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 – II

DR.A.DEVI

Associate Professor

Department of Computer Applications

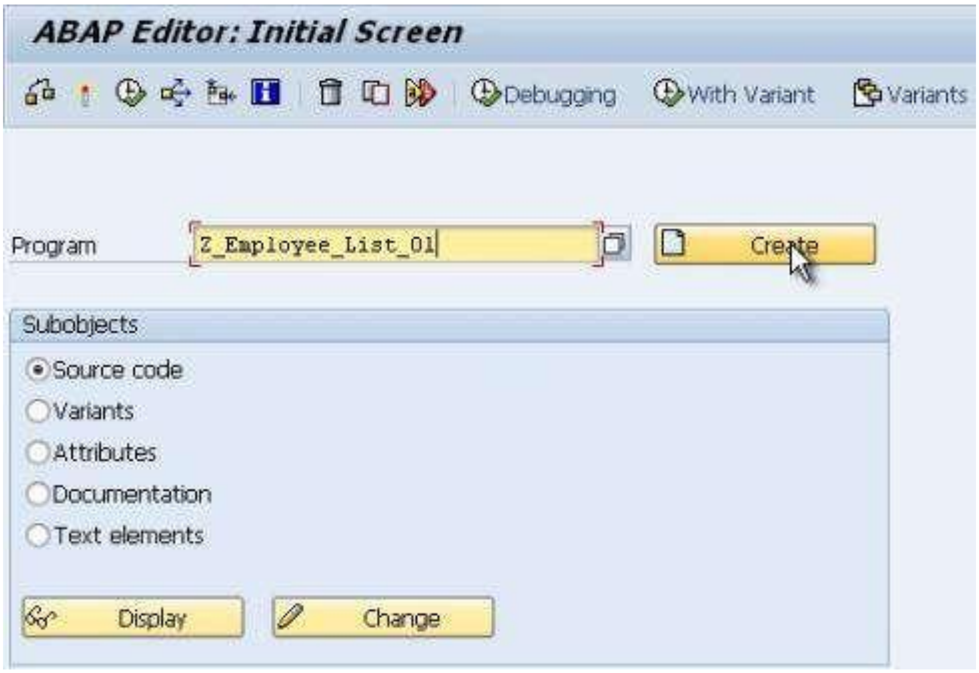
DRSNSRCAS

Creating a Program

To begin creating a program, access the ABAP Editor either via transaction code SE38, or by navigating the SAP menu tree to Tools ABAP Workbench Development, in which the ABAP Editor is found. Double-click to execute.

A note to begin: it is advisable to keep the programs created as simple as possible. Do not make them any more complicated than is necessary. This way, when a program is passed on to another developer to work with, fix bugs and so on, it will be far easier for them to understand. Add as many comments as possible to the code, to make it simpler for anyone who comes to it later to understand what a program is doing, and the flow of the logic as it is executed.

The program name must adhere to the customer naming conventions, meaning that here it must begin with the letter Z. In continuation of the example from the previous chapter, in this instance the program will be titled ‘Z_Employee_List_01’, which should be typed into the ‘Program’ field on the initial screen of the ABAP Editor. Ensure that the ‘Source code’ button is checked, and then click ‘Create’:



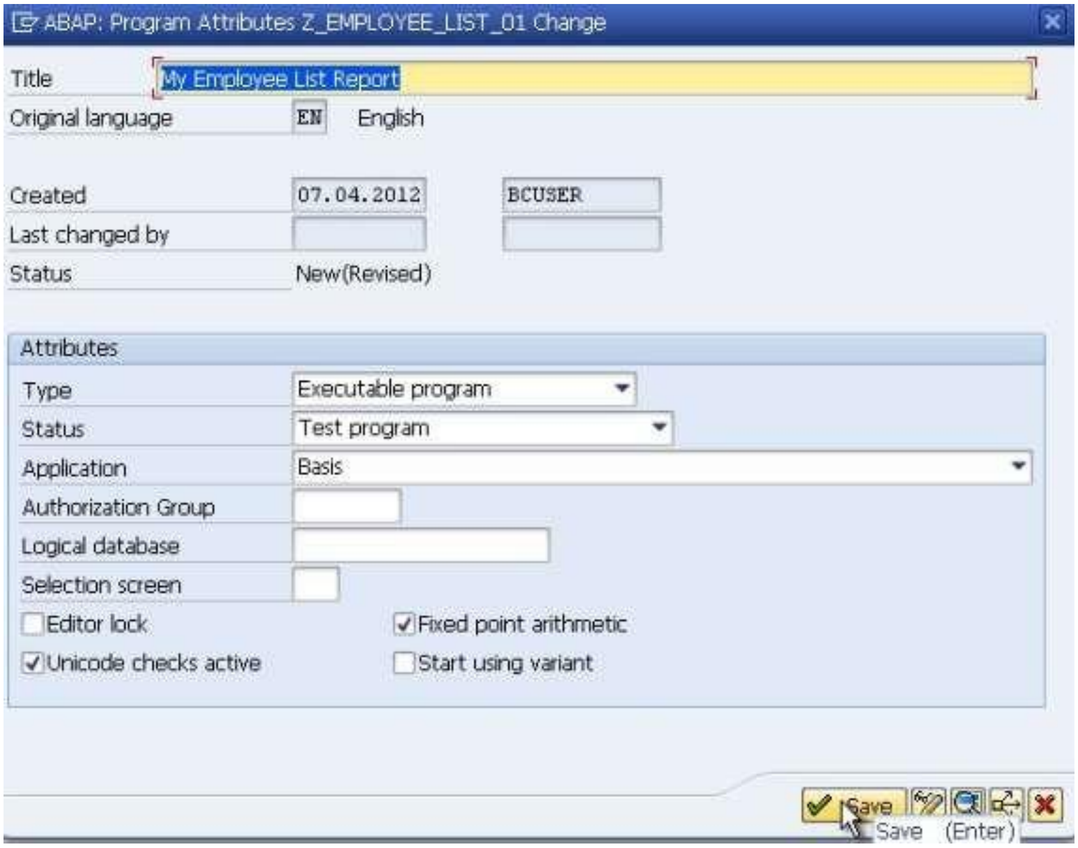
Program Z_Employee_List_01

Subobjects: (●) Source code

A ‘Program Attributes’ window will then appear. In the ‘Title’ box, type a description of what the program will do. In this example, “My Employee List Report”. The Original language should be set to EN, English by default, just check this, as it can have an effect on the text entries displayed within certain programs. Any text entries created within the program are language-specific, and can be maintained for each country using a translation tool. This will not be examined at length here, but is something to bear in mind.

In the ‘Attributes’ section of the window, for the ‘Type’, click the drop-down menu and select ‘Executable program’, meaning that the program can be executed without the use of a transaction code, and also that it can be run as a background job. The ‘Status’ selected should be ‘Test program’, and the ‘Application’ should be ‘Basis’. These two options help to manage the program within the SAP system itself, describing what the program will be used for, and also the program development status.

For now, the other fields below these should be left empty. Particularly ensure that the ‘Editor Lock’ box is left clear (selection of this will prevent the program from being edited). ‘Unicode checks active’ should be selected, as should ‘Fixed point arithmetic’ (without this, any packed-decimal fields in the program will be rounded to whole numbers). Leave the ‘Start using variant’ box blank. Then, click the Save button.


ABAP: Program Attributes Z_EMPLOYEE_LIST_01 Change

Title	My Employee List Report	
Original language	EN	English
Created	07.04.2012	BCUSER
Last changed by		
Status	New(Revised)	

Attributes

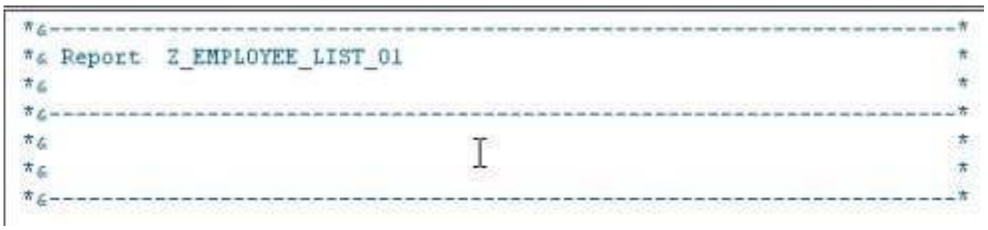
Type	Executable program
Status	Test program
Application	Basis
Authorization Group	
Logical database	
Selection screen	

☐ Editor lock ✓ Fixed point arithmetic
✓ Unicode checks active ☐ Start using variant

The familiar ‘Create Object Directory Entry’ box from the previous section should appear now, click the ‘Local object’ option as before to assign the program to the temporary development class. Once this is achieved, the coding screen is reached.

Code Editor

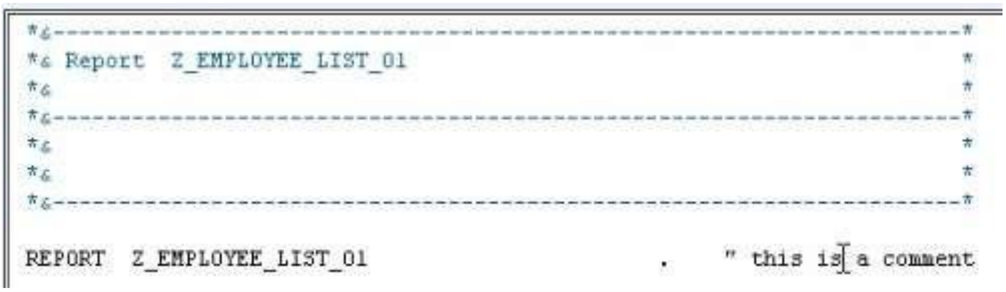
Here, focus will be put on the coding area. The first set of lines visible here are comment lines. These seven lines can be used to begin commenting the program. In ABAP, comments can appear in two ways. Firstly, if a * is placed at the beginning of a line, it turns everything to its right into a comment.



```
*&---------------------------------------------------------------------*
*& Report  Z_EMPLOYEE_LIST_01                                          *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&                                                                     *
*&---------------------------------------------------------------------*
```

Note that the * <u>must</u> be in the first column on the left. If it appears in the second column or beyond, the text will cease to be a comment.

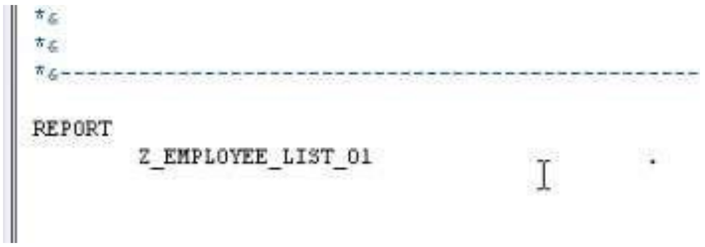
A comment can also be written within a line itself, by using a ". Where this is used, everything to the right again becomes a comment. This means that it is possible to add comments to each line of a program, or at least a few lines of comments for each section.



```
*&---------------------------------------------------------------------*
*& Report  Z_EMPLOYEE_LIST_01                                          *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  Z_EMPLOYEE_LIST_01                      .    " this is a comment
```

The next line of code, visible above, begins with the word REPORT. This is called a STATEMENT, and the REPORT statement will always be the first line of any executable program created. The statement is followed by the program name which was created previously. The line is then terminated with a full stop (visible to the left of the comment).

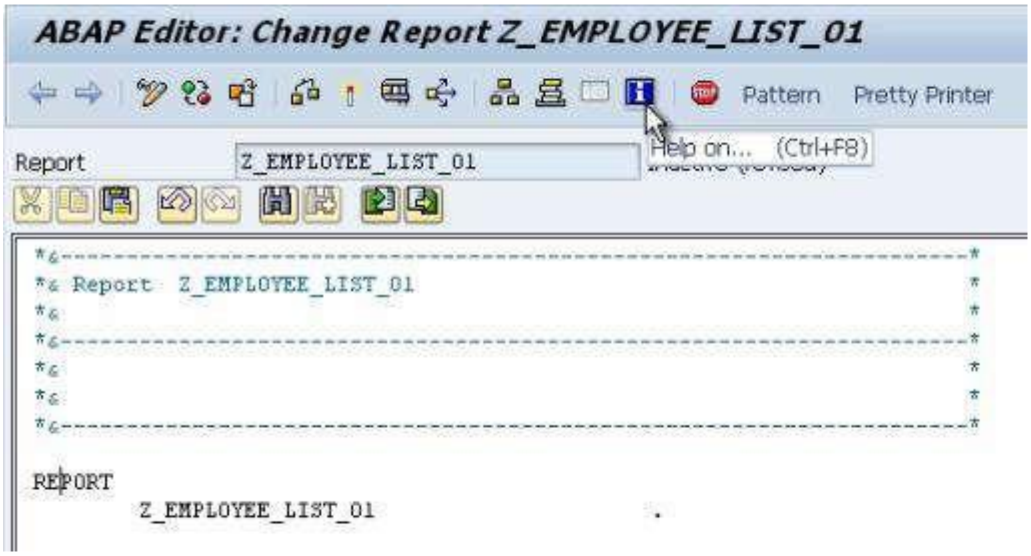
Every statement in ABAP <u>must</u> be followed by a full stop, or period. This allows the statement to take up as many lines in the editor as it needs, so for example, the REPORT statement here could look like this:



```
*&
*&
*&---------------------------------------------

REPORT
        Z_EMPLOYEE_LIST_01                  .
```

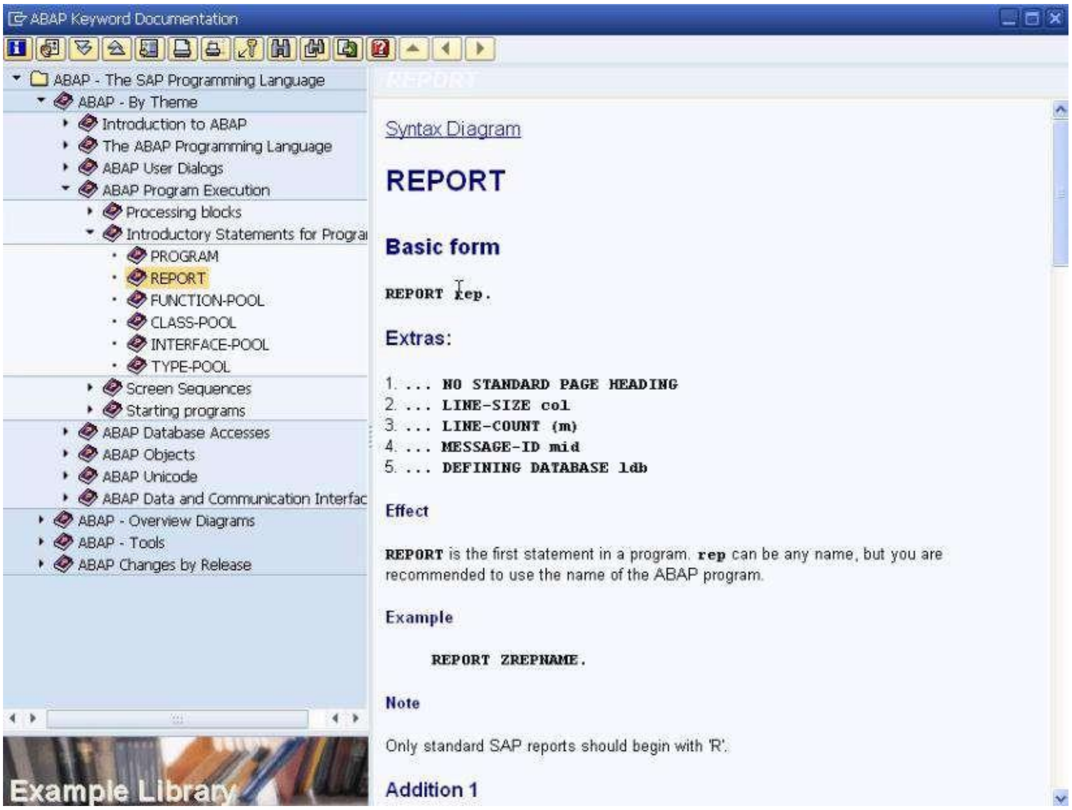
As long as the period appears at the end of the statement, no problems will arise. It is this period which marks where the statement finishes.

If you require help with a statement, place the cursor within the statement and choose the 'Help on...' button in the top toolbar:



```
*&---------------------------------------------------------------------*
*& Report  Z_EMPLOYEE_LIST_01                                          *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT
      Z_EMPLOYEE_LIST_01                   .
```

A window will appear with the ABAP keyword automatically filled in. Click the continue button and the system will display help on that particular statement, giving an explanation of what it is used for and the syntax. This can be used for every ABAP statement within an SAP system. Alternatively, this can be achieved by clicking the cursor within the statement, and pressing the F1 key:



REPORT

Basic form

```
REPORT rep.
```

Extras:

```
1. ... NO STANDARD PAGE HEADING
2. ... LINE-SIZE col
3. ... LINE-COUNT (m)
4. ... MESSAGE-ID mid
5. ... DEFINING DATABASE ldb
```

Effect

REPORT is the first statement in a program. rep can be any name, but you are recommended to use the name of the ABAP program.

Example

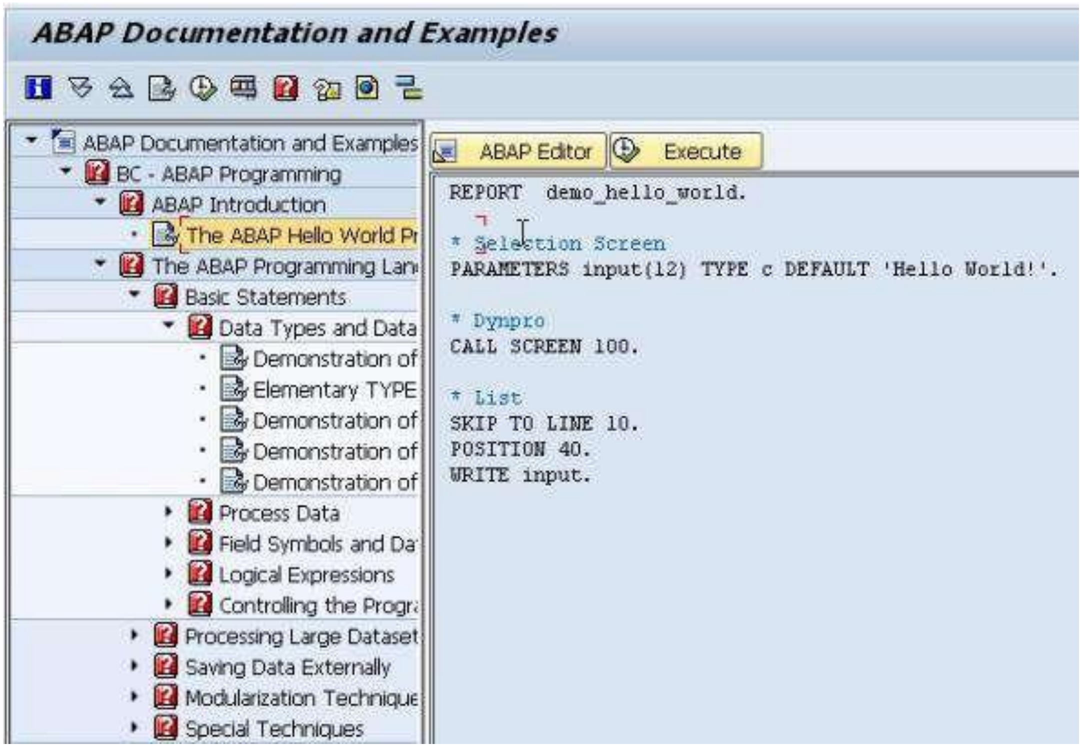
```
REPORT ZREPNAME.
```

Note

Only standard SAP reports should begin with 'R'.

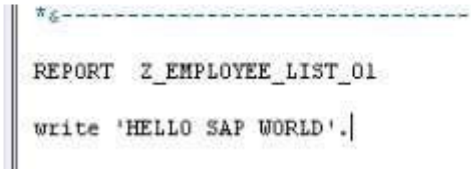
Addition 1

A further tip in this vein is to use the ‘ABAP Documentation and Examples’ page, which can be accessed by entering transaction code ABAPDOCU into the transaction code field. The menu tree to the left hand side on this screen allows you to view example code, which one’s own code can later be based upon. This can either be copied and pasted into the ABAP editor, or experimented with inside the screen itself using the Execute button to run the example code:



```
REPORT  demo_hello_world.

* Selection Screen
PARAMETERS input(12) TYPE c DEFAULT 'Hello World!'.

* Dynpro
CALL SCREEN 100.

* List
SKIP TO LINE 10.
POSITION 40.
WRITE input.
```

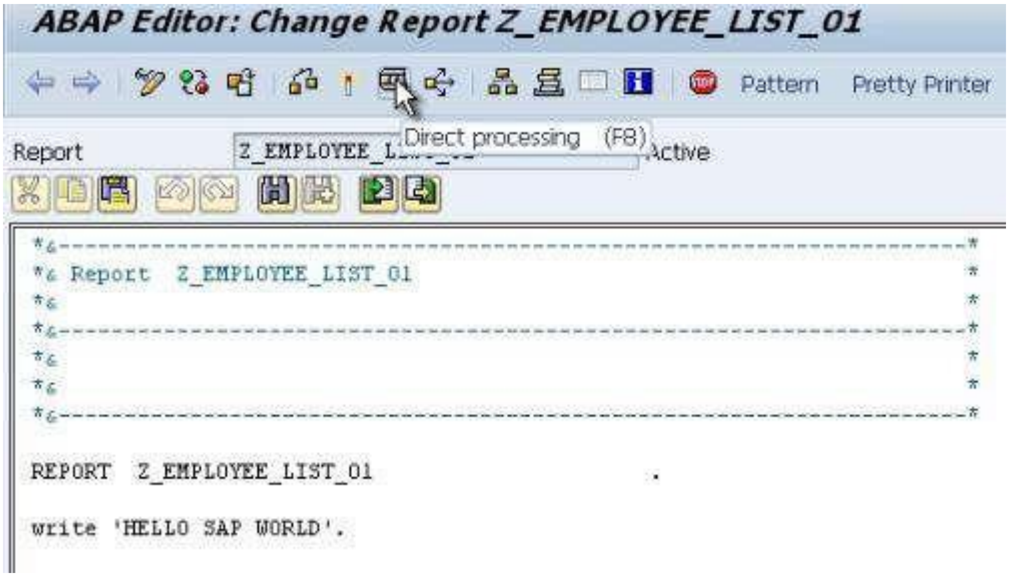
Returning to the ABAP editor now, the first line of code will be written. On the line below the REPORT statement, type the statement: ***write 'HELLO SAP WORLD'.***



```
*&-------------------------------

REPORT  Z_EMPLOYEE_LIST_01

write 'HELLO SAP WORLD'.
```

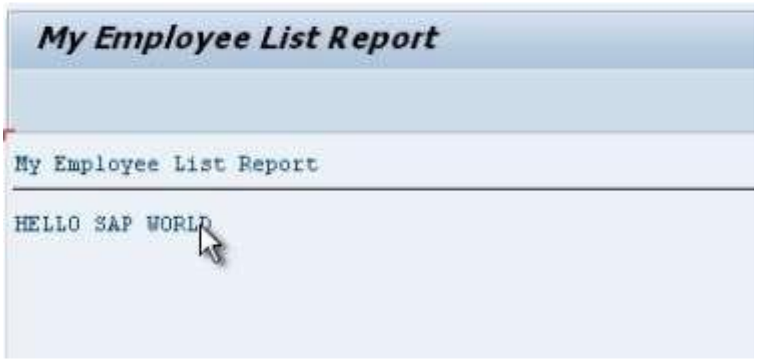
The write statement will, as you might expect, write whatever is in quotes after it to the output window (there are a number of additions which can be made to the write statement to help format the text, which we will return in a later chapter).

Save the program, and check the syntax with the 'Check' button in the toolbar (or via CTRL + F2). The status bar should display a message reading "Program Z_EMPLOYEE_LIST_01 is syntactically correct". Then, click the 'Activate' button, which should add the word 'Active' next to the program name. Once this is done, click the 'Direct processing' button to test the code:



```
*&---------------------------------------------------------------------*
*& Report  Z_EMPLOYEE_LIST_01                                          *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  Z_EMPLOYEE_LIST_01                .

write 'HELLO SAP WORLD'.
```

The report title and the text output should appear like this, completing the program:

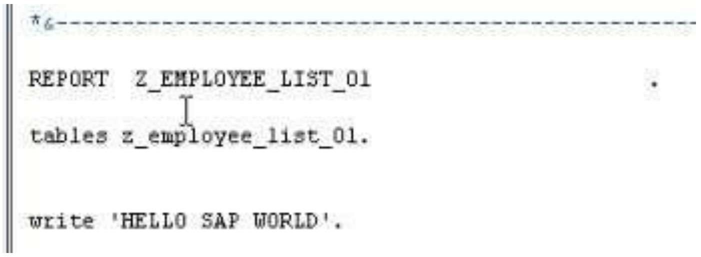


```
My Employee List Report

HELLO SAP WORLD
```

Write Statements

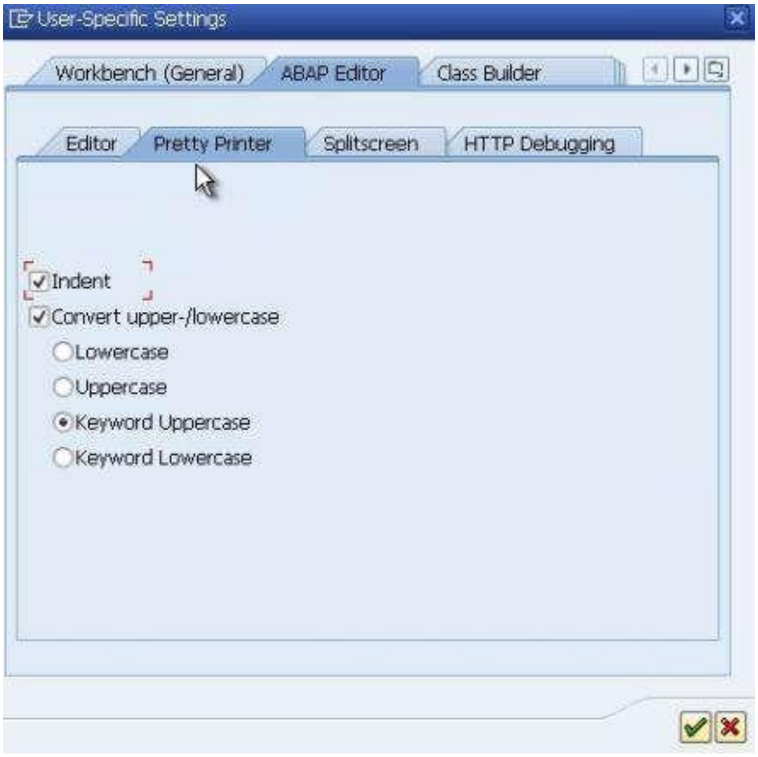
Now that the first program has been created, it can be expanded with the addition of further ABAP statements. Use the Back button to return from the test screen to the ABAP editor.

Here, the tables which were created in the ABAP Dictionary during the first stage will be accessed. The first step toward doing this is to include a table's statement in the program, which will be placed below the REPORT statement. Following this, the table name which was created is typed in, z_employee_list_01, and, as always, a period to end the statement:



```
*&---------------------------------------------

REPORT  Z_EMPLOYEE_LIST_01                  .

tables z_employee_list_01.


write 'HELLO SAP WORLD'.
```

While not essential, to keep the format of the code uniform, the Pretty Printer facility can be used. Click the 'Pretty Printer' button in the toolbar to automatically alter the text in line with the Pretty Printer settings (which can be accessed through the Utilities menu, Settings, and the Pretty Printer tab in the ABAP Editor section):



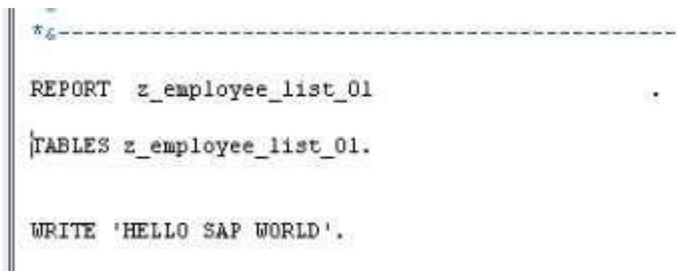
User-Specific Settings

Workbench (General) | ABAP Editor | Class Builder

Editor | Pretty Printer | Splitscreen | HTTP Debugging

[✓] Indent
[✓] Convert upper-/lowercase
- ○ Lowercase
- ○ Uppercase
- ● Keyword Uppercase
- ○ Keyword Lowercase

Once these settings have been applied, the code will look slightly tidier, like this:

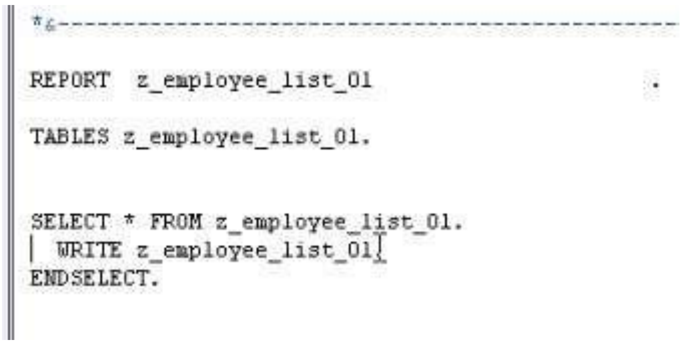


```
*&---------------------------------------------------

REPORT  z_employee_list_01                          .

TABLES z_employee_list_01.


WRITE 'HELLO SAP WORLD'.
```

Let us now return to the TABLES statement. When the program is executed, the TABLES statement will create a table structure in memory based on the structure previously defined in the ABAP Dictionary. This table structure will include all of the fields previously created, allowing the records from the table to be read and stored in a temporary structure for the program to use.

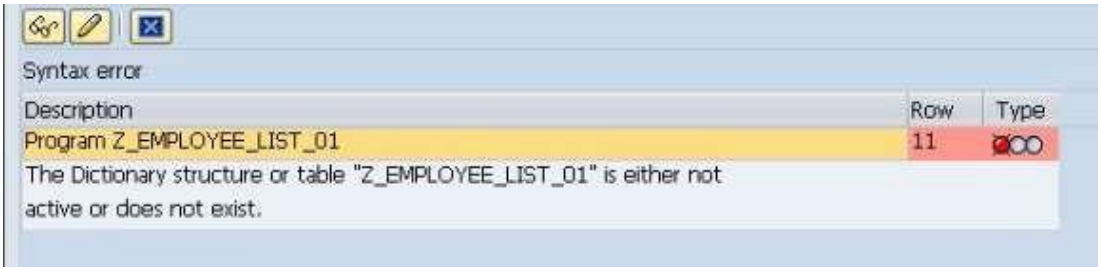
To retrieve from our data dictionary table and place them into the *table structure*, the SELECT statement will be used.

Type ***SELECT * from z_employee_list_01.*** This is telling the system to select everything (the * refers to all-fields) from the table. Because the SELECT statement is a loop, the system must be told where the loop ends. This is done by typing the statement ENDSELECT. Now we have created a select loop let's do something with the data we have are looping through. Here, the WRITE command will be used again. Replace the "***write 'HELLO SAP WORLD'.***" line with "***write z_employee_list_01.***" to write every row of the table to the output window:



```
*&---------------------------------------------------

REPORT  z_employee_list_01                          .

TABLES z_employee_list_01.


SELECT * FROM z_employee_list_01.
  WRITE z_employee_list_01.
ENDSELECT.
```

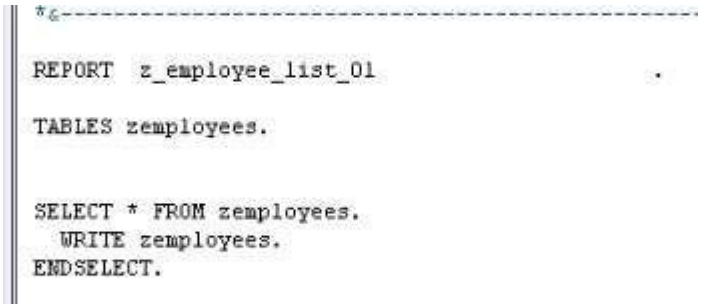
Check the code with the 'Check' button, and it will state that there is a syntax error:



Syntax error

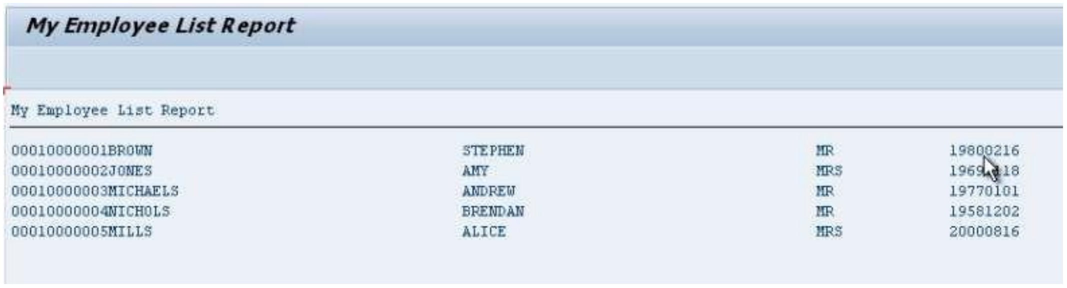
Description	Row	Type
Program Z_EMPLOYEE_LIST_01	11	●
The Dictionary structure or table "Z_EMPLOYEE_LIST_01" is either not active or does not exist.		

The cursor will have moved to the TABLES statement which was identified, along with the above warning. The name "Z_EMPLOYEE_LIST_01" appears to be incorrect. To check this, open a new session via the New Session button in the toolbar . Execute the ABAP Dictionary with transaction code SE11, search for Z* in the 'Database table' box and it will bring back the table ZEMPLOYEES, meaning that the initial table name Z_EMPLOYEE_LIST_01 was wrong. Close the new session and the syntax error window and type in the correct table name 'ZEMPLOYEES' after the TABLES state. Your screen should look like this:



```
*&---------------------------------------------------

REPORT  z_employee_list_01                .

TABLES zemployees.


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

Save the program and check the code, ensuring the syntax error has been removed, and then click the Test button (F8) and the output window should display every row of the table:

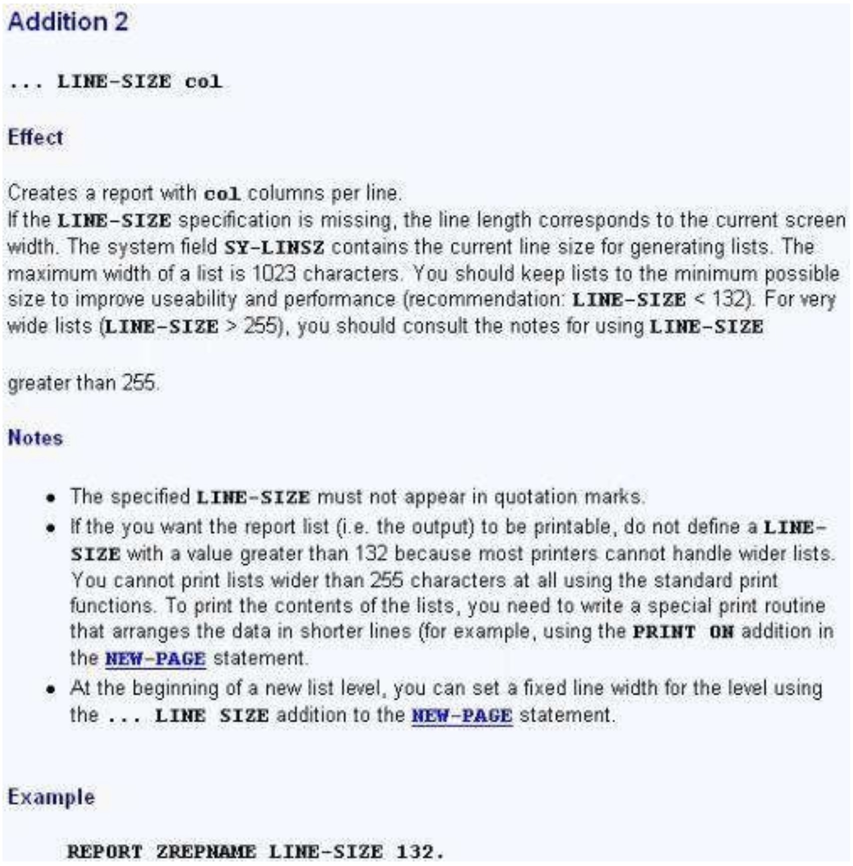


My Employee List Report

```
My Employee List Report
---------------------------------------------------------------------------------------
00010000001BROWN                    STEPHEN                   MR          19800216
00010000002JONES                    AMY                       MRS         1969[illegible]18
00010000003MICHAELS                 ANDREW                    MR          19770101
00010000004NICHOLS                  BRENDAN                   MR          19581202
00010000005MILLS                    ALICE                     MRS         20000816
```

Look at the data in the output window. The system has automatically put each line from the table on a new row. The WRITE statement in the program did not know that each row was to be output on a new line; this was forced by some of the default settings within the system regarding screen settings, making the line length correspond to the width of the screen. If you try to print the report, it could be that there are too many columns or characters to fit on a standard sheet of A4. With this in mind, it is advisable to use an addition to the REPORT statement regarding the width of each line.

Return to the program, click the REPORT statement and press the F1 key and observe the LINE SIZE addition which can be included:



Addition 2

... LINE-SIZE col

Effect

Creates a report with **col** columns per line.
If the **LINE-SIZE** specification is missing, the line length corresponds to the current screen width. The system field **SY-LINSZ** contains the current line size for generating lists. The maximum width of a list is 1023 characters. You should keep lists to the minimum possible size to improve useability and performance (recommendation: **LINE-SIZE** < 132). For very wide lists (**LINE-SIZE** > 255), you should consult the notes for using **LINE-SIZE**

greater than 255.

Notes

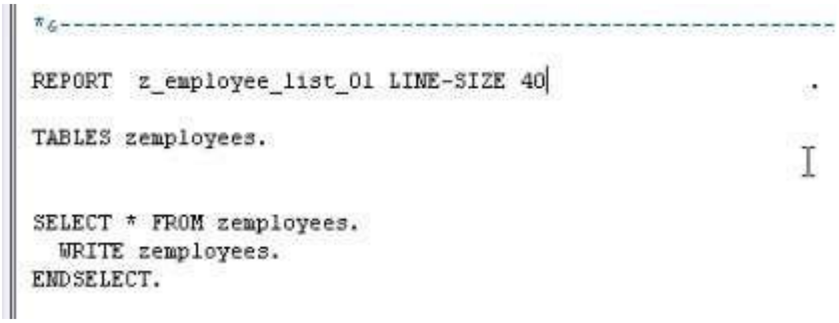
- The specified **LINE-SIZE** must not appear in quotation marks.
- If the you want the report list (i.e. the output) to be printable, do not define a **LINE-SIZE** with a value greater than 132 because most printers cannot handle wider lists. You cannot print lists wider than 255 characters at all using the standard print functions. To print the contents of the lists, you need to write a special print routine that arranges the data in shorter lines (for example, using the **PRINT ON** addition in the **NEW-PAGE** statement.
- At the beginning of a new list level, you can set a fixed line width for the level using the **... LINE SIZE** addition to the **NEW-PAGE** statement.

Example

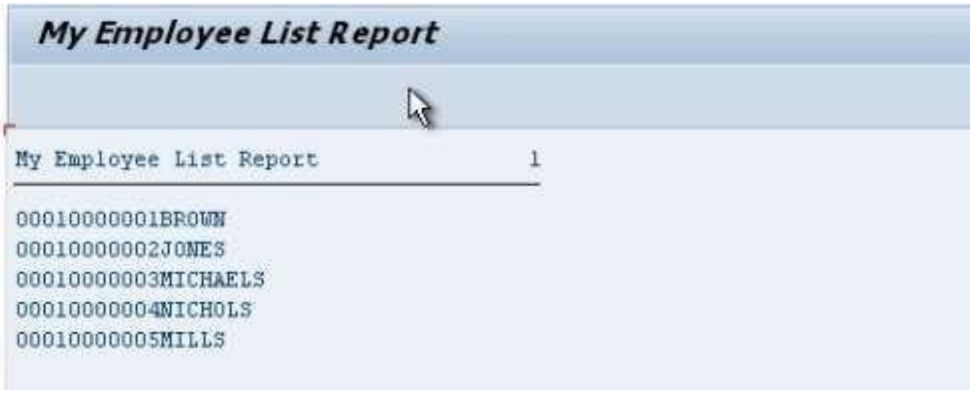
```
REPORT ZREPNAME LINE-SIZE 132.
```

In this example, add the LINE-SIZE addition to the REPORT statement. Here, the line will be limited to 40 characters. Having done this, see what difference it has made to the output window. The lines have now been broken at the 40 character limit, truncating the output

of each line:

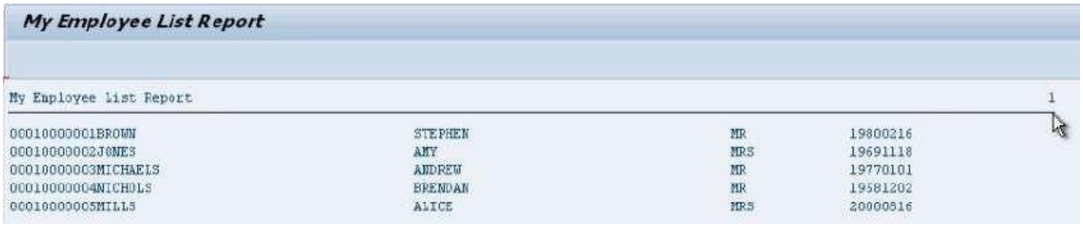


```
*&---------------------------------------------------------------

REPORT  z_employee_list_01 LINE-SIZE 40                    .

TABLES zemployees.


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



```
My Employee List Report                1
----------------------------------------
00010000001BROWN
00010000002JONES
00010000003MICHAELS
00010000004NICHOLS
00010000005MILLS
```

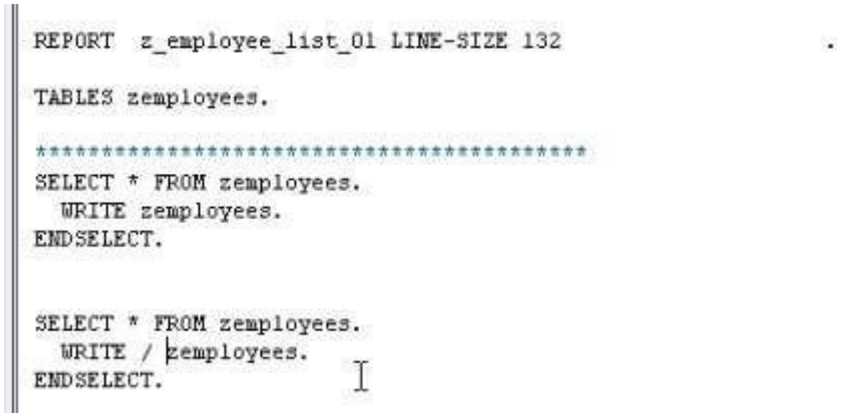
Bear these limits in mind so as to avoid automatic truncation when printing reports. For a standard sheet of A4 this limit will usually be 132 characters. When the limit is set to this for the example table here, the full table returns, but the line beneath the title ‘*My Employee List Report*’ displays the point at which the output is limited:



```
My Employee List Report                                                                                                              1
------------------------------------------------------------------------------------------------------------------------------------
00010000001BROWN                                     STEPHEN                                      MR              19800216
00010000002JONES                                     AMY                                          MRS             19691118
00010000003MICHAELS                                  ANDREW                                       MR              19770101
00010000004NICHOLS                                   BRENDAN                                      MR              19581202
00010000005MILLS                                     ALICE                                        MRS             20000816
```

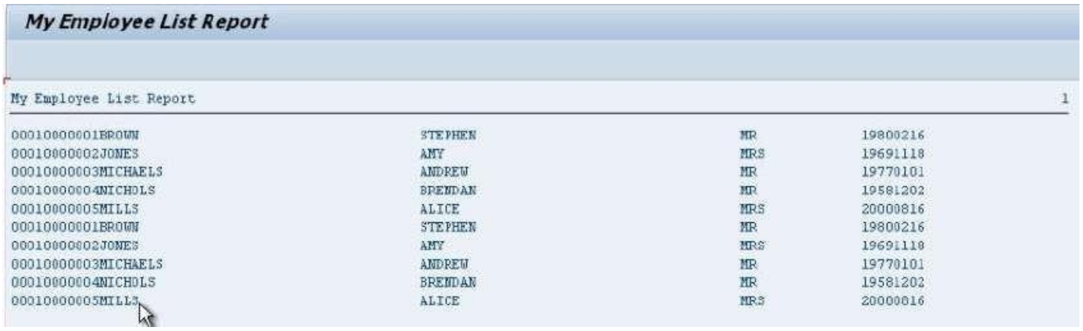
Next, the program will be enhanced somewhat, by adding specific formatting additions to the WRITE statement. First, a line break will be inserted at the beginning of every row that is output.

Duplicate the previous SELECT – ENDSELECT statement block of code and place a '/' after the WRITE statement. This will trigger a line break:



```
REPORT  z_employee_list_01 LINE-SIZE 132                    .

TABLES zemployees.

*******************************************
SELECT * FROM zemployees.
  WRITE zemployees.
ENDSELECT.


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

Save and execute the code. The output window should now look like this:

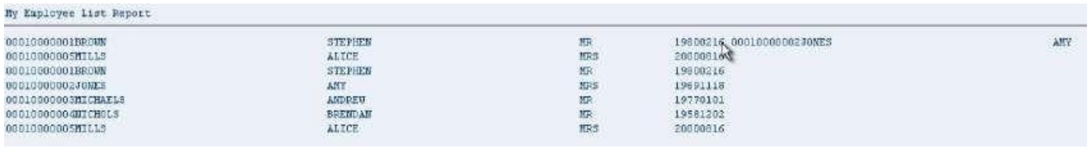


My Employee List Report

```
My Employee List Report                                                                                    1

00010000001BROWN                    STEPHEN                      MR          19800216
00010000002JONES                    AMY                          MRS         19691118
00010000003MICHAELS                 ANDREW                       MR          19770101
00010000004NICHOLS                  BRENDAN                      MR          19581202
00010000005MILLS                    ALICE                        MRS         20000816
00010000001BROWN                    STEPHEN                      MR          19800216
00010000002JONES                    AMY                          MRS         19691118
00010000003MICHAELS                 ANDREW                       MR          19770101
00010000004NICHOLS                  BRENDAN                      MR          19581202
00010000005MILLS                    ALICE                        MRS         20000816
```

The first SELECT loop has created the first five rows, and the second has output the next five.

Both look identical. This is due to the LINE-SIZE limit in the REPORT statement, causing the first five rows to create a new line once they reached 132 characters. If the LINE-SIZE is increased to, for example 532, the effects of the different WRITE statements will be visible:

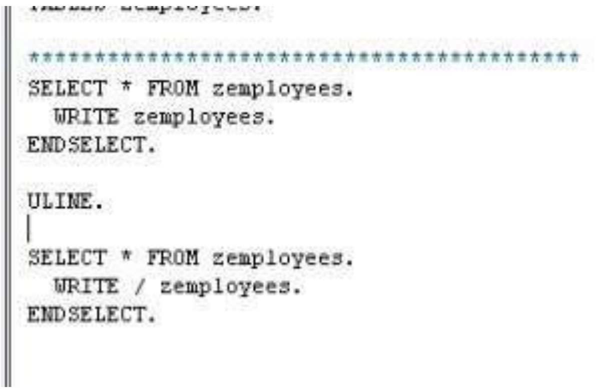


```
My Employee List Report

00010000001BROWN                    STEPHEN                      MR          19800216 00010000002JONES                        AMY
00010000005MILLS                    ALICE                        MRS         20000816
00010000001BROWN                    STEPHEN                      MR          19800216
00010000002JONES                    AMY                          MRS         19691118
00010000003MICHAELS                 ANDREW                       MR          19770101
00010000004NICHOLS                  BRENDAN                      MR          19581202
00010000005MILLS                    ALICE                        MRS         20000816
```

The first five rows, because they do not have a line break in the WRITE statement, have appeared on the first line up until the point at which the 532 character limit was reached and a new line was forced. The first four records were output on the first line. The 5th record appears on a line of its own followed by the second set of five records, having had a line break forced before each record was output.

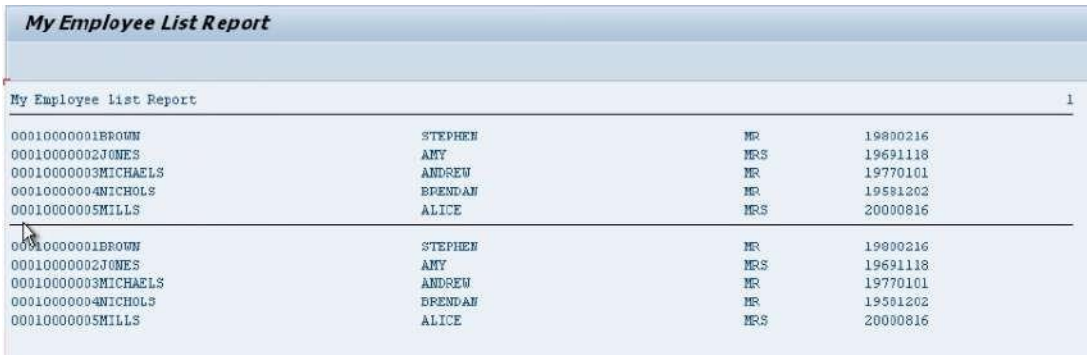
Return the LINE-SIZE to 132, before some more formatting is done to show the separation between the two different SELECT loops.

Above the second SELECT loop, type ***ULINE***. This means underline.



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

ULINE.

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

Click the ULINE statement and press F1 for further explanation from the Documentation window, which will state "Writes a continuous underline in a new line." Doing this will help separate the two different SELECT outputs in the code created. Execute this, and it should look like so:

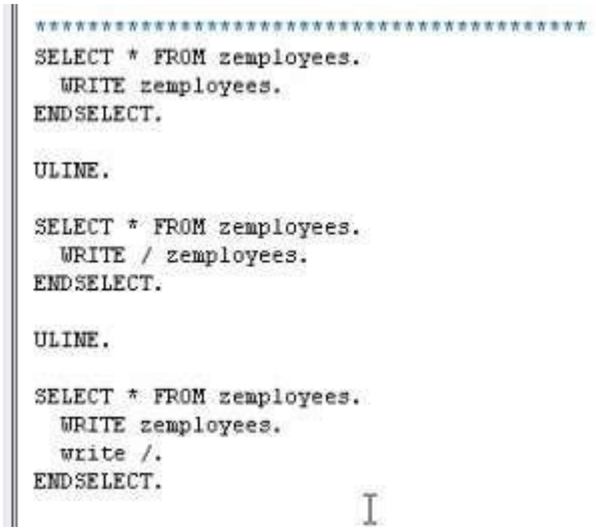


My Employee List Report

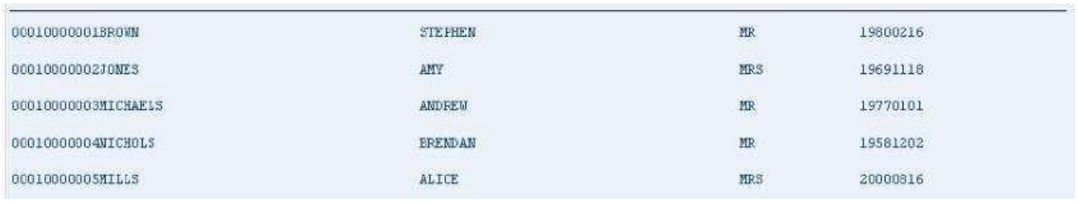
```
My Employee list Report                                                                                        1
------------------------------------------------------------------------------------------------------------------
00010000001BROWN                          STEPHEN                        MR            19800216
00010000002JONES                          AMY                            MRS           19691118
00010000003MICHAELS                       ANDREW                         MR            19770101
00010000004NICHOLS                        BRENDAN                        MR            19581202
00010000005MILLS                          ALICE                          MRS           20000816
------------------------------------------------------------------------------------------------------------------
00010000001BROWN                          STEPHEN                        MR            19800216
00010000002JONES                          AMY                            MRS           19691118
00010000003MICHAELS                       ANDREW                         MR            19770101
00010000004NICHOLS                        BRENDAN                        MR            19581202
00010000005MILLS                          ALICE                          MRS           20000816
```

Duplicate the previous SELECT – ENDSELECT statement block of code again, including the

ULINE, to create a third SELECT output. In this third section, remove the line break from the WRITE statement and, on the line below, type “WRITE /.” This will mean that a new line will be output at the end of the previous line. Execute this to see the difference in the third section:

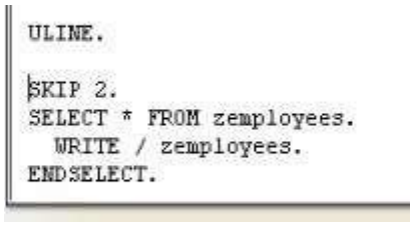


```
*******************************************
SELECT * FROM zemployees.
  WRITE zemployees.
ENDSELECT.

ULINE.

SELECT * FROM zemployees.
  WRITE / zemployees.
ENDSELECT.

ULINE.

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

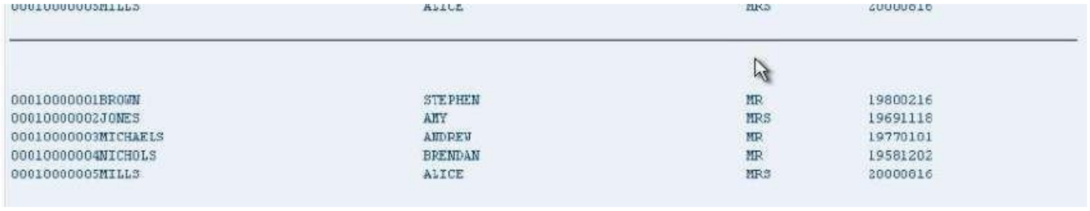


```
00010000001BROWN                 STEPHEN                MR        19800216
00010000002JONES                 AMY                    MRS       19691118
00010000003MICHAELS              ANDREW                 MR        19770101
00010000004NICHOLS               BRENDAN                MR        19581202
00010000005MILLS                 ALICE                  MRS       20000816
```

Now, create another SELECT loop by duplicating the second SELECT loop. This time the WRITE statement will be left intact, but a new statement will be added before the SELECT loop: ***SKIP***, which means to skip a line. This can have a number added to it to specify how many lines to skip, in this case 2. If you press F1 to access the documentation window it will explain further, including the ability to skip to a specific line. The code for this section should look like the first image, and when executed, the second:

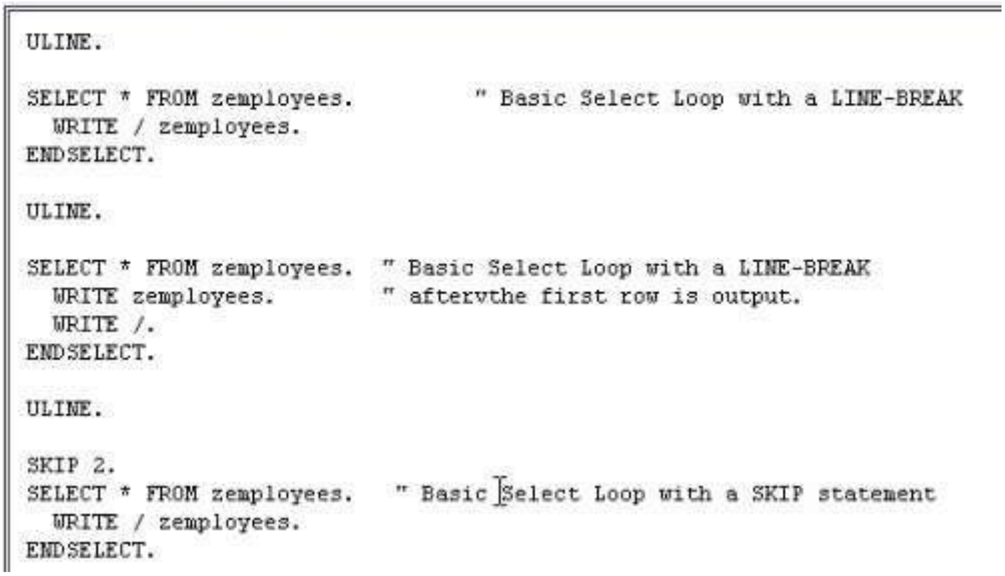


```
ULINE.

SKIP 2.
SELECT * FROM zemployees.
  WRITE / zemployees.
ENDSELECT.
```



```
00010000005MILLS                 ALICE                    MRS        20000816

00010000001BROWN                 STEPHEN                  MR         19800216
00010000002JONES                 AMY                      MRS        19691118
00010000003MICHAELS              ANDREW                   MR         19770101
00010000004NICHOLS               BRENDAN                  MR         19581202
00010000005MILLS                 ALICE                    MRS        20000816
```

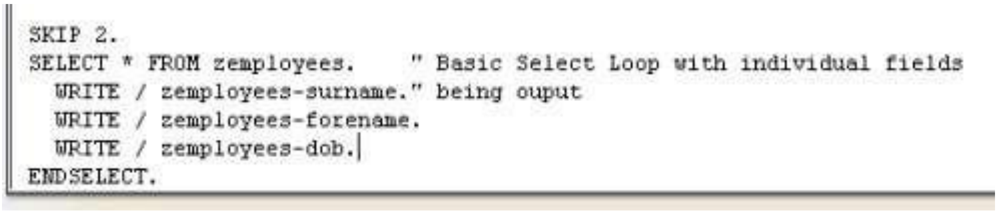
Our program should now look as shown below. Comments have been added to help differentiate the examples.



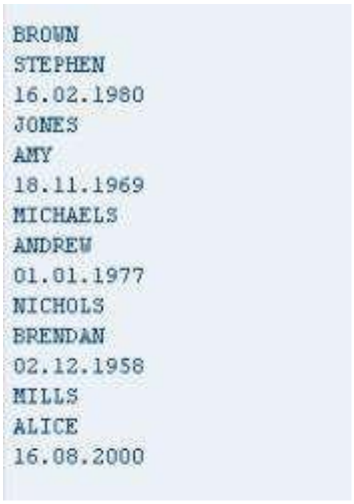
```
ULINE.

SELECT * FROM zemployees.         " Basic Select Loop with a LINE-BREAK
  WRITE / zemployees.
ENDSELECT.

ULINE.

SELECT * FROM zemployees.  " Basic Select Loop with a LINE-BREAK
  WRITE zemployees.        " aftervthe first row is output.
  WRITE /.
ENDSELECT.

ULINE.

SKIP 2.
SELECT * FROM zemployees.   " Basic Select Loop with a SKIP statement
  WRITE / zemployees.
ENDSELECT.
```

Output Individual Fields

Create another SELECT statement. This time, instead of outputting entire rows of the table, individual fields will be output. This is done by specifying the individual field after the WRITE statement. On a new line after the SELECT statement add the following line ***WRITE / zemployees-surname.*** Repeat this in the same SELECT loop for fields Forename and DOB. Then execute the code:

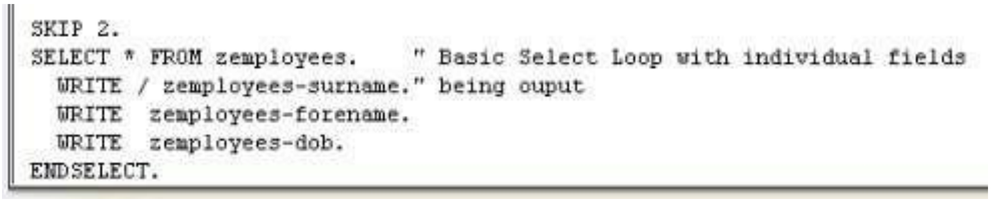


```
SKIP 2.
SELECT * FROM zemployees.    " Basic Select Loop with individual fields
  WRITE / zemployees-surname." being ouput
  WRITE / zemployees-forename.
  WRITE / zemployees-dob.
ENDSELECT.
```

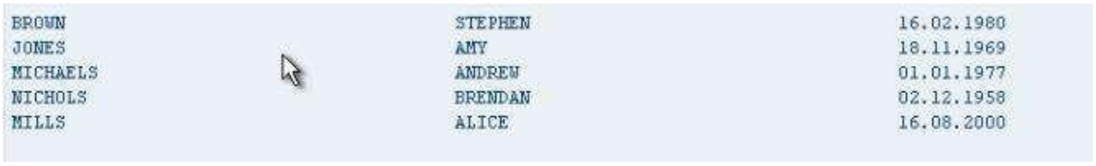


```
BROWN
STEPHEN
16.02.1980
JONES
AMY
18.11.1969
MICHAELS
ANDREW
01.01.1977
NICHOLS
BRENDAN
02.12.1958
MILLS
ALICE
16.08.2000
```

To tidy this up a little remove the **/** from the last 2 WRITE statements which will make all 3 fields appear on 1 line.



```
SKIP 2.
SELECT * FROM zemployees.    " Basic Select Loop with individual fields
  WRITE / zemployees-surname." being ouput
  WRITE  zemployees-forename.
  WRITE  zemployees-dob.
ENDSELECT.
```

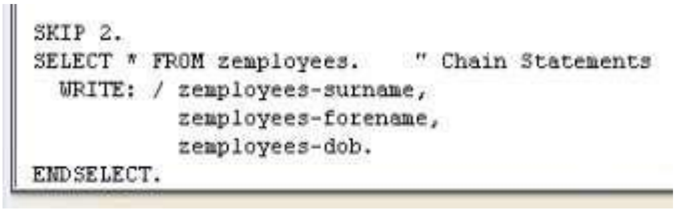


```
BROWN                              STEPHEN                            16.02.1980
JONES                              AMY                                18.11.1969
MICHAELS                           ANDREW                             01.01.1977
NICHOLS                            BRENDAN                            02.12.1958
MILLS                              ALICE                              16.08.2000
```

Chaining Statements Together

We have used the WRITE statement quite a lot up to now and you will see it appear on a regular basis in many standard SAP programs. To save time, the WRITE statements can be chained together, avoiding the need to duplicate the WRITE statement on every line.

To do this, duplicate the previous SELECT loop block of code. After the first WRITE statement, add "**:**" This tells the SAP system that this WRITE statement is going to write multiple fields (or text literals). After the "zemployees-surname" field change the period (.) to a comma (,) and remove the second and third WRITE statements. Change the second period (.) to comma (,) also but leave the last period (.) as is to indicate the end of the statement. This is how we chain statements together and can also be used for a number of other statements too.



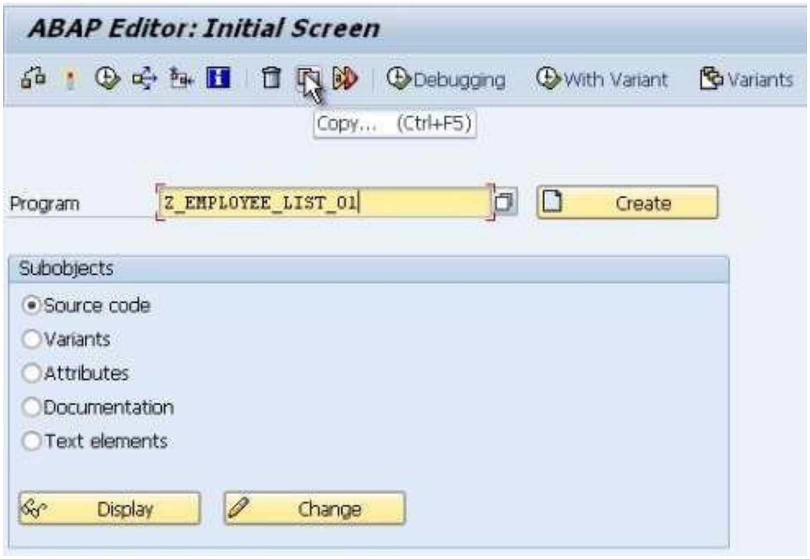
```
SKIP 2.
SELECT * FROM zemployees.    " Chain Statements
  WRITE: / zemployees-surname,
           zemployees-forename,
           zemployees-dob.
ENDSELECT.
```

Execute the code, and the output should appear exactly the same as before.

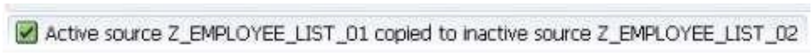
Copy Your Program

Let's now switch focus a little and look at creating fields within the program. There are two types of field to look at here, Variables and Constants.

Firstly, it will be necessary to generate a new program from the ABAP Editor. This can be done either with the steps from the previous section, or by copying a past program. The latter option is useful if you plan on reusing much of your previous code. To do this, launch transaction SE38 again and enter the original program's name into the 'Program' field of the 'Initial' screen, and then click the **Copy** button (CTRL + F5):

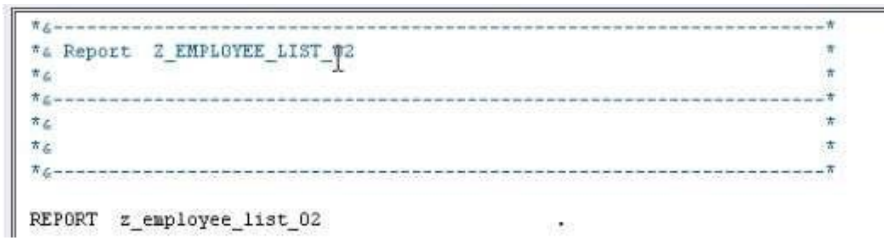


A window will appear asking for a name for the new program, in this instance, enter ***Z_EMPLOYEE_LIST_2*** in the 'Target Program' input box, then press the **Copy** button. The next screen will ask if any other objects are to be copied. Since none of the objects here have been created in the first program, leave these blank, and click **Copy**. The 'Create Object Directory Entry' screen will then reappear and, as before you should assign the entry to 'Local object'. The status bar will confirm the success of the copy:


Active source Z_EMPLOYEE_LIST_01 copied to inactive source Z_EMPLOYEE_LIST_02

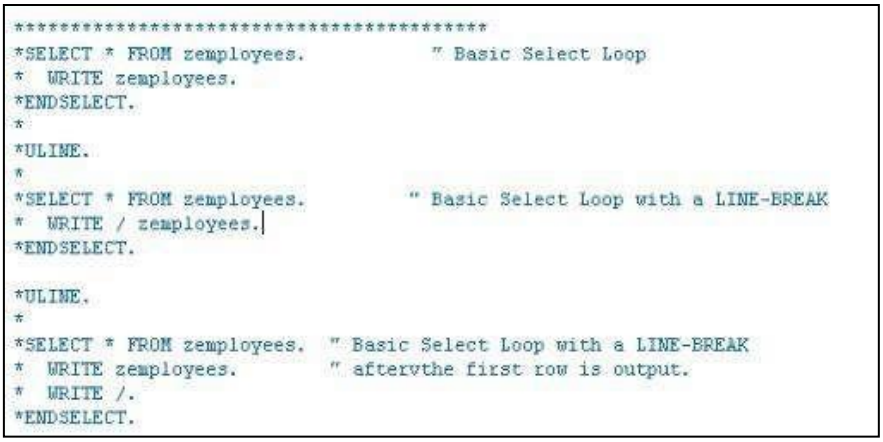
The new program name will then appear in the 'Program' text box of the ABAP Editor Initial screen. Now click the **Change** button to enter the coding screen.

The copy function will have retained the previous report name in the comment space at the top of your program and in the initial REPORT statement, so it is important to remember to update these. Also, delete the LINE-SIZE limit, so that this does not get in the way of testing the program.



```
*&---------------------------------------------------------------------*
*& Report  Z_EMPLOYEE_LIST_02                                          *
*&                                                                     *
*&---------------------------------------------------------------------*
*&                                                                     *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  z_employee_list_02          .
```

Because there are a number of SELECT and WRITE statements in the program, it is worth looking at how to use the fast comment facility. This allows code to be, in practical terms, removed from the program without deleting it, making it into comments, usually by inserting an asterisk (*) at the beginning of each line. To do this quickly, highlight the lines to be made into comment and hold down CTRL + <. This will automatically comment the lines selected. Alternatively, the text can he highlighted and then in the 'Utilities' menu, select 'Block/Buffer' and then 'Insert Comment *'. The selected code is now converted to comment:



```
******************************************
*SELECT * FROM zemployees.          " Basic Select Loop
*  WRITE zemployees.
*ENDSELECT.
*
*ULINE.
*
*SELECT * FROM zemployees.         " Basic Select Loop with a LINE-BREAK
*  WRITE / zemployees.
*ENDSELECT.

*ULINE.
*
*SELECT * FROM zemployees.  " Basic Select Loop with a LINE-BREAK
*  WRITE zemployees.        " aftervthe first row is output.
*  WRITE /.
*ENDSELECT.
```

Delete most of the code from the program now, retaining one section to continue working with.

Declaring Variables

A field is a temporary area of memory which can be given a name and referenced within programs. Fields may be used within a program to hold calculation results, to help control the logic flow and, because they are temporary areas of storage (usually held in the RAM), can be accessed very fast, helping to speed up the program's execution. There are, of course, many other uses for fields.

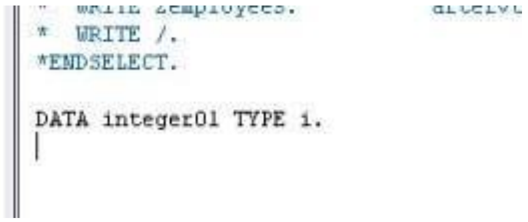
The next question to examine is that of variables, and how to declare them in a program. A variable is a field, the values of which change during the program execution, hence of course the term variable.

There are some rules to be followed when dealing with variables:

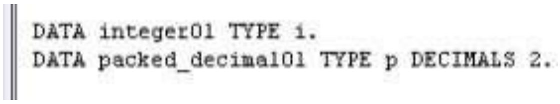
- They must begin with a letter.
- Can be a maximum size of 30
- characters, Cannot include + , : or ( )
- in the name, Cannot use a reserved word.

When creating variables, it is useful to ensure the name given is meaningful. Naming variables things like A1, A2, A3 and so on is only likely to cause confusion when others come to work with the program. Names like, in the example here, 'Surname', 'Forename', 'DOB' are much better, as from the name it can be ascertained exactly what the field represents.

Variables are declared using the DATA statement. The first variable to be declared here will be an integer field. Below the section of code remaining in your program, type the statement **DATA** followed by a name for the field - **integer01**. Then, the data type must be declared using the word **TYPE** and for integers this is referred to by the letter **i**. Terminate the statement with a period.


```
*  WRITE /.
*ENDSELECT.

DATA integer01 TYPE i.
```

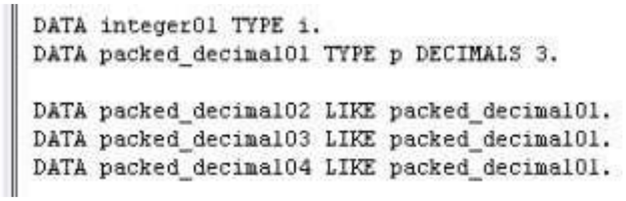
Try another, this time named **packed_decimal01**, the data type for which is **p**. A packed decimal field is there to help store numbers with decimal places. It is possible to specify the number of decimal places you want to store. After the 'p', type the word **decimals** and then the number desired, in this instance, 2 (packed decimal can store up to 14 decimal places). Type all of this, then save the program:


```
DATA integer01 TYPE i.
DATA packed_decimal01 TYPE p DECIMALS 2.
```

These data types used are called elementary. These types of variables have a fixed length in ABAP, so it is not necessary to declare how long the variables need to be.

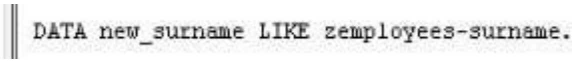
There is another way of declaring variables, via the **LIKE** addition to the DATA statement. Declare another variable, this time with the name **packed_decimal02** but, rather than using the TYPE addition to define the field type, use the word LIKE, followed by the previous variable's name "packed_decimal01". This way, you can ensure subsequent variables take on exactly the same properties as a previously created one. Copy and paste this several times to create packed_decimal03 and 04.

If you are creating a large number of variables of the same data type, by using the LIKE addition, a lot of time can be saved. If, for example, the DECIMALS part were to need to change to 3, it would then only be necessary to change the number of decimals on the original variable, not all of them individually:



```
DATA integer01 TYPE i.
DATA packed_decimal01 TYPE p DECIMALS 3.

DATA packed_decimal02 LIKE packed_decimal01.
DATA packed_decimal03 LIKE packed_decimal01.
DATA packed_decimal04 LIKE packed_decimal01.
```

Additionally, the LIKE addition does not only have to refer to variables, or fields, within the program. It can also refer to fields that exist in tables within the SAP system. In the table we created there was a field named 'Surname'. Create a new variable called **new_surname** using the DATA statement. When defining the data type use the LIKE addition followed by **zemployees-surname.** Defining fields this way saves you from having to remember the exact data type form every field you have to create in the SAP system.

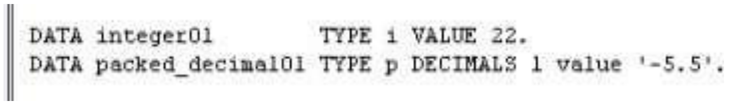


```
DATA new_surname LIKE zemployees-surname.
```

Check this for syntax errors to make sure everything is correct. If there are no errors remove the *new_surname, packed_decimal02, 03* and *04* fields as they are no longer needed.

With another addition which can be made to the DATA statement, one can declare initial values for the variables defined in the program. For the "integer01" variable, after “TYPE i”, add the following addition: **VALUE 22**. This will automatically assign a value of 22 to "integer01" when the program starts.

For packed decimal fields the process is slightly different. The VALUE here must be specified within single quotation marks, **'5.5'** as without these, the ABAP statement would be terminated by the period in the decimal. Note that one is not just limited to positive numbers. If you want to declare a value of a negative number, this is entirely possible:

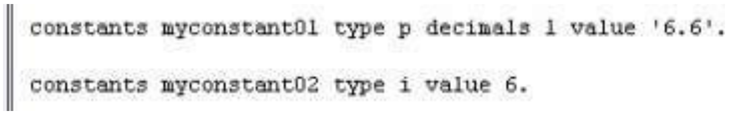


```
DATA integer01        TYPE i VALUE 22.
DATA packed_decimal01 TYPE p DECIMALS 1 value '-5.5'.
```

Constants

A constant is a variable whose associated value cannot be altered by the program during its execution, hence the name. Constants are declared with the CONSTANTS statement (where the DATA statement appeared for variables). When writing code then, the constant can only ever be referred to; its value can never change. If you do try to change a Constant's value within the program, this will usually result in a runtime error.

The syntax for declaring constants is very similar to that of declaring variables, though there are a few differences. You start with the statement **CONSTANTS**. Use the name **myconstant01** for this example. Give it a type p as before with 1 decimal place and a value of **'6.6'**. Copy and paste and try another with the name **myconstant02**, this time a standard integer (type '**i**') with a value of **6**:



```
constants myconstant01 type p decimals 1 value '6.6'.

constants myconstant02 type i value 6.
```

(*A note: one cannot define constants for data types XSTRINGS, references, internal tables or structures containing internal tables.)*

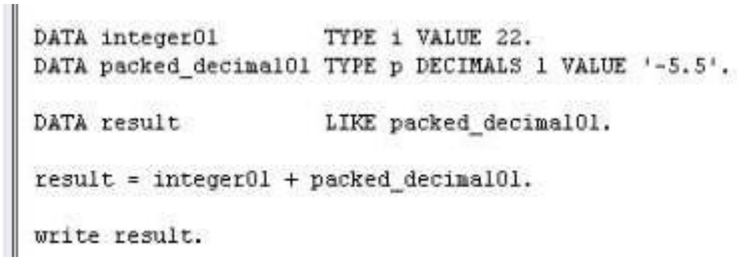
Arithmetic – Addition

Now that the ability to create variables has been established, these can be used for calculations within a program. This chapter will begin by looking at some of the simple arithmetical calculations within ABAP.

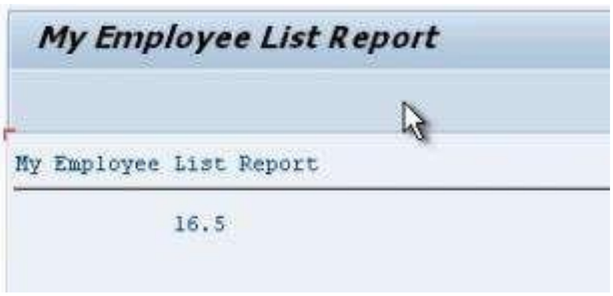
Our program will be tidied up by removing the two constants which were just created. If a program needs to add two numbers together and each number is stored as its own unique variable, the product of the two numbers can be stored in a brand new variable titled “result”.

Create a new DATA statement, name this “**result**” and use the LIKE statement to give it the same properties as packed_decimal01, terminating the line with a period.

To add two numbers together, on a new line, type “**result = integer01 + packed_decimal01.**” On a new line enter, “**WRITE result.**” Activate and test the program, and the result will appear in the output screen:



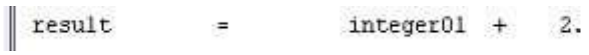
```
DATA integer01        TYPE i VALUE 22.
DATA packed_decimal01 TYPE p DECIMALS 1 VALUE '-5.5'.

DATA result           LIKE packed_decimal01.

result = integer01 + packed_decimal01.

write result.
```



```
My Employee List Report

          16.5
```

Things to remember: For any arithmetical operation, the calculation itself must appear to the right of the =, and the variable to hold the result to the left. This ensures that only the result variable will be updated in the execution. If the variable titled "result" had been assigned a value prior to the calculation, this would be overwritten with the new value. Spaces must always be inserted on either side of the = and + signs. This applies to all arithmetical operators, including parentheses ( ), which will start to be used as the calculations become more complicated. Note that one space is the minimum, and multiple spaces can be used, which may help in lining code up to make it more readable, and indeed where calculations may be defined over many lines of code.

It is not just the products of variables which can be calculated in calculations, but also individual literal values, or a mixture of the two, as shown here:

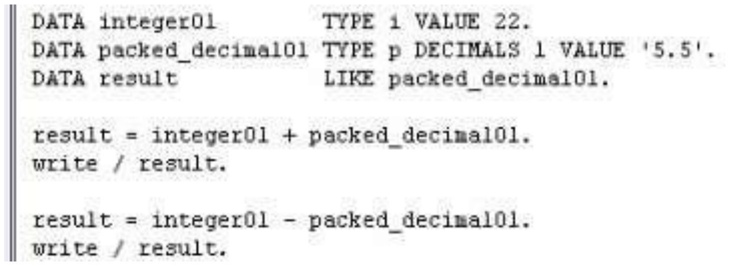


```
result        =        integer01  +   2.
```

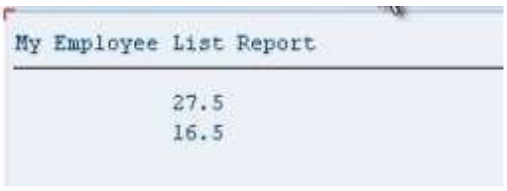
Arithmetic – Subtraction

To subtract numbers, the same method is used, replacing the **+** with a **-**. Copy and paste the previous calculation and make this change. Also, to make this simpler to understand, change the value of packed_decimal01 from -5.5 to 5.5. One can see by doing this the way that changing the initial variable will alter the calculation.

Execute the code:



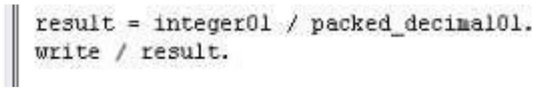
```
DATA integer01        TYPE i VALUE 22.
DATA packed_decimal01 TYPE p DECIMALS 1 VALUE '5.5'.
DATA result           LIKE packed_decimal01.

result = integer01 + packed_decimal01.
write / result.

result = integer01 - packed_decimal01.
write / result.
```



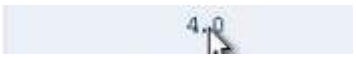
```
My Employee List Report

          27.5
          16.5
```

Arithmetic – Division

To divide numbers, the same method is followed, but the arithmetical operator this time will be a **/**



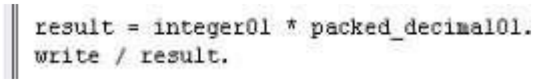
```
result = integer01 / packed_decimal01.
write / result.
```



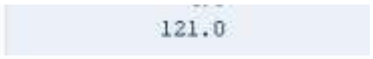
```
4.0
```

Arithmetic – Multiplication

To multiply, the operator is a *****

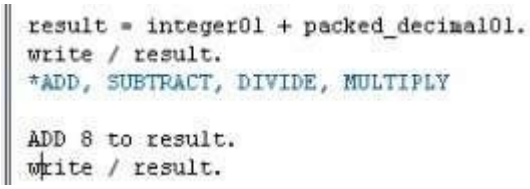


```
result = integer01 * packed_decimal01.
write / result.
```

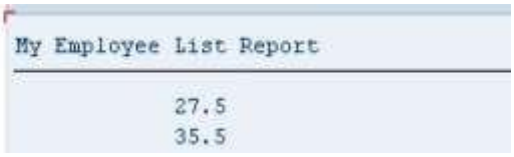


```
121.0
```

Additionally to these methods, the statements **ADD**, **SUBTRACT**, **DIVIDE** and **MULTIPLY** can be used. The syntax for these is slightly different. Beneath the first calculation (where integer01 and packed_decimal01 where added), write a new line of code "**ADD 8 to result.**" (Ignore the comment line in the image):



```
result = integer01 + packed_decimal01.
write / result.
*ADD, SUBTRACT, DIVIDE, MULTIPLY

ADD 8 to result.
write / result.
```



```
My Employee List Report

        27.5
        35.5
```

While this is a legitimate method for calculations, it must be added that this is very rarely used, as the initial method is much simpler.

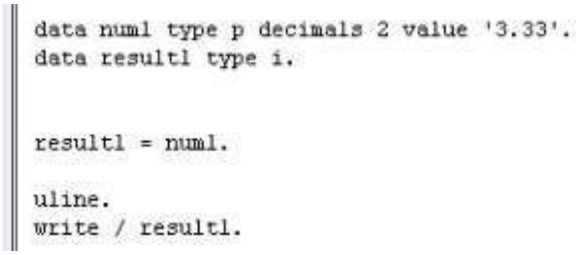
Conversion Rules

In this program, different data types have been used when declaring variables. It is the responsibility of the programmer to ensure the data types used are compatible with one another when used for calculations or moving data to and from objects. One should not attempt calculations with variables and numbers which do not match.

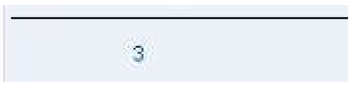
For example, a variable defined as an integer cannot be multiplied by a character, as these two data types are incompatible. This would cause the system to generate syntax and runtime errors when the program is executed. While SAP has built in automatic data type conversions for many of the standard data types within ABAP, there are scenarios where the inbuilt conversion rules are not appropriate. It is important to become familiar with the inbuilt conversion rules and know when to manipulate the data prior to using them in calculations. Here, some examples of conversion rules will be given, so that they can be used throughout programs created.

Conversion rules are pre-defined logic that determine how the contents of the source field can be entered into a target field. If one attempts to insert an integer field containing the value of 1 to a character string, the built-in conversion rules will determine exactly how this should be done without any syntax or runtime errors.

For example, create a DATA statement with the name "**num1**" of **TYPE p** (packed decimal) with **DECIMALS 2** and a **VALUE** of '**3.33**'. Then create another variable with the name "**result1**" of type **i** (integer). Attempt the calculation "**result1 = num1**". The conversion rule here would round the number to the closest integer, in this case 3.



```
data num1 type p decimals 2 value '3.33'.
data result1 type i.


result1 = num1.

uline.
write / result1.
```



```
3
```

As you work with different data types, these kinds of conversion rules will often be applied automatically, and it is up to you, the programmer, to understand these conversion rules and the data types used within the program to ensure no runtime errors occur.

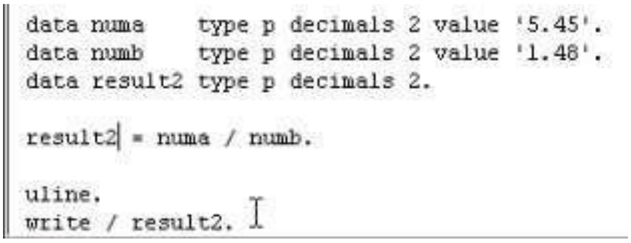
Division Variations

Now, a slight step back will be taken to discuss the division operator further. In ABAP, there are three ways in which numbers can be divided:

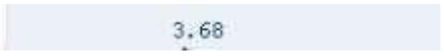
- The standard result with decimal
- places The remainder result
- The integer result.

The standard form of division.

Create 2 variables, "**numa**" and "**numb**", with values of **5.45** and **1.48** respectively, then create the variable "**result2**" (also with 2 decimal places). Then insert the calculation "**result2 = numa / numb.**" followed by a **WRITE** statement for result2. Execute the program.



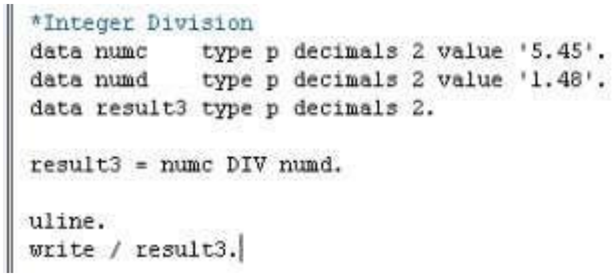
```
data numa    type p decimals 2 value '5.45'.
data numb    type p decimals 2 value '1.48'.
data result2 type p decimals 2.

result2 = numa / numb.

uline.
write / result2.
```



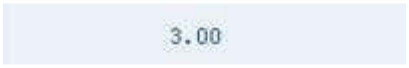
```
3.68
```

The integer form of division.

Copy the initial calculation; change the initial variables to "**numc**" and "**numd**" and the resulting variable to "**result3**". For integer division, rather than using the standard **/**, use the operator **DIV**. This will give the result of the calculation's integer value, without the decimal places.



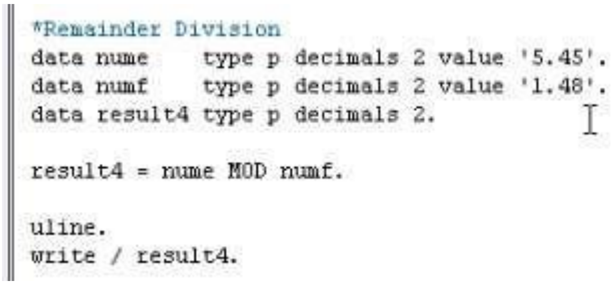
```
*Integer Division
data numc    type p decimals 2 value '5.45'.
data numd    type p decimals 2 value '1.48'.
data result3 type p decimals 2.

result3 = numc DIV numd.

uline.
write / result3.
```



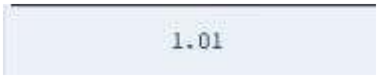
```
3.00
```

The remainder form of division.

Follow the steps from the integer form, this time with "**nume**", "**numf**" and "**result4**". For this type of division, the arithmetical operator should be **MOD**. This, when executed, will show the remainder value.



```
*Remainder Division
data nume    type p decimals 2 value '5.45'.
data numf    type p decimals 2 value '1.48'.
data result4 type p decimals 2.

result4 = nume MOD numf.

uline.
write / result4.
```



```
1.01
```

Declaring C and N Fields

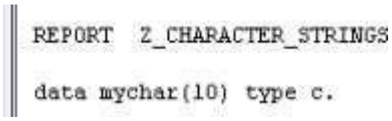
This chapter will discuss character strings. When creating programs, fields defined as char- acter strings are almost always used. In SAP, there are two elementary data types used for character strings. These are data type **C**, and data type **N**.

Data type C.

Data type C variables are used for holding alphanumeric characters, with a minimum of 1 character and a maximum of 65,535 characters. By default, these are aligned to the left.

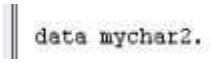
Begin this chapter by creating a new program. From the ABAP Editor's initial screen, cre- ate a new program, named "**Z_Character_Strings**". Title this "**Character Strings Exam- ples**", set the Type to '**Executable program**', the Status to '**Test program**', the Application to '**Basis**', and Save.

Create a new DATA field, name this "**mychar**" and, without any spaces following this, give a number for the length of the field in parentheses. Then, include a space and define the TYPE as **c**



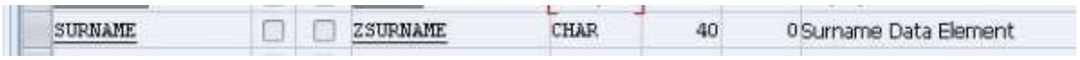
```
REPORT  Z_CHARACTER_STRINGS

data mychar(10) type c.
```

This is the long form of declaring a type c field. Because this field is a generic data type, the system has default values which can be used so as to avoid typing out the full length of the declaration. If you create a new field, named "**mychar2**" and wish the field to be 1 character in size, the default field size is set to 1 character by default, so the size in brack- ets following the name is unnecessary. Also, because this character field is the default type used by the system, one can even avoid defining this. In the case of mychar2, the variable can be defined with only the field name. The code in the image below performs exactly the same as if it was typed "**data mychar2(1) type c**":



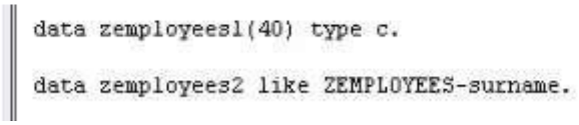
```
data mychar2.
```

In the previous chapter, the table “zemployees” included various fields of type c, such as “zsurname”. If one uses the TABLES statement followed by zemployees, then by double- clicking the table name to use forward navigation and view the table, one can see that the “surname” field is of data type CHAR, with length 40. This declaration can be replicated within the ABAP program:



SURNAME	☐	☐	ZSURNAME	CHAR	40	0	Surname Data Element

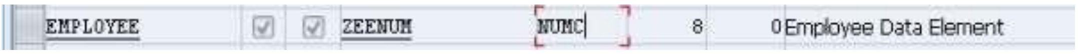
Return to the program, and in place of mychar2, create a new field named “**zemploy- ees1**”, with a length of **40** and type **c**. This will have exactly the same effect as the previous declaration. Referring back to previous chapter, another way of doing this would be to use the LIKE statement to declare zemployees (or this time zemployees2) as having the same properties as the “surname” field in the table:



```
data zemployees1(40) type c.

data zemployees2 like ZEMPLOYEES-surname.
```

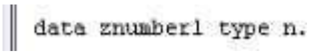
Data type N.

The other common generic character string data type is N. These are by default right- aligned. If one looks at the initial table again, using forward navigation, the field named “employee”, which refers to employee numbers, is of the data type NUMC, with a length of 8. NUMC, or the number data type, works similarly to the character data type, except with the inbuilt rule to only allow the inclusion of numeric characters. This data type, then, is ideal when the field is only to be used for numbers with no intention of carrying out cal- culations.



EMPLOYEE	✓	✓	ZEENUM	NUMC	8	0	Employee Data Element

To declare this field in ABAP, create a new DATA field named “**znumber1**”, TYPE **n**. Again, alternatively this can be done by using the LIKE statement to refer back to the original field in the table.



```
data znumber1 type n.
```

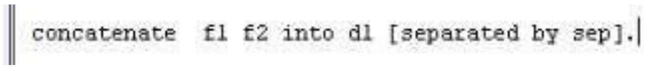
String Manipulation

Like many other programming languages, ABAP provides the functionality to interrogate and manipulate the data held in character strings. This section will look at some of the popular statements which ABAP provides for carrying out these functions:

- Concatenating String Fields
- Condensing Character Strings
- Finding the Length of a String
- Searching for Specific
- Characters The SHIFT
- statement
- Splitting Character Strings SubFields

Concatenate

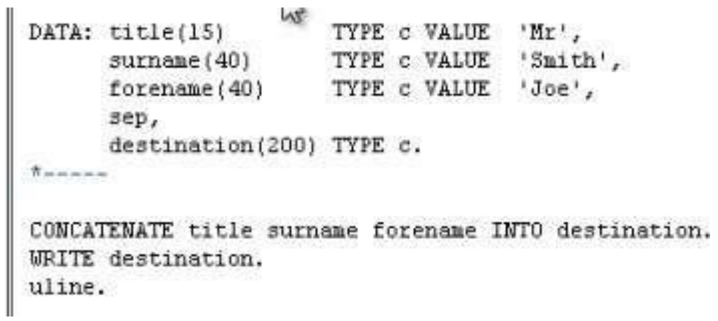
The concatenate statement allows two character strings to be joined so as to form a third string. First, type the statement **CONCATENATE** into the program, and follow this by speci- fying the fields, here “**f1**”, “**f2**” and so on. Then select the destination which the output string should go to, here “**d1**”. If one adds a subsequent term, **[separated by sep]** (“sep” here is an example name for the separator field), this will allow a specified value to be in- serted between each field in the destination field:



```
concatenate  f1 f2 into d1 [separated by sep].
```

Note: If the destination field is shorter than the overall length of the input fields, the char- acter string will be truncated to the length of the destination field, so ensure when using the CONCATENATE statement, the string data type is being used, as these can hold over 65,000 characters.

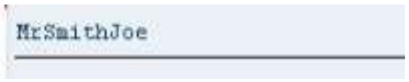
As an example, observe the code in the image below.



```
DATA: title(15)          TYPE c VALUE  'Mr',
      surname(40)        TYPE c VALUE  'Smith',
      forename(40)       TYPE c VALUE  'Joe',
      sep,
      destination(200) TYPE c.
*-----

CONCATENATE title surname forename INTO destination.
WRITE destination.
uline.
```

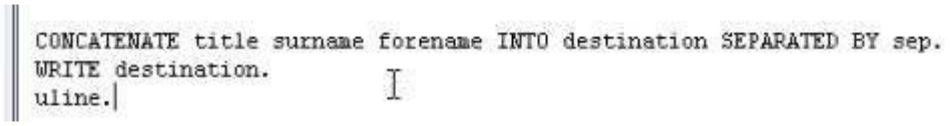
The first 3 fields should be familiar by now. The fourth is the separator field, here again called “sep” (the size of sep has not been defined here, and so it will take on the default which the system uses - 1 character). The last field is titled “destination”, 200 characters long and of data type c.

Below this section is the CONCATENATE statement, followed by the fields to combine to- gether into the destination field. The WRITE statement is then used to display the result. Executing this code will output the following:

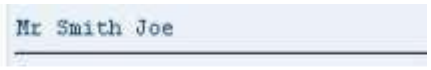


```
MrSmithJoe
```

Note that the text has been aligned to the left, as it is using data type c. Also, the code did not include the **SEPARATED BY** addition, and so the words have been concatenated with- out spaces. This can be added, and spaces will appear in the output:



```
CONCATENATE title surname forename INTO destination SEPARATED BY sep.
WRITE destination.
uline.
```



```
Mr Smith Joe
```