deleted Correctly
```

Note that, if one uses the following piece of code, without specifying the logic addition, all of the records will in fact be deleted:

```
DELETE FROM zemployees.
```

Chapter 10 – Program Flow Control and Logical Expressions

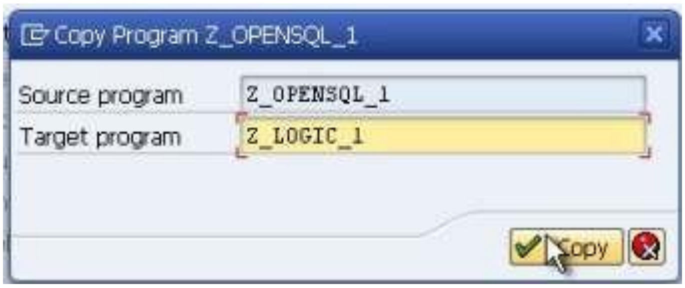
Control Structures

This section will look at program flow control and logical expressions. It could be argued that this is really the main aspect of ABAP programming, where the real work is done. How one structures a program using logical expressions will determine the complete flow of the program and in what sequence actions are taken.

First, a look will be taken at control structures. When a program is created it is broken up into many tasks and subtasks. One controls how and when the sections of a program are executed using logical expressions and conditional loops, often referred to as control structures.

If Statement

Copy you program previous chapter in which to test some of the logic which is to be built. Here I copy the program Z_OPENSQL_1 to Z_LOGIC_1:

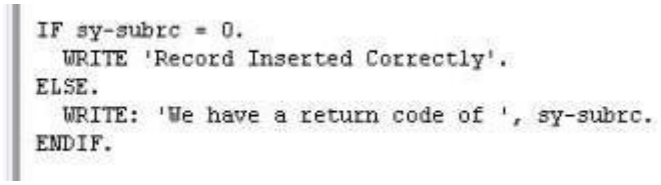

Copy Program Z_OPENSQL_1

Source program	Z_OPENSQL_1
Target program	Z_LOGIC_1

Remove all of the code from the program, leaving only the first example INSERT statement and its validation test.

When one talks of control structures, this refers to large amounts of code which allows one to make decisions, resulting in a number of different outcomes based on the decisions taken. Take a look at the IF statement to explain the basic logic at work here.

The IF statement is probably the most common control structure, found in just about every programming language. The syntax may vary between languages, but its use is just about universal:



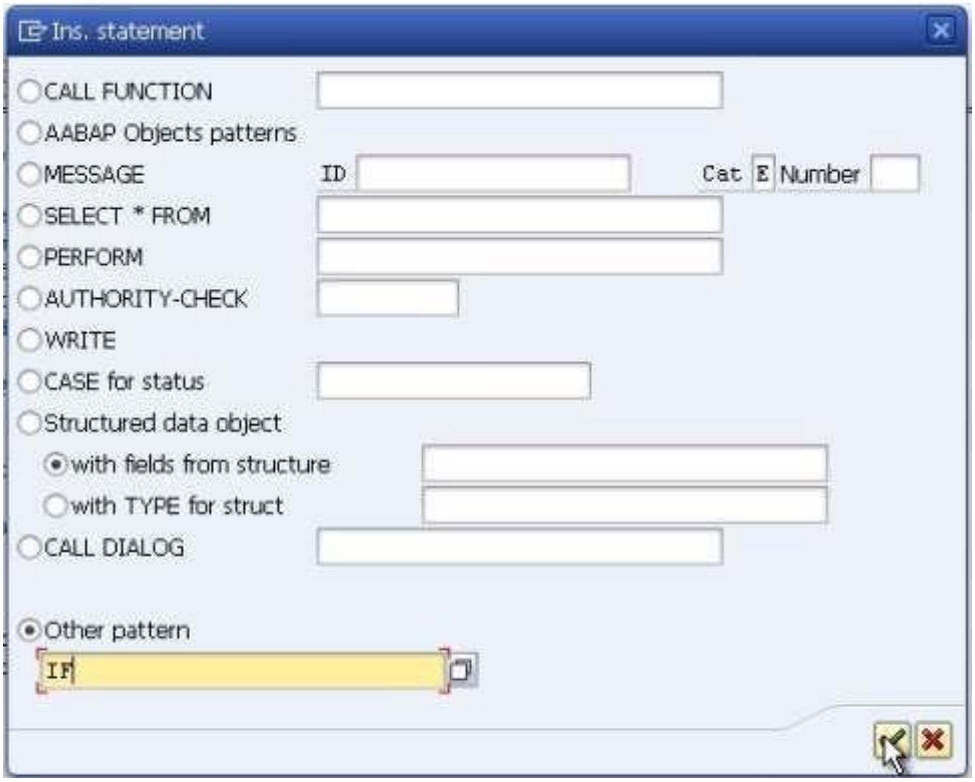
```
IF sy-subrc = 0.
  WRITE 'Record Inserted Correctly'.
ELSE.
  WRITE: 'We have a return code of ', sy-subrc.
ENDIF.
```

This IF statement tells the program that IF (a logical expression), do something. The ELSE addition means that should this logical expression not occur, do something else. Then the statement is ended with the ENDIF statement.

The IF and ENDIF statements belong together, and every control structure created will take a similar form, with a start and an end. Control structures can be very large, and may contain other, smaller control structures within them, having the system perform tasks within the framework of a larger task. The code between the start and end of a control structure defines the subtasks within it. Tasks can be repeated, in what are called loops.

From here on, control structures will be used to control the flow, create tasks, subtasks and branches within a program, and to perform loops.

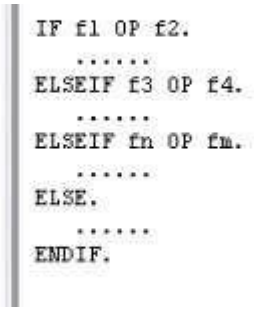
Comment out all of the preceding code, and click the ‘Pattern’ button, in the toolbar by Pretty Printer. A window will appear, and just select the ‘Other pattern’ field, and type “IF”. The structure of an IF statement will then appear in the code, which can be followed as a guide:



Ins. statement

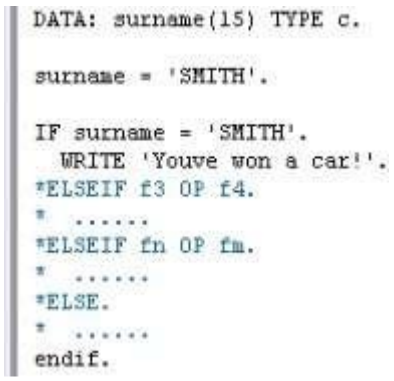
- CALL FUNCTION
- AABAP Objects patterns
- MESSAGE ID Cat E Number
- SELECT * FROM
- PERFORM
- AUTHORITY-CHECK
- WRITE
- CASE for status
- Structured data object
  - [✓ with fields from structure]
  - with TYPE for struct
- CALL DIALOG
- [✓ Other pattern]

IF



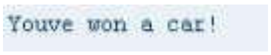
```
IF f1 OP f2.
   ......
ELSEIF f3 OP f4.
   ......
ELSEIF fn OP fm.
   ......
ELSE.
   ......
ENDIF.
```

Create a DATA statement, 15 characters of type ‘c’, and name this “surname”. Then on a new line give this the value ‘SMITH’. Then edit the auto-generated IF statement so that it looks like this.



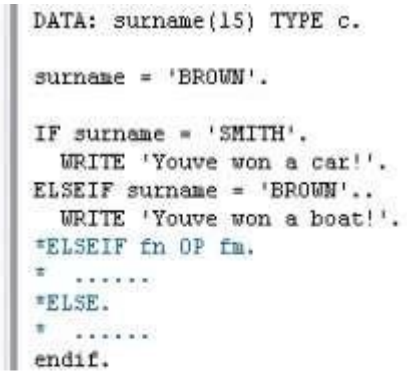
```
DATA: surname(15) TYPE c.

surname = 'SMITH'.

IF surname = 'SMITH'.
  WRITE 'Youve won a car!'.
*ELSEIF f3 OP f4.
*  ......
*ELSEIF fn OP fm.
*  ......
*ELSE.
*  ......
endif.
```

The IF statement here takes the form that if the value of ”surname” is ‘SMITH’, text will be displayed stating “Youve won a car!” (note that an apostrophe cannot be placed correctly in You’ve without making the code invalid). Then execute the code. The result should be:

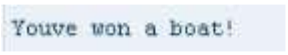


```
Youve won a car!
```

Next, this will be extended to include the ELSEIF statement which has been commented out above. Change the value of “surname” to ‘BROWN’. Then, add to the ELSEIF statement so that if the value of “surname” is ‘BROWN’, the output text will read “Youve won a boat!”:

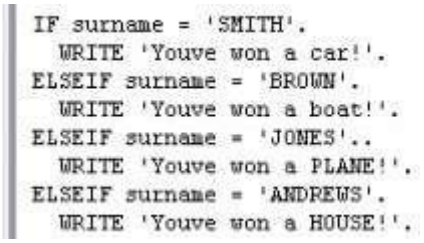


```
DATA: surname(15) TYPE c.

surname = 'BROWN'.

IF surname = 'SMITH'.
  WRITE 'Youve won a car!'.
ELSEIF surname = 'BROWN'..
  WRITE 'Youve won a boat!'.
*ELSEIF fn OP fm.
*  ......
*ELSE.
*  ......
endif.
```



```
Youve won a boat!
```

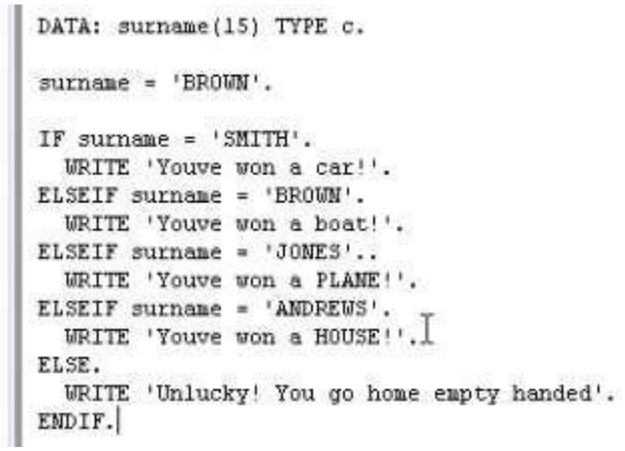
In this example, the first IF statement was not true, as the surname was not Smith. Hence this branch was not executed. The ELSEIF statement was true, so the text output assigned here appeared. The ELSEIF statement can be added to an IF statement any number of times, to designate the action taken in a number of situations:



```
IF surname = 'SMITH'.
  WRITE 'Youve won a car!'.
ELSEIF surname = 'BROWN'.
  WRITE 'Youve won a boat!'.
ELSEIF surname = 'JONES'..
  WRITE 'Youve won a PLANE!'.
ELSEIF surname = 'ANDREWS'.
  WRITE 'Youve won a HOUSE!'.
```

Depending on what the value of ‘surname’ is at any given time, a different branch will be executed.

There is also the ELSE statement. This is used for the last piece of the IF block, and is used if none of the values in the IF and ELSEIF statement are matched. The full block of code is shown below:

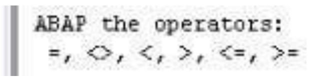


```
DATA: surname(15) TYPE c.

surname = 'BROWN'.

IF surname = 'SMITH'.
  WRITE 'Youve won a car!'.
ELSEIF surname = 'BROWN'.
  WRITE 'Youve won a boat!'.
ELSEIF surname = 'JONES'..
  WRITE 'Youve won a PLANE!'.
ELSEIF surname = 'ANDREWS'.
  WRITE 'Youve won a HOUSE!'.
ELSE.
  WRITE 'Unlucky! You go home empty handed'.
ENDIF.
```

With this block as it is now, there will always be an output, regardless of the value of ‘surname’, every possibility is now taken care of. The value will either match one of the first four, or the ELSE statement’s text will be displayed. The IF statement is very important for determining the flow of a program and will be used on a regular basis.

Linking Logical Expressions Together

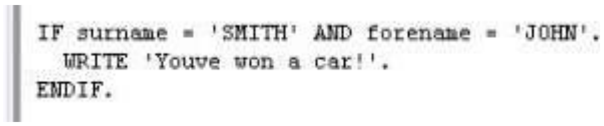
There are a whole set of ABAP operators which can be used with logic statements. With the IF statement so far the equals (=) operator has been used. The following can also be used here:



```
ABAP the operators:
 =, <>, <, >, <=, >=
```

(from left to right: equal to, NOT equal to, less than, greater than, less than OR equal to, greater than OR equal to. These can also be written with their text equivalents, in order: EQ, NE, LT, GT, LE, GE. The text versions are not commonly used.)

Logical expressions can be linked with the operators OR, AND and NOT. For example, one could add to the previous IF statement:

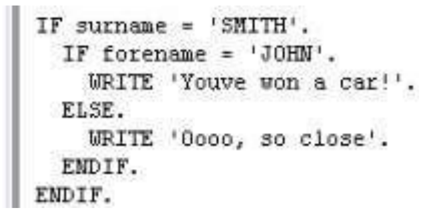


```
IF surname = 'SMITH' AND forename = 'JOHN'.
  WRITE 'Youve won a car!'.
ENDIF.
```

OR and NOT operate can also be used in exactly the same way

Nested If Statements

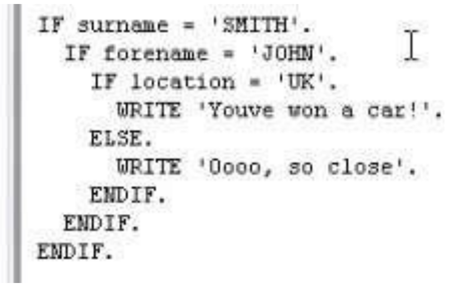
Nested IF statements allow one to include IF statements inside other IF statements, for example:



```
IF surname = 'SMITH'.
  IF forename = 'JOHN'.
    WRITE 'Youve won a car!'.
  ELSE.
    WRITE 'Oooo, so close'.
  ENDIF.
ENDIF.
```

Here, the first IF statement will discount records where the Surname field value does not equal ‘SMITH’. For all records with a Surname = ‘SMITH’, the second IF statement checks to see if the record being processed has a Forename = ‘JOHN’. If it does the message “Youve won a car!” will be output to the screen. If not, a consolatory message will be output instead.

You are not limited to just one nested IF statement. Nesting can continue down as many levels / branches as is required by the program being written, for example:



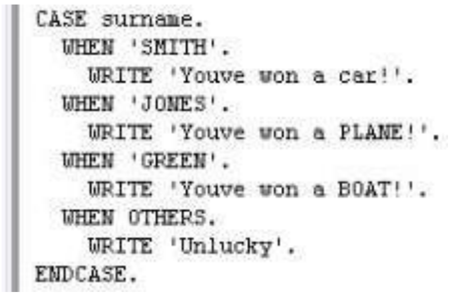
```
IF surname = 'SMITH'.
  IF forename = 'JOHN'.
    IF location = 'UK'.
      WRITE 'Youve won a car!'.
    ELSE.
      WRITE 'Oooo, so close'.
    ENDIF.
  ENDIF.
ENDIF.
```

Also, you do not simply have to nest statements one after another, but can put any other statements you need between, as long as the control structures are terminated correctly with, in this case, the ENDIF statement.

Case Statement

When logical expressions are created, and linked together, it is always important to make the code as readable as possible. Generating many logical expressions on one line can often be confusing. While the code will still work without problems, it is preferable to structure your code across multiple lines and make use of other control structures if possible.

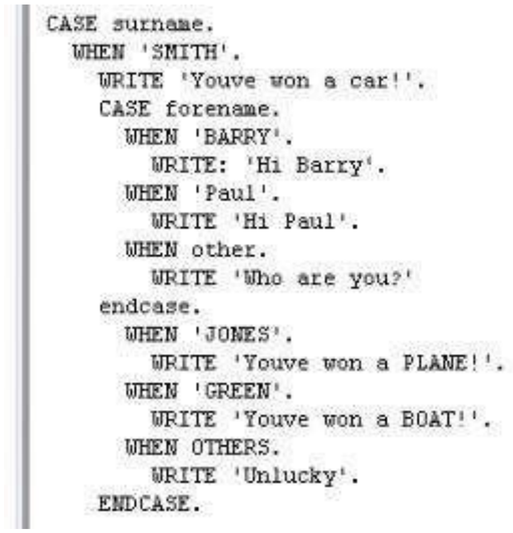
This is where the CASE statement can help. This does similar work to the IF statement but with the flexibility to make the code much more readable, but is at the same time limited to one logical expression. Here is an example code block for the CASE statement:



```
CASE surname.
  WHEN 'SMITH'.
    WRITE 'Youve won a car!'.
  WHEN 'JONES'.
    WRITE 'Youve won a PLANE!'.
  WHEN 'GREEN'.
    WRITE 'Youve won a BOAT!'.
  WHEN OTHERS.
    WRITE 'Unlucky'.
ENDCASE.
```

Like the IF statement, here the contents of the surname field are searched by the CASE statement, checking its contents and performing an action. The WHEN addition is used to check the field for different values, and WHEN OTHERS accounts for all values which are not specified elsewhere. The ENDCASE statement closes this control structure. This is in many ways much easier to read than a large amount of nested IFs and ELSEIFs.

You also have the facility to nest multiple CASE statements.



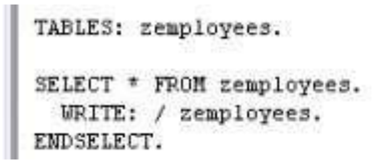
```
CASE surname.
  WHEN 'SMITH'.
    WRITE 'Youve won a car!'.
    CASE forename.
      WHEN 'BARRY'.
        WRITE: 'Hi Barry'.
      WHEN 'Paul'.
        WRITE 'Hi Paul'.
      WHEN other.
        WRITE 'Who are you?'
    endcase.
      WHEN 'JONES'.
        WRITE 'Youve won a PLANE!'.
      WHEN 'GREEN'.
        WRITE 'Youve won a BOAT!'.
      WHEN OTHERS.
        WRITE 'Unlucky'.
    ENDCASE.
```

Select Loops

This next section will discuss iteration statements, otherwise known as looping state- ments. These are used to execute a block of ABAP code multiple times.

Create another new program and call it Z_ITERATIONS_1.

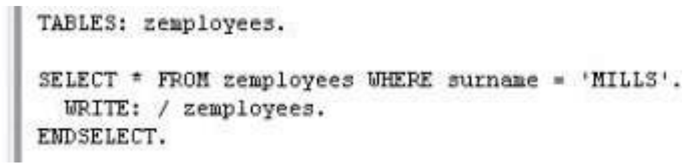
There are various ways to loop through blocks of code in an ABAP program, and these can be separated into those which have conditions attached and those which do not. The SELECT statement is a form of loop which has already been used. This statement allows you to iterate through a record set.



```
TABLES: zemployees.

SELECT * FROM zemployees.
  WRITE: / zemployees.
ENDSELECT.
```

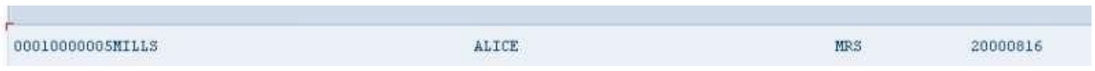
The asterisk (*) tells the program to select everything from the zemployees table, and this is followed by a WRITE statement to write the table to the output screen. The SELECT loop closed with ENDSELECT, at which point the loop returns to the start, writing each record in turn until there are no more records to process in the table.

This last example had no conditions attached. To add a condition is quite simple:



```
TABLES: zemployees.

SELECT * FROM zemployees WHERE surname = 'MILLS'.
  WRITE: / zemployees.
ENDSELECT.
```

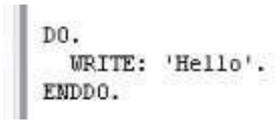
Here, only records where the surname is Mills will be selected and written to the output screen:



```
00010000005MILLS                              ALICE                         MRS          20000816
```

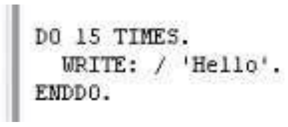
Do Loops

The DO loop is a simple statement, here declare DO. Add a WRITE statement, and then ENDDO:

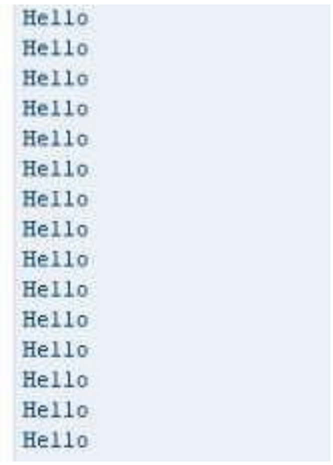


```
DO.
  WRITE: 'Hello'.
ENDDO.
```

You will notice there is nothing to tell the loop to end. If one tries to execute the code, the program will get stuck in a continuous loop endlessly writing out ‘Hello’ to the output screen. The transaction must be stopped and the code amended. A mechanism must be added to the DO loop to tell it when to stop processing the code inside it. Here, the TIMES addition is used. Amend the code as follows so that the system knows the loop is to be processed 15 times. Also here a ‘new line’ has been added before ‘Hello’:

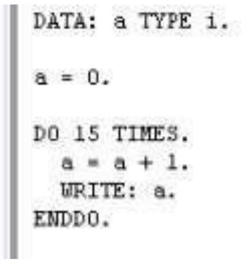


```
DO 15 TIMES.
  WRITE: / 'Hello'.
ENDDO.
```

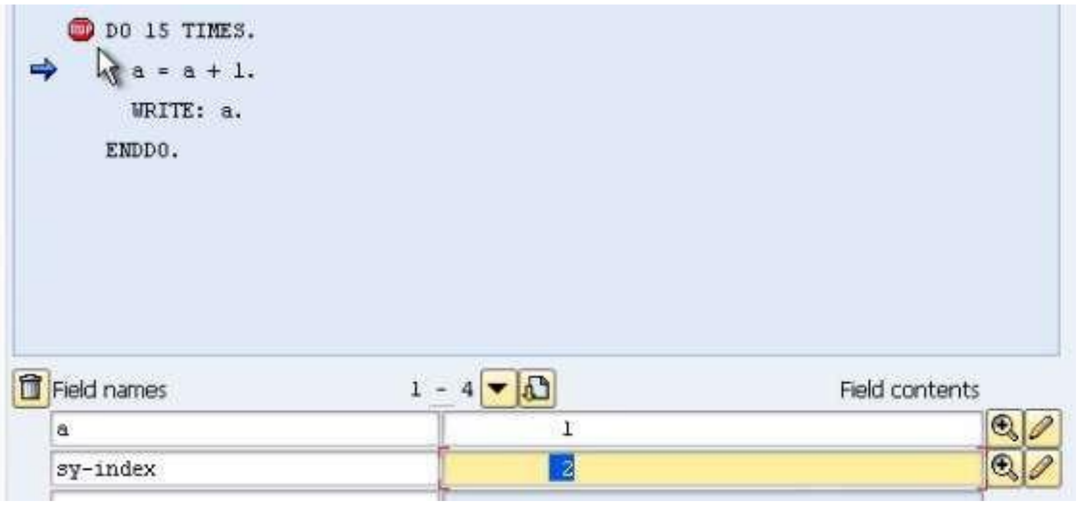

```
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

The DO statement is useful for repeating a particular task a specific number of times. Just remember to always include the TIMES addition.

Now try some processing with the DO loop. Create a DATA variable named ‘a’, of type integer, and set the value of this to 0. Then, inside the DO loop, include the simple calculation “a = a + 1”.

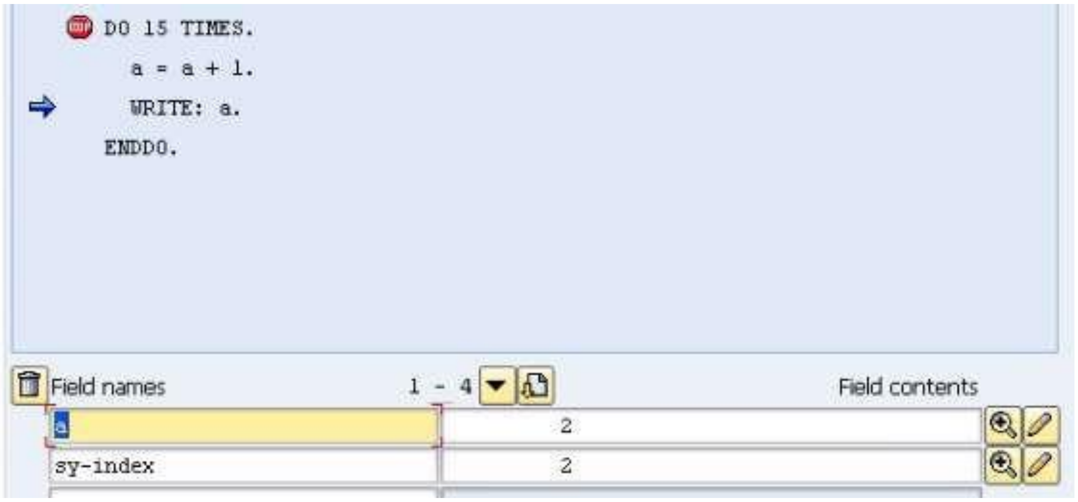

```
DATA: a TYPE i.

a = 0.

DO 15 TIMES.
  a = a + 1.
  WRITE: a.
ENDDO.
```

The system also contains its own internal counter for how many times a DO loop is executed, which can be seen when this is executed in debug mode. Set a breakpoint on the DO line, then execute the code, keeping an eye on the ‘a’ field in the Field names section, and also includes ‘sy-index’ in one of these fields. You will see that ‘a’ keeps a running count of how many times the DO loop executes as well as the system variable sy-index. The values will be the same for both, going up by 1 each time the loop completes. The sy- index variable will in fact update a line of code before the ‘a’ variable, as it counts the DO loops, and the ‘a’ refers to the calculation on the next line of code:



```
DO 15 TIMES.
  a = a + 1.
  WRITE: a.
ENDDO.
```

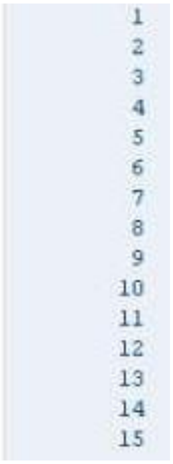
Field names 1 - 4	Field contents
a	1
sy-index	2



```
DO 15 TIMES.
  a = a + 1.
  WRITE: a.
ENDDO.
```

Field names 1 - 4	Field contents
a	2
sy-index	2

Note that here the blue arrow cursor has moved down a line in the second image, executing the next line of code. If one adds a new line to the WRITE statement in the initial code, the output window will appear like this:



```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

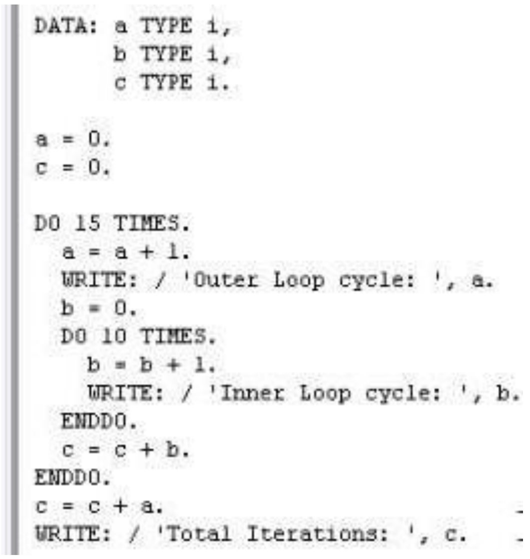
Nested Do Loops

DO loops can also be nested. If this is done, each nested loop will have its own sy-index created and monitored by the system. Be aware that when nesting many loops, it is important to consider how much work the system is being asked to do.

Add to the WRITE statement from the previous section a small amount of text reading ‘Outer Loop cycle:’ before outputting the value of ‘a’. This will allow ‘a’ to be monitored.

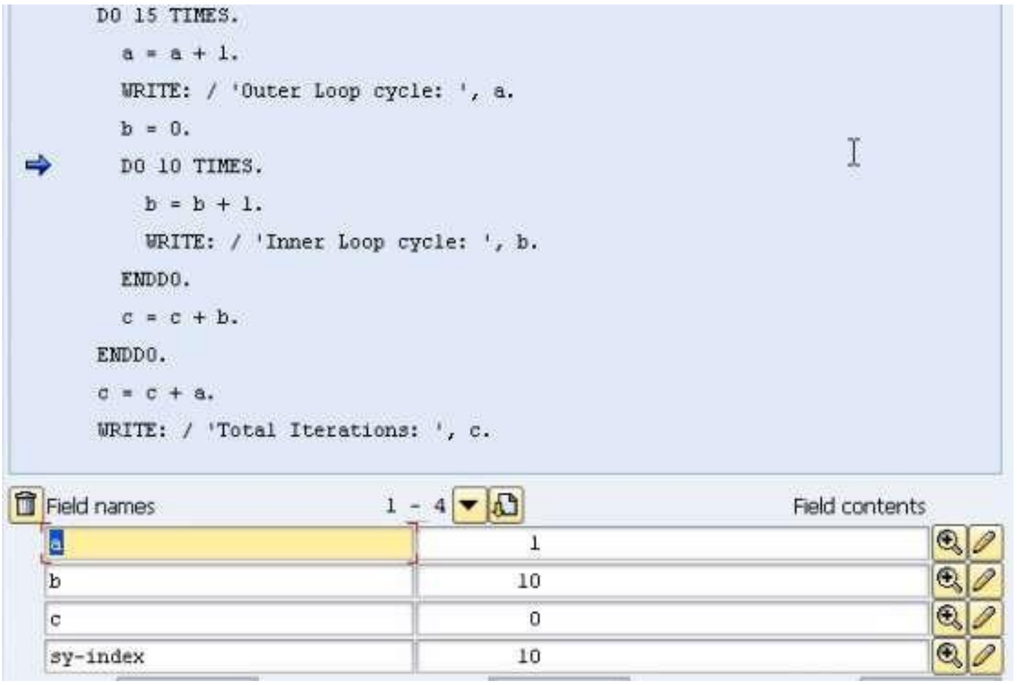
Then, under the WRITE statement, add a new DO statement to create the inner loop cycle, as below, as well as adding the extra data variables. The main loop will execute 15 times, but within each of these loops, the nested loop will execute 10 times. The variable named ‘c’ will count how many times the loop has occurred. Around 150 loops will execute here.

While the SAP system will certainly be able to handle this instantly, you should bear in mind that if this number was significantly larger and included more intensive processing than simple counting, this could take much longer:



```
DATA: a TYPE i,
      b TYPE i,
      c TYPE i.

a = 0.
c = 0.

DO 15 TIMES.
  a = a + 1.
  WRITE: / 'Outer Loop cycle: ', a.
  b = 0.
  DO 10 TIMES.
    b = b + 1.
    WRITE: / 'Inner Loop cycle: ', b.
  ENDDO.
  c = c + b.
ENDDO.
c = c + a.
WRITE: / 'Total Iterations: ', c.
```

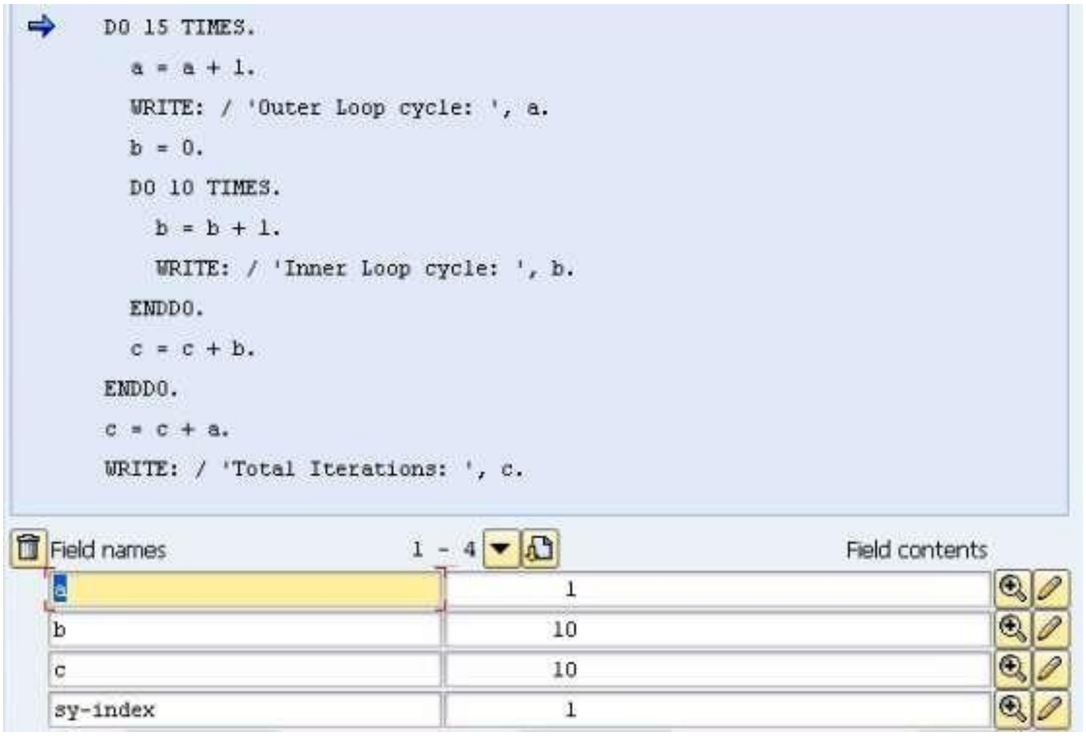
Set a breakpoint and execute this code in debug mode, keeping an eye on the values of a, b, c and sy-index in the Fields mode. As the DO loop is entered, the sy-index field will be- gin counting. Here, the inner loop has just occurred for the 10th time, noted by the 10 in sy-index (and indeed the value of ‘b’).



```
DO 15 TIMES.
  a = a + 1.
  WRITE: / 'Outer Loop cycle: ', a.
  b = 0.
  DO 10 TIMES.
    b = b + 1.
    WRITE: / 'Inner Loop cycle: ', b.
  ENDDO.
  c = c + b.
ENDDO.
c = c + a.
WRITE: / 'Total Iterations: ', c.
```

Field names 1 - 4	Field contents
a	1
b	10
c	0
sy-index	10

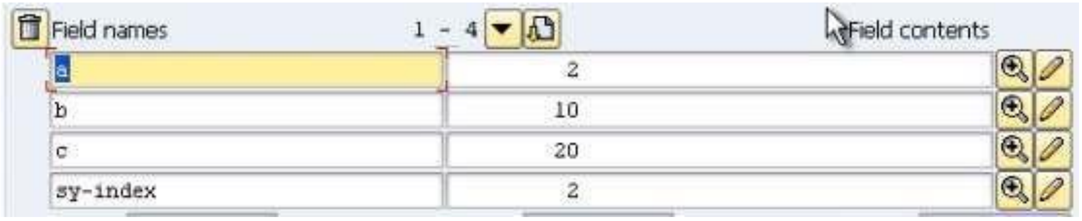
Then the full loop has completed once, the sy-index field displays 1 and the ‘c’ field has been filled in:



```
DO 15 TIMES.
  a = a + 1.
  WRITE: / 'Outer Loop cycle: ', a.
  b = 0.
  DO 10 TIMES.
    b = b + 1.
    WRITE: / 'Inner Loop cycle: ', b.
  ENDDO.
  c = c + b.
ENDDO.
c = c + a.
WRITE: / 'Total Iterations: ', c.
```

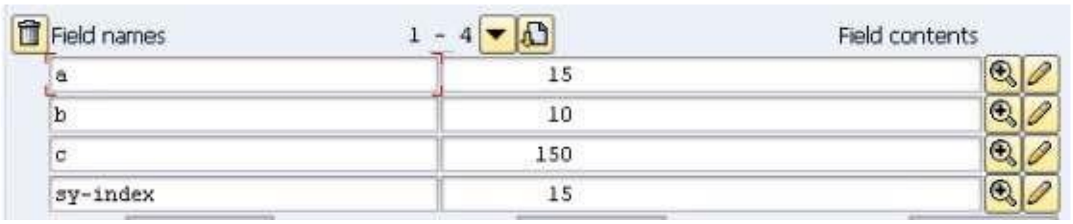
Field names 1 - 4	Field contents
a	1
b	10
c	10
sy-index	1

After the second full loop, sy-index and ‘a’ will display 2, ‘b’ will be 10 again (as its value is reset to 0 at the beginning of each loop) and ‘c’ will display 20 representing the number of calculations completed all together:



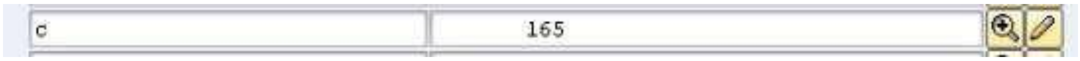
Field names 1 - 4	Field contents
a	2
b	10
c	20
sy-index	2

After the full 15 outer loops are completed, it will look like this:



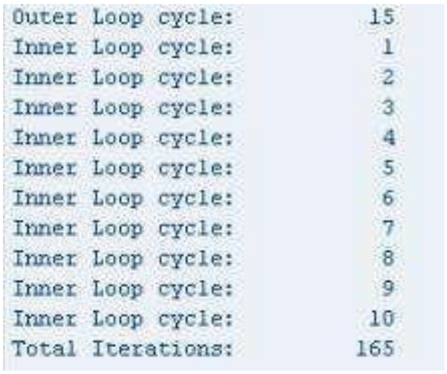
Field names	Field contents
a	15
b	10
c	150
sy-index	15

The value of ‘a’ is then added to ‘c’ to give the total number of both outer and inner loops completed:



c	165

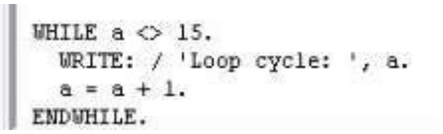
When the results are viewed in the output window, the last full loop will look like this:



```
Outer Loop cycle:         15
Inner Loop cycle:          1
Inner Loop cycle:          2
Inner Loop cycle:          3
Inner Loop cycle:          4
Inner Loop cycle:          5
Inner Loop cycle:          6
Inner Loop cycle:          7
Inner Loop cycle:          8
Inner Loop cycle:          9
Inner Loop cycle:         10
Total Iterations:        165
```

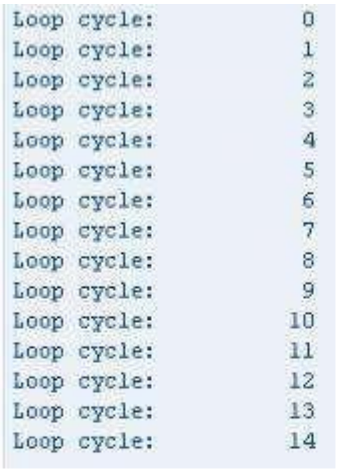
While Loops

The next looping statement to be examined is the WHILE loop. This differs from the DO loop in that it checks for a predefined condition within the loop before executing any code. All the code between the WHILE and ENDWHILE statements will be repeated as long as the conditions are met. As soon as the condition is false the loop terminates. Here, again the sy-index field can be monitored to see how many times the loop has executed.



```
WHILE a <> 15.
  WRITE: / 'Loop cycle: ', a.
  a = a + 1.
ENDWHILE.
```

So here, the loop will again cause the value of ‘a’ to take the form of incremental counting, and each time the loop is executed the value of ‘a’ will be written. The loop will continue as long as the value of ‘a’ is not equal to 15, and once it is, it will stop:

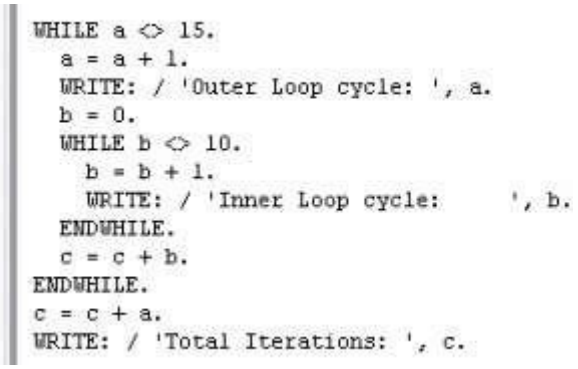


```
Loop cycle:          0
Loop cycle:          1
Loop cycle:          2
Loop cycle:          3
Loop cycle:          4
Loop cycle:          5
Loop cycle:          6
Loop cycle:          7
Loop cycle:          8
Loop cycle:          9
Loop cycle:         10
Loop cycle:         11
Loop cycle:         12
Loop cycle:         13
Loop cycle:         14
```

If one runs this in the debugger mode one will see that on the 15th loop, when the value of ‘a’ is 15, the code inside the statement is skipped over and the cursor jumps straight from WHILE to ENDWHILE.

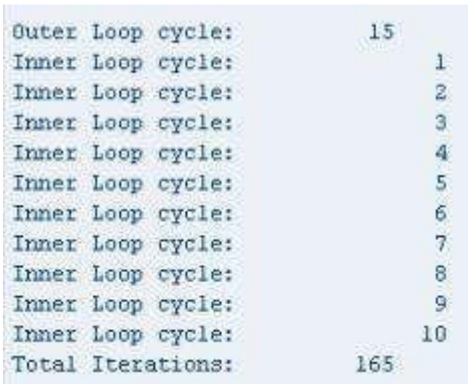
Nested While Loops

Just as with DO loops, WHILE loops can be nested. The process is exactly the same for both. Below is an example of nested WHILE loop statements.



```
WHILE a <> 15.
  a = a + 1.
  WRITE: / 'Outer Loop cycle: ', a.
  b = 0.
  WHILE b <> 10.
    b = b + 1.
    WRITE: / 'Inner Loop cycle:     ', b.
  ENDWHILE.
  c = c + b.
ENDWHILE.
c = c + a.
WRITE: / 'Total Iterations: ', c.
```

The output for this code would appear exactly the same as our nested DO loop example. The values of ‘b’ have also been indented slightly here for ease of reading:



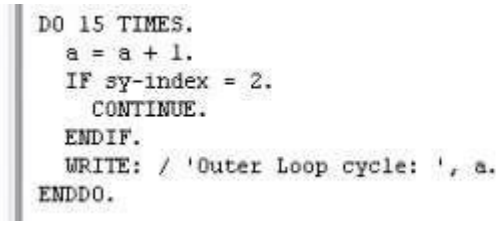
```
Outer Loop cycle:          15
Inner Loop cycle:              1
Inner Loop cycle:              2
Inner Loop cycle:              3
Inner Loop cycle:              4
Inner Loop cycle:              5
Inner Loop cycle:              6
Inner Loop cycle:              7
Inner Loop cycle:              8
Inner Loop cycle:              9
Inner Loop cycle:             10
Total Iterations:         165
```

Loop Termination – CONTINUE

Up until now, the loop statements set up have been allowed to use the conditions inside them to determine when they are terminated. ABAP also includes termination statements which allow loops to be ended prematurely. There are two categories of these, those which apply to the loop and those which apply to the entire processing block in which the loop occurs.

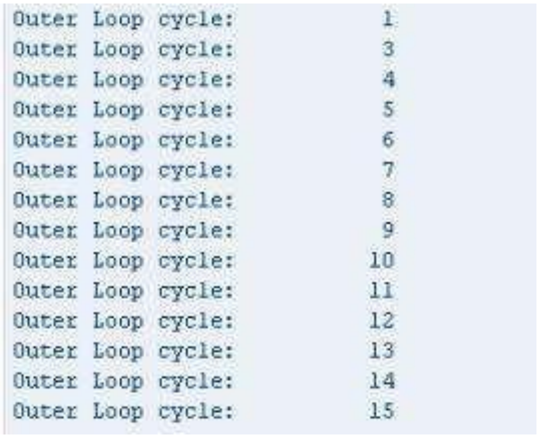
First, we will looks at how to terminate the processing of a loop. The first statement of importance here is the CONTINUE statement. This allows a loop pass to be terminated unconditionally. As the syntax shows, there are no conditions attached to the statement itself. It tells the program to end processing of the statements in the loop at the point where it appears and go back to the beginning of the loop again. If it is included within a loop, any statements after it will not be executed.

For the simple DO loop ,include an IF statement which includes CONTINUE inside it, like this:



```
DO 15 TIMES.
  a = a + 1.
  IF sy-index = 2.
    CONTINUE.
  ENDIF.
  WRITE: / 'Outer Loop cycle: ', a.
ENDDO.
```

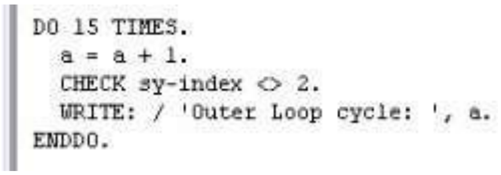
With this code, the second iteration of the loop (when the sy-index field, like the value of a, will read 2) will hit the CONTINUE statement and go back to the top, missing the WRITE statement. When this is output, the incremental counting will go from 1 to 3. *As with many of these statements, in debug mode, the operation can be observed more closely by executing the code line by line*.



```
Outer Loop cycle:          1
Outer Loop cycle:          3
Outer Loop cycle:          4
Outer Loop cycle:          5
Outer Loop cycle:          6
Outer Loop cycle:          7
Outer Loop cycle:          8
Outer Loop cycle:          9
Outer Loop cycle:         10
Outer Loop cycle:         11
Outer Loop cycle:         12
Outer Loop cycle:         13
Outer Loop cycle:         14
Outer Loop cycle:         15
```

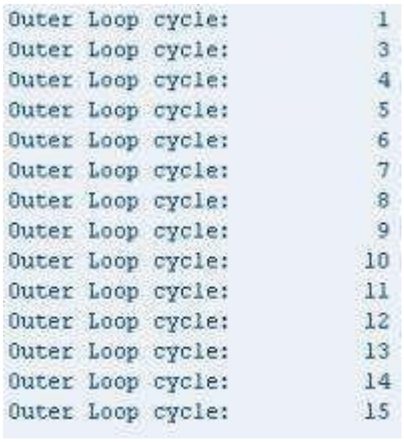
Loop Termination – CHECK

The CHECK statement works similarly to the CONTINUE statement, but this time allows you to check specific conditions. When the logic of a CHECK statement is defined, if the condition is not met, any remaining statements in the block will not be executed and processing will return to the top of the loop. It can be thought of as a combination of the IF and CONTINUE statements. To use the CHECK statement to achieve the same ends as in the example above, the syntax would look like this:

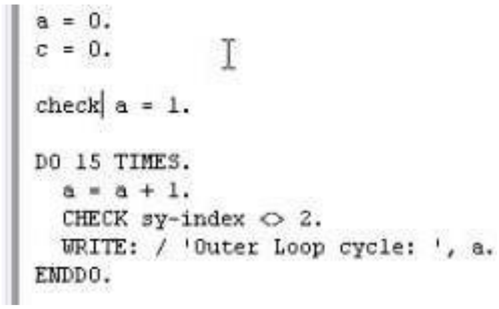


```
DO 15 TIMES.
  a = a + 1.
  CHECK sy-index <> 2.
  WRITE: / 'Outer Loop cycle: ', a.
ENDDO.
```

The program will check that the sy-index field does not contain a value equal to 2, and where it does not, will continue executing the code. When it does contain 2, the condition attached will not be true and the CHECK statement will cause the loop to start again, missing the WRITE statement. This can be executed in debug mode to closely observe how it works. The output window, once this is complete, will again appear like this:


```
Outer Loop cycle:          1
Outer Loop cycle:          3
Outer Loop cycle:          4
Outer Loop cycle:          5
Outer Loop cycle:          6
Outer Loop cycle:          7
Outer Loop cycle:          8
Outer Loop cycle:          9
Outer Loop cycle:         10
Outer Loop cycle:         11
Outer Loop cycle:         12
Outer Loop cycle:         13
Outer Loop cycle:         14
Outer Loop cycle:         15
```

When you are looking at programs created by other people, do not be surprised to see the CHECK statement used outside loops. It is not only used to terminate a loop pass, but can check, and terminate other processing blocks at any point if its particular conditions are not met. You must be aware of where the CHECK statement is being used, as putting it in the wrong place can even cause the entire program to terminate. For example here, the statement will only allow processing to continue if the value of ‘a’ is equal to 1. Since the value of ‘a’ equals 0, it will always terminate the program before the DO loop is reached:

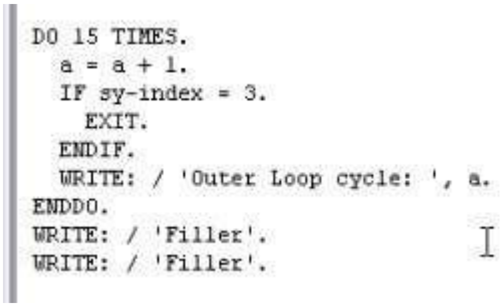

```
a = 0.
c = 0.

check a = 1.

DO 15 TIMES.
  a = a + 1.
  CHECK sy-index <> 2.
  WRITE: / 'Outer Loop cycle: ', a.
ENDDO.
```

Loop Termination – EXIT

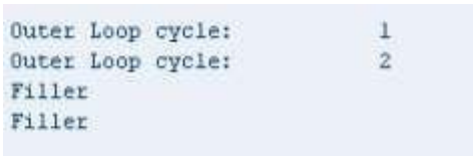
The EXIT statement can also be used to terminate loops. This again allows the loop to be terminated immediately without conditions. Unlike the CONTINUE statement though, it does not then return to the beginning of a loop but, terminates the loop entirely once it is reached. The program will then continue process the code immediately following the *end* statement of the loop.

If the exit statement is used within a nested loop, it will only be that nested loop which is terminated and the statement following the end of the nested loop will execute next in the higher level loop. Additionally it can, like the CHECK statement, be used outside loops, though again one must be careful doing this.

In the next example, regardless of the number of times the DO statement is told to be executed, on the third loop when the sy-index field contains the number 3, the loop will be terminated and the statement after ENDDO will be executed, writing “Filler” to the output screen.



```
DO 15 TIMES.
  a = a + 1.
  IF sy-index = 3.
    EXIT.
  ENDIF.
  WRITE: / 'Outer Loop cycle: ', a.
ENDDO.
WRITE: / 'Filler'.
WRITE: / 'Filler'.
```



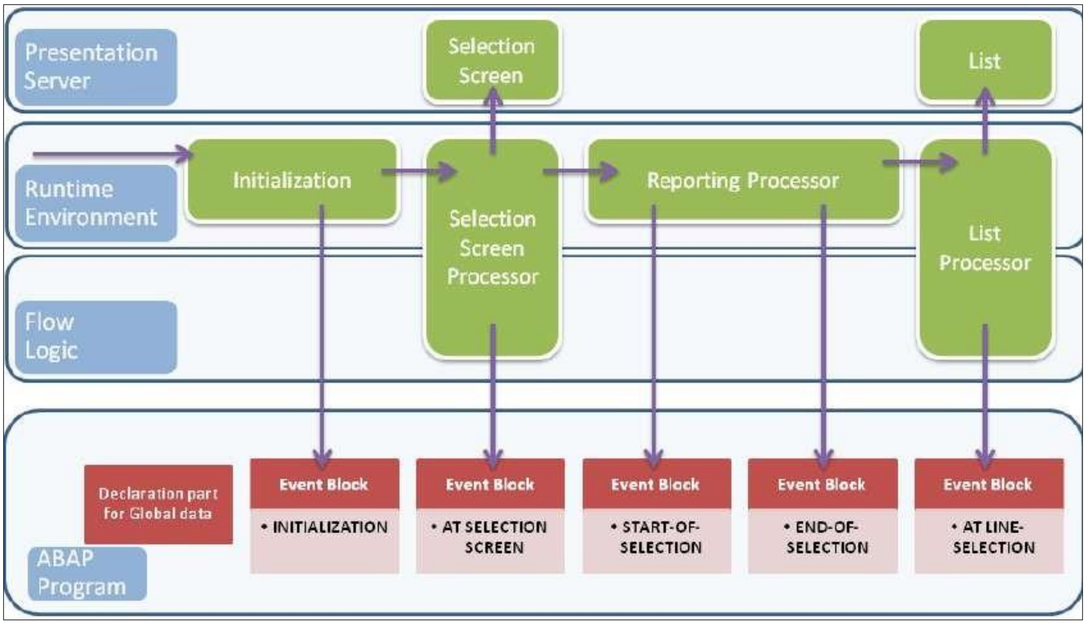
```
Outer Loop cycle:          1
Outer Loop cycle:          2
Filler
Filler
```

Selection Screens

Events

For selection screens to be built and used in a program, the first things to understand are events. Events are processing blocks, sections of code specific to the selection screens. The structure of an event starts with the event keyword, but does not have an ending keyword. The end of the event block of code is implicit, because the beginning of the next event will terminate the first, or the code itself will end.

When executable programs are run, they are controlled by a predefined process in the runtime environment, and a series of processes are called one after another. These processes trigger events, for which event blocks can be defined within the program. When a program starts, certain events work in a certain order.



At the top level is the SAP Presentation Server (Usually the SAP GUI), seen by the end user, with its selection screen and list output. When a program starts, from the left, with the declaration of global variables, the system will check to see if any processing blocks are included and will follow the sequence of events detailed above to execute these.

The initialization event block of code will only be run once, and will include things like the setting up of initial values for fields in the selection screen. It will then check whether a selection screen is included in the program. If at least one input field is present, control will be passed to the selection screen processor.

This will display the screen to the user, and it can then be interacted with. Once this is complete, the ‘at selection screen’ event block will process the information, and this is where one can write code to check the entries which have been made. If incorrect values have been entered, the code can catch these and can force the selection screen to be dis- played again until correct values are entered. Error messages can be included so that the user then knows where corrections must be made.

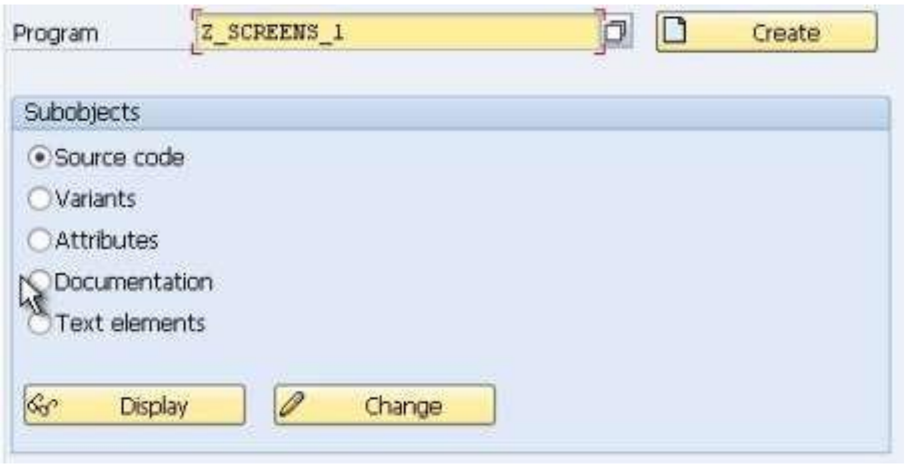
The ‘start of selection’ event block then takes control once the selection screen is filled correctly. This can contain code for, for example, setting up the values of internal tables or fields. There are other event blocks, which are visible in the diagram and there could be a number of others. The ones discussed here though, tend to be the main ones which would be used when working with selection screens to capture user input, which will then be used to process the rest of the program.

Once all of these event blocks have been processed, control is handed to the list proces- sor, which will output the report to the screen for the user to see. The list screen occa- sionally can be interactive itself, and the code in the event block ‘at line selection’ visible in the diagram takes responsibility for this.

This chapter will focus on creating the selection screen and making sure the user enters the correct values for the report, as well as ensuring the selection screen has a good inter- face.

Intro to Selection Screens

ABAP reports have 2 types of screens, selection screens and list output screens. The out- put window has already been used to produce list output screens. Selection screens are very commonly used. Indeed, when entering the ABAP editor, you are using a type of se- lection screen:



We will focus on reproduced this type of screen for use by our programs. These will allow the user to select data which will be used as parameters in the program. When one cre- ates a selection screen, in fact a dialogue screen is being created, but one does not have to write the dynpro code oneself. Only specific statements need to be used, and the sys- tem will take care of the screen flow logic itself.

List screens and selection screens are both dialogue programs. Every one of these has at least one dynpro which is held in what is called a module pool. A dynpro report program called ‘standard selection screen’ is called and controlled automatically by the runtime environment while the program is executed. The dynpro number itself is 1000. The user will only see the screen when the programmer includes the parameters in their program using specific ABAP statements. It is these ABAP statements which cause the screen to be generated and displayed to the user. This means it is easy for the programmer to start writing their own programs without having to think about code to control the screen.

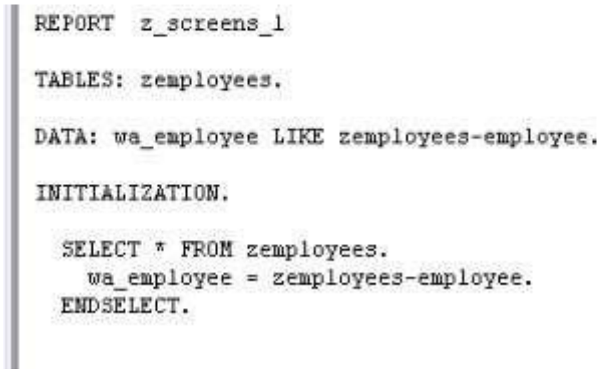
Creating Selection Screens

Create a brand new program in the ABAP editor, called Z_SCREENS_1.

First, the initialization event will be looked at. This is the first thing to be triggered in a program. In this example, imagine one wanted to know the last employee number which was used to create a record in the zemployees table. The initialization event is the correct place for this type of code, so that this information can then be displayed on the selection screen, alerting the user that values greater than this should not be entered as they will not return results.

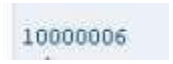
Begin by declaring the TABLES statement for zemployees. Then declare a DATA statement to hold the value of the last employee number that has been used in the table. This can be done with a work area declared LIKE the employee number field of the table.

Type “**INITITIALIZATION.**”, to begin the event block, followed by a SELECT statement where all records from zemployees are selected, and the work area is populated with the employee number field:



```
REPORT  z_screens_1

TABLES: zemployees.

DATA: wa_employee LIKE zemployees-employee.

INITIALIZATION.

  SELECT * FROM zemployees.
    wa_employee = zemployees-employee.
  ENDSELECT.
```

Then add a WRITE statement for the work area to output to the screen after the loop. Note that as the SELECT statement is a loop and does not contain a WRITE statement in- side it, the WRITE statement at the end only writes the final employee number which populates wa_employee, the last one which was used.



```
10000006
```

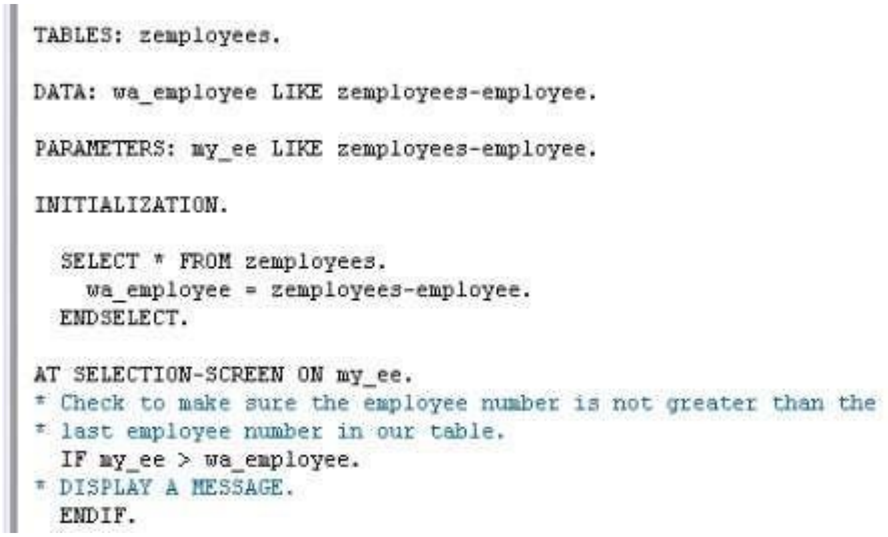
At Selection Screen

The “**at selection screen**” event is the next event block in the process. This will always be executed, if present, before the report can be processed. This, then, would be the ideal place to check the value which has been entered by the user as a new employee number. The entry screen will be looked at later, but here some code will be written which will al- low some kind of error message to be shown if an incorrect value is entered, telling the user to correct their entry.

The PARAMETERS statement will be used, though will not be gone in detail until later. This statement, allows you to declare a parameter input box which will appear on the screen. This works similarly to a DATA statement - “PARAMETERS: my_ee LIKE zemployees-employee.”, declaring the parameter as having the same properties as the employee number field.

Then declare the **AT SELECTION-SCREEN** event. This is declared with the addition **ON**, and my_ee added. This specifies that the 'at selection screen' block refers specifically to this parameter.

After this, an IF statement can be written, displaying an error message if the parameter value my_ee entered by the user is greater than the value held in wa_employee, the last employee number used:

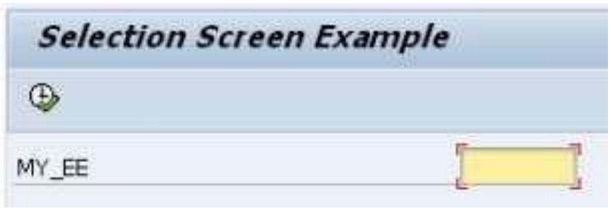


```
TABLES: zemployees.

DATA: wa_employee LIKE zemployees-employee.

PARAMETERS: my_ee LIKE zemployees-employee.

INITIALIZATION.

  SELECT * FROM zemployees.
    wa_employee = zemployees-employee.
  ENDSELECT.

AT SELECTION-SCREEN ON my_ee.
* Check to make sure the employee number is not greater than the
* last employee number in our table.
  IF my_ee > wa_employee.
* DISPLAY A MESSAGE.
  ENDIF.
```

As mentioned earlier, there is no need to terminate event blocks, as they are terminated automatically when a new one begins. Hence, the INITIALIZATION block ends as soon as the AT SELECTION-SCREEN block begins.

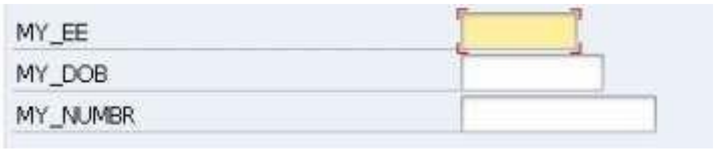
Parameters

Now, the PARAMETERS statement will be looked at in greater detail. Having defined the my_ee variable using this statement, the system will now automatically know that a selection screen is going to be generated. This statement is all that is necessary to display a field in a selection screen. If you display just the PARAMETERS variable on the screen, it will appear like this:



The syntax for PARAMETERS is very similar to the DATA statement. A name is given to the variable, a type can be given or the LIKE statement can be used to give the same proper- ties as another field already declared. An example appears below, followed by the output screen when this is executed:

```
PARAMETERS: my_ee LIKE zemployees-employee,
            my_DOB like zemployees-dob,
            my_numbr type i.
```



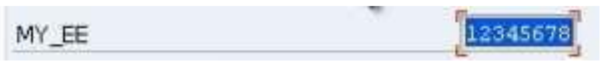
The DOB parameter takes on the same attributes as the *DOB* field in the table, to the ex- tent that it will even offer a drop-down box to select a date. The *my_numbr* parameter is not related to another field as has been declared as an integer type parameter. Addition- ally, note that parameter names are limited to 8 characters. Also, just like the DATA statement, a parameter can hold any data type, with the one exception, floating point numbers. You will notice also that the parameters in the output are automatically given text labels. The name of the parameter from the program, converted to upper case is used by default.

Now, some additions to the PARAMETERS statement will be examined.

DEFAULT

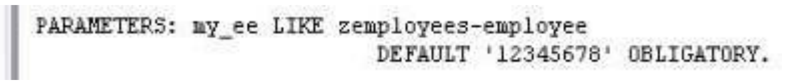
If you add this to the end of the statement follow by a value, the value will appear in the input box on the output screen giving a *default* value that the user can change if they wish.

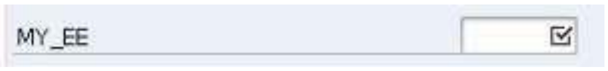
```
PARAMETERS: my_ee LIKE zemployees-employee DEFAULT '12345678'.
```

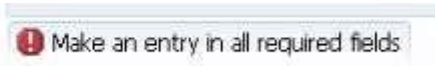

MY_EE 12345678

OBLIGATORY

To make the field mandatory for the user, the addition OBLIGATORY is used. A small tick- box will then appear in the field when empty, to indicate that a value must be inserted here. If one tries to enter the report with this empty, the status bar will display a message telling the user an entry must appear in this field:

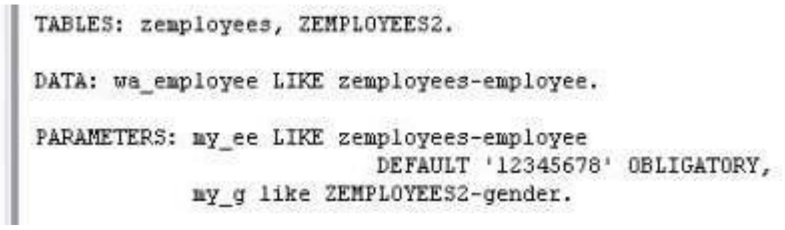

```
PARAMETERS: my_ee LIKE zemployees-employee
                     DEFAULT '12345678' OBLIGATORY.
```


MY_EE

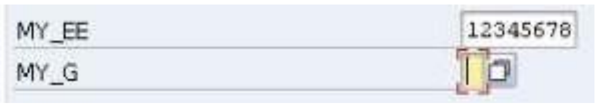

Make an entry in all required fields

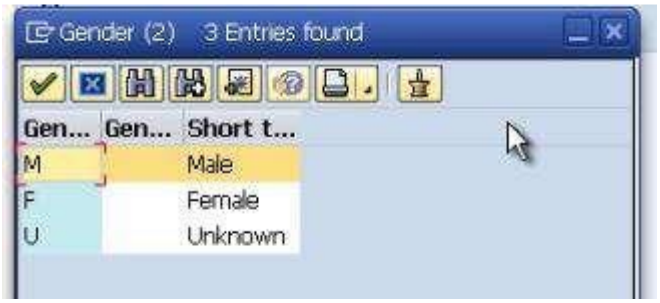
Automatic Generation of Drop-Down fields

For the next parameter, the zemployees2 table will be used. This must be added to the TABLES statement at the top of the program. A new parameter, named **my_g** here is set up for gender:


```
TABLES: zemployees, ZEMPLOYEES2.

DATA: wa_employee LIKE zemployees-employee.

PARAMETERS: my_ee LIKE zemployees-employee
                     DEFAULT '12345678' OBLIGATORY,
            my_g like ZEMPLOYEES2-gender.
```

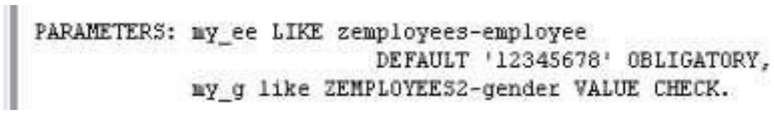
Since a number of values allowed to be entered for the gender field have been suggested in the table itself, a drop down box will appear by the parameter in the output window. Here one can see the ABAP dictionary working in tandem with the program to ensure that values entered into parameters correspond with values which have been set for the field in the table:


MY_EE 12345678
MY_G

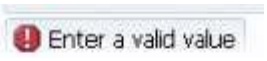


Gen...	Gen...	Short t...
M		Male
F		Female
U		Unknown

If one manually types an illegitimate entry into the gender box, an error message will not appear. Here, the VALUE CHECK addition is useful, as it will check any entry against the valid value list which is created in the ABAP dictionary. Now if one tries to enter an invalid value for the field, an error message is shown in the status bar:



```
PARAMETERS: my_ee LIKE zemployees-employee
                           DEFAULT '12345678' OBLIGATORY,
            my_g like ZEMPLOYEES2-gender VALUE CHECK.
```

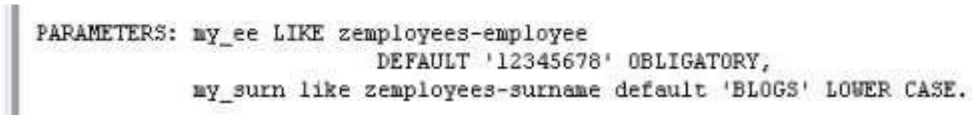


Enter a valid value

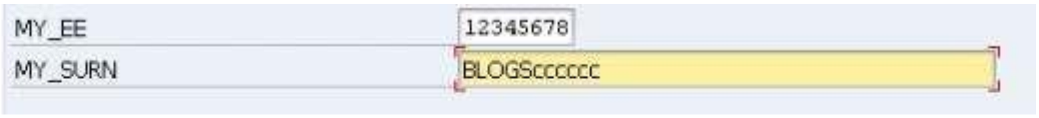
(*After this example, the zemployees2 table and gender parameter can be removed.*)

LOWER CASE

By default parameter names are converted to upper case, to get around this one must use the LOWER CASE addition. Create a new parameter named **my_surn** and make it LIKE *zemployees-surname* field. Give this a default value of 'BLOGS' and then add the LOWER CASE addition. When this is output, BLOGS still appears in upper case, but lower case letters can be added to the end of it. If these were entered without the LOWER CASE addition, they would automatically convert to upper case:



```
PARAMETERS: my_ee LIKE zemployees-employee
                           DEFAULT '12345678' OBLIGATORY,
            my_surn like zemployees-surname default 'BLOGS' LOWER CASE.
```



MY_EE	12345678
MY_SURN	BLOGScccccc

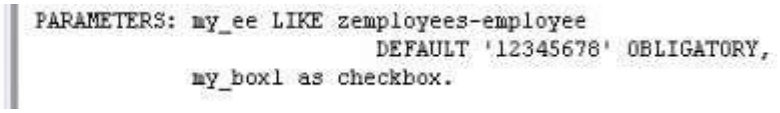
There are other additions which can be included with parameters, but these are generally the most common ones. To look at others, one can simply select the PARAMETERS statement, and press F1 for the ABAP help screen, which will explain further additions which can be used.

Check Boxes and Radio Button Parameters

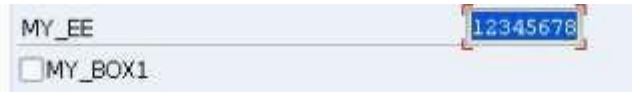
Check boxes and radio buttons can both be used to simplify the task of data entry for the end user. These are both forms of parameters.

A check box must always be of the character type ‘c’ with a length of 1. The contents stored in this parameter will either be an ‘x’, when it is checked, or empty when it is blank.

Define a new parameter called my_box1. Since this is type c, the type does not have to be declared. The field name is then followed by “**as checkbox**”. Note that the output differs slightly from other parameters by seeing the box on the left and the text to its right:



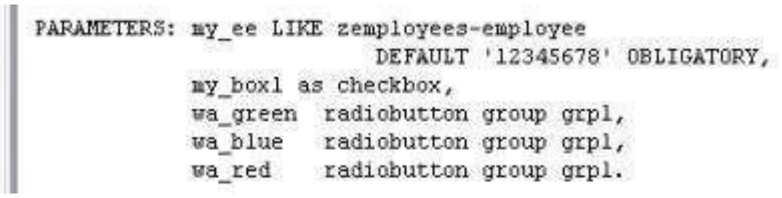
```
PARAMETERS: my_ee LIKE zemployees-employee
                       DEFAULT '12345678' OBLIGATORY,
            my_box1 as checkbox.
```



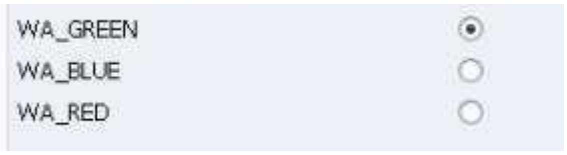
```
MY_EE                                12345678
☐MY_BOX1
```

Radio buttons are another common method for controlling the values stored in fields. A normal parameter field allows any value to be entered, while a check box limits the values to 2. Radio buttons, however, give a group of values which the user must choose one option from. Again, these are of data type c with 1 character.

To create a group of 3 radio buttons, 3 parameter fields must be set up. Each radio button must be given a name, in this example to select between colours (don’t forget, parameter names are limited to 8 characters), followed by “radiobutton”. These are then linked together by adding the word “group”, followed by a name for the group, here “grp1”. This can be seen in the image below:



```
PARAMETERS: my_ee LIKE zemployees-employee
                          DEFAULT '12345678' OBLIGATORY,
            my_box1 as checkbox,
            wa_green  radiobutton group grp1,
            wa_blue   radiobutton group grp1,
            wa_red    radiobutton group grp1.
```



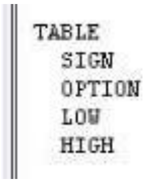
WA_GREEN ●
WA_BLUE ○
WA_RED ○

Select-Options

Next we will take a look at **SELECT-OPTIONS**. Parameters are useful for allowing the user to select individual values.. However, when multiple values are required, rather than set- ting up many different parameters, the select-options statement can be used.

The first thing to consider here is that internal tables will be used to store the values entered by the user. A detailed discussion regarding internal tables will be returned to, but for now, only what is necessary for select options will be looked at.

When a user wants to enter multiple individual values, or select a value range, these must be stored in a table in memory which the program can use. The internal tables to be used here are, similarly to parameters, limited to 8 characters and contain 4 fields which are defined when the statement is created. These fields are “**sign**”, “**option**”, “**low**” and “**high**”. The image below demonstrates the structure of this table:



```
TABLE
  SIGN
  OPTION
  LOW
  HIGH
```

When a user makes a choice, filling in a selection field on the screen, whether this is a single value or a range of values, a record is generated and put into this internal table. This table allows the user to enter as many records as they wish, which can then be used to filter the data.

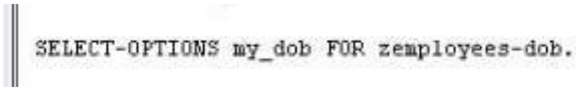
The “sign” field has a data type of c, and a length of 1. The data stored in this field determines, for each record, whether it is to be included or excluded from the result set that the final report selects from. The possible values to be held in this field are ‘I’ and ‘E’, for ‘inclusive’ and ‘exclusive’.

The “option” field also has a type of c, but this time a length of 2. This field holds the selection operator, such as EQ, NE, GT, LT, GE, LE (in order, as discussed previously: equal to, not equal to, greater than, less than, greater than or equal to, less than or equal to), as well as CP and NP. If a wild card statement is included here (such as * or +), the system will default this to CP.

The “low” field holds the lower limit for a range of values a user can enter, while the “high” field is the upper limit. The type and length of these will be the same as those for the database table to which the selection criteria are linked.

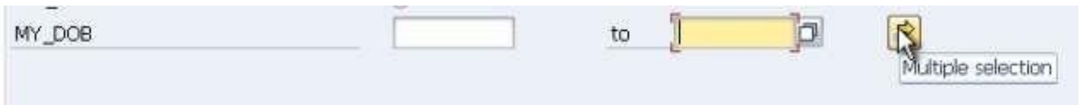
The reason for using select-options is that parameters only allow for one individual specific value to be used. If for example, one is using parameters to select from the DOB field in the zemployees table, these are very specific and so are likely to return, at best, one result, requiring the user to know the exact date of birth for every employee. The select-options statement allows one to set value ranges, wild cards and so on so that any selec- tion within that will return results.

First, type the statement SELECT-OPTIONS and then give a name to the field to be filled, for example **my_dob**. To declare the type, the addition **FOR** is used. This then link this to zemployees-dob:

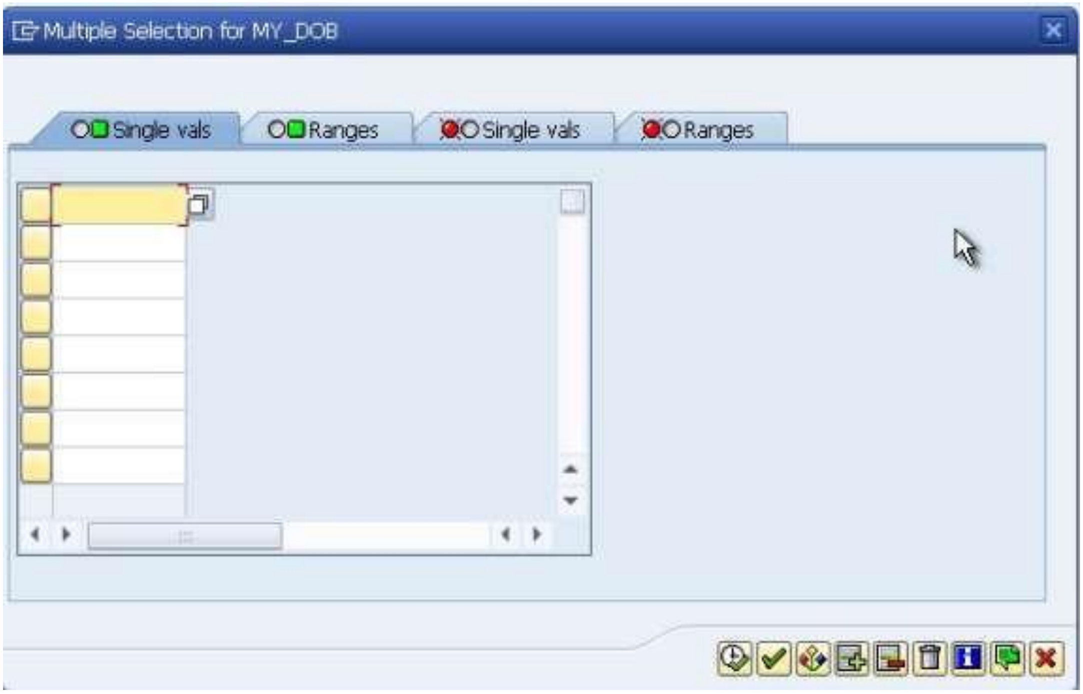


```
SELECT-OPTIONS my_dob FOR zemployees-dob.
```

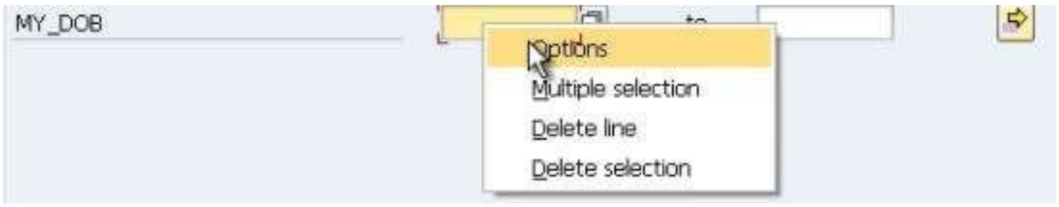
When this is output, 2 fields will appear, plus a ‘Multiple selection’ button:

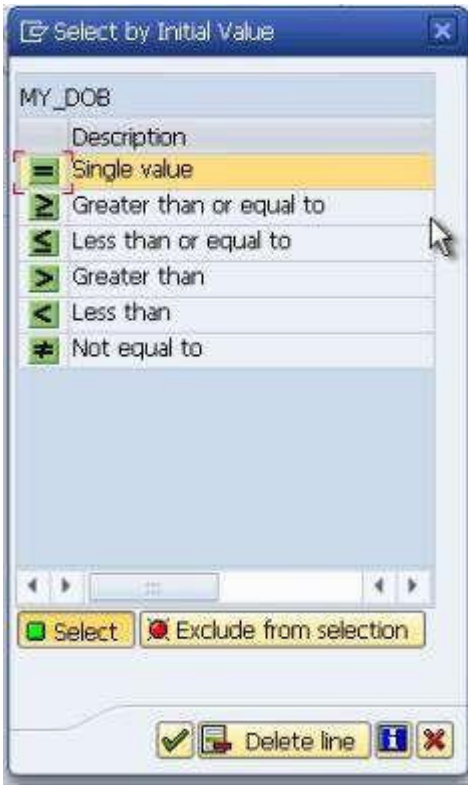


A value range can be selected by entering the low value into the left field and the high value in the right field. These two fields both include calendar drop down menus, making entry here even easier. If the ‘multiple selection’ button is clicked, a new pop-up box appears:



The fields here allow multiple single records, or value ranges to be searched for, as well as, in the case of the latter two tabs, excluded from one’s search results. All of the fields here as well correspond to the initial data type, and so will all feature calendar drop-downs. The buttons along the bottom add functionality, allowing values to be copied and pasted into the rows available, and indeed to create and delete rows among other options. Addi- tionally on the selection screen, if one right-clicks either field and chooses ‘options’, a list of the logical operators will be offered, allowing further customisation of the value ranges selected. This can also be done in the multiple selection box:





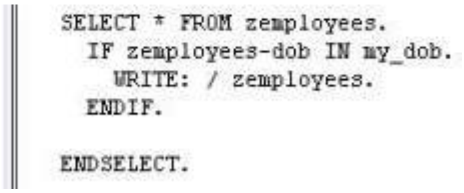
By filling in the fields offered via the SELECT-OPTIONS statement on the selection screen, each of the fields of the internal table can then be filled depending on the options chosen, telling the system exactly which values it should (and should not) be searching for.

Select-Option Example

With the select-options defined, some code will now be added.

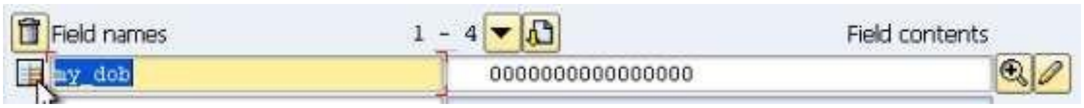
Create a SELECT statement, selecting all the records from zemployees. Then, inside the loop, add an IF statement, so that if a record from the zemployees table matches the value range selected at the selection screen, the full record is written in the output screen.

The IN addition ensures that only records which meet the criteria of my_dob, held in the internal table, will be included, and where they do not, the loop will begin again:

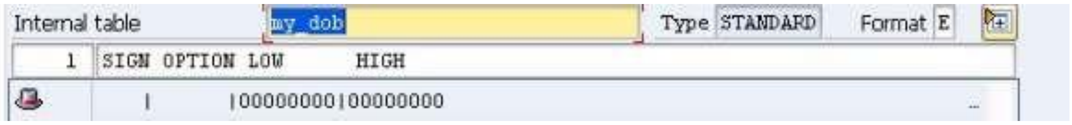


```
SELECT * FROM zemployees.
  IF zemployees-dob IN my_dob.
    WRITE: / zemployees.
  ENDIF.

ENDSELECT.
```

Put a breakpoint on the SELECT statement, so that you can watch the code's operation in debug mode. When you execute the code the selection screen will be displayed. Initially, do not enter any values for the DOB field. Execute the program and the debugger will ap- pear. Double click the my_dob field in the *field* mode. It will be shown to be empty and an icon will appear to the left indicating that it represents an internal table. If this is double clicked, the contents of the internal table are shown. Here, all fields are empty as no val- ues were inserted:



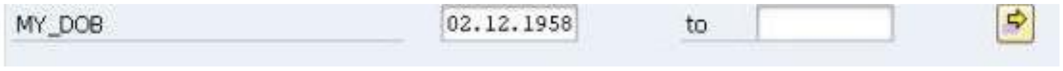
```
my_dob        0000000000000000
```



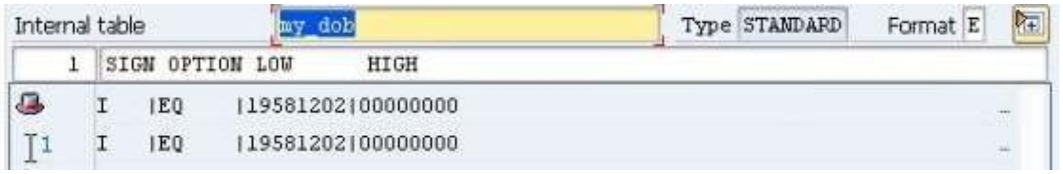
```
Internal table  my_dob   Type STANDARD   Format E
   1 SIGN OPTION LOW      HIGH
          |        |00000000|00000000
```

Run through the code and all of the records from the table should be written to the out- put screen, as no specific selection criteria were set.

Run the program again but this time include a value in the DOB field of the selection screen. This one corresponds to one of the records in the table:

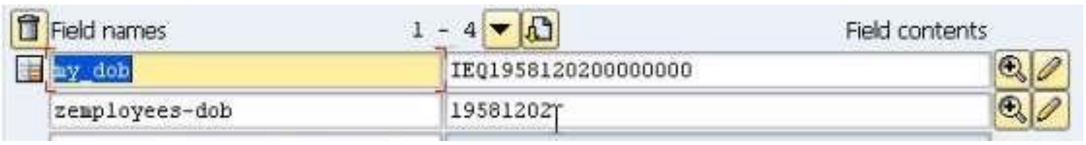


```
MY_DOB          02.12.1958     to
```

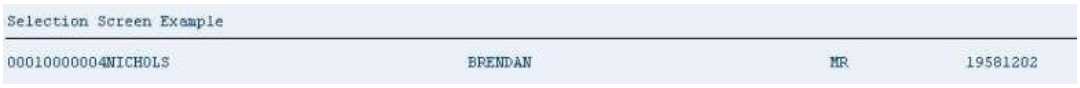


```
Internal table  my_dob   Type STANDARD   Format E
   1 SIGN OPTION LOW      HIGH
     I    |EQ     |19581202|00000000
  1  I    |EQ     |19581202|00000000
```

As the select loop is processed, eventually a matching record will be found. When this oc- curs, rather than skip back to the beginning of the loop, the WRITE statement is executed:

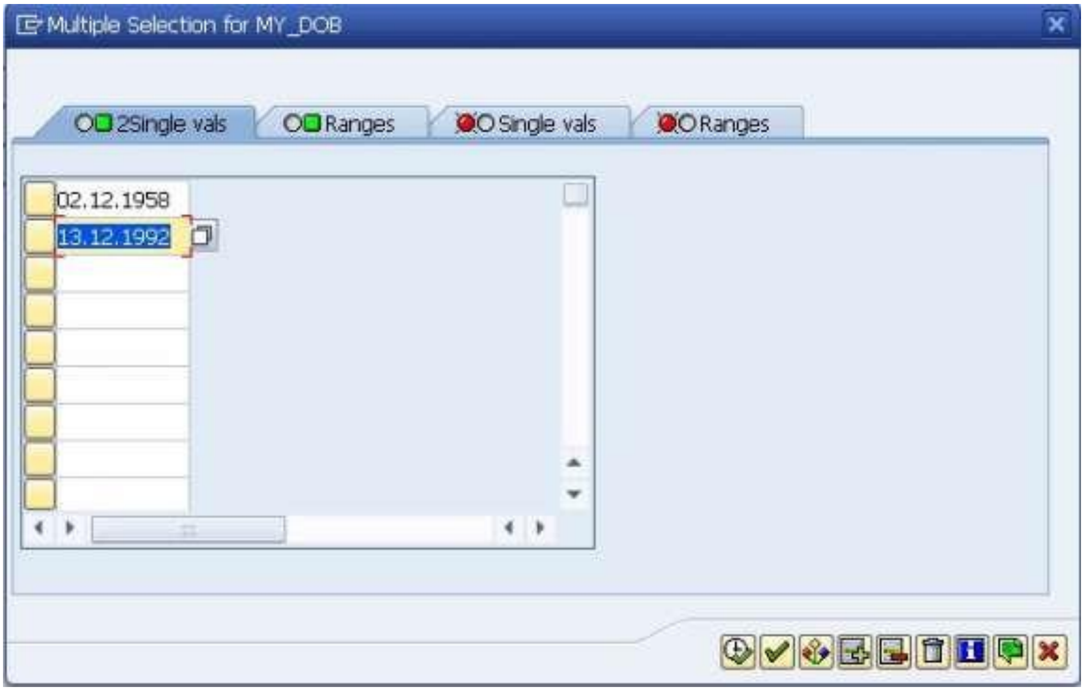


Field names	Field contents
my_dob	IEQ1958120200000000
zemployees-dob	19581202

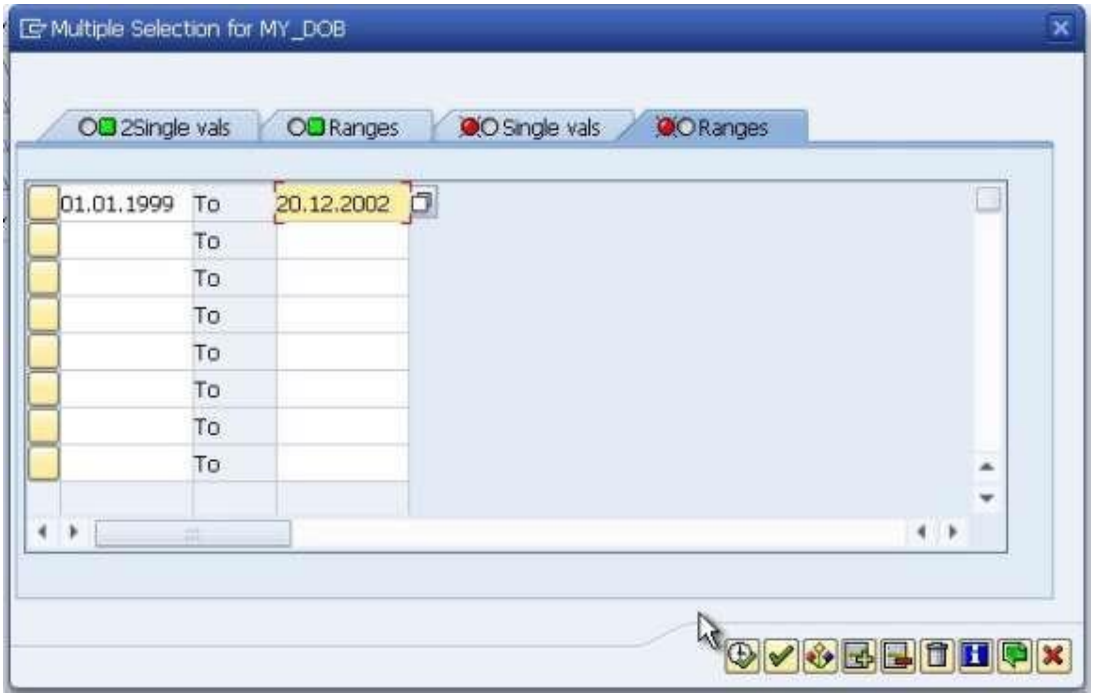


```
Selection Screen Example

00010000004NICHOLS                    BRENDAN                MR          19581202
```

Run the program again but this time try using the multiple selection tool to select several values for the DOB field, as well as excluding some:

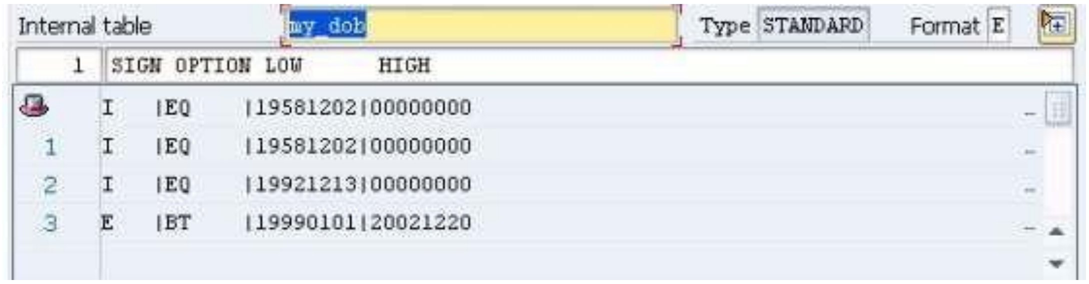


```
02.12.1958
13.12.1992
```



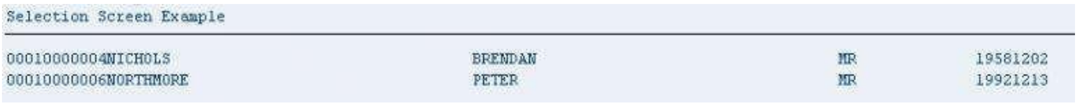
```
01.01.1999  To  20.12.2002
            To
            To
            To
            To
            To
            To
            To
```

The internal table now contains several entries for values to search for and to exclude from its search:



```
Internal table   my_dob                 Type STANDARD   Format E
    1  SIGN OPTION LOW      HIGH
       I    |EQ    |19581202|00000000
    1  I    |EQ    |19581202|00000000
    2  I    |EQ    |19921213|00000000
    3  E    |BT    |19990101|20021220
```

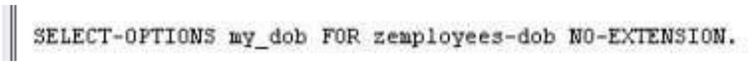
The records stored in the select-option table for my_dob show the different types of data the system uses to filter records depending on the entries we make in the multiple selec- tions window. Once the program is fully executed the output window then appears like this:



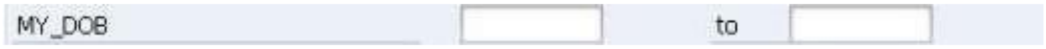
```
Selection Screen Example

00010000004NICHOLS                            BRENDAN                      MR          19581202
00010000006NORTHMORE                          PETER                        MR          19921213
```

Select-Option Additions

As with most statements, there are a number of additions which can be appended to SE- LECT-OPTIONS. Similarly to PARAMETERS, one can here use OBLIGATORY and LOWER CASE, and others in exactly the same way. Unique to this statement, however, is **NO- EXTENSION**, which prevents the multiple selection option from being offered to the user. The ability to select a value range still exists, but extending this via multiple selections is prevented:



```
SELECT-OPTIONS my_dob FOR zemployees-dob NO-EXTENSION.
```



Text Elements

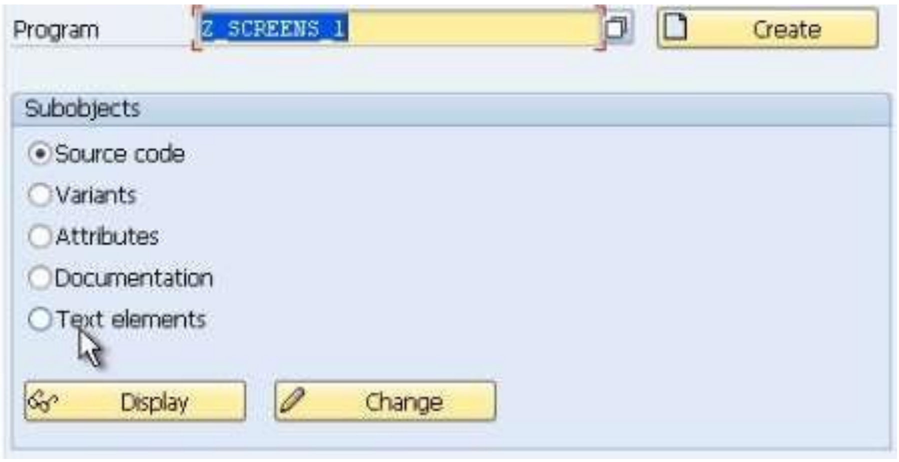
We have already touched on the fact that when parameters and select-options are de- clared the fields are labelled with the technical names given in the code. These fields still must be referenced using the technical name. However, it will be much preferable for the user to see some more descriptive text. Let's see how we can do this by using Text Ele- ments.

Every ABAP program is made up of sub-objects, like text elements. When one copies a program, the list of options offered asks which parts of the program one wants to copy. The source code and text elements here are mandatory, these are the elements which are essential to the program.

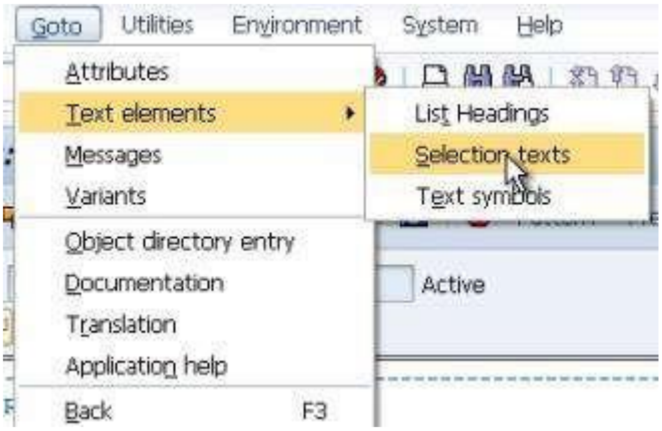
When text elements are created, they are created in *text pools*, which hold all of the text elements of the program. Every program created is language independent, meaning that the text elements created can be quickly and easily translated to other languages without the need for the source code to be changed.

There are three kinds of text elements which can be used in a program, selection texts, mentioned above, are one. The other two are text symbols and list headings. Text symbols can be created for a program so that one does not have to hard code literals into the source code. List headings, as the name indicates, refer to the headings used when creat- ing a report. By using these instead of hard coding them into the program, one can be cer- tain that they will be translated if the program is then used in another language. Also, if the headings need to be changed later on, one can just change the list headings set rather than going into the code and doing this manually.

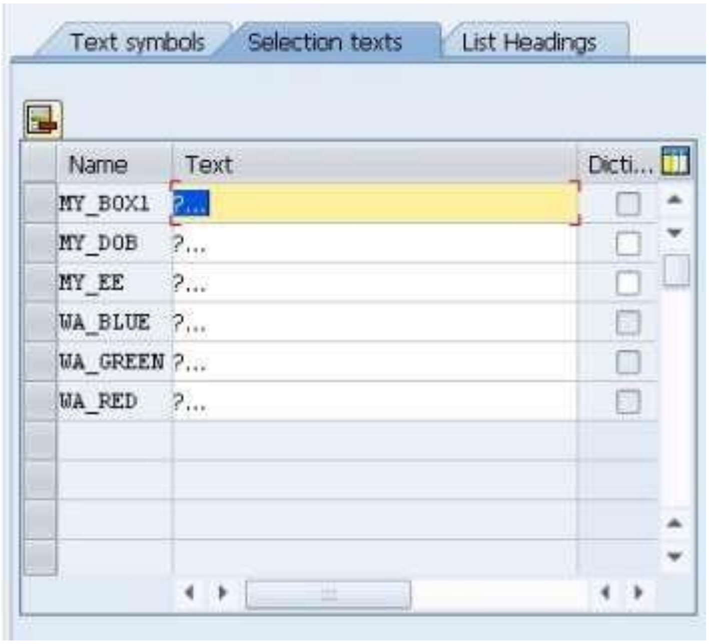
Selection texts allow text elements to be displayed on the screen so that the user does not have to see the technical names for fields and the like. There are several ways to navigate to the screen where these can be created and changed. At the initial ABAP editor screen, there is in fact an option for creating text elements:



Alternatively, if one is already inside the program, this can be reached through the 'Goto' menu, 'Text elements' and select 'Selection texts':

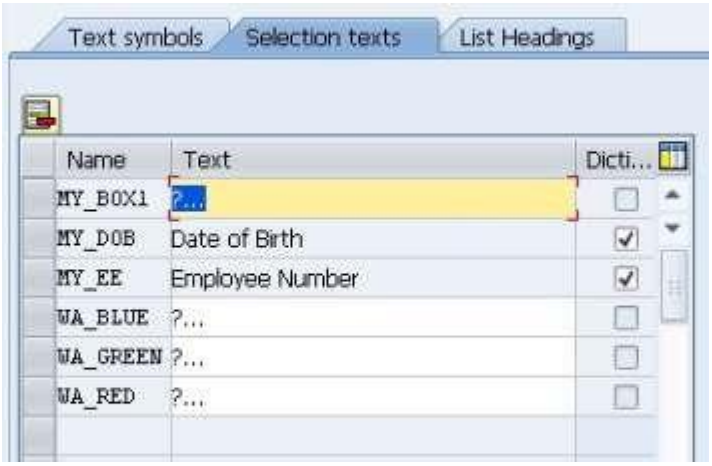


If this is clicked, a screen will appear where selection texts can be created for all of the technical field names which appear at the selection screen:



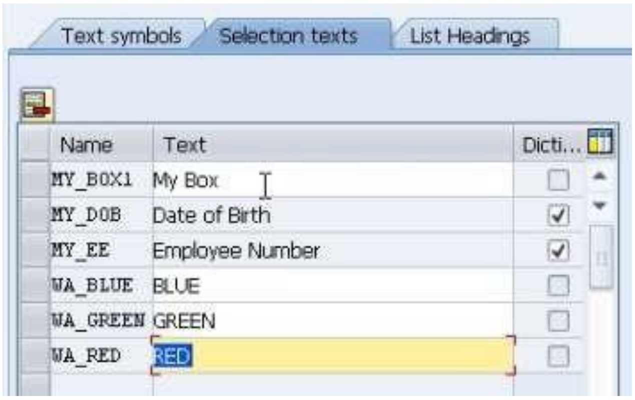
Name	Text	Dicti...
MY_BOX1	?...	☐
MY_DOB	?...	☐
MY_EE	?...	☐
WA_BLUE	?...	☐
WA_GREEN	?...	☐
WA_RED	?...	☐

The third column here is for ‘Dictionary reference’, which recognises that some of these fields are linked to fields already created in the ABAP dictionary. If one checks this box and clicks save, the field names from the initial fields and the ABAP dictionary automatically appear. You can of course choose not to use the text here and overwrite it yourself.



Name	Text	Dicti...
MY_BOX1	?...	☐
MY_DOB	Date of Birth	✓
MY_EE	Employee Number	✓
WA_BLUE	?...	☐
WA_GREEN	?...	☐
WA_RED	?...	☐

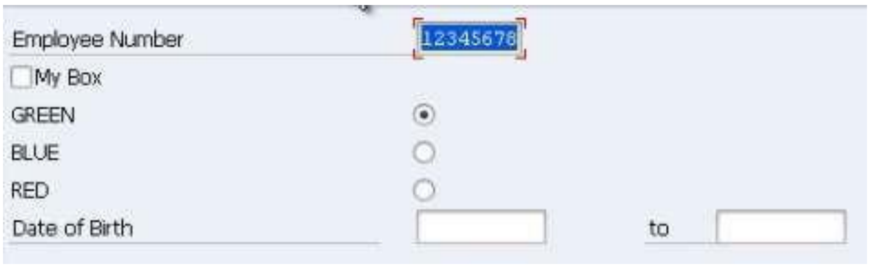
For the others fields, the text must be manually typed in, up to a 30 character limit:



Text symbols | Selection texts | List Headings

Name	Text	Dicti...
MY_BOX1	My Box	☐
MY_DOB	Date of Birth	✓
MY_EE	Employee Number	✓
WA_BLUE	BLUE	☐
WA_GREEN	GREEN	☐
WA_RED	RED	☐

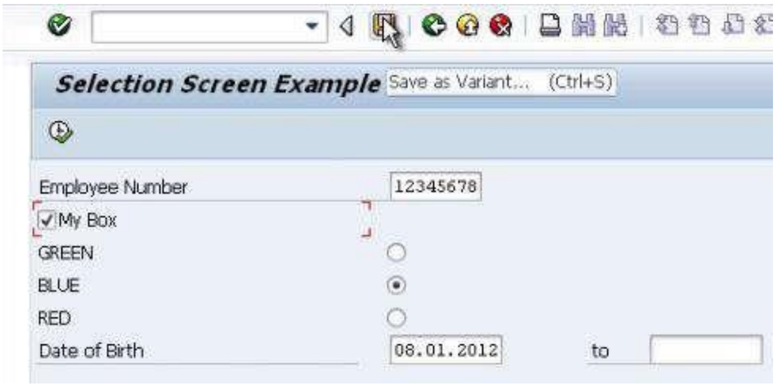
Text Elements must then be activated and once this is done, they are automatically saved and will appear on the selection screen in place of the technical names. The output screen will now look like this:



Employee Number	12345678		
☐ My Box			
GREEN	●		
BLUE	○		
RED	○		
Date of Birth		to	

Variants

When a user fills in a selection screen, there is the option of saving the entry. This is called a variant:

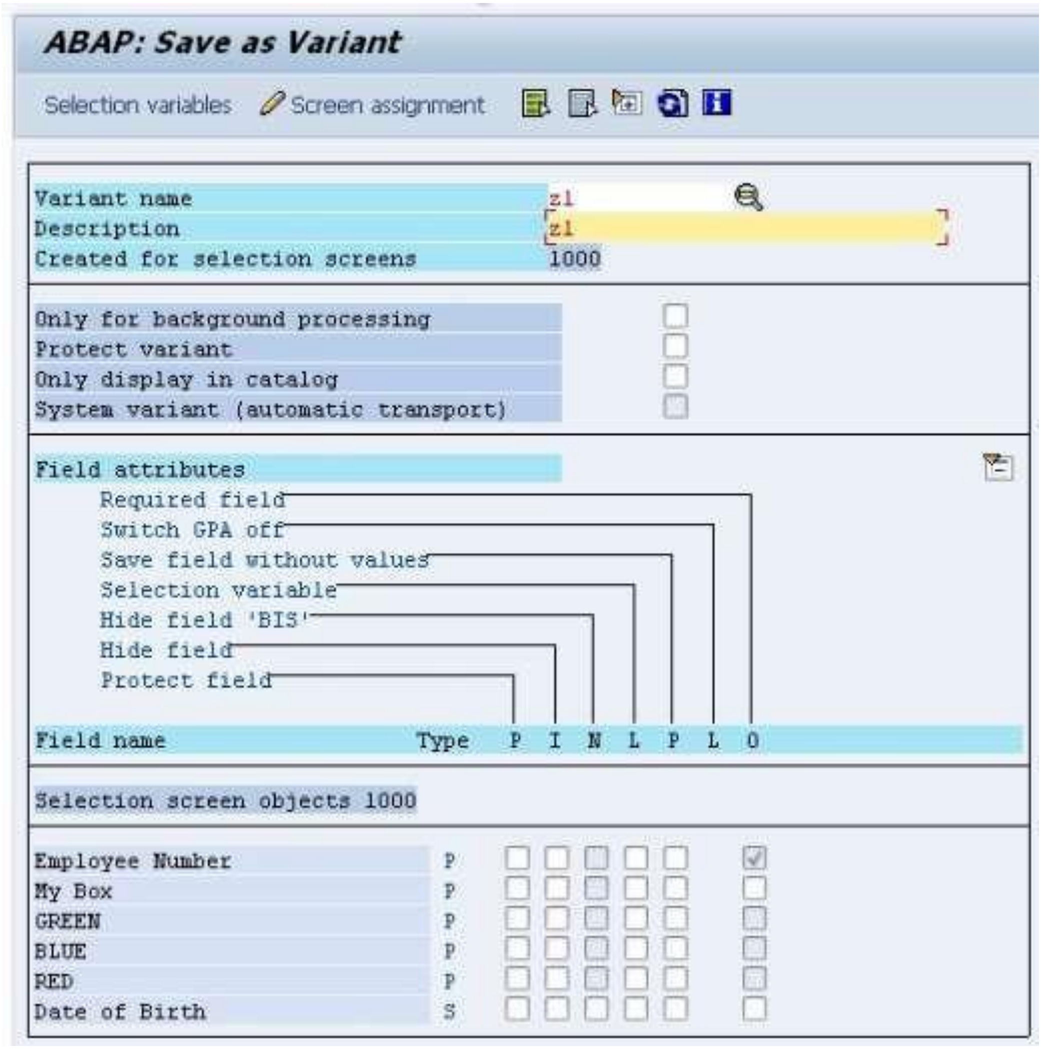


Selection Screen Example

Save as Variant... (Ctrl+S)

Employee Number	12345678		
✓ My Box			
GREEN	○		
BLUE	●		
RED	○		
Date of Birth	08.01.2012	to	

Once this is done, a new screen appears. As long as a name and description are given, this can be saved for use later on:



ABAP: Save as Variant

Variant name	z1
Description	z1
Created for selection screens	1000

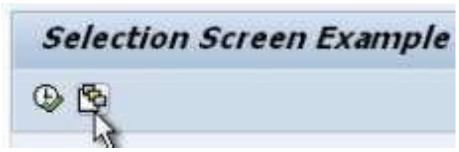
Only for background processing	☐
Protect variant	☐
Only display in catalog	☐
System variant (automatic transport)	☐

Field attributes
Required field
Switch GPA off
Save field without values
Selection variable
Hide field 'BIS'
Hide field
Protect field

Selection screen objects 1000

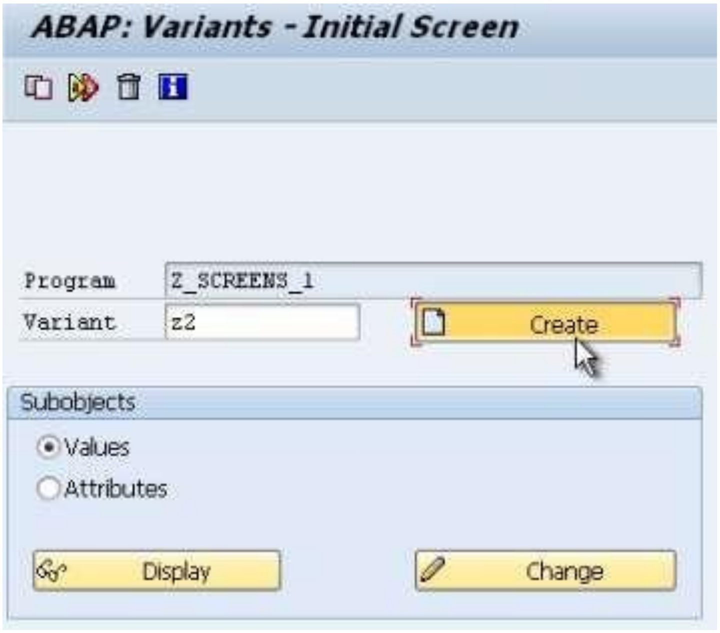
Field name	Type	P	I	N	L	P	L	O
Employee Number	P	☐	☐	☐	☐	☐		✓
My Box	P	☐	☐	☐	☐	☐		☐
GREEN	P	☐	☐	☐	☐	☐		☐
BLUE	P	☐	☐	☐	☐	☐		☐
RED	P	☐	☐	☐	☐	☐		☐
Date of Birth	S	☐	☐	☐	☐	☐		☐

Once saved a new button appears on the selection screen next to the execute button, named ‘**Get variant**’ allowing the variant entry to be recalled.



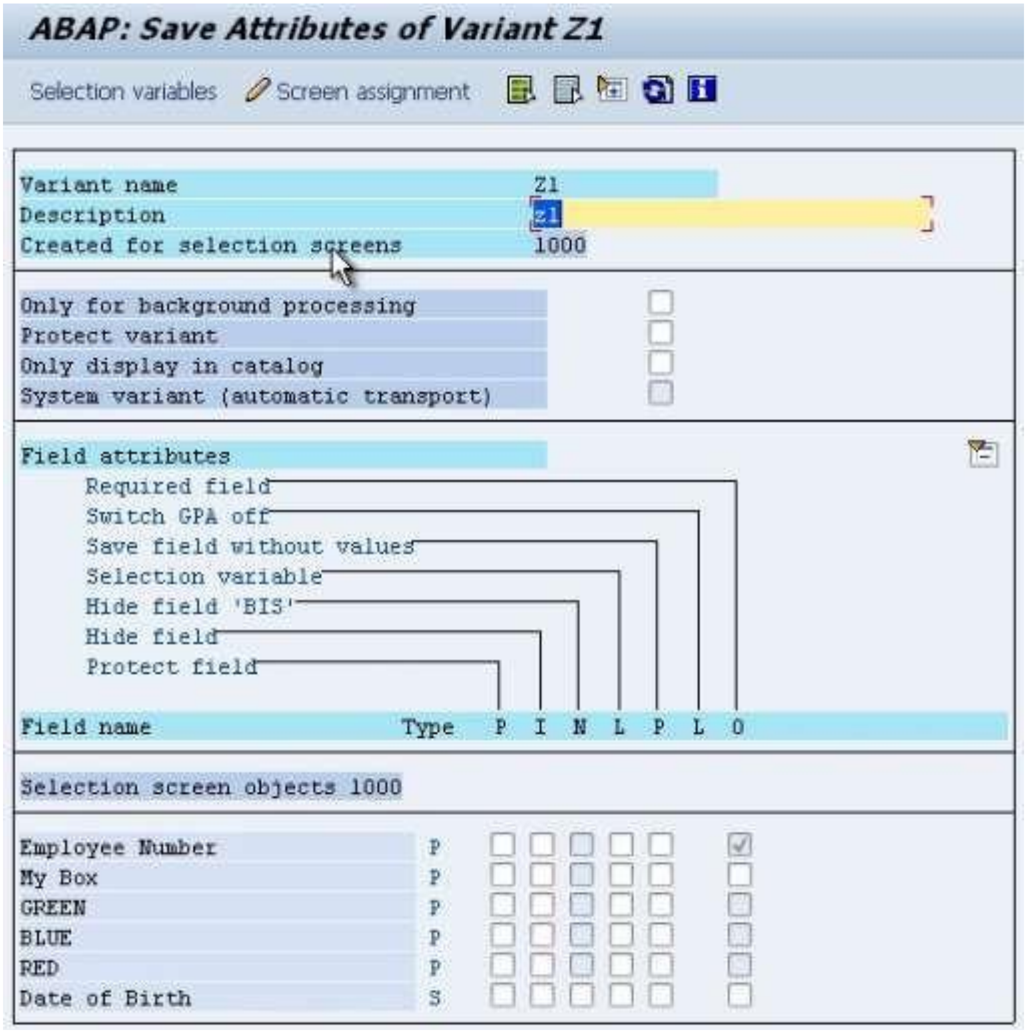
A box appears allowing a variant to be selected and when selected, the fields are populated with the data from that particular entry. Another way to create variants is via the initial ABAP editor screen.

Choose the 'Variants' option. A new variant name can be entered and then the variant can be created:



Program Z_SCREENS_1
Variant z2
(● Values) ○ Attributes

Once 'Create' is clicked, the selection screen appears and you can proceeds as normal, saving the attributes of the new variant once the entries have been made. You can then choose between displaying and changing the values and attributes of the variant ('Values' will show the selection screen, 'Attributes' the screen below. These two views can be switched between):



Variant name	Z1
Description	z1
Created for selection screens	1000

Only for background processing	☐
Protect variant	☐
Only display in catalog	☐
System variant (automatic transport)	☐

Field attributes
Required field
Switch GPA off
Save field without values
Selection variable
Hide field 'BIS'
Hide field
Protect field

Selection screen objects 1000

Field name	Type	P	I	N	L	P	L	O
Employee Number	P	☐	☐	☐	☐	☐		✓
My Box	P	☐	☐	☐	☐	☐		☐
GREEN	P	☐	☐	☐	☐	☐		☐
BLUE	P	☐	☐	☐	☐	☐		☐
RED	P	☐	☐	☐	☐	☐		☐
Date of Birth	S	☐	☐	☐	☐	☐		☐

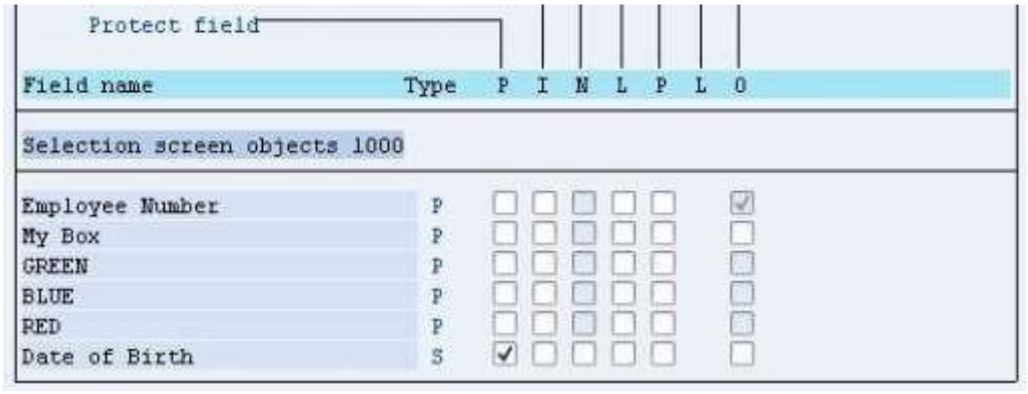
The '**Only for background processing'** check box allows you to tell the system to only use this variant as part of a background job. Here, a job can be scheduled to run overnight so the program does not in fact have to be monitored.

The '**Protect variant**' option prevents other users from being able to select this variant and using it on their reports.

'Only display in catalog' effectively makes the variant inactive, it will exist, but when a user views the drop-down menu of existing variants, it will not appear.

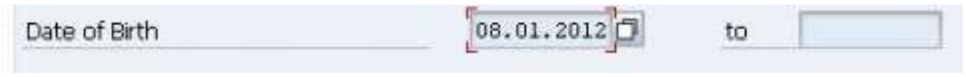
The **‘Field attributes’** section allows the list of possible attributes displayed to be assigned to the fields in the bottom section of the screen, via the check boxes. Experiment with the different options available and see the results. For example, you can see that the ‘Re- quired field’ check box for ‘Employee number’ has been filled here, as this was labelled OBLIGATORY in the program. The P’s and one S which appear by the fields simply refer to whether each field is a parameter or select-option.

Choose ‘**Protect field**’ for the Date of Birth field; it will no longer be possible to change the value set until such time as this box is un-checked. In the image below you can see this field has been greyed out and cannot be changed:



Protect field

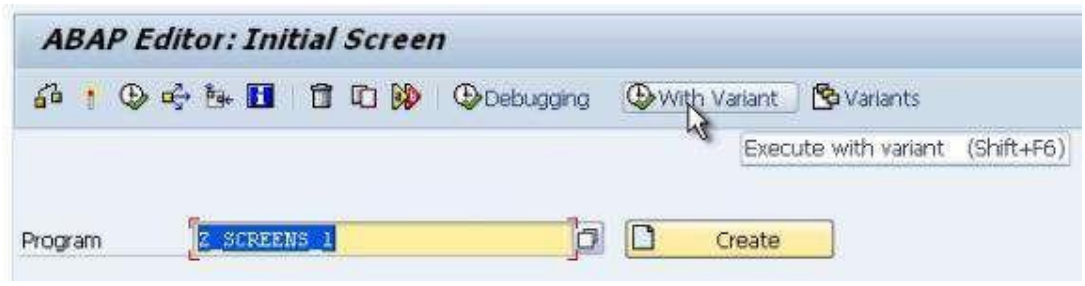
Field name	Type	P	I	N	L	P	L	O
Selection screen objects 1000								
Employee Number	P	☐	☐	☐	☐	☐		✓
My Box	P	☐	☐	☐	☐	☐		☐
GREEN	P	☐	☐	☐	☐	☐		☐
BLUE	P	☐	☐	☐	☐	☐		☐
RED	P	☐	☐	☐	☐	☐		☐
Date of Birth	S	✓	☐	☐	☐	☐		☐



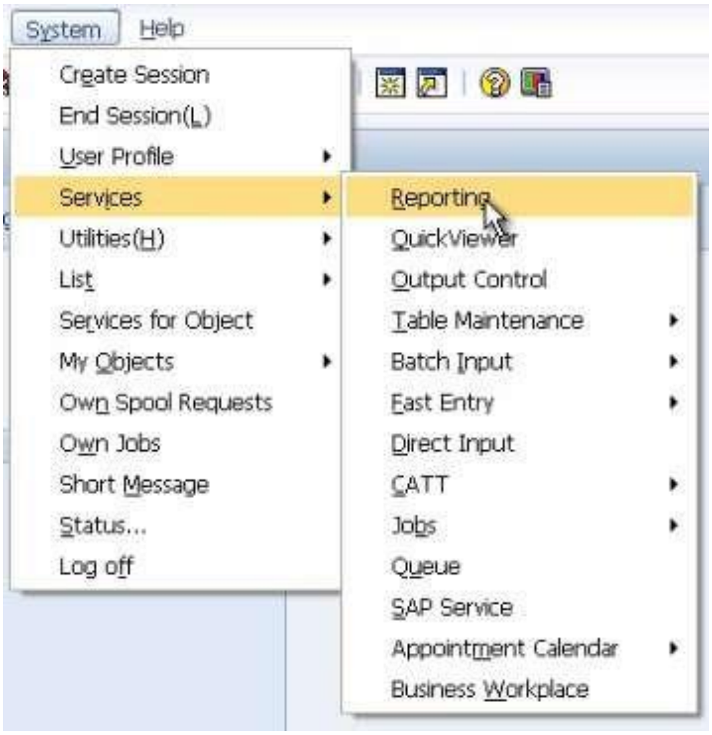
Date of Birth 08.01.2012 to

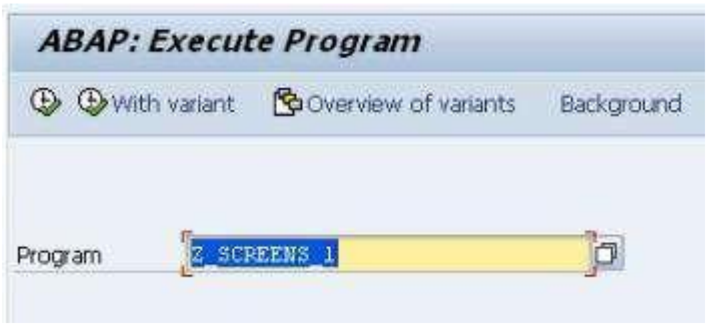
When large selection screens are created, users will regularly create variants so that, if necessary, the same data can be used repeatedly when running reports, saving the time it would take to fill in the information again and again. Unnecessary fields, or fields which will always hold the same value can be protected so that filling in the screen becomes a much simpler and less time consuming task for the end user.

At the ABAP editor’s initial screen, there is in fact a button which allows the program to run with a variant, directing one straight to the selection screen with the variant’s values already present:

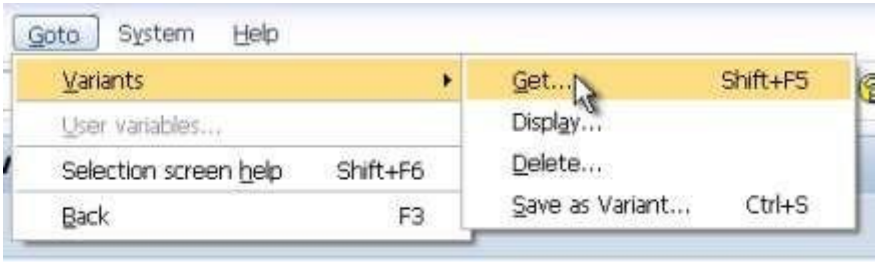


The ABAP editor will likely not be accessed by the user but reports can be accessed via the ‘System’ menu, ‘Services’, and then ‘Reporting’. Selecting this presents the ’ABAP: Execute Program’ screen, which could be described as a cut-down version of the ABAP editor screen, minus the editing functionality. From here the program can again either be exe- cuted directly or executed using a variant which can be selected from the menu which is offered:





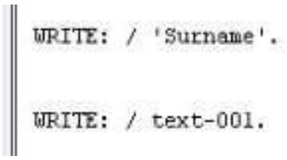
If the program is executed directly and the user then wants to use a variant, this can also be done via the ‘Goto’ menu:



Text Symbols

We will now take a look at other text objects starting with Text Symbols. These are used to replace literals in a program. For example, when the WRITE statement is used, one can choose to use text symbols to reuse text which has already been set up. This also gives the added functionality of being able to use translated text within the program. This allows hard coded literals to be avoided and text symbols used in their place.

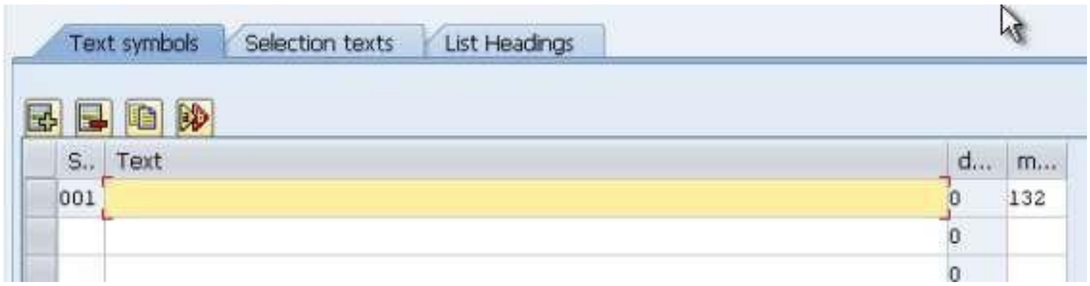
Text symbols effectively function as placeholders for text. So, rather than having “**WRITE: / ‘Surname’.**” multiple times in the code, you can avoid using the literal by using “**WRITE: / text-001.**” which here would refer to a text symbol which can be set up with the text “Surname” itself.



```
WRITE: / 'Surname'.

WRITE: / text-001.
```

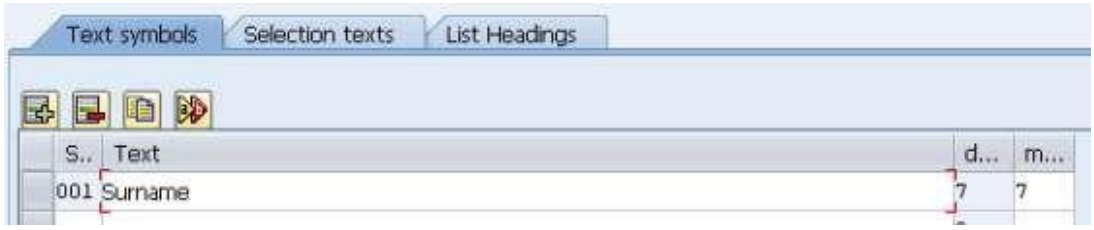
Text symbols are always declared with the word ‘text’ followed by a dash and a three digit number. This means that up to 1000 text symbols can theoretically be used in a program, of which each one can be translated into as many languages as one wishes. One thing to remember here is that text symbols are always limited to 132 characters in length.

To create a text symbol, you can use the ‘Goto’ menu, select ‘Text elements’ and then ‘Text symbols’, or you can use forward navigation. Just double-click ‘*text-001*’. A window will then appear asking if you want to create this object, select ‘Yes’. The Text Elements window will then appear and text can be entered for the new text symbol.



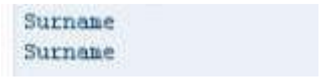
S..	Text	d...	m...
001		0	132
		0	
		0	

Here, include the word ‘Surname’. The column on the left references the text symbol id ‘001’. The two columns on the right note the text’s length and maximum length:



S..	Text	d...	m...
001	Surname	7	7

This can then be activated and you can step back to the program. If the code is then executed, the word ‘Surname’ will be output twice, the first from the WRITE statement with the literal, the second from the WRITE statement with the newly created text symbol:



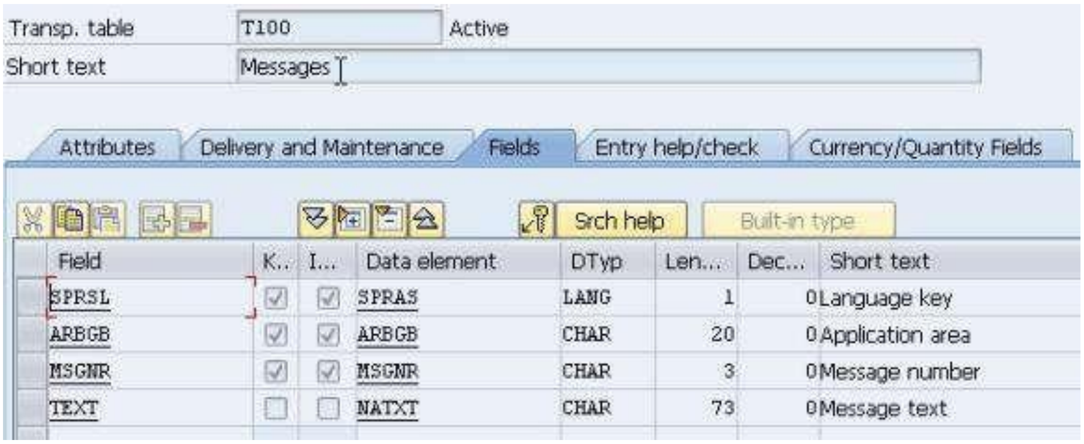
```
Surname
Surname
```

It is advisable to use text symbols rather than literals as often as possible as it is much easier to change the text symbol once than to sift through the code to find and change many literal values. Additionally, using text symbols gives the added benefit of translatability.

Text Messages

The next thing to be examined here is messages. When one wants to give feedback to the user, literals can be used, but as stated above, this is to be avoided as far as possible. To use messages then, these must first be stored in a message class, which is in turn stored in a database table called T100.

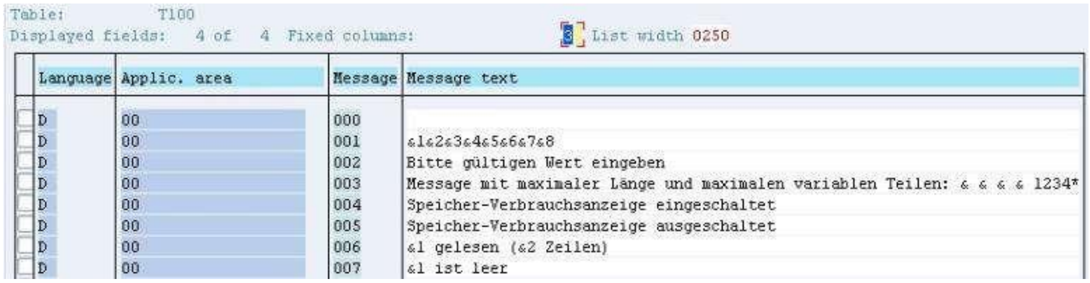
At the ABAP dictionary's initial screen, type 'T100' into the database table field and choose 'Display':



Transp. table	T100	Active
Short text	Messages	

Field	K...	I...	Data element	DTyp	Len...	Dec...	Short text
SPRSL	✓	✓	SPRAS	LANG	1	0	Language key
ARBGB	✓	✓	ARBGB	CHAR	20	0	Application area
MSGNR	✓	✓	MSGNR	CHAR	3	0	Message number
TEXT			NATXT	CHAR	73	0	Message text

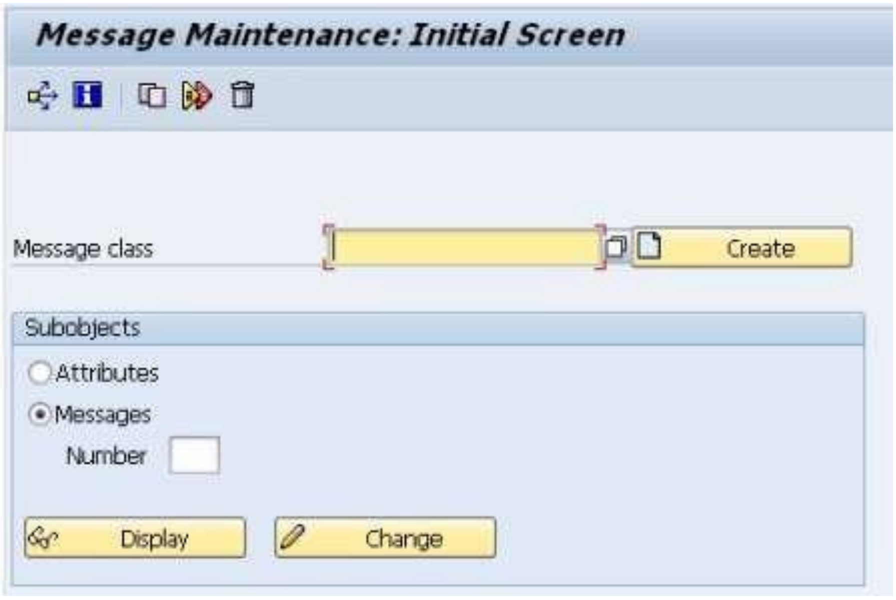
If one views the contents of this, one can see the four fields displayed. One for language (here D, referring to German), one for the application area, one for the message code and one for the message text:



```
Table:          T100
Displayed fields:   4 of   4  Fixed columns:                3  List width 0250
```

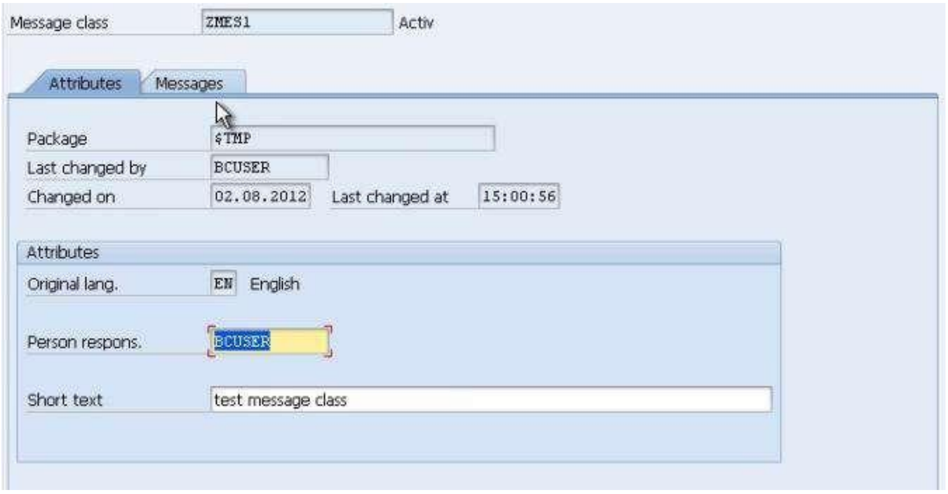
Language	Applic. area	Message	Message text
D	00	000	
D	00	001	&1&2&3&4&5&6&7&8
D	00	002	Bitte gültigen Wert eingeben
D	00	003	Message mit maximaler Länge und maximalen variablen Teilen: & & & & 1234*
D	00	004	Speicher-Verbrauchsanzeige eingeschaltet
D	00	005	Speicher-Verbrauchsanzeige ausgeschaltet
D	00	006	&1 gelesen (&2 Zeilen)
D	00	007	&1 ist leer

To create new messages to be used in your program, forward navigation can be used, or the transaction SE91 can be directly accessed:



First, create a message class. These must again follow the customer name space rules, here beginning with the letter Z. Let’s call this **ZMES1** and choose Create. Messages are distinct from text elements as they are not themselves part of the program created. They exist independently. They are instead stored in the T100 table. This means that messages can be reused across many programs.

The attributes must be filled in, creating a short text. Then, in the messages tab, the text to be used can be created:

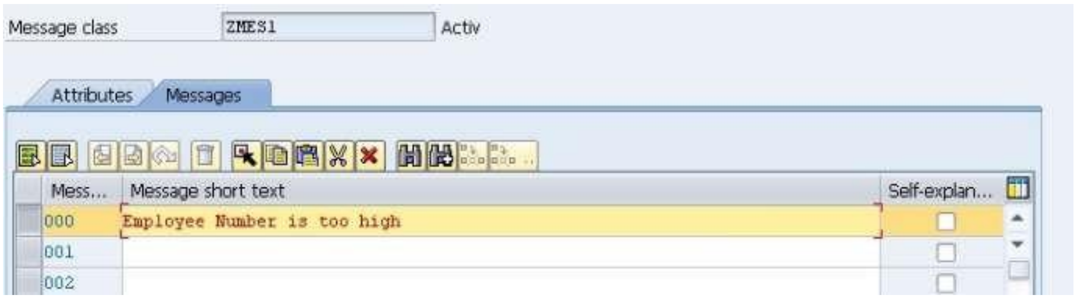


Message class ZMES1 Activ

Package $TMP
Last changed by BCUSER
Changed on 02.08.2012 Last changed at 15:00:56

Attributes
Original lang. EN English
Person respons. BCUSER
Short text test message class

Remember that, when the AT SELECTION-SCREEN event was created, an IF statement was used so that if the employee number given by the user was greater than the last employee number used in the table, a message would be displayed. Here, the text for that message can be created:


Message class ZMES1 Activ

Mess...	Message short text	Self-explan...
000	Employee Number is too high	☐
001		☐
002		☐

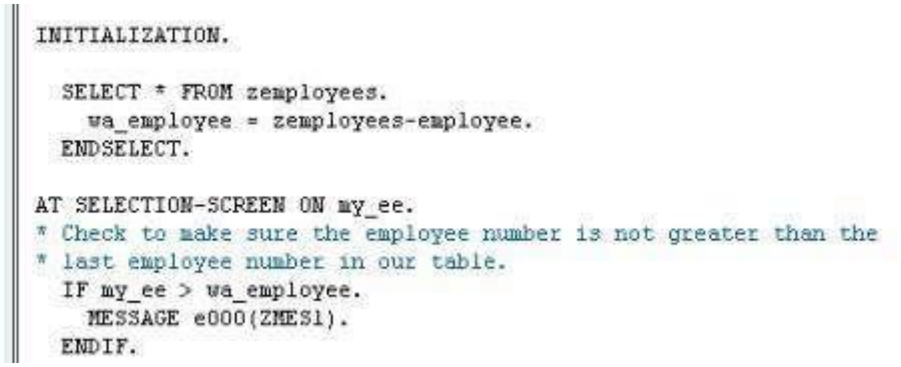
Once the text is entered, it can be saved.

There are a number of message types which can be used, as this table explains:

A	**Termination Message**	The message appears in a dialog box, and the program terminates. When the user has confirmed the message, control returns to the next-highest area menu.
E	**Error Message**	**Depending on the program context**, an error dialog appears or the program terminates.
I	**Information**	The message appears in a dialog box. Once the user has confirmed the message, the program continues immediately after the **MESSAGE** statement.
S	**Status Message**	The program continues normally after the **MESSAGE**statement, and the message is displayed in the status bar of the next screen.
W	**Warning**	**Depending on the program context**, an error dialog appears or the program terminates.
X	**Exit**	No message is displayed, and the program terminates with a short dump. Program terminations with a short dump normally only occur when a runtime error occurs. Message type X allows you to force a program termination. The short dump contains the message ID.

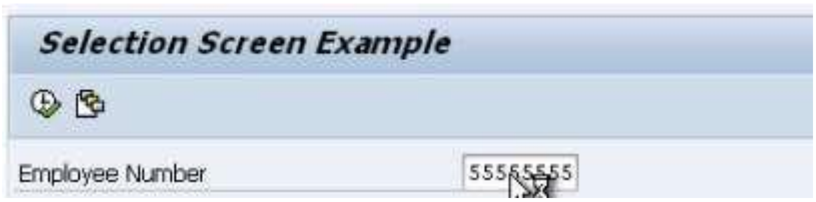
For this example, type E, an error message, will be used. Depending on where this type of message is used, it will have a different effect. Here, it will be used at the “at selection-screen” and the program’s execution will pause, the error message will be displayed and the user will be allowed to amend their entry. When the new entry appears, the event will begin again. If an error message is used elsewhere, outside of an event in the main body of the code, when this is triggered the program will terminate entirely.

To include the newly created message in the code, then, the syntax is “MESSAGE e000(ZMES1).” The ‘e’ refers to the error message type, the ‘000’ to the number assigned to the message in the message class, and then ‘ZMES1’ to the class itself:



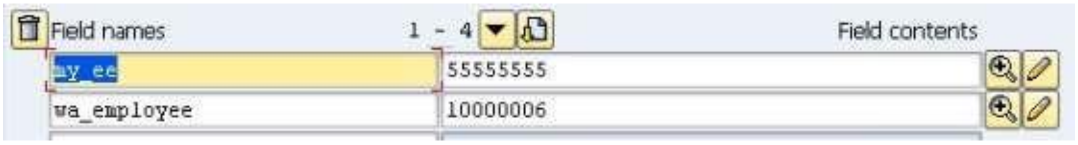
```
INITIALIZATION.

  SELECT * FROM zemployees.
    wa_employee = zemployees-employee.
  ENDSELECT.

AT SELECTION-SCREEN ON my_ee.
* Check to make sure the employee number is not greater than the
* last employee number in our table.
  IF my_ee > wa_employee.
    MESSAGE e000(ZMES1).
  ENDIF.
```

The INITIALIZATION event will populate wa_employee with the last, highest employee number used in the table, and then, at the AT SELECTION-SCREEN event, the value entered can be checked against this. If it is higher, the error message will display. You can monitor these values in debug mode to watch the code in action. Here, the number is higher so, once executed, the selection screen will be returned to and the message dis- played in the status bar:

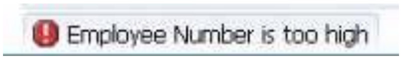


Selection Screen Example

Employee Number 55555555



Field names	Field contents
my_ee	55555555
wa_employee	10000006

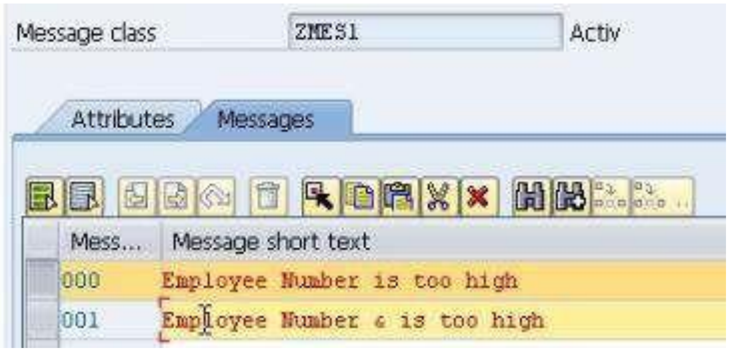


Employee Number is too high

Once a legitimate, lower value is entered, the program will continue as normal without triggering the error message.

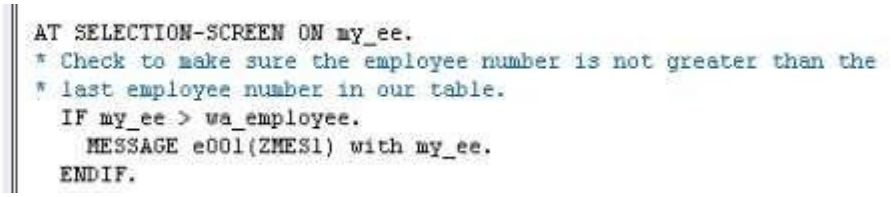
An addition which can be used with the MESSAGES statement is **WITH**. Here, a field can be specified, for example to display the invalid value which was entered by the user in the message itself. The WITH addition allows up to 4 parameters to be included in the error message. To do this, one must ensure the error message is compatible.

Create another message in the message class screen, this time with an **&** character. When used in conjunction with the WITH addition, this character will then be replaced by the value in the specified parameter:

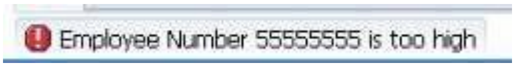


Mess...	Message short text
000	Employee Number is too high
001	Employee Number & is too high

Save the new message, add “WITH my_ee” to the MESSAGES statement and change the number of the message referenced in the code to the new 001 message:



```
AT SELECTION-SCREEN ON my_ee.
* Check to make sure the employee number is not greater than the
* last employee number in our table.
  IF my_ee > wa_employee.
    MESSAGE e001(ZMES1) with my_ee.
  ENDIF.
```



Employee Number 55555555 is too high

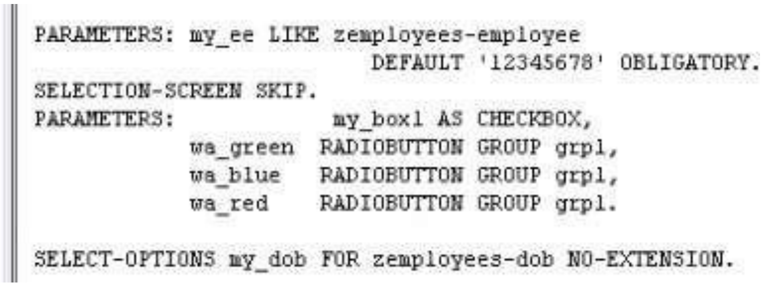
As messages created are not specific to the program itself, but can be used across the entire system, it is usually worth checking if an appropriate message for the task you are performing already exists, rather than continually setting up new messages.

Skip Lines and Underline

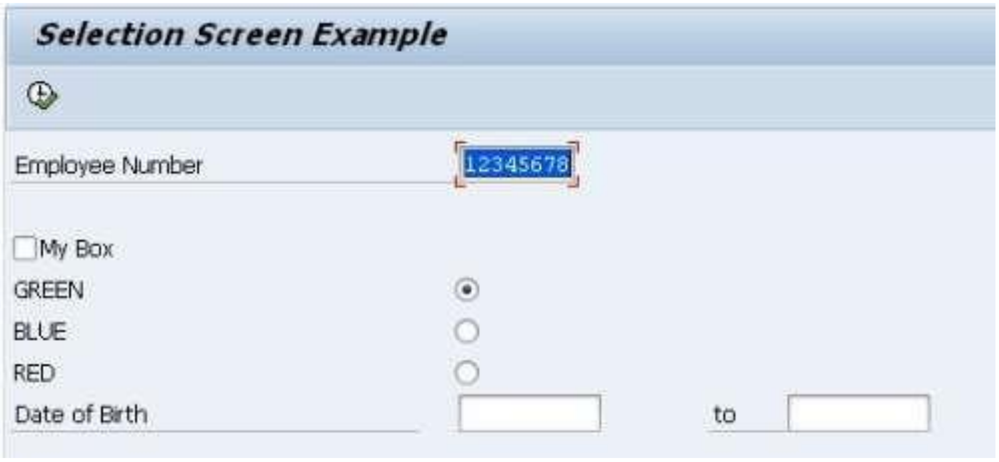
Now, a look will be taken at formatting selection screens. This will allow the screen to be a lot easier to navigate and so on for the end user. Parameters and select-options have already been set up, but as yet no layout options have been implemented allowing the system to place the objects by itself. This is generally not sufficient. For example, when a group of radio buttons appear, they should be distinct and positioned in a group on their own, clearly separated from other parts of the screen.

The SELECTION-SCREEN statement, and its associated additions allow this kind of format- ting to be done. One must locate where in the code the screen layout begins to be re- ferred to. Here, this is at the top when PARAMETERS is declared. In the line above this, type the statement SELECTION-SCREEN. Additions must then be added.

First, to add blank lines you can use the SKIP addition, followed by the number of lines to be skipped. If you only want to skip 1 line then the number can be omitted as this is the default values. This line of code must then be moved to the place where you want the line to be skipped. Place it under the *my_ee* parameter. Note that the PARAMETERS chain is now broken, so another PARAMETERS statement must be added:



```
PARAMETERS: my_ee LIKE zemployees-employee
                           DEFAULT '12345678' OBLIGATORY.
SELECTION-SCREEN SKIP.
PARAMETERS:            my_box1 AS CHECKBOX,
            wa_green  RADIOBUTTON GROUP grp1,
            wa_blue   RADIOBUTTON GROUP grp1,
            wa_red    RADIOBUTTON GROUP grp1.

SELECT-OPTIONS my_dob FOR zemployees-dob NO-EXTENSION.
```



Employee Number 12345678

My Box

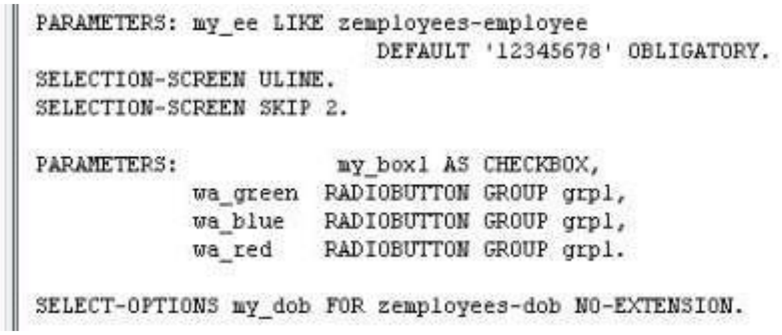
GREEN

BLUE

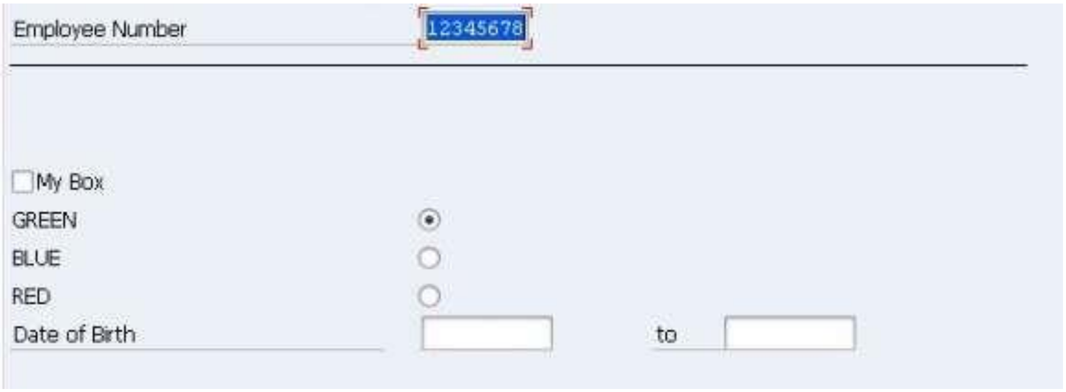
RED

Date of Birth to

To add a horizontal line, the ULINE addition can be used:



```
PARAMETERS: my_ee LIKE zemployees-employee
                         DEFAULT '12345678' OBLIGATORY.
SELECTION-SCREEN ULINE.
SELECTION-SCREEN SKIP 2.

PARAMETERS:            my_box1 AS CHECKBOX,
            wa_green  RADIOBUTTON GROUP grp1,
            wa_blue   RADIOBUTTON GROUP grp1,
            wa_red    RADIOBUTTON GROUP grp1.

SELECT-OPTIONS my_dob FOR zemployees-dob NO-EXTENSION.
```



Employee Number 12345678

My Box

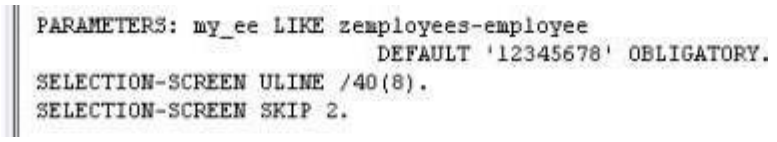
GREEN

BLUE

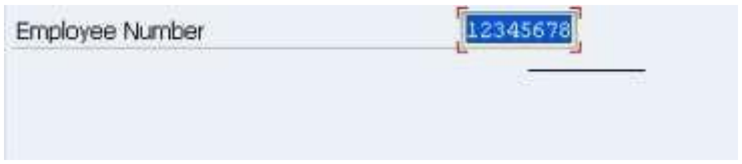
RED

Date of Birth to

There are further additions which can be added to ULINE to determine its position and length. The code in the image below sets the position of the line to the 40th character from the left of the screen, and its length is set to 8 characters:



```
PARAMETERS: my_ee LIKE zemployees-employee
                          DEFAULT '12345678' OBLIGATORY.
SELECTION-SCREEN ULINE /40(8).
SELECTION-SCREEN SKIP 2.
```



Employee Number 12345678