Beginner's Guide To

SAP ABAP

An Introduction To Programming SAP Applications Using ABAP

PETER MOXON

BEGINNER'S GUIDE TO SAP ABAP

AN INTRODUCTION TO PROGRAMMING SAP APPLICATIONS USING ABAP

PETER MOXON

PUBLISHED BY:

SAPPROUK Limited

http://www.saptraininghq.com

Table of Contents

Contact the Author

As the reader of this book you are my most important critic and commentator. I would love to hear from you to let me know what you did and did not like about this book, as well as to what you think I could do in future books to make them stronger.

E-mail: pete@sappro.co.uk

Please note that although I cannot personally help you learn SAP ABAP, I am available for corporate hire for project management, technical lead and mentoring programs.

Refer to my website http://www.saptraininghq.com to see all the training material I have available and to get a good overview of my expertise.

Introduction

This book has been written with SAP Super-User and Consultants in mind. Whether your current job title is functional consultant, system support analyst, business consultant, project manager for something entirely different, if you are responsible for all have an interest in creating ABAP programs, then this book is for you.

Much of the book is written in the "***How-To***" style and will allow anybody to follow along and create ABAP programs from scratch. It is written in such a way that each chapter builds on the last so that you become familiar in lots of different aspects of SAP ABAP programming to enable you to then start creating your own programs and understand programs you will find in your own SAP system.

The principles and guidelines apply across all SAP modules whether you're writing programs for HR, FI, SD or one of the many other modules within SAP.

Over my years of working with SAP systems I have had the great pleasure of working with some top-notch functional and technical consultants who know how to document, plan and develop SAP programs of all types. Likewise I have had the unpleasant experience of working with lower quality consultants, who either race through or stumble and stutter through their SAP work copying and pasting from one program or another resulting in difficult to support programs. This ultimately often results in project delays and cost overruns.

The aim of this book is to help you understand how SAP ABAP programs are put together and developed so that you will produce detailed concise understandable and functional programs that correspond with your specifications and most importantly delivered on time and on budget.

How to Use This Book

There are several ways to go through this book and the best way depends on your situation.

If you are new to writing SAP programs then I suggest starting at the very beginning and working through each chapter one after another.

If you are familiar with some SAP ABAP programming then you may want to use the table of contents and jump to the chapter that interests you, but remember each chapter builds on the previous chapter so some of the examples shown do require you to have knowledge of the database tables we create in this book.

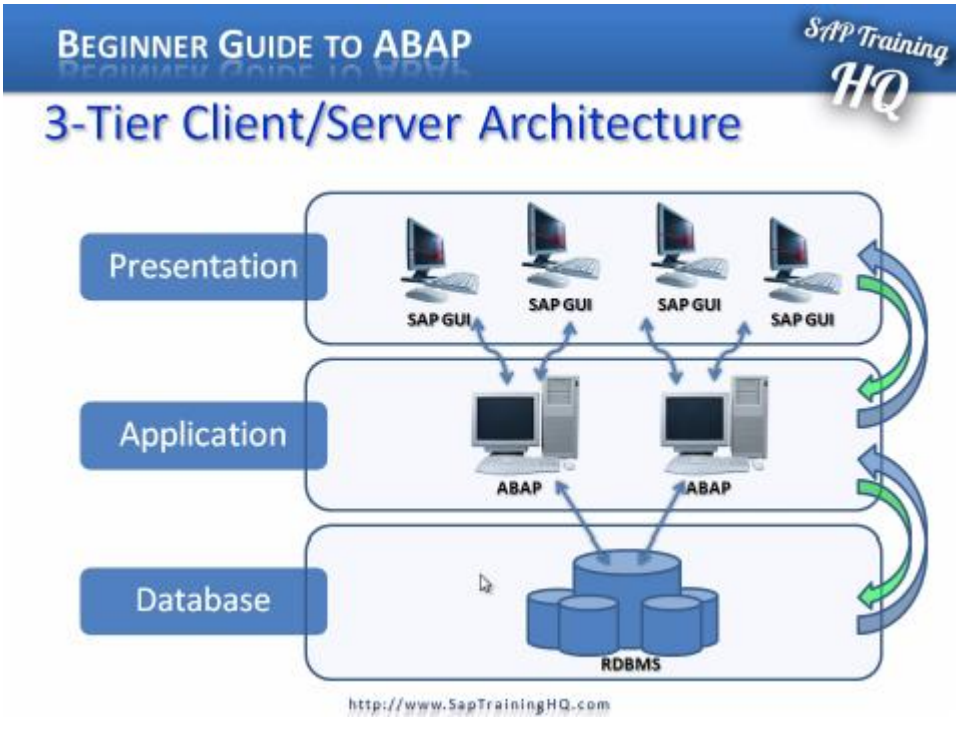
Chapter 1: SAP System Overview

We will start out by covering the high-level architecture of an SAP system, including the technical architecture and platform independence. We will dig into the environment that our ABAP programs run in, which include the work processes and the basic structures of an ABAP program. Then we can focus on a running SAP system, discuss the business model overview, and begin looking at the ABAP workbench.

SAP System Architecture

First, the Technical Architecture of a typical SAP system will be discussed, before moving on to the Landscape Architecture, and a discussion of why the landscape should be broken into multiple systems.

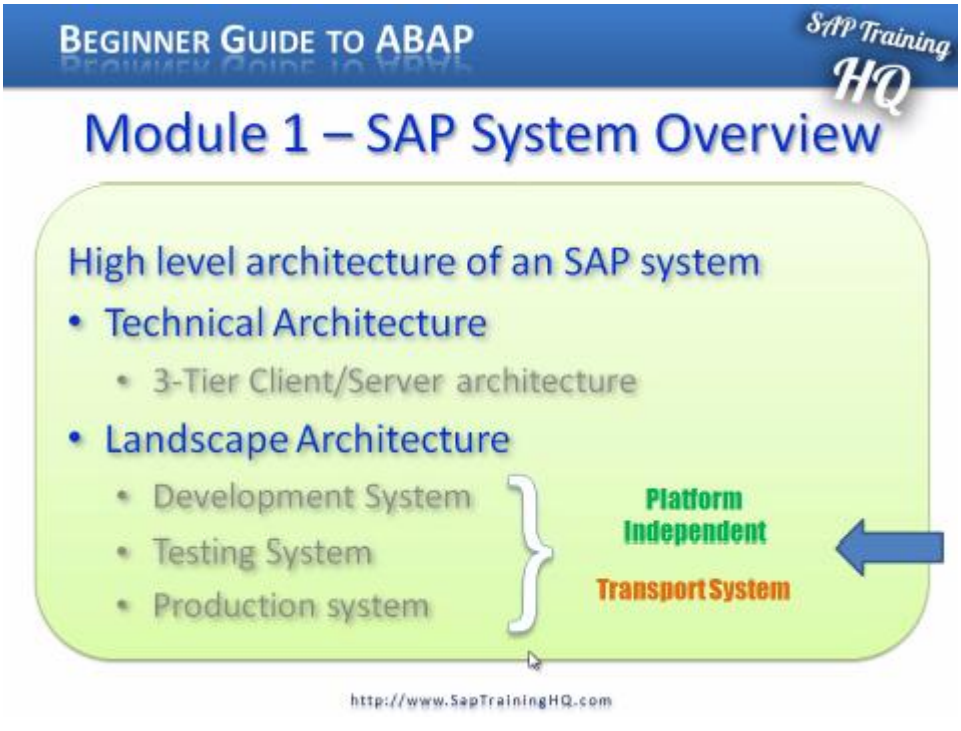
This diagram shows the 3-tier Client/Server architecture of a typical SAP system:



At the top is the Presentation server, which is any input device that can be used to control an SAP system (the diagram shows the SAP GUI, but this could equally be a web browser, a mobile device, and so on). The Presentation layer communicates with the Application server, and the Application server is the 'brains' of an SAP system, where all the central processing takes place. The Application server is not just one system in itself, but can be made up of multiple instances of the processing system. The Application server, in turn, communicates with the Database layer.

The Database is kept on a separate server, mainly for performance reasons, but also for security, providing a separation between the different layers of the system.

Communication happens between each layer of the system, from the Presentation layer, to the Application server, to the Database, and then back up the chain, through the Application server again, for further processing, until finally reaching the Presentation layer.



A typical Landscape Architecture - Typical here is subjective, in practical terms there is not really any such thing as a standard, 'typical' landscape architecture which most companies

use. However, it is common to find a Development system, a Testing system and a Production system:

The reason for this is fairly simple. All the initial development and testing is done on a Development system, which ensures other systems are not affected. Once developments are at a stage where they may be ready to be tested by an external source, or someone within the company whose role is to carry out testing, the developments are moved, using what is called a Transport System, to the next system (here, the Testing system).

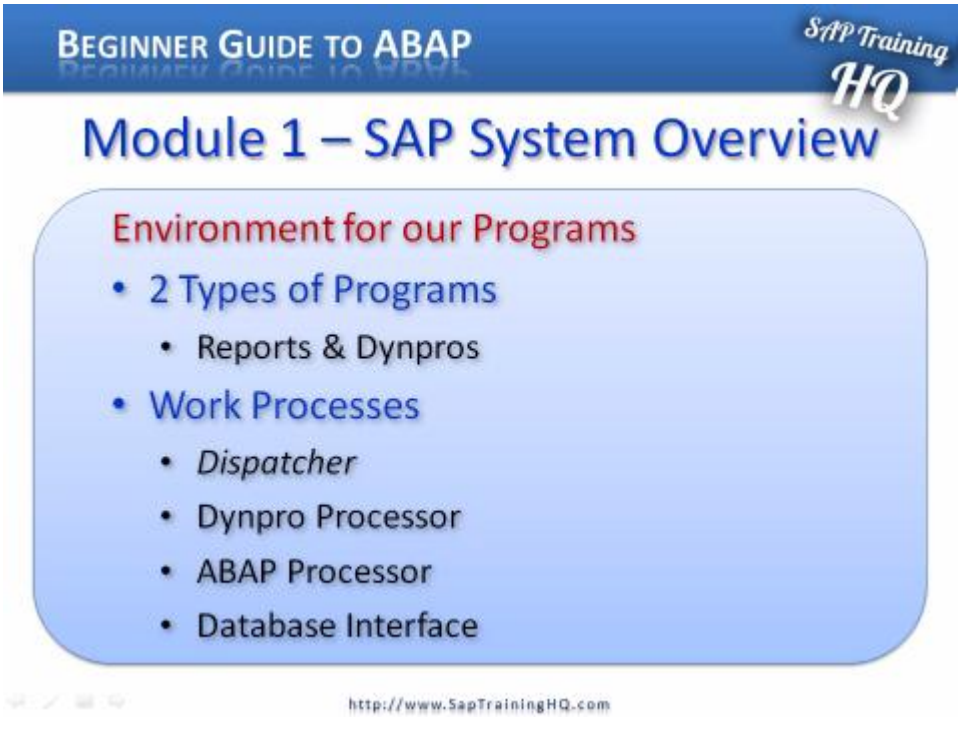
Normally, no development at all is done on the testing system; it is just used for testing the developments from the development system. If everything passes through the Testing system, a Transport system is used again to move the developments into the Production environment. When code enters the Production environment, this is the stage at which it is turned on, and used within the business itself.

The landscape architecture is not separated just for development purposes; the company may have other reasons. This could be the quantity of data in the Production system, which may be too great to be used in the development environment (normally the Development and Testing systems are not as large as the Production system, only needing a subset of data to test on). Also, it could be for security reasons. More often than not, companies do not want developers to see live production data, for data security reasons (for example, the system could include employee data or sales data, which a company would not want people not employed in those areas to see). Normally, then, the Development and Testing systems would have their own set of data to work with.

The three systems described here, normally, are a minimum. It can increase to four systems, perhaps with the addition of a Training system, or perhaps multiple projects are running simultaneously, meaning there may be two separate Development systems, or Testing systems, even perhaps a Consolidation system before anything is passed to the Production environment. This is all, of course, dependent on the company, but commonly each system within the Landscape architecture will have its own Application server and its own Database server, ensuring platform independence.

Environment for Programs

Next, we have the environment which programs run in, the Work Processes, and the structure of an ABAP program.



Within an SAP system, or at least the example used here, there are two types of programs, Reports and Dynpro's.

Reports, as the name would suggest, are programs which generate lists of data. They may involve a small amount of interactivity, but mainly they supply data to the front-end interfaces, the SAP GUI and so on. When a user runs a report, they typically get a selection screen. Once they enter their selection parameters and execute the report, they normally cannot intervene in the execution of the program. The program runs, and then displays the output.

Dynpro's are slightly different. They are dynamic programs, and allow the user to intervene in the execution of the program, by processing a series of screens, called

Dialogue screens. The user determines the flow of the program itself by choosing which buttons or fields to interact with on the screen. Their action then triggers different functions which have been coded within the flow logic of the program. While reports are being created, interfaces are also to be generated which are classed as Dynpro's, for all the selection criteria.

Most of the work done by people involved with ABAP is done within Report programs, and even though these programs are labelled 'Reports', they do not always generate output. The Report programs are there to process the logic, reading and writing to the Database, in order to make the system work.

Work Processes

Every program that runs in an SAP system runs on what are called Work Processes, which run on the Application server. Work Processes themselves work independently of the computer's operating system and the Database that it interacts with, giving the independence discussed earlier with regard to the Technical architecture. When an SAP system is initially set up, the basis consultants (who install the system, keep it running, manage all the memory and so on) configure SAP in such a way that it automatically sets the number of Work Processes programs use when they start, the equivalent of setting up a pre-defined number of channels or connections to the Database system itself, each of which tend to have their own set of properties and functions.

The Dispatcher

You might come across something referred to as the Dispatcher. The SAP system has no technical limits as to the number of users who can log on and use it, generally the number of users who can access an SAP system is much larger than the number of available Work Processes the system is configured for. This is because not everybody is sending instructions to the Application server at exactly the same time. Because of this, users cannot be assigned a certain number of processes while they are logged on.

The Dispatcher controls the distribution of the Work Processes to the system users. The Dispatcher keeps an eye on how many Work Processes are available, and when a user triggers a transaction, the Dispatcher's job is to provide that user with a Work Process to use. The Dispatcher tries to optimise things as far as possible, so that the same Work Process receives the sequential Dialogue steps of an application. If this is not possible, for example because the user takes a long time between clicking different aspects of the

screen, it will then select a different Work Process to continue the processing of the Dialogue program. It is the Work Process which executes an application, and it is the Work Process which has access to the memory areas that contain all of the data and objects an application uses. It also makes three very important elements available.

The first is the Dynpro processor. All Dynpro programs have flow and processing logic, and it is the Dynpro processor's job to handle the flow logic. It responds to the user's interactions, and controls the further flow of the program depending on these interactions. It is responsible for Dialogue control and the screen itself, but it is important to remember that it cannot perform calculations; it is purely there to manage the flow logic of a program.

The next important element is the ABAP processor, which is responsible for the processing logic of the programs. It receives screen entries from the Dynpro processor, and transmits the screen output to the program. It is the ABAP processor which can perform the logical operations and arithmetical calculations in the programs. It can check authorisations, and read and write to the Database, over the Database Interface.

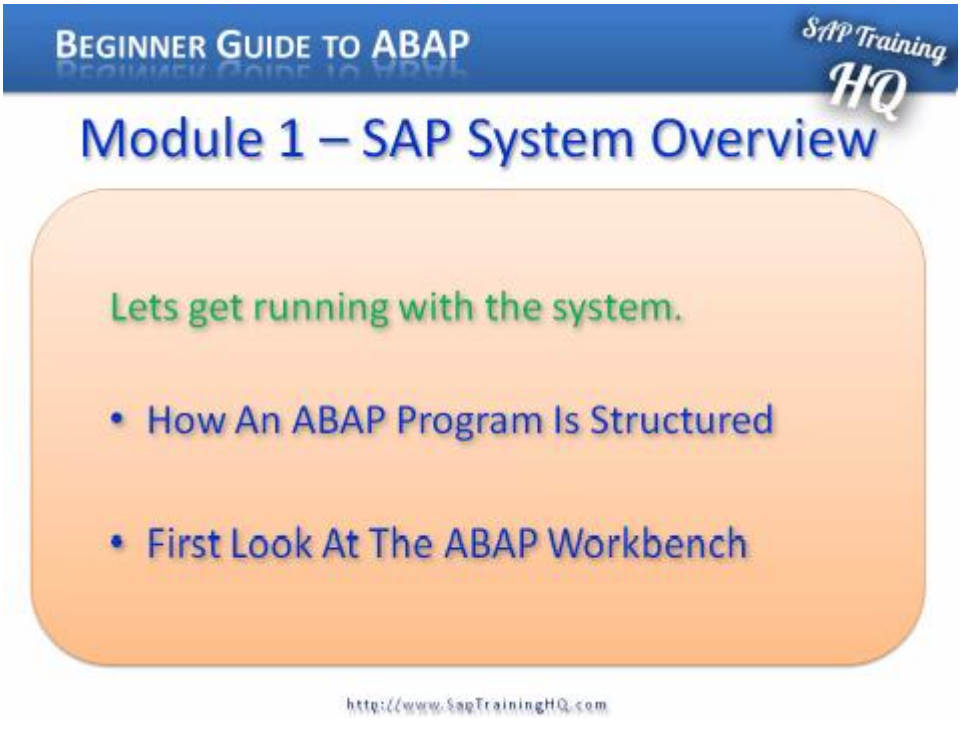
The Database Interface

The Database Interface is the third important element. It is a set of ABAP statements that are Database independent. What this means is that a set of ABAP statements can be used that, in turn, can communicate with any type of Database that has been installed when the system was set up. Whether this is, for example, a Microsoft SQL server or an Oracle Database, you can use the same ABAP statements, called Open SQL, to control the entire Database reading and writing over the Database Interface. The great advantage of this is that the ABAP statements have encapsulation, meaning the programmers do not need to know which physical Database system the ABAP system they are using actually supports.

There are times when you may want to use a specific SQL statement native to the database which is installed. ABAP is designed in such a way that if this type of coding is necessary, this facility is available. It is possible to directly access the Database through the programs using native SQL statements, but this is not encouraged. Normally, when systems are set up, the system administrator will forbid these practices, due to the security and stability risks to the system which may be introduced. If you are going to be programming ABAP, make sure Open SQL is used, because then anyone subsequently looking at the programs will understand what is trying to be achieved.

First look at the ABAP Workbench

It is now time to take a first look at an SAP ABAP program. The following section will look at the SAP System and introduce the ABAP Workbench. But before doing so, let's take a look at the structure of an ABAP program.



Like many other programming languages, ABAP programs are normally structured into two parts.

The first is what is considered to be the Declaration section. This is where you define the data types, structures, tables, work area variables and the individual fields to be used inside the programs. This is also where you would declare global variables that will be available throughout the individual subsections of the program. When creating an ABAP program, you do not only declare global variables, but you also have the option to declare

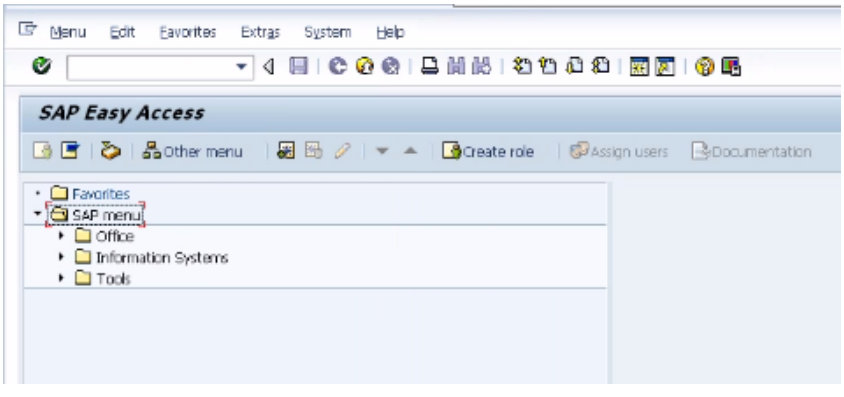
variables that are only valid within specific sections inside the programs. These sections are commonly referred to as internal Processing Blocks.

The Declaration part of the program is where you define the parameters used for the selection screens for the reports. Once you have declared tables, global variables and data types in the Declaration section of the program, then comes the second part of the ABAP program, where all of the logic for the program will be written. This part of an ABAP program is often split up into what are called Processing Blocks.

The Processing Blocks defined within programs can be called from the Dynpro processor, which were discussed previously, depending on the specific rules created within the program. These Processing Blocks are almost always just small sections of programming logic which allow the code to be encapsulated.

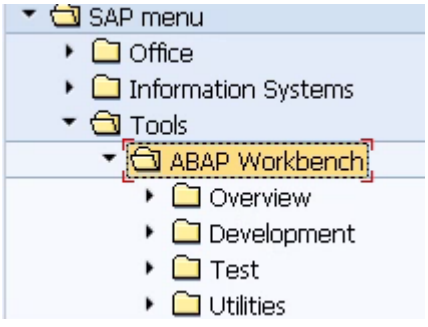
First Look

When logged into an SAP system it will look something similar to the image below.

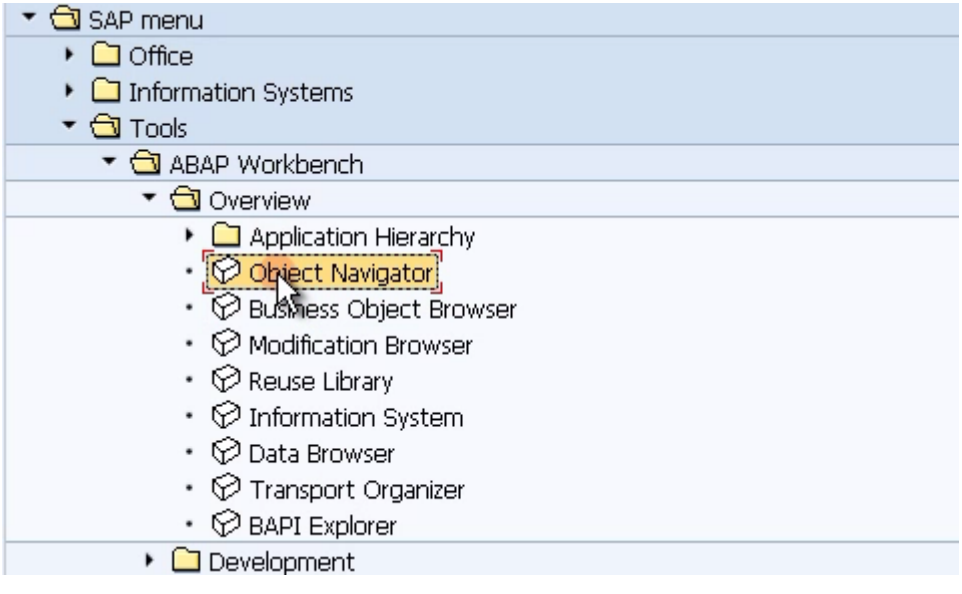


The way the SAP GUI looks may vary, the menu to the side may be different, but here the display show a minimal menu tree which will be used throughout this book.

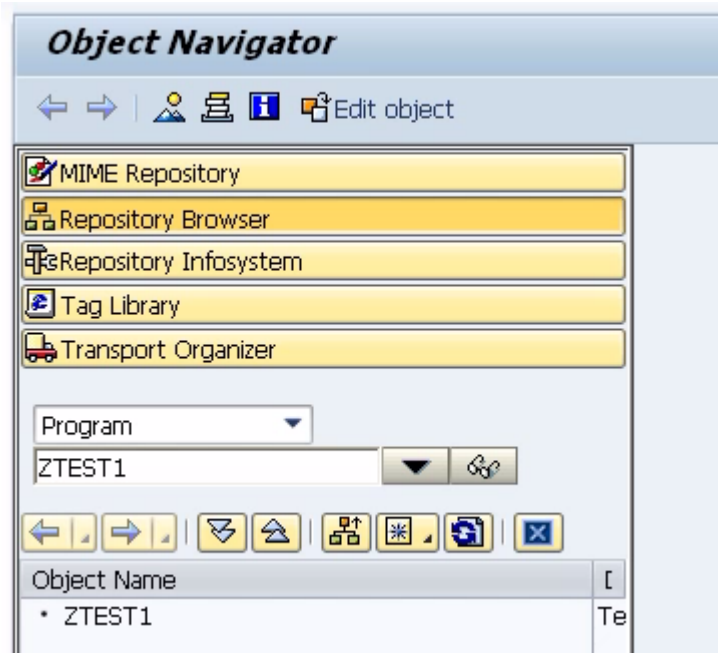
The first thing to do here is look at the ABAP Workbench. To access this, you use the menu on the left hand side. Open the SAP menu, choose Tools and open the ABAP Workbench, where there will be four different options.



The first thing to look at is a quick overview of how to run a transaction in SAP. There are two ways to do this. Firstly, if the overview folder is opened, any item which does not look like a folder itself, is a transaction which can be run. In this instance, we can see the Object Navigator:

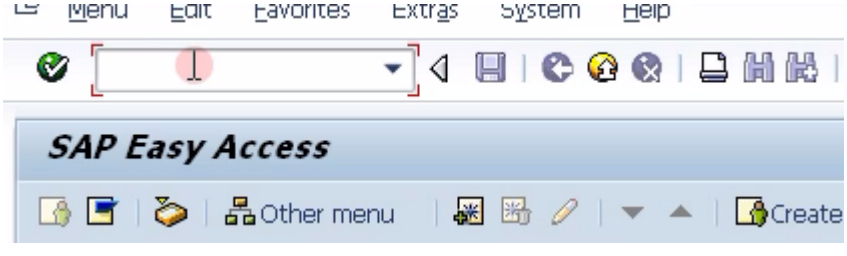


Double click this, and the transaction will open:

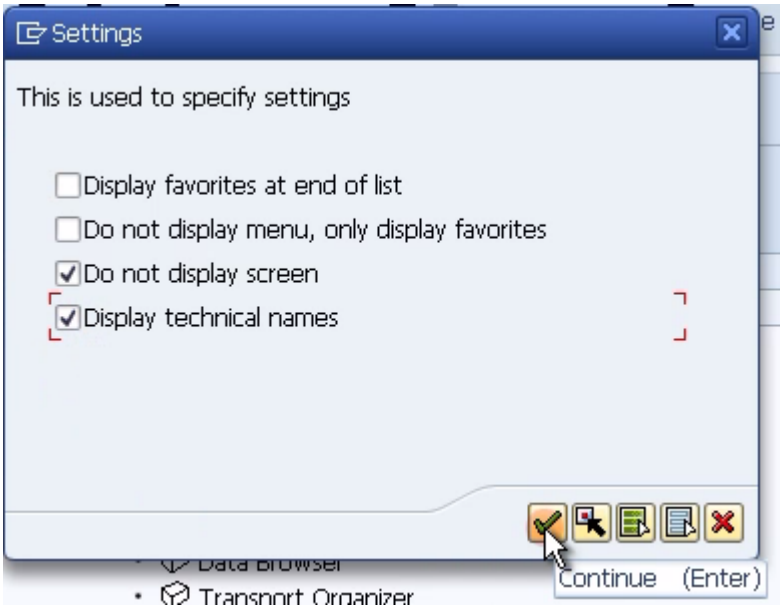


To exit out of the transaction, click the Back button:

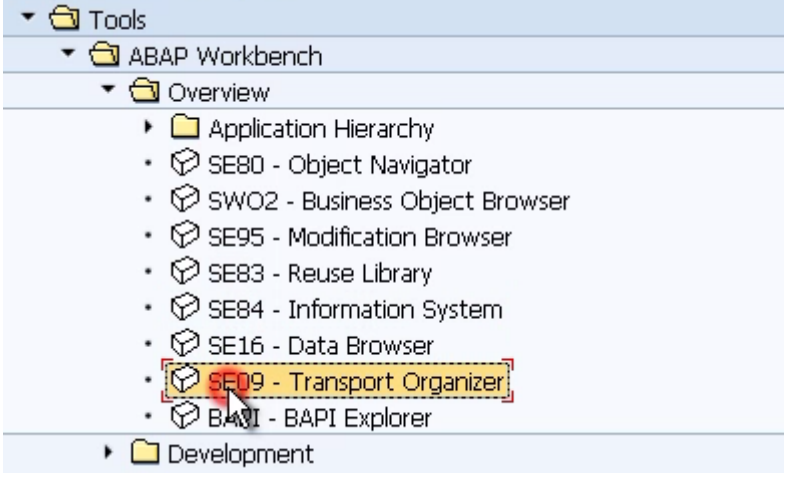
The second way of running a transaction is to enter the transaction code into the transaction code input area:



A useful tip to become familiar with the names of transactions is to look at the Extras menu --> open Settings and in the dialogue box which appears, select the option 'Display technical names' and click the 'Continue' icon:



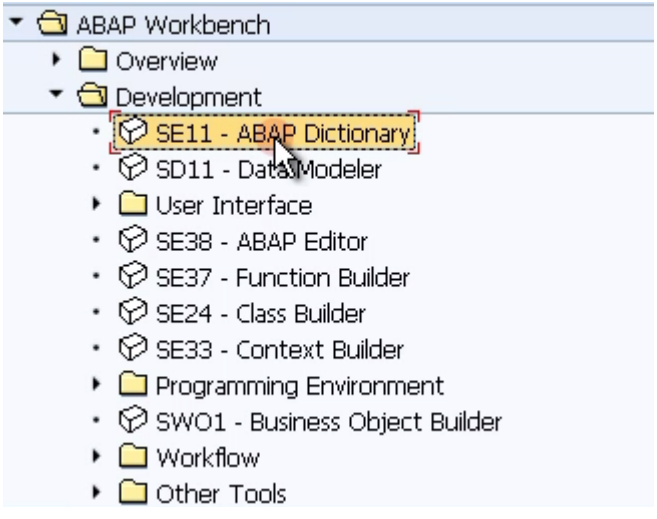
The menu tree will be refreshed, and when the 'Overview' folder is opened, the transaction codes will be made visible. It is now possible to become familiar with them, and enter them directly into the transaction code input area:



Now, a step-by-step look will be taken through the major transactions of the ABAP Workbench to become familiar with, and use, as an ABAP developer.

ABAP Dictionary

One thing most programs will have in common is that they will read and write data to and from the Database tables within the SAP system. The ABAP Workbench has a transaction to allow the creation of Database tables, view the fields which make up these tables and browse the data inside. This is called the ABAP Dictionary. The ABAP Dictionary can be found by expanding the ABAP Workbench menu tree --> 'Development'. The transaction code to run the ABAP dictionary directly is SE11:



ABAP Editor

The next and probably most commonly used part of the ABAP Workbench is the ABAP Editor, which much of this course will focus upon. The ABAP Editor is where all of the code is created, the logic built and, by using forward navigation (a function within an SAP system which will be discussed later), function modules defined, screens created and so on. The ABAP Editor can be found under the 'Development' menu, as shown above and with transaction code SE38.

Function Builder

The next important part of the Workbench is the Function Builder, which is similar to the ABAP Editor. Its main function is to define specific tasks that can be called from any other program. Interfaces are created in the Function Builder, where the different data elements and different types of tables are defined, that can be passed to and from the Function which is built. The Function Builder will be discussed a little later on, when the programs created are encapsulated into function modules. The Function Builder can be

called with transaction code SE37.

Menu Painter

The next item to look at here is called the Menu Painter, which can be found in the 'User Interface' folder inside the 'Development' menu, or with transaction code SE41. This is a tool which can be used to generate menu options, buttons, icons, menu bars, transaction input fields, all of which can trigger events within the program. You can define whether events are triggered using a mouse click, or with a keyboard-based shortcut. For example, in the top menu bar here, the 'Log off' button can be seen, which can be triggered by using (Shift + F3):

Screen Painter

While the Menu Painter is used for building menu items, menu bars and so on, the next item on the list is the Screen Painter, transaction code SE51, which allows you to define the user input screen, meaning that you can define text boxes, drop-down menus, list boxes, input fields, tabbed areas of the screen and so on. It allows you to define the whole interface which the user will eventually use, and behind the initial elements that are put on the screen, you can also define the individual functions which are called when the user interacts with them.

Object Navigator

The last item to look at here is the Object Navigator, a tool which brings together all the previous tools, providing a highly efficient environment in which to develop programs. When building large programs, with many function modules, many screens, the Object Navigator is the ideal tool to use to navigate around the development. It can be found in the 'Overview' menu of the ABAP Workbench, with transaction code SE80.

These are the main features of the ABAP Workbench interacted with during this course. In the SAP menu tree, there are evidently many more transactions which can be used to help develop programs, but these cover the vast majority of development tools which will be used.

Chapter 2: Data Dictionary

Introduction

This chapter will focus specifically on the Data Dictionary. This is the main tool used to look at, understand and enhance the Database and Database tables which are used by the SAP system. You can view standard tables delivered by SAP using this tool, create new tables and enhance the existing tables delivered by SAP with new fields. There are many other features involved in the Data Dictionary, but the focus here will be on the basic ones so as to build on this later on when creating ABAP programs.

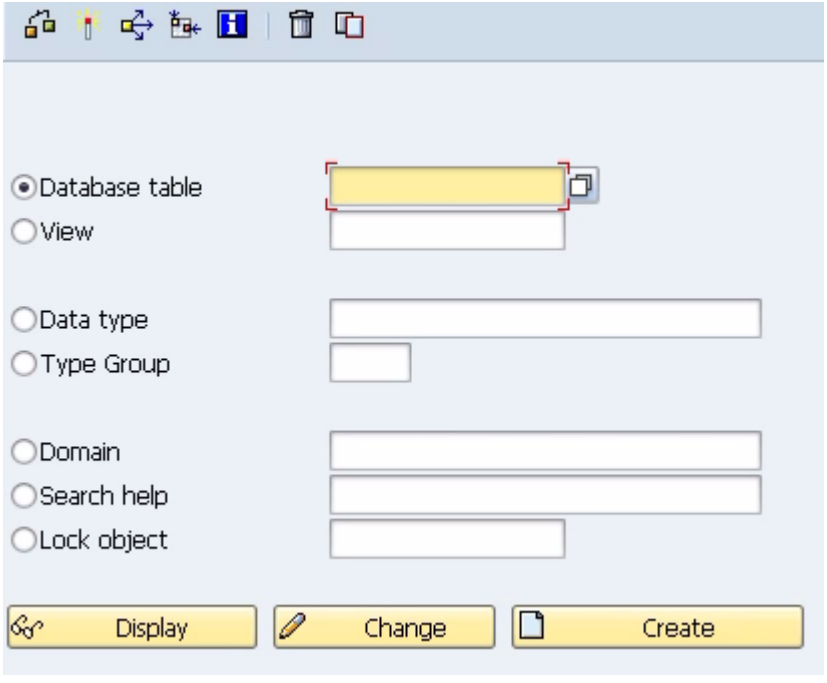
First, a database table will be created, involving the creation of fields, data elements and domains. An explanation of what each of these is, and why they are necessary to the tables built will be given. During the building of the tables, the tools used to check for errors will be shown. Once these errors are eradicated, the tables can be activated so that they can be used within the system.

After this, a look will be taken at maintaining the technical settings of the table created, which will allow the entry of data, before finally looking at the data which has been entered using standard SAP transactions available in the SAP system.

Creating a Table

With the SAP GUI open, you will be able find the Data Dictionary in the SAP menu tree. This is done via the Tools menu. Open the ABAP Workbench and click the 'Development' folder, where the ABAP Dictionary can be found and double-clicked. Alternatively, use the transaction code SE11:

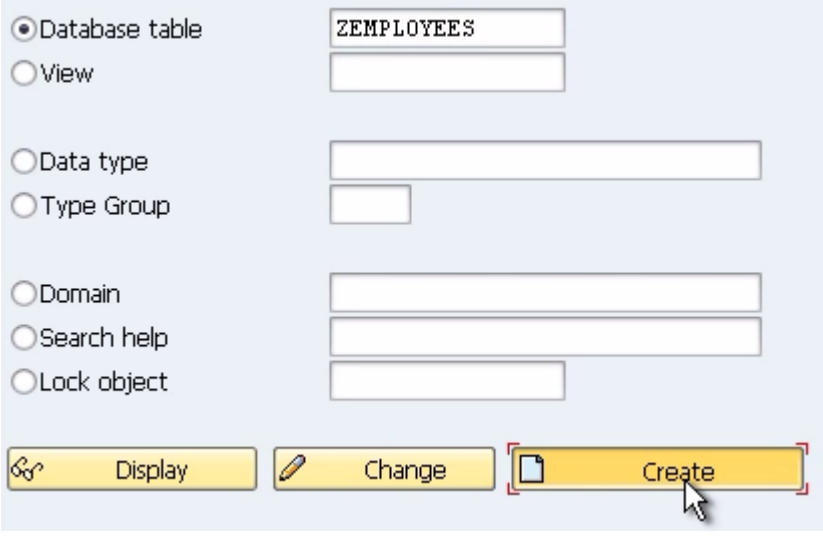
Now, the initial screen of the ABAP Dictionary will appear:

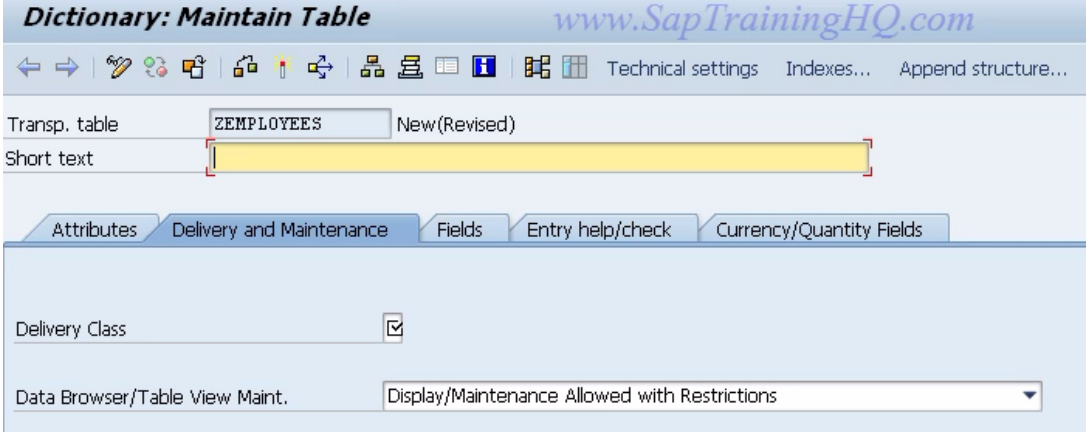


To create a table, select the 'Database table' option. In this exercise a transparent table will be created. Other types of table do exist (such a cluster tables and pool tables), but at this early stage the transparent table variety is the important one to focus upon.

The table name must adhere to the customer-defined name space, meaning that the name must begin with the letter Z or Y, most commonly this will be Z. In this example, the table will show a list of employees within a company, so, in the 'Database table' area, type 'ZEMPLOYEES' and click the 'Create' button.

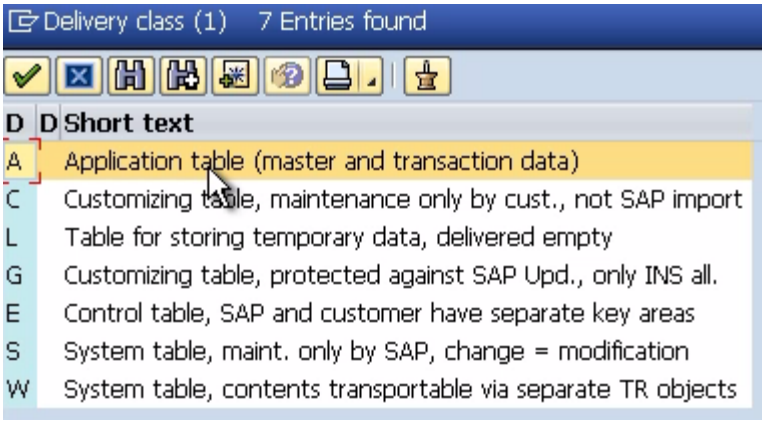
Once this is done, a new screen will appear:





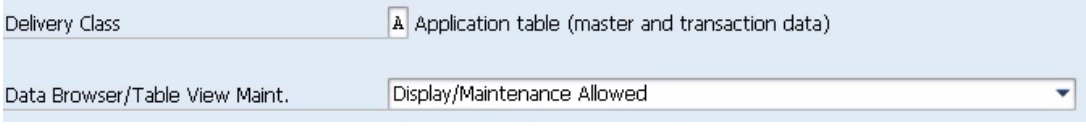
In the 'Short text' field, a description for the table must be included, enter 'Employees':

In the 'Delivery and Maintenance' tab (which opens by default), look at the 'Delivery class' section, select the field and then click the drop-down button, where a list of Delivery classes will be shown and selected:



For the table being created here, choose 'Application table', as the data held in the table fits the description 'master and transaction data'.

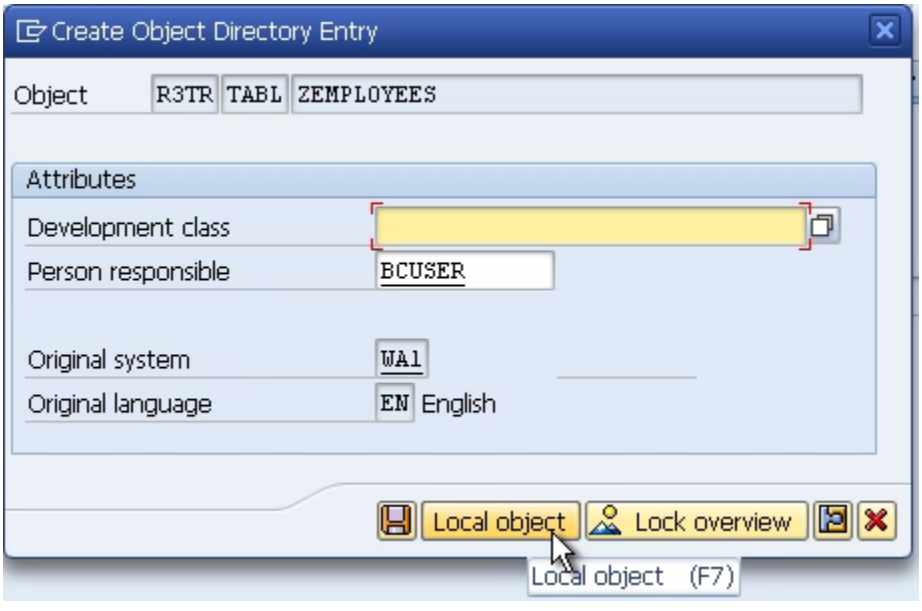
In the field below this, labelled 'Data Browser/Table View Maint.', choose the 'Display/Maintenance allowed' option, which will allow for data entry directly into the table later on. It should look like this:



Before going any further, click the 'Save' button:

A window appears titled 'Create Object Directory Entry'.

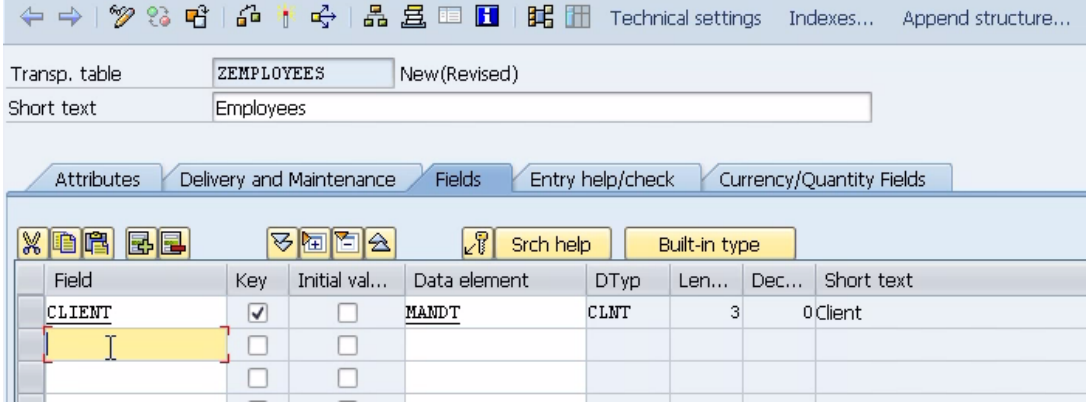
Nearly all development work done with SAP is usually done within a development environment, before being moved on to, for example, a quality assurance environment and on further to production. This window allows you to choose the appropriate Development class which is supported by other systems where the work may be moved on to. In this example scenario, though, developments will not be moved on to another system, so click **'Local object'**, so as to indicate to the system (via the phrase '$TMP' which appears) that the object is only to exist within the development system and not to be transported elsewhere. Once this is done, the status bar at the bottom will show that the object has been saved:



To check everything has worked as we want, select the 'Go to' menu and selects the 'Object directory entry' option, a similar pop-up box to the previous one will appear, where the 'Development class' field will show '$TMP', confirming this has been done correctly.

Creating Fields

The next step is to begin creating Field names for the table, in the 'Fields' tab:



Fields, unlike the name of the table, can begin with any letter of the alphabet, not just Z and Y and can contain up to 16 characters.

Tables must include at least one Key field, which is used later for the searching and sorting of data, and to identify each record as being unique.

An Initial value can be assigned to each field, for example, in the case of a field called *Employee Class* you could say the majority of employees are Regular Staff ('S'), but some are Directors, with a code of 'D'. The standard initial value would be 'S', but the user could change some of these to a 'D' later on.

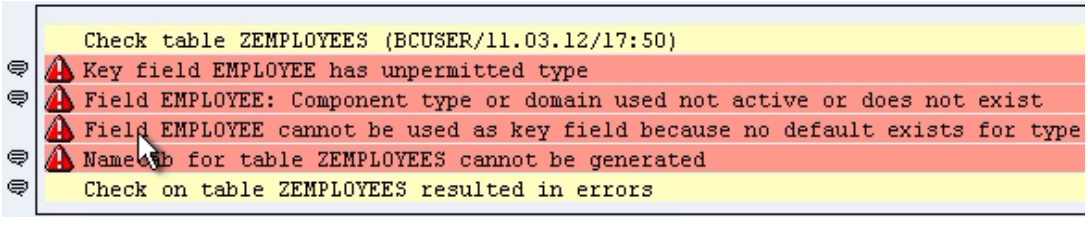
Data Elements

Every Field in the table is made up of what is called a Data Element, which defines specific attributes of each field.

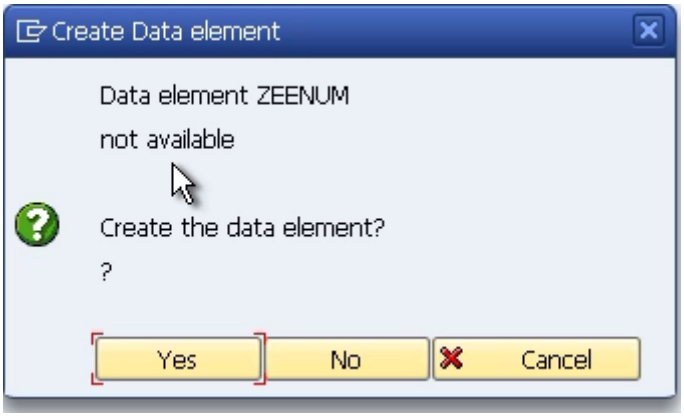
The first Field to be created here is an important one within an SAP system, and identifies the client which the records are associated with. In the Field name, enter 'Client', and in the Data Element, type 'MANDT'. This Data Element already exists in the system, and after entering it, the system automatically fills in the Data Type, the Length, Number of Decimals and Short text for the Data Element itself. Ensure that the 'Client' field is made a Key field in the table by checking the 'Key' box.

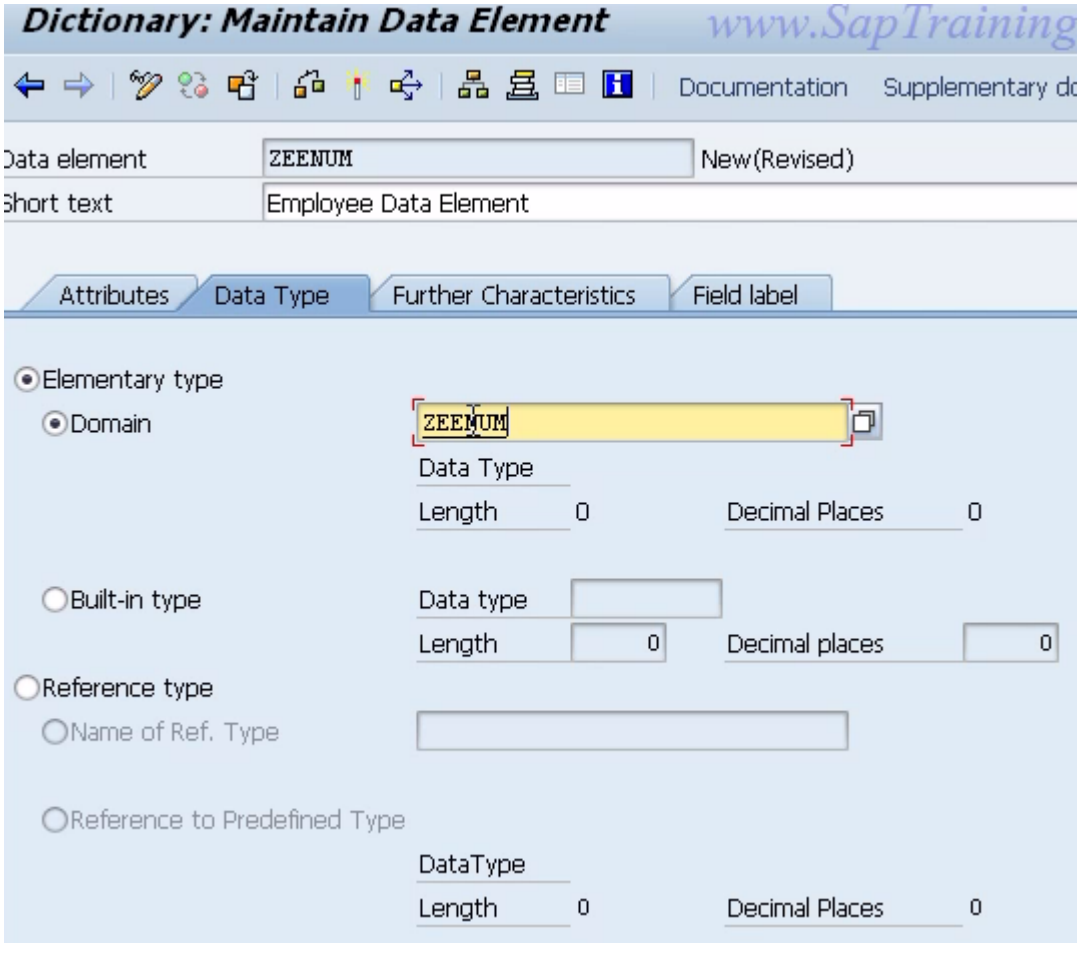
The next field will be called 'Employee'. Again, check the box to make this a Key field, and enter the new Data Element 'ZEENUM' (Data Elements broadly must adhere to the customer name space by beginning with Z or Y). Once this is done, click the save button.

Next, because the Data Element 'ZEENUM' does not yet exist, it must be created. If you try to activate or even check the table (via the 'Check' button), an error message is displayed:



Until the Data Element 'ZEENUM' is created, it cannot be used within the system. To do this, forward navigation is used. Double-click the new Data Element, and a window labelled 'Create Data element' appears. Answer 'Yes' to this, and the 'Maintain Data Element' window comes up.


Create Data element
Data element ZEENUM
not available
Create the data element?
?
Yes
No
Cancel

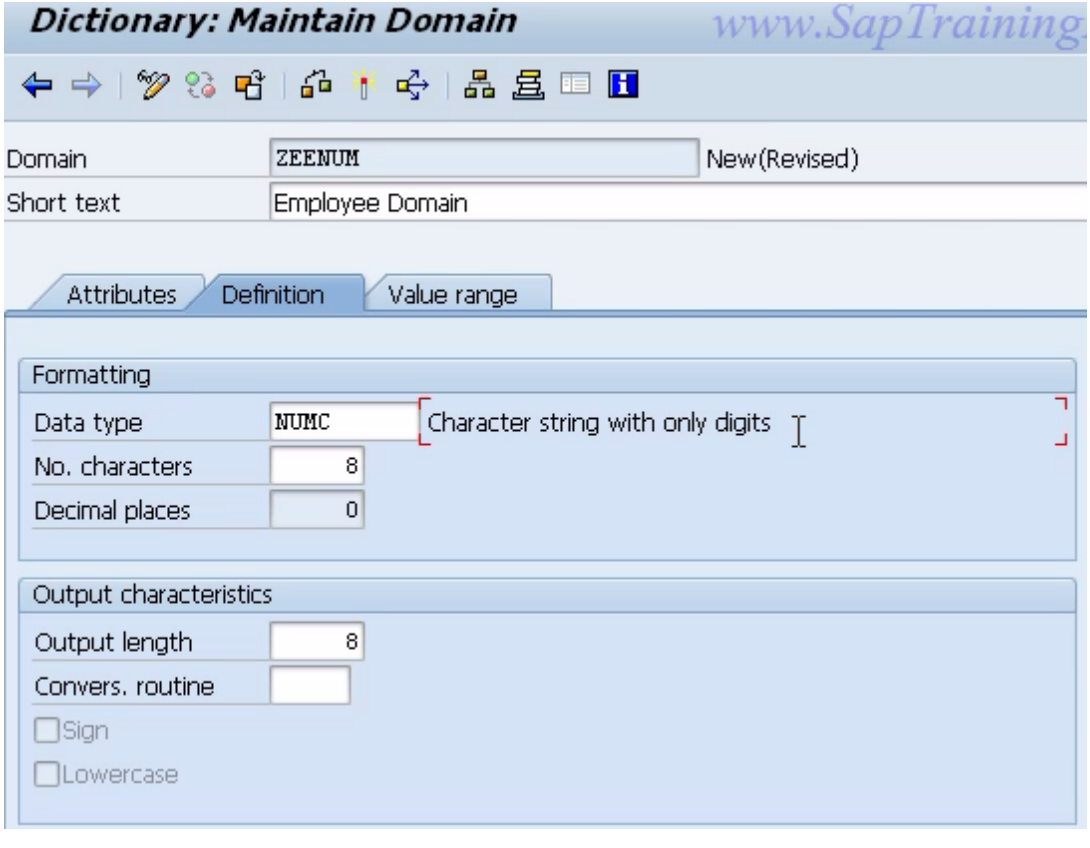

Dictionary: Maintain Data Element
www.SapTraining
Documentation
Supplementary do
Data element
ZEENUM
New(Revised)
Short text
Employee Data Element
Attributes
Data Type
Further Characteristics
Field label
Elementary type
Domain
ZEENUM
Data Type
Length
0
Decimal Places
0
Built-in type
Data type
Length
0
Decimal places
0
Reference type
Name of Ref. Type
Reference to Predefined Type
DataType
Length
0
Decimal Places
0

In the 'Short text' area, enter 'Employee Data Element'. Next, the Elementary data type, called the 'Domain', must be defined for the new Data element. Domains must adhere to the customer name space, so in this instance the same name as the Data element will be given: 'ZEENUM', (though giving both the same name is not imperative). Again, forward navigation will be used to create the Domain.

Data Domains

Double-click the entry ('ZEENUM') in the Domain area, and agree to save the changes made. Now, the 'Create Object Directory Entry' window will re-appear and again it is important to save this development to the '$TMP' development class, via the 'save' or 'local object' button visible in this window.

After doing this, a window will appear stating that the new Domain 'ZEENUM' does not exist. Choose 'Yes' to create the Domain, and in the window which appears, type into the 'Short text' box a description of the Domain. In this example, 'Employee Domain':



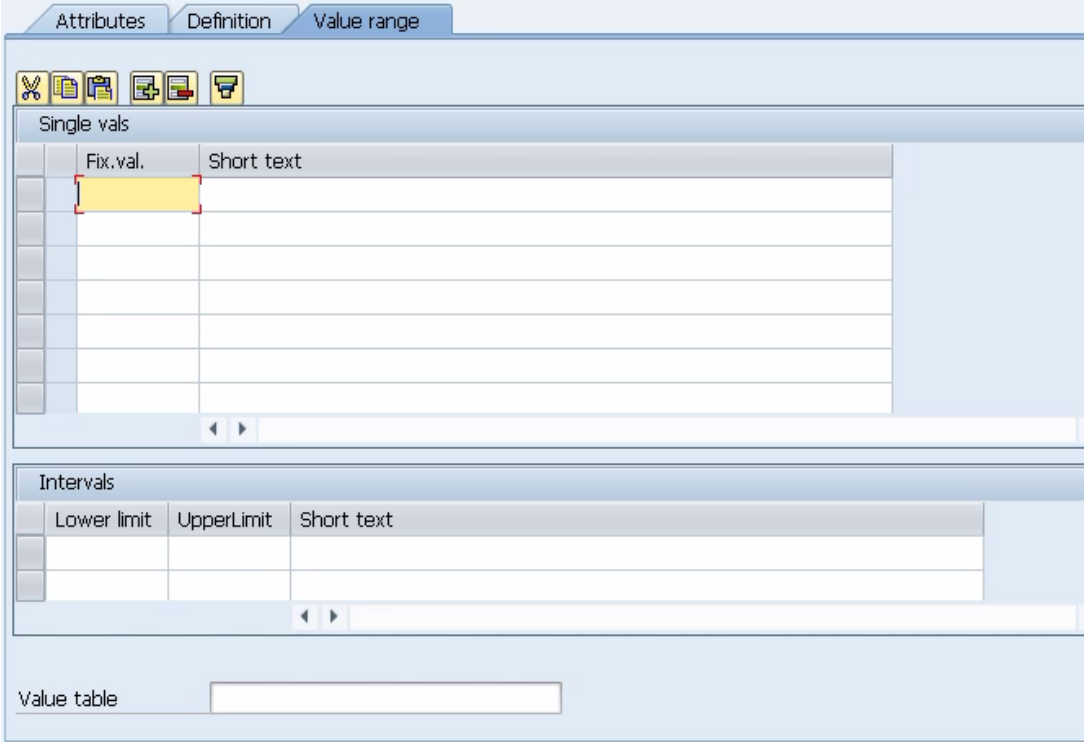
The 'Definition' tab, which, as shown above, opens automatically. The first available field here is 'Data type', click inside the box and select the drop-down menu, and a number of generic data types already existing within the ABAP dictionary will appear.

The 'NUMC' type is the one to be used here for the Employee data, a “character string with only digits”. Once this selection is double-clicked, it will appear in the 'Data type' area in the 'Definition' tab.

Next, in the 'No. characters' field, enter the number 8, indicating that the field will contain a maximum of 8 characters, and in the 'Decimal places' area, enter 0. An Output length of 8 should be selected, and then press Enter.

The 'NUMC' field's description should re-appear, confirming that this is a valid entry.

Next, select the 'Value range' tab, which is visible next to the 'Definition' tab just used:



This is where you set valid value ranges for the Domain created. Once this is set, any subsequent user entering values outside the valid value range will be shown an error

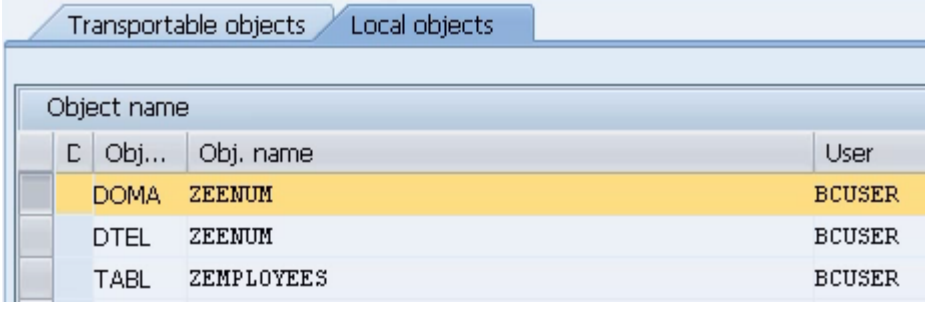
message and be requested to enter a valid entry. Here, there are three options.

- First, where you can see 'Single values', it is possible to enter a list of individual valid values which can be entered by the user.
- Second, 'Intervals', where you can enter a lower and upper limit for valid values, for example 1 and 9, which saves the effort of entering 9 individual single values in the 'Single values' section.
- Last, the 'Value table' box visible at the bottom. When there are a large number of possible entries, this is a common method (to do this you must specify a complete valid value table entry list, in which case it is also necessary to introduce foreign keys to the table, to ensure the user's entries are tested against the value stored in the value table created).

This example Domain, however, does not require any Value range entry, so just click the save button and, again, assign it as a 'Local object'.

The next step is to Activate the object, allowing other Data elements to use this domain going forward. In the toolbar click the small matchstick icon (also accessible by pressing CTRL +F3).

A pop-up window appears, listing the 3 currently inactive objects:

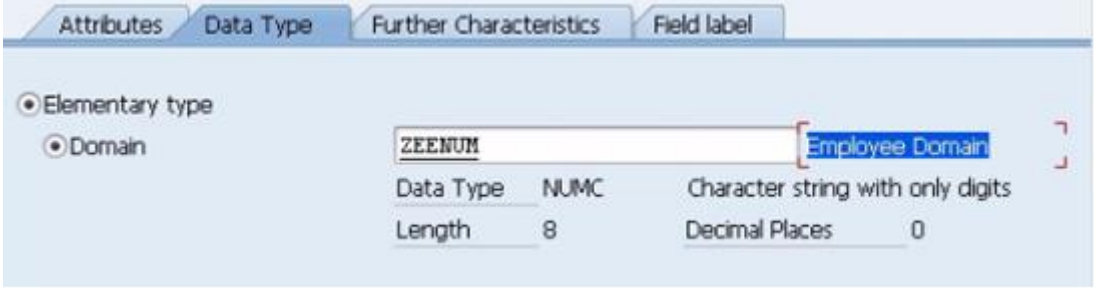


It may be possible to activate all of the objects together, but this is not advised. In a typical development environment, a number of people will be creating developments simultaneously, and quite often, others' objects will appear in this list.

At this point, it is only the Domain which is to be activated, the top entry labelled 'DOMA', with the name 'ZEENUM'. When this is highlighted, click the green tick continue button. The window should disappear, and the status bar will display the message 'Object(s)

activated'

Now it is possible to proceed with the creation of the table. Forward navigation was used for generating the Domain, so click the 'Back' button, or press F3 to return to the 'Maintain Data Element' screen. As the domain is active, the description entered previously should appear by the area where 'ZEENUM' was typed, along with other Domain properties which have been created:

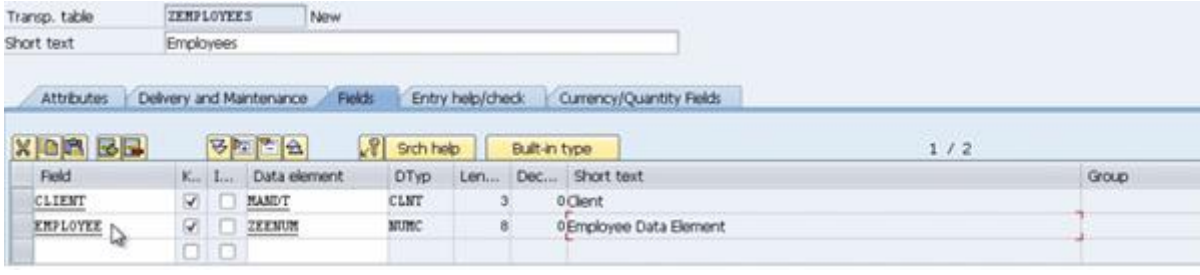


Next, the Field labels must be created, so click that tab. The Field labels entered here will appear as field labels in the final table. In this example they should read 'Employee', or better, 'Employee Number'. If this does not fit within the area given, just tailor it so that it still makes sense, for example typing 'Employee N' into the 'Short' Field label box. Once the text has been put into the Field label spaces, press enter, and the 'Length' section will automatically be filled in:

Once this is complete, Save and Activate the element via the toolbar at the top. The inactive objects window will reappear, where two inactive objects will remain. Highlight the Data element (labelled 'DTEL') and click the green tick
Continue button at the bottom.

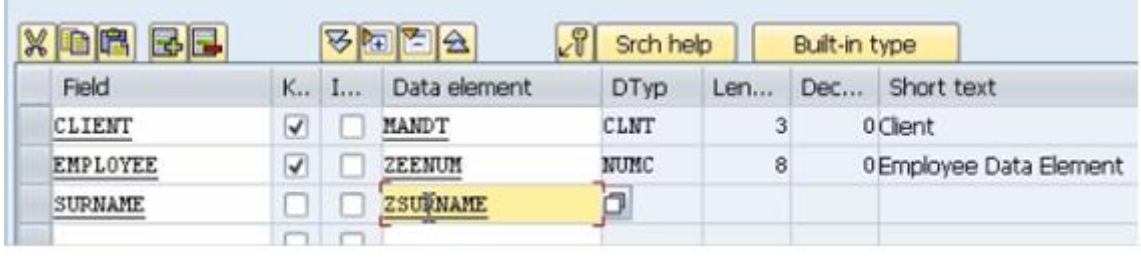
Again, the status bar should display 'Object(s) activated'.

Press the back button to return to the Table maintenance screen. Here you will now see that the 'EMPLOYEE' column has the correct Data Type, Length, Decimals and Short text, thus indicating the successful creation of a Data element and Domain being used for this Field.



Next, the same practices will be used to create four additional fields.

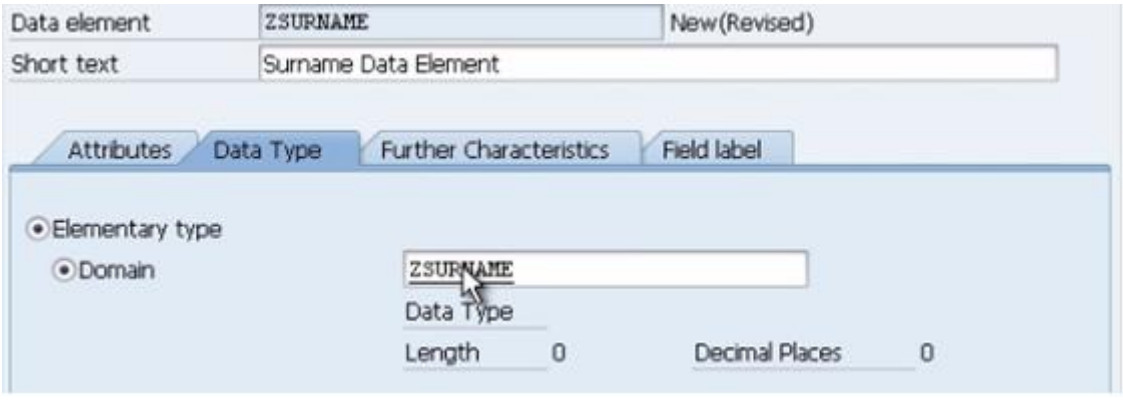
The next field to create should be titled 'SURNAME'. This time it should *not* be selected as a Key field, so do not check the box. The Data element, in this instance, is labelled 'ZSURNAME':



Now, forward navigation will again be used. Double-click ‘ZSURNAME’; choose 'Yes' to save the table and 'Yes' again to create the new Data element. The 'Maintain Data Element' window will appear which will be familiar from the previous steps.

In the 'Short text' box this time type 'Surname Data Element' and title the new domain

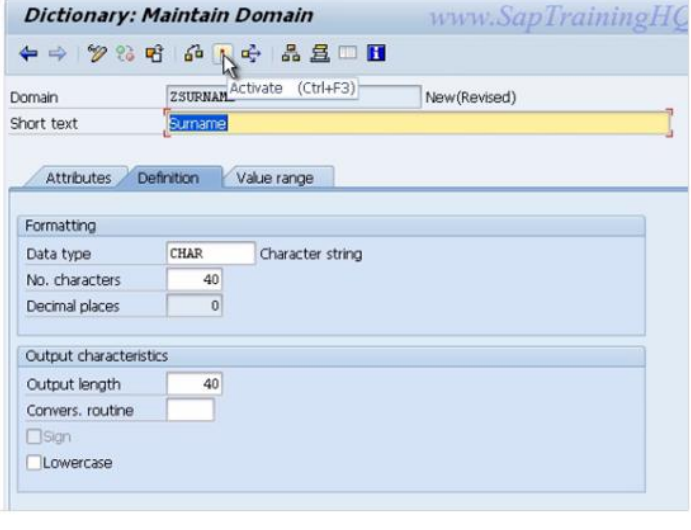
'ZSURNAME':



Double-click the new domain and save the Data element, assigning it a 'Local object' and then choose 'Yes' to create the new Domain.

The Domain maintenance screen will reappear. Enter the short text 'Surname' and, this time; the Data type to select is 'CHAR', a Character string.

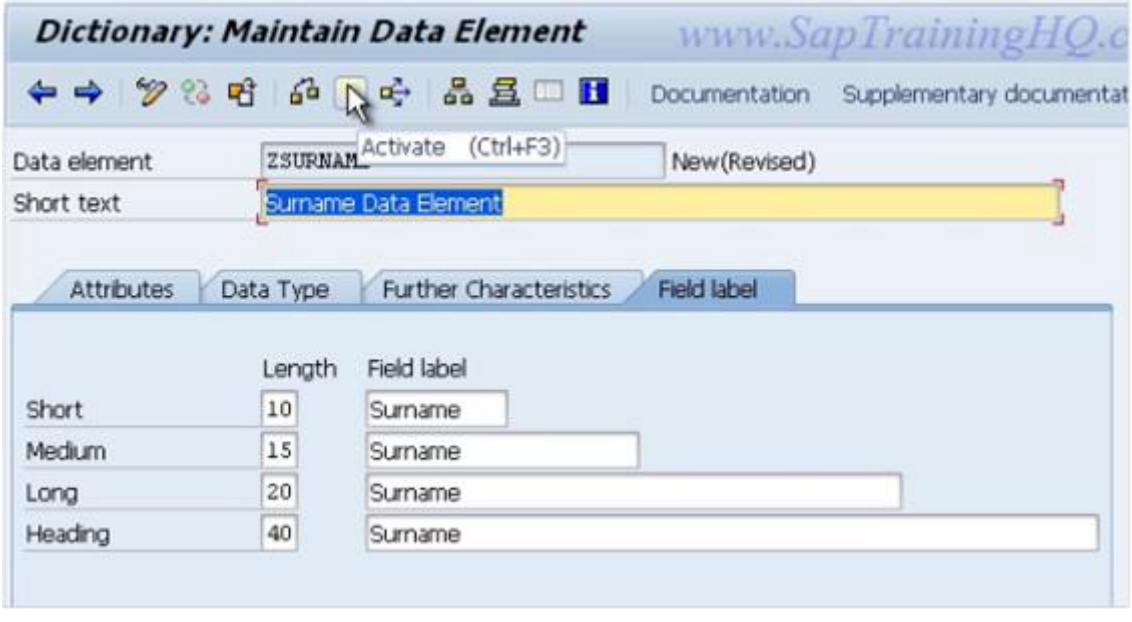
The number of characters and output length should both be set to 40, then press enter to be sure everything has worked, and click the Activate button.



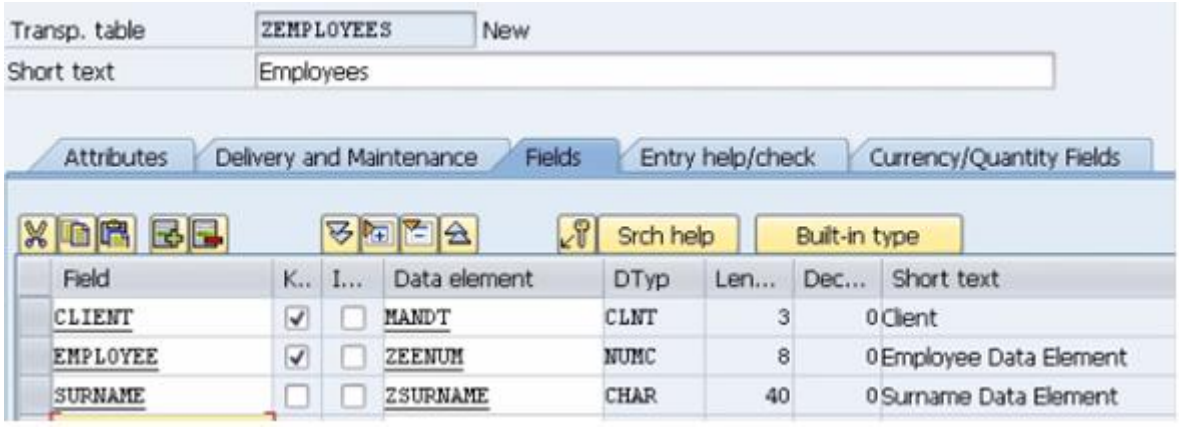
Note that the Save button has not been pressed this time, as the Activate button will also save the work automatically. Ensure you assign the object to the $TMP development class as usual.

In the Activate menu, select the object (the domain (labelled 'DOMA') named 'ZSURNAME') to be activated, and click the green tick continue button. The status bar should read 'Object saved and activated'.

Following this, click Back or F3 to return to the Maintain Data element screen. Ensure the domain attributes have appeared (Short text, Data type, Length and so on). In the Field Label tab, enter 'Surname' in each box and press Enter to automatically fill the 'Length' boxes and then activate the Data element (in the Activate menu, the 'DTEL' object named 'ZSURNAMES'), checking the status bar to ensure this has occurred with any errors:



Again, press Back to return to the Maintain Table screen, where the new Data element will be visible:



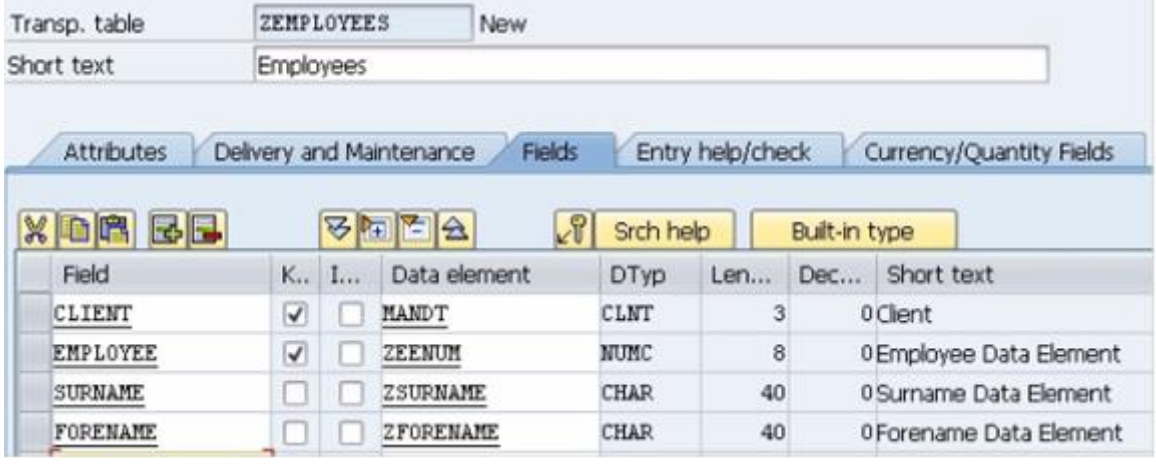
Field	K..	I...	Data element	DTyp	Len...	Dec...	Short text
CLIENT	☑	☐	MANDT	CLNT	3	0	Client
EMPLOYEE	☑	☐	ZEENUM	NUMC	8	0	Employee Data Element
SURNAME	☐	☐	ZSURNAME	CHAR	40	0	Surname Data Element

The next field to be created is titled 'FORENAME', and the data element 'ZFORENAME'. Click to create the Data element and follow the steps above again.

In the Maintain Data Element screen, the Short text should read 'Forename Data Element' and the domain 'ZFORENAME'. Save this and choose 'Yes' to create the domain.

The domain's short text should read 'Forename'. Use the CHAR data type again and a Length and Output length of 40. Next, Activate the Domain as before.

Return to the Maintain Data Element screen. Type 'Forename' into the four Field label boxes. Press *enter* to fill the length boxes and then Activate the Data Element named 'ZFORENAME' as before. Go back again to see the table:



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

The next field will be called 'Title' and the Data Element 'ZTITLE', follow the steps above again to create this field with the following information:

The Data element short text should read 'Title Data Element' and the domain should be named 'ZTITLE'.

The Domain Short text should be 'Title' and the Data type is again 'CHAR'. This time the Length and Output length will be 15.

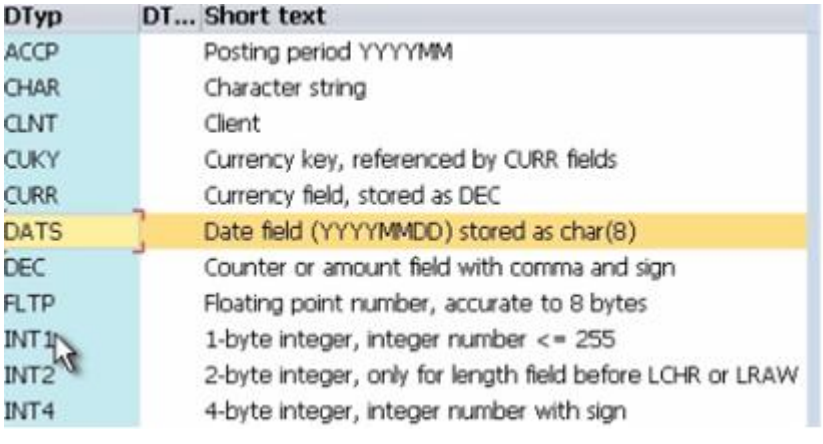
The Field labels should all read 'Title'.

Activate all of these and go back to view the new, fifth field in the Table.

The final field which will be created for this table is for Date of Birth. In the Field box type 'DOB' and create the Data element 'ZDOB' using the steps from the previous section and this information:

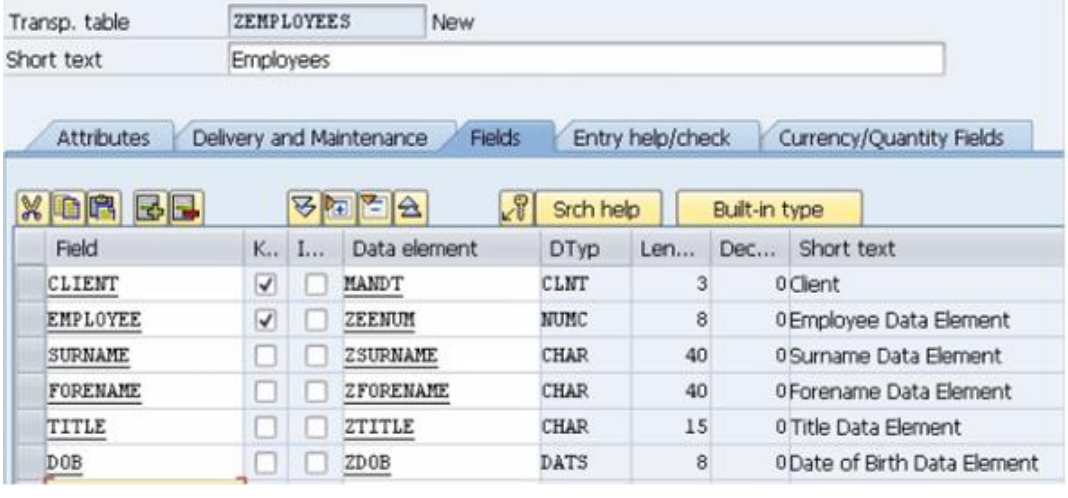
The Data element short text should read 'Date of Birth Data Element' and the domain should be named 'ZDOB'.

The Domain Short text should be 'Date of Birth' and the Data type is, this time, 'DATS', after which an information box will appear to confirm this. Click the green tick to continue:



For the DATS data type, the Length and Output lengths are set automatically at 8 and 10 (the Output length is longer as it will automatically output dividers between the day, month and year parts of the date).

The Field labels should all read 'Date of Birth', except the 'Short' label where this will not fit, so just type 'DOB' here. Activate the Domain and Data element, and return to the table.



Field	K..	I...	Data element	DTyp	Len...	Dec...	Short text
CLIENT	☑	☐	MANDT	CLNT	3	0	Client
EMPLOYEE	☑	☐	ZEENUM	NUMC	8	0	Employee Data Element
SURNAME	☐	☐	ZSURNAME	CHAR	40	0	Surname Data Element
FORENAME	☐	☐	ZFORENAME	CHAR	40	0	Forename Data Element
TITLE	☐	☐	ZTITLE	CHAR	15	0	Title Data Element
DOB	☐	☐	ZDOB	DATS	8	0	Date of Birth Data Element

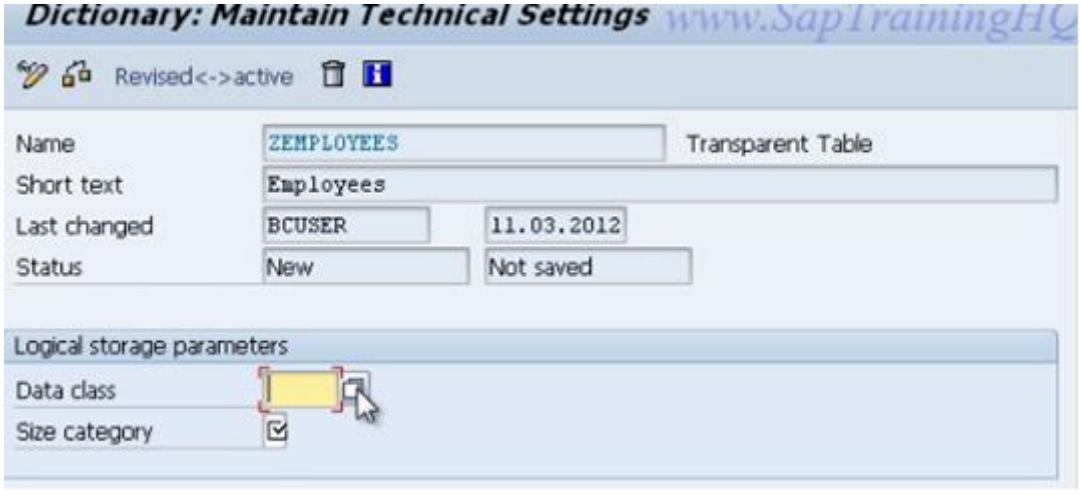
Technical Settings

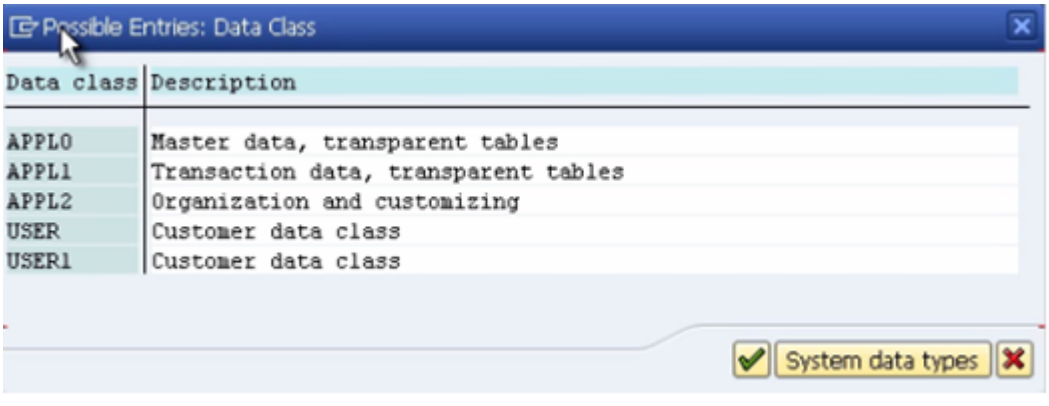
Once this has been saved, the next step is to move on to maintaining the technical settings of the Table. Before creating the final Database table, SAP will need some more information about the table being created.

Select 'Technical settings' via the toolbar above the table, through the 'Go to' menu, or

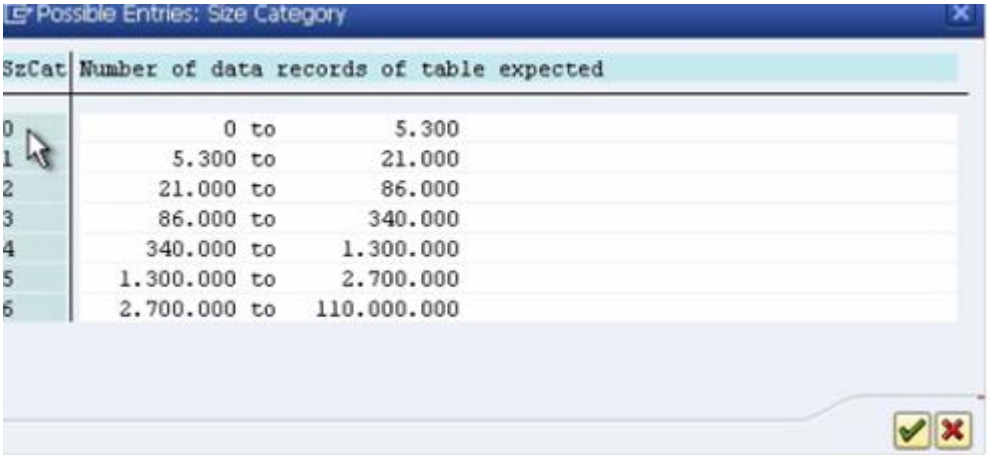
with the shortcut CTRL+SHIFT+F9.

Here, it is important to tell the system what Data class is to be used, so select the drop down button. There are five different options, with accompanying descriptions. For this table, select the first, labelled 'APPL0', and double-click it:

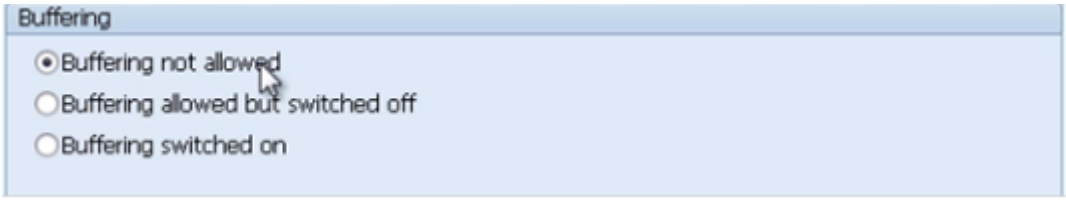




For the 'Size category' field, again click the drop-down button. Here, you have to make an estimate as to the amount of data records which will be held within the table so that the system has some idea of how to create the tables in the underlying database. In this instance, it will be a relatively small amount of information, so select the first size category, labelled 0:



Below this are the Buffering options. Here, 'Buffering not allowed' should be selected:

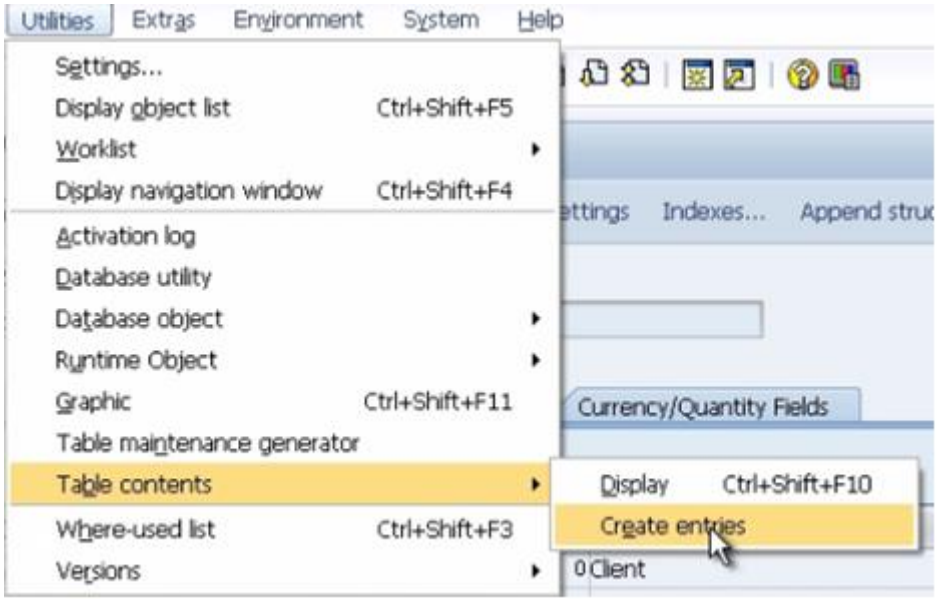


This prevents the table contents from being loaded into memory for reading, stopping the table from being read in advance of the selection of the records in the program. You may, correctly, point out that it may be advantageous to hold the table in the memory for speed efficiency, but in this example, this is not necessary. If speed was an issue in a development, buffering would then be switched on, ensuring the data is read into memory. In the case of large tables which are accessed regularly but updated infrequently, this is the option to choose.

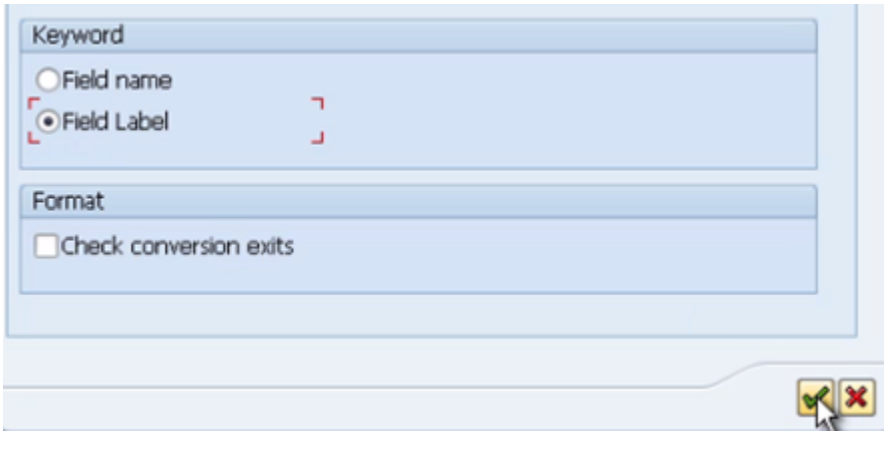
Nothing else on the 'Maintain Technical Settings' screen needs to be filled at this point, so click Save and then go back to the table itself. If all of this is successful, then the table should now be in a position to be activated and the entry of records can begin. Click the Activate icon to activate the table and check the status bar, which should again read 'Object Activated'.

Entering Records into a Table

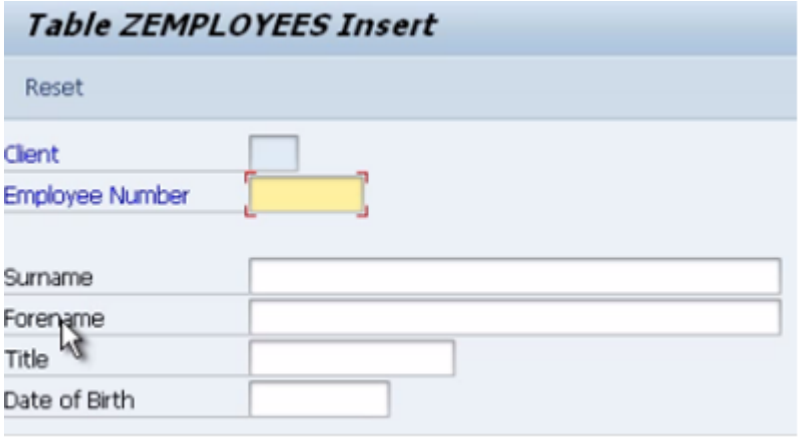
Now that the table has been created, data can be entered. To do this, enter the 'Utilities' menu, scroll to 'Table contents', and then 'Create Entries':



A Data-entry screen will appear which has automatically been generated from the table created. The field names correspond here to the technical names given when we created them. To change these to the Field labels which we set up, enter the 'Settings' menu and select 'User Parameters'. This facility allows you to tailor how tables look for your own specific user ID. Select the 'Field label' radio button and click 'Continue':

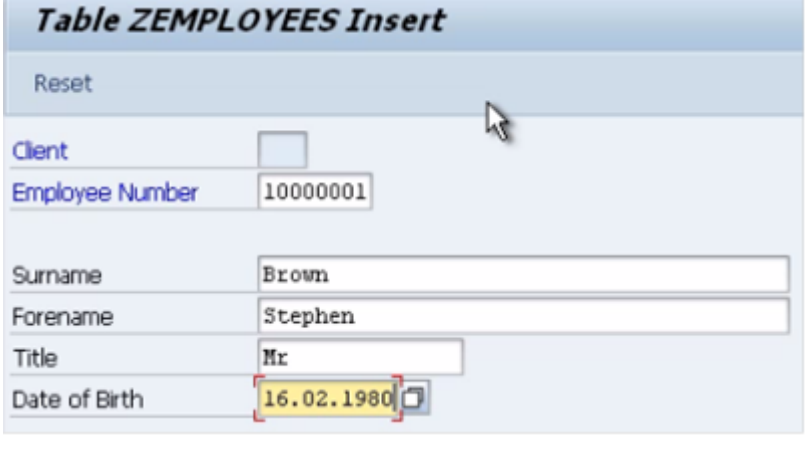


The Field labels created will now appear as they were defined when creating the table:



The Employee Number field is limited to 8 characters, and the data type was set to NUMC, so only numerical characters can be entered. Create a record with the following data:

- Employee Number: '10000001'
- Surname: 'Brown'
- Forename: 'Stephen'
- Title: 'Mr'
- Date of Birth: '16.02.1980':



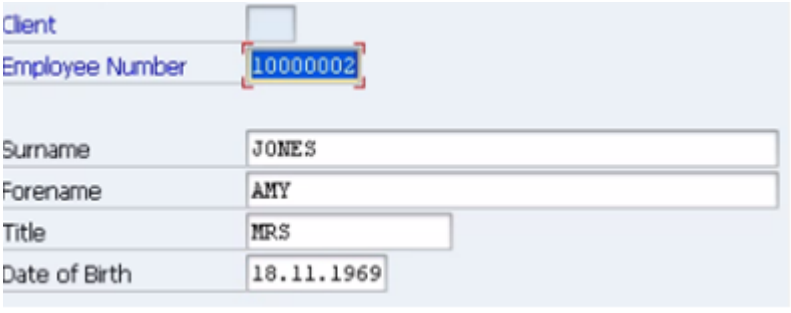
Press Enter and the system will automatically put the names in upper case, and validate each field to ensure the correct values were entered:

Click Save and the status bar should state 'Database record successfully created'. Next, click the 'Reset' button above the data entry fields to clear the fields for the next entry.

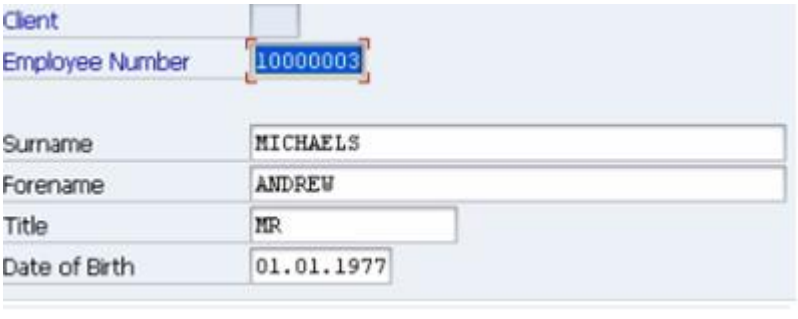
Create another record with the following data:

- Employee Number '10000002'
- Surname 'Jones'
- Forename 'Amy'
- Title 'Mrs'
- Date of Birth '181169'.

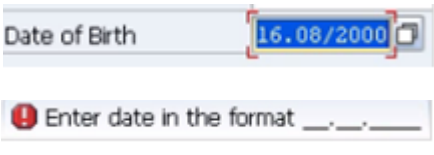
Note that this time the Date of Birth has been filled in without the appropriate dividers. When Enter is pressed, the system automatically validates all fields, correcting the Date of Birth field to the correct formatting itself:



Save, Reset, and then further records can be entered following the same steps:



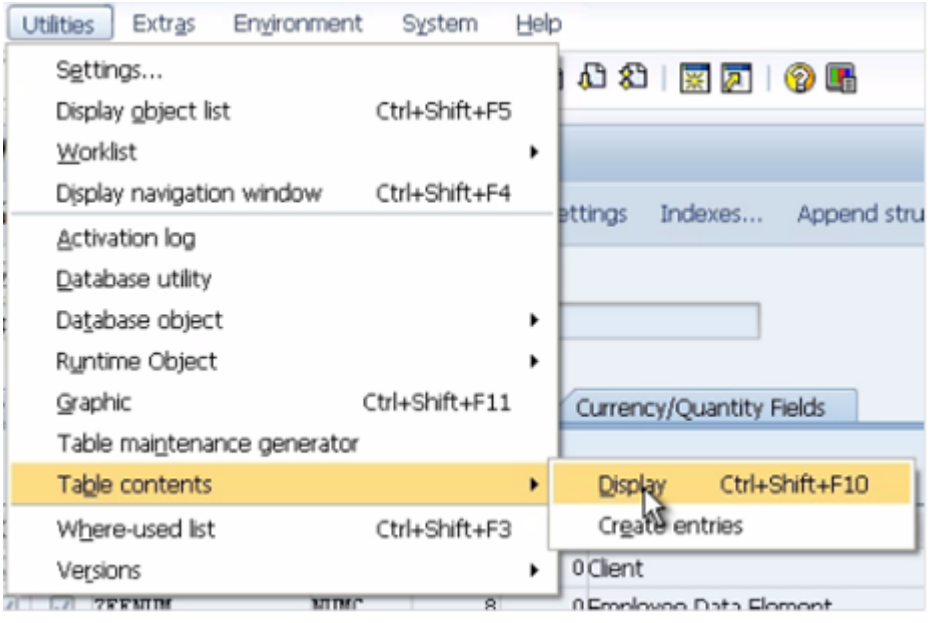
Note that if dates are entered in the wrong format, an error message will appear in the status bar:



Viewing the Data in a Table

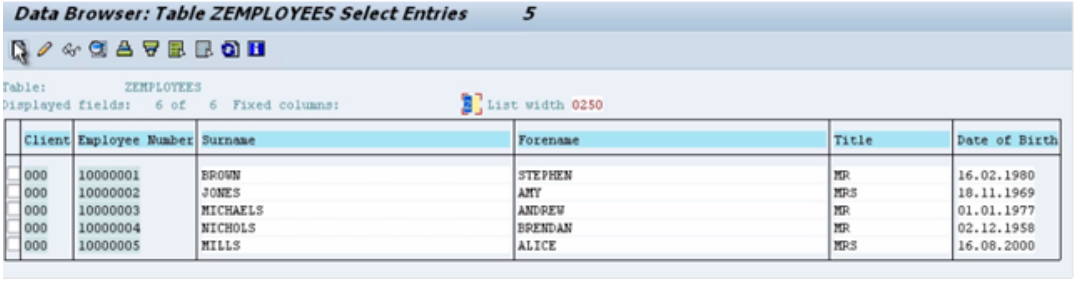
Now that data has been entered into the table, the final few steps will allow this data to be viewed.

Having entered several data records in the manner discussed previously, click the Back key to return to the 'Dictionary: Display Table' screen. To view the table created with the data entered, from the 'Utilities' menu, select 'Table contents' and then 'Display':



A selection screen will then appear, allowing you to enter or choose filter values for the fields you created. The selection screen is very useful when you have lots of data in your table. In this case, though, only five records have been entered, so this is unnecessary. However, for example if you were to only want to focus on a single employee number, or a small range, these figures can be selected from this screen:

To view all of the records, do not enter any data here. Just click the 'Execute' button, which is displayed in the top left corner of the image above, or use the shortcut F8. You will now see a screen showing the data records you entered in the previous section:

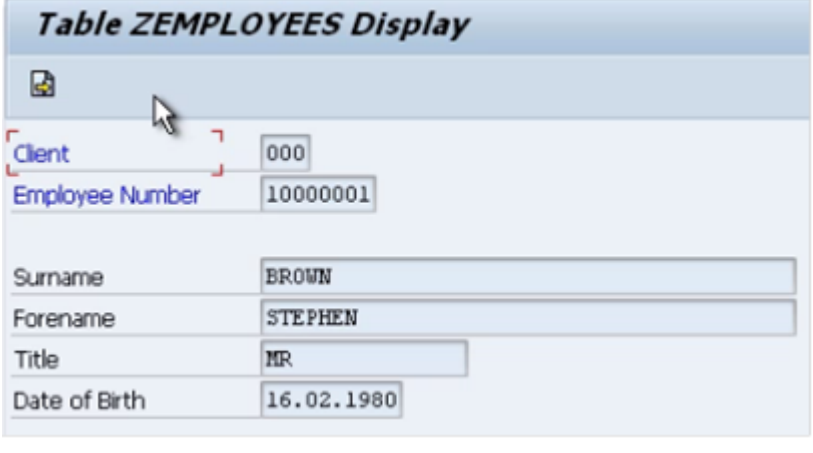


Client	Employee Number	Surname	Forename	Title	Date of Birth
000	10000001	BROWN	STEPHEN	MR	16.02.1980
000	10000002	JONES	AMY	MRS	18.11.1969
000	10000003	MICHAELS	ANDREW	MR	01.01.1977
000	10000004	NICHOLS	BRENDAN	MR	02.12.1958
000	10000005	MILLS	ALICE	MRS	16.08.2000

If further fields were to exist, the screen would scroll further to the right, meaning not all fields could be displayed simultaneously due to field size properties.

If you want to see all of the data for one record, double-click on the record and this will be shown. Alternatively, several records can be scrolled through by selecting the desired records via the check-boxes to the left of the 'Client' column and then clicking the 'Choose' icon on the toolbar: 

These can then be individually viewed and scrolled through with the 'Next entry' button:



To return to the full table then, simply click the Back button, or press F3.

Experiment with the table created, using the toolbar's range of options to filter and sort the information in a number of ways:

For example, to organise alphabetically by forename, click to select the 'Forename' field, and then click the 'Sort ascending' button: 

There are a number of things which can be achieved in this table view, and it can be a useful tool for checking the data within an SAP system without going through the transaction screens themselves.

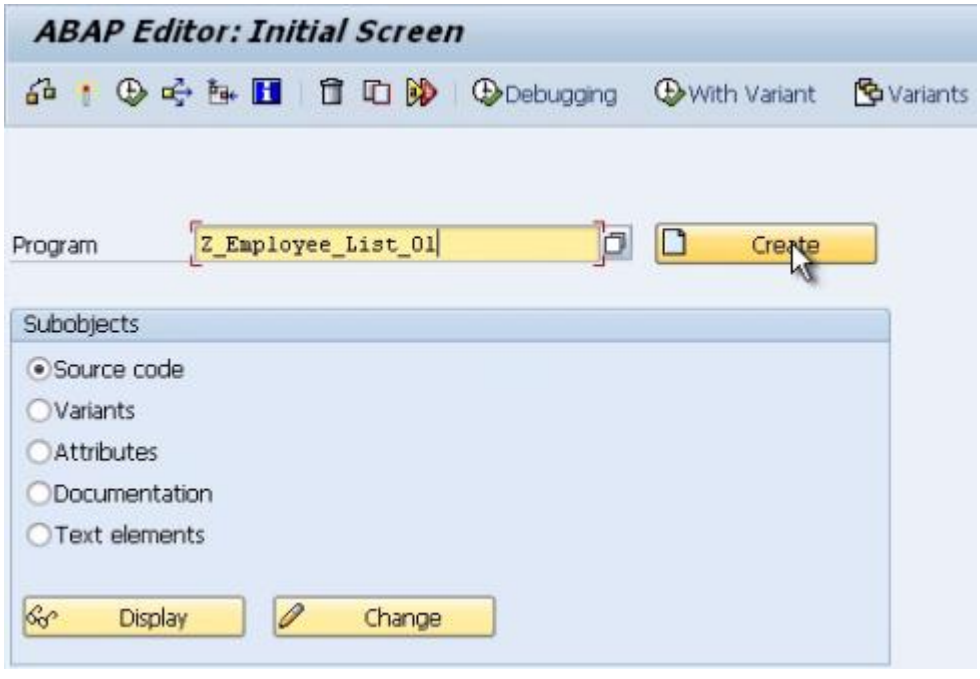
Chapter 3

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.

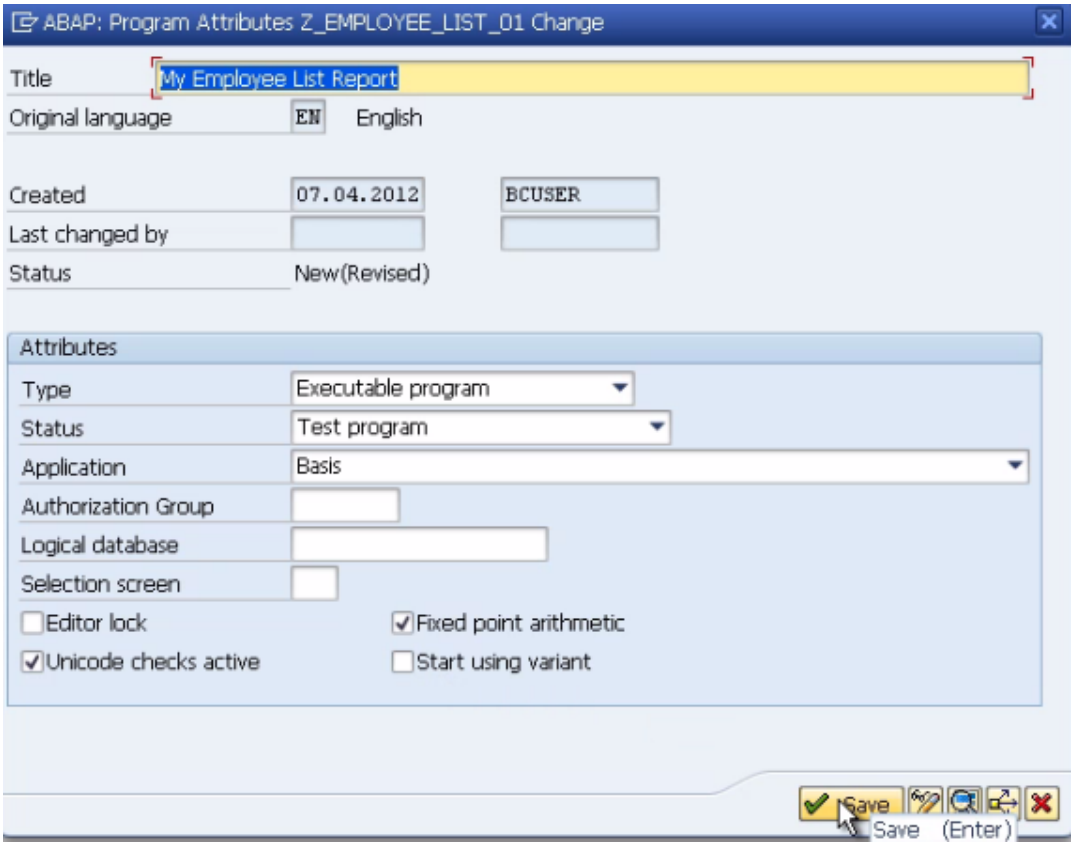
The program name must adhere to the customer naming conventions, meaning that here it must begin with the letter Z. In continuation of the example from the previous chapter, in this instance the program will be titled 'Z_Employee_List_01', which should be typed into the 'Program' field on the initial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



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.

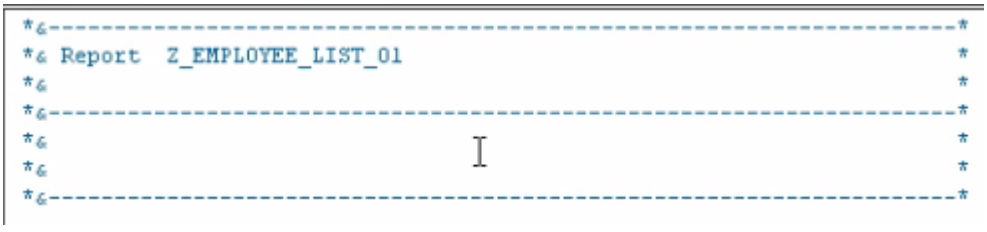
For now, the other fields below these should be left empty. Particularly ensure that the 'Editor Lock' box is left clear (selection of this will prevent the program from being edited). 'Unicode checks active' should be selected, as should 'Fixed point arithmetic' (without this, any packed-decimal fields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



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.

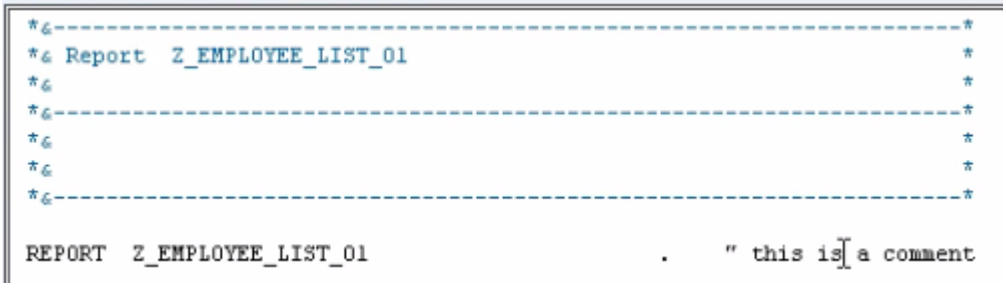
Code Editor

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.



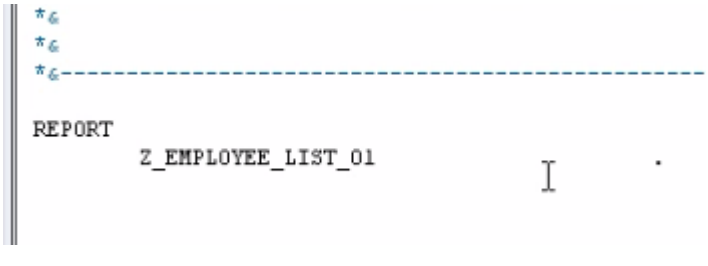
Note that the * must be in the first column on the left. If it appears in the second column or beyond, the text will cease to be a comment.

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.



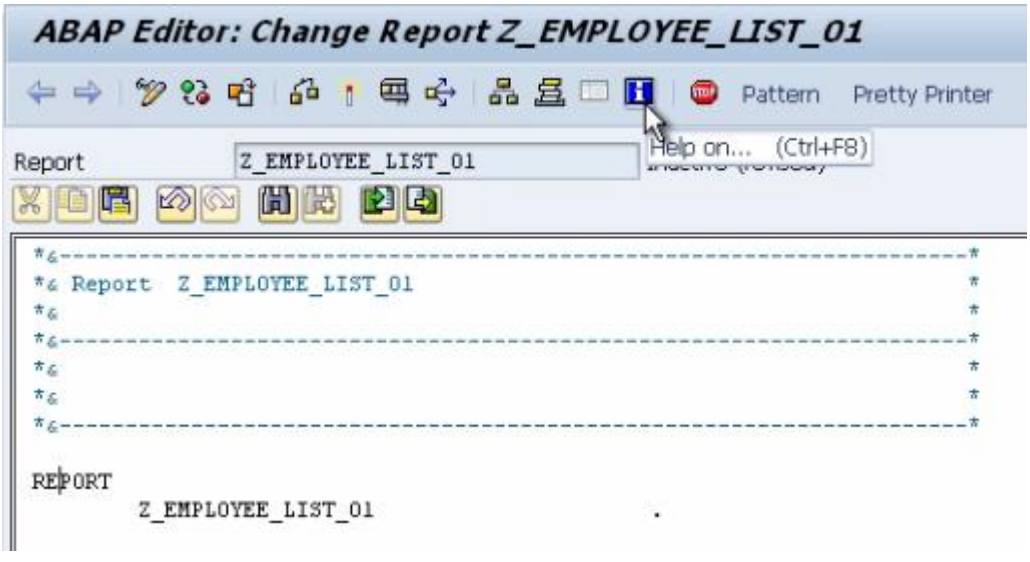
The next line of code, visible above, begins with the word REPORT. This is called a STATEMENT, and the REPORT statement will always be the first line of any executable program created. The statement is followed by the program name which was created previously. The line is then terminated with a full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

Every statement in ABAP must 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:

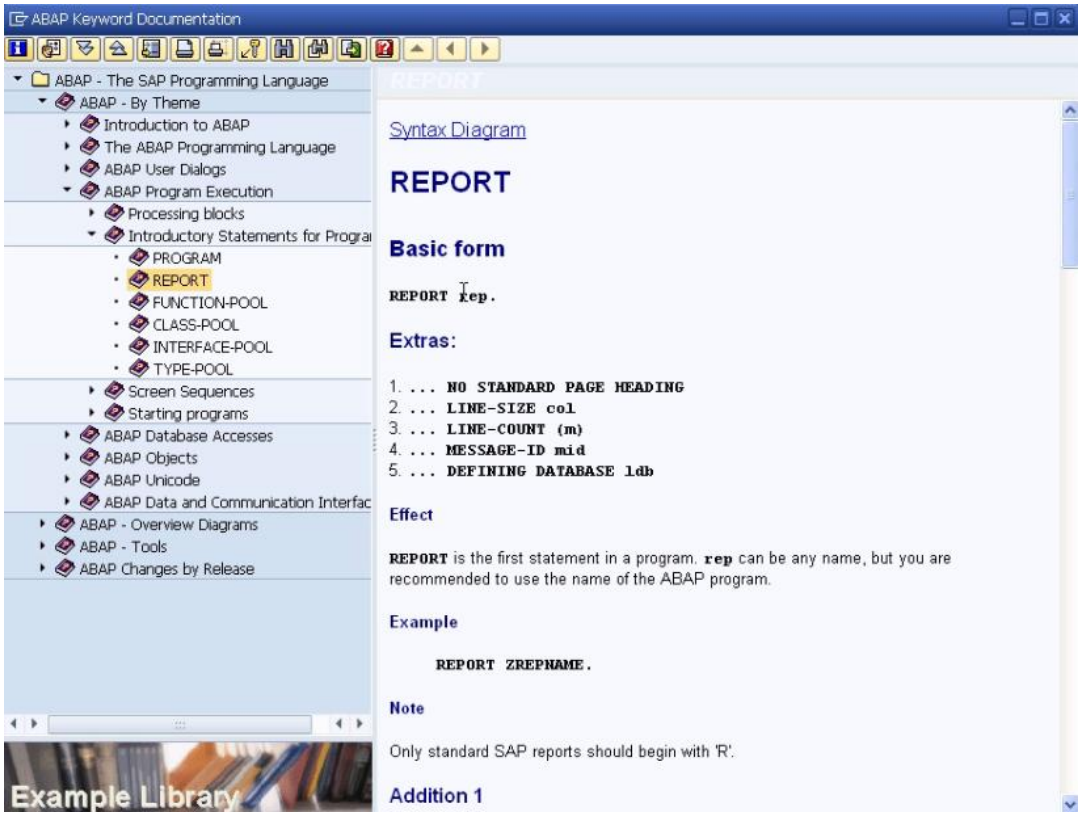


As long as the period appears at the end of the statement, no problems will arise. It is this period which marks where the statement finishes.

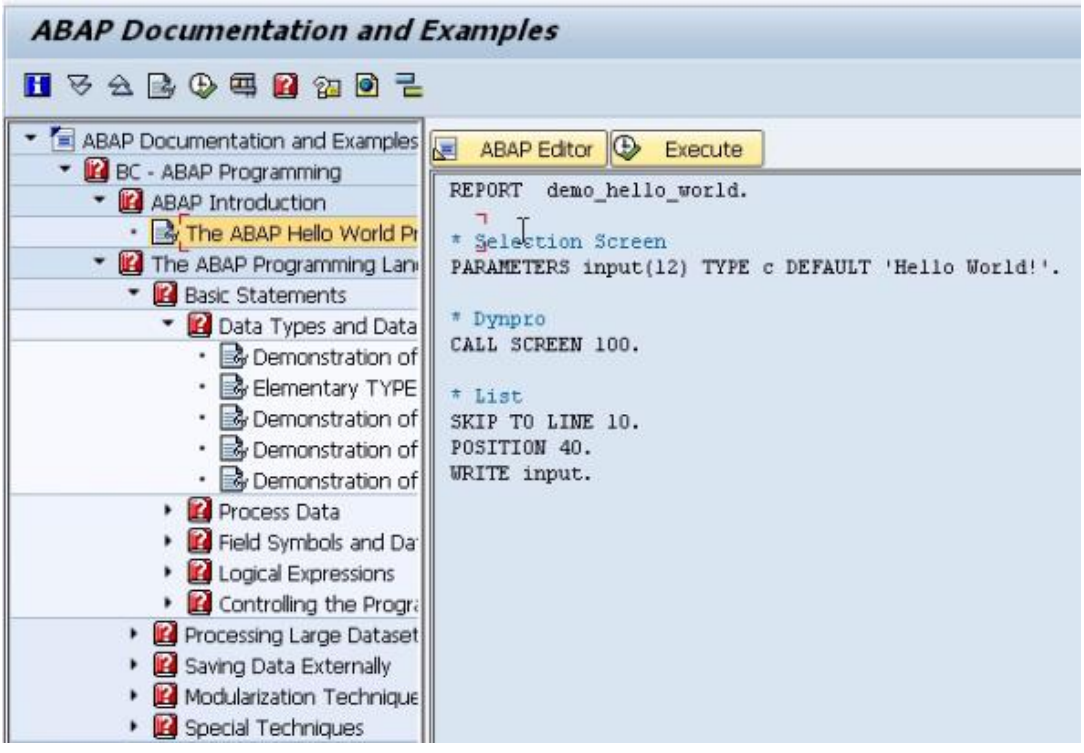
If you require help with a statement, place the cursor within the statement and choose the 'Help on...' button in the top toolbar:



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:



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:

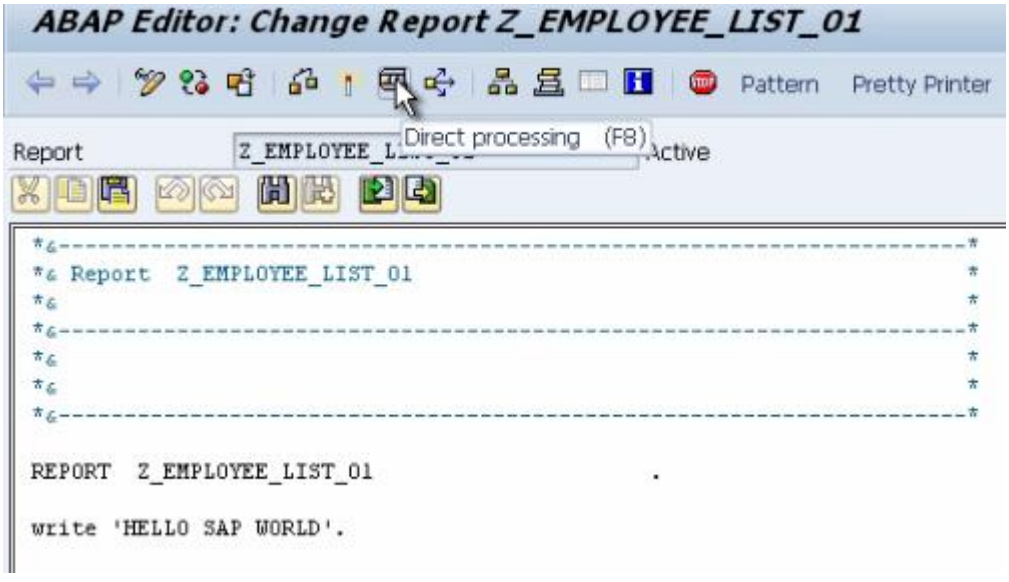


Returning to the ABAP editor now, the first line of code will be written. On the line below the REPORT statement, type the statement: ***write 'HELLO SAP WORLD'.***

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

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:



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

Write Statements

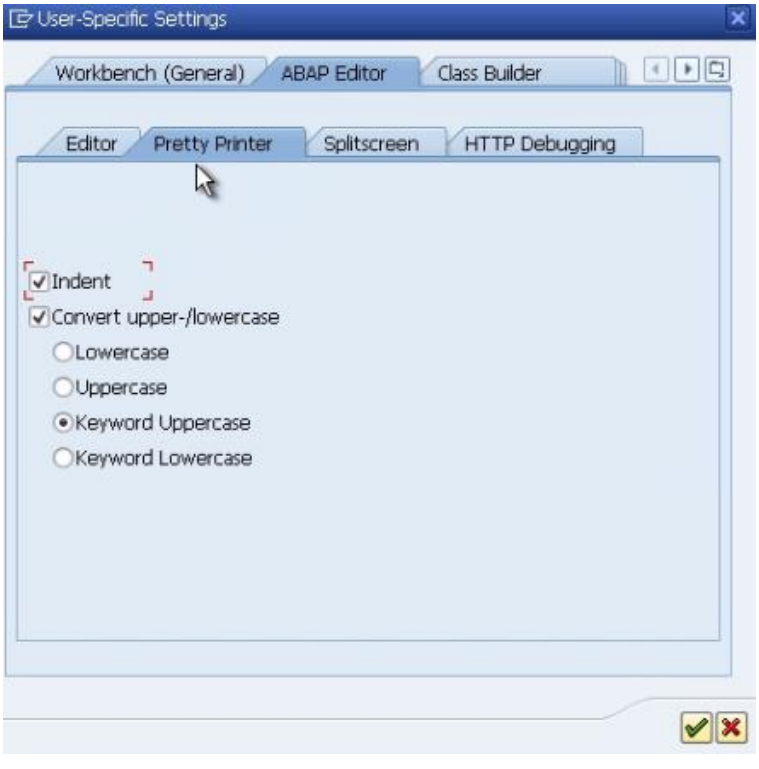
Now that the first program has been created, it can be expanded with the addition of further ABAP statements. Use the Back button to return from the test screen to the ABAP editor.

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):



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'.
```

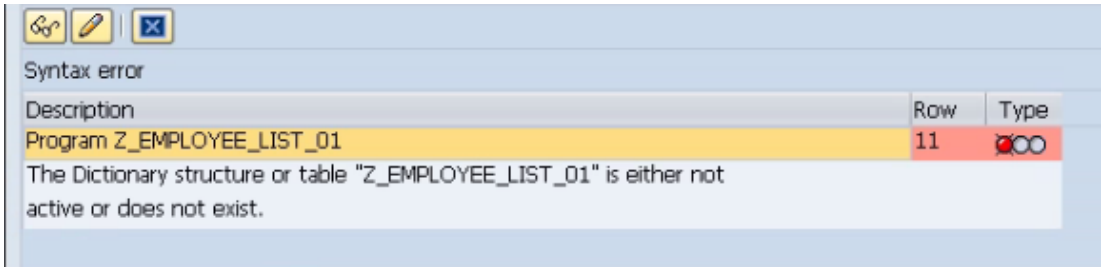
Let us now return to the TABLES statement. When the program is executed, the TABLES statement will create a table structure in memory based on the structure previously defined in the ABAP Dictionary. This table structure will include all of the fields previously created, allowing the records from the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

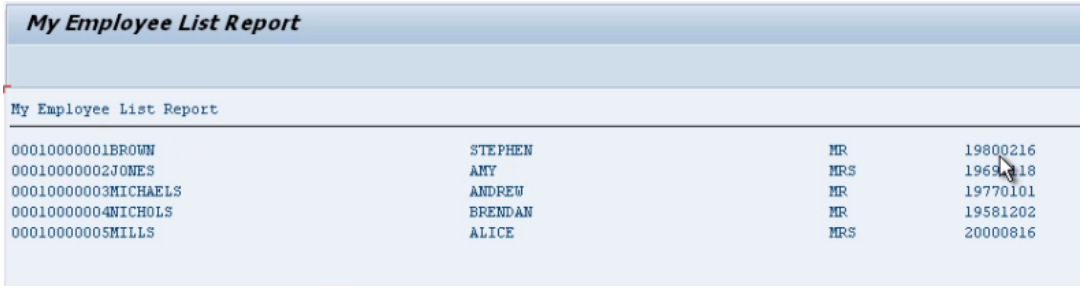
Check the code with the 'Check' button, and it will state that there is a syntax error:



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:



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.
```

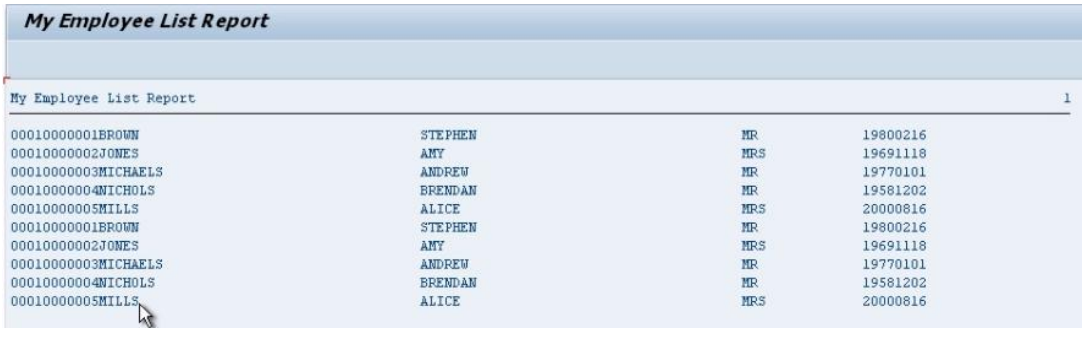
Bear these limits in mind so as to avoid automatic truncation when printing reports. For a standard sheet of A4 this limit will usually be 132 characters. When the limit is set to this for the example table here, the full table returns, but the line beneath the title '*My Employee List Report*' displays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

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:

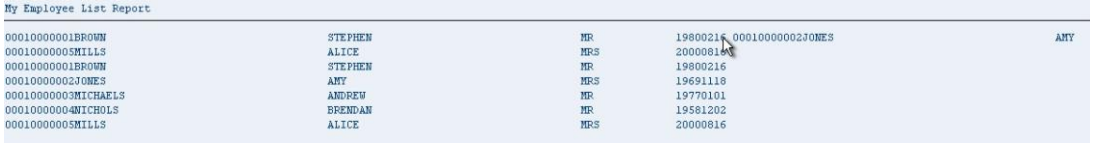
```
REPORT  z_employee_list_01 LINE-SIZE 132                    .

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:



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:



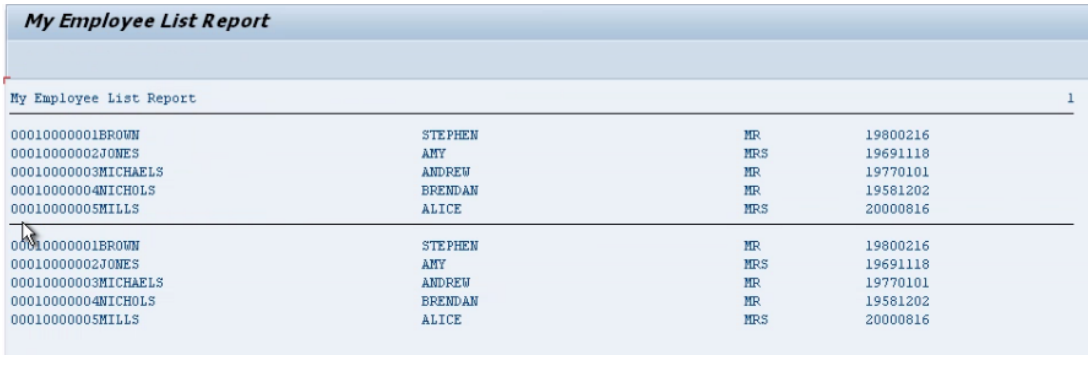
The first five rows, because they do not have a line break in the WRITE statement, have appeared on the first line up until the point at which the 532 character limit was reached and a new line was forced. The first four records were output on the first line. The 5th record appears on a line of its own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



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

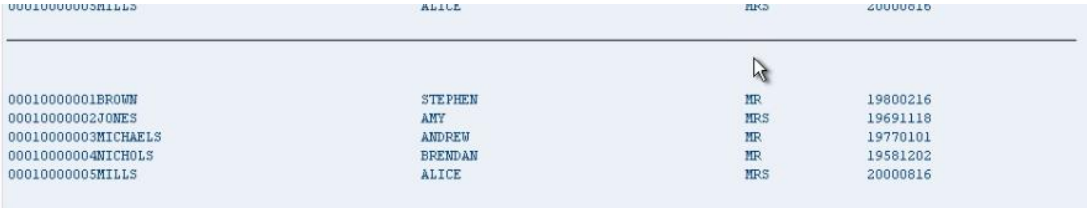
ULINE, to create a third SELECT output. In this third section, remove the line break from the WRITE statement and, on the line below, type “WRITE /.” This will mean that a new line will be output at the end of the previous line. Execute this to see the difference in the third section:

```
*****************************************
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.
```



```
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.

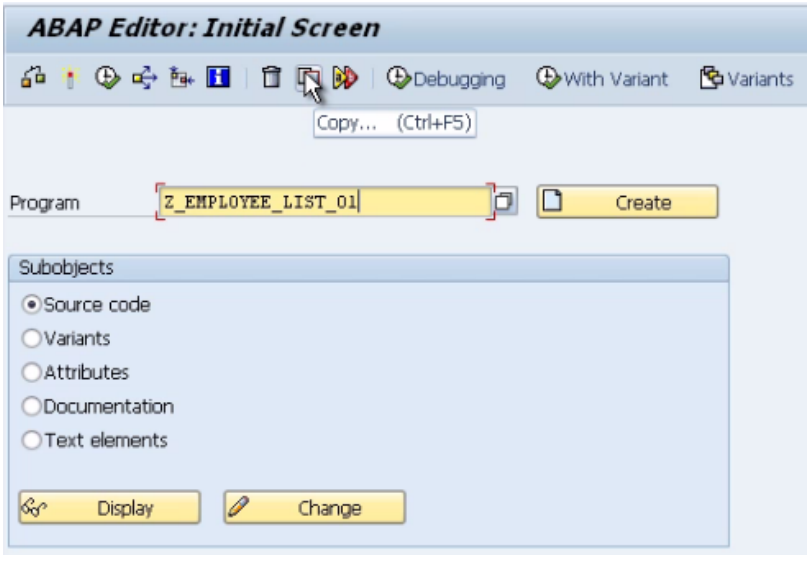
```
SKIP 2.
SELECT * FROM zemployees.     " Chain Statements
  WRITE: / zemployees-surname,
           zemployees-forename,
           zemployees-dob.
ENDSELECT.
```

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

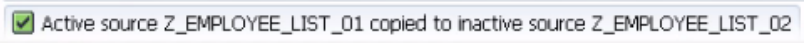
Copy Your Program

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:



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.

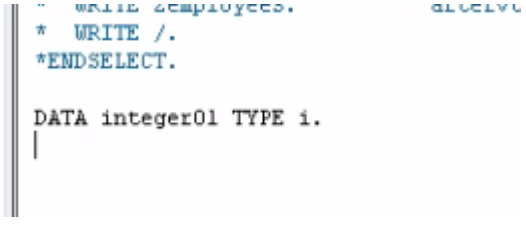
The next question to examine is that of variables, and how to declare them in a program. A variable is a field, the values of which change during the program execution, hence of course the term variable.

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

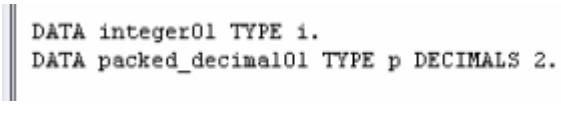
- They must begin with a letter.
- Can be a maximum size of 30 characters,
- Cannot include + , : or ( ) in the name,
- Cannot use a reserved word.

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.



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:



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.)*

Chapter 4

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.
```

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:

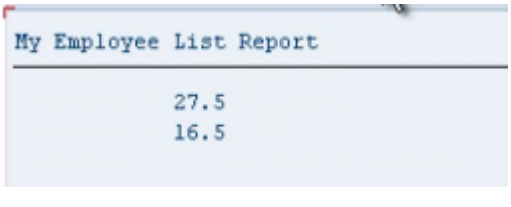
```
result        =         integer01  +    2.
```

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.
```



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
```

Chapter 5 – Character Strings

Declaring C and N Fields

This chapter will discuss character strings. When creating programs, fields defined as character 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, create a new program, named "**Z_Character_Strings**". Title this "**Character Strings Examples**", 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 brackets 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:

Return to the program, and in place of mychar2, create a new field named “**zemployees1**”, 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 calculations.

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 specifying 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 inserted 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 character 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 together 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 without 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
```

Condense

Next, the CONDENSE statement. Often an ABAP program will have to deal with large text fields, with unwanted spaces. The CONDENSE statement is used to remove these blank characters.

Now, observe the code 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,
      spaced_name(20)  type c VALUE  'Mr      Joe   Smith'.
*-----
```

This should, of course, be mostly familiar from the last section, with the addition of the new, 20-character "**spaced_name**" field, with large spaces between the individual words. Below we have an example of using the CONDENSE statement using our new variable:

```
CONDENSE spaced_name.
WRITE spaced_name.
uline.
```

The CONDENSE statement will remove the blank spaces between words in the variable, leaving only 1 character's space:

```
Mr Joe Smith
```

NO-GAPS

An optional addition to the CONDENSE statement is **NO-GAPS**, which as you may guess, removes all spaces from our variable.

```
CONDENSE spaced_name NO-GAPS.
WRITE spaced_name.
ULINE.
```

```
MrJoeSmith
```

Find the Length of a String

To find the length of a string, a function rather than a statement is used. Added beneath the previous data fields here, is a new one titled "**len**", with a TYPE **i**, so as to just hold the integer value of the string length.

```
      spaced_name(20)  TYPE c VALUE  'Mr      Joe   Smith',
      len              TYPE i.
```

The code to find the length of the 'surname' field and display it in the 'len' field appears like this, with "**strlen**" defining the function:

```
len = strlen( surname ).
WRITE: 'The length of the SURNAME field is', len.
ULINE.
```

The output, then, will appear like this:

```
The length of the SURNAME field is          5
```

Replace

Below I have created the "**surname2**" field and is 40 characters in length. Note that no TYPE has been defined, so the system will use the default type, c:

```
      len              TYPE i,
      surname2(40).
```

Some text is then moved into the field after which the **REPLACE** statement is used to replace the comma with a period:

```
surname2 = 'Mr, Joe Smith'.
REPLACE ',' WITH '.' INTO surname2.
WRITE: surname2.
ULINE.
```

```
Mr. Joe Smith
```

One thing to note here is that the REPLACE statement will only replace the first occurrence in the string. So if, for example, the surname2 field read "Mr, Joe, Smith", only the first comma would be changed. All occurrences of comma's could be replaced by making use of a *while loop*, which will be discussed later on.

Search

Next, a look will be taken at searching for specific character strings within fields. Unsurprisingly, the statement **SEARCH** is used for this.

All that is needed is to enter **SEARCH** followed by the field which is to be searched, in this instance the surname2 field. Then the string which is to be searched for, for example, '**Joe**':

```
surname2 = 'Mr Joe Smith'.
SEARCH surname2 for 'Joe'.
```

Note that here no variable has been declared to hold the result. In the case of the SEARCH statement, two system variables are used. The first is "**sy-subrc**", which identifies whether the search was successful or not, and the second is "**sy-fdpos**", which, if the search is successful, is set to the position of the character string searched for in *surname2*. Below, a small report is created to show the values of the system variables.

```
surname2 = 'Mr Joe Smith'.

WRITE: / 'Searching: "Mr Joe Smith"'.
SKIP.
```

SEARCH Example 1

The first SEARCH statement, below, indicates that the surname2 field is being searched, for the character string 'Joe '. The Search statement will ignore the blank spaces. The output will show the string being searched for, followed by the system variables and the value results. In this case, the search should be successful.

```
SEARCH surname2 FOR 'Joe    '.
WRITE: / 'Searching for "Joe    "'.
WRITE: / 'sy-subrc: ', sy-subrc, / 'sy-fdpos: ', sy-fdpos.
ULINE.
```

SEARCH Example 2

The next example is very similar, but the full stops either side of '**.Joe .**' mean that the blank spaces this time will not be ignored and the system will search for the full string, including the blanks. Here, the search will be unsuccessful, as the word '**Joe**' in the *Surname2* field is not followed by four blank spaces.

```
SEARCH surname2 FOR '.Joe    .'.
WRITE: / 'Searching for ".Joe    ."'.
WRITE: / 'sy-subrc: ', sy-subrc, / 'sy-fdpos: ', sy-fdpos.
ULINE.
```

SEARCH Example 3

This third search uses a wild card character '*****' and will search for any words ending in '**ith**'. This, again, should be successful.

```
SEARCH surname2 FOR '*ith'.
WRITE: / 'Searching for "*ith"'.
WRITE: / 'sy-subrc: ', sy-subrc, / 'sy-fdpos: ', sy-fdpos.
ULINE.
```

SEARCH Example 4

The last example also uses the wild card facility, this time to search for words beginning with '**Smi**', which again should be successful. Compare the places in the code where the * appears in this and the previous example.

```
SEARCH surname2 FOR 'Smi*'.
WRITE: / 'Searching for "Smi*"'.
WRITE: / 'sy-subrc: ', sy-subrc, / 'sy-fdpos: ', sy-fdpos.
ULINE.
```

Run a test on these searches, and output returns as follows:

```
Searching: "Mr Joe Smith"

Searching for "Joe    "
sy-subrc:       0
sy-fdpos:       3
---------------------------------------
Searching for ".Joe    ."
sy-subrc:       4
sy-fdpos:       0
---------------------------------------
Searching for "*ith"
sy-subrc:       0
sy-fdpos:       7
---------------------------------------
Searching for "Smi*"
sy-subrc:       0
sy-fdpos:       7
---------------------------------------
```

When the **sy-subrc** = 0 this refers to a successful search. When sy-subrc = 4 in the second example this indicates that the search was unsuccessful.

In the first search, the **sy-fdpos** value of 3 refers to the third character in the surname2 field, *the offset*, and the search term appears one character after this. The failure of the second search means that a 0 is displayed in the sy-fdpos field. The value of 7 in the sy-fdpos fields for the final two searches both mean that the word 'Smith' was found, corresponding to the search terms, and that the searched word appears 1 character after the offset value.

Shift

The **SHIFT** statement is a simple statement that allows one to move the contents of a character string left or right, character by character. In this example, a field's contents will be moved to the left, deleting leading zeros. Declare a new DATA variable as follows: "**empl_num**", **10** characters long, and set the content of the field to '**0000654321**', filling all 10 characters of the field:

```
      sep,
      destination(200) TYPE c,
      spaced_name(20)  TYPE c VALUE  'Mr      Joe   Smith',
      len              TYPE i,
      surname2(40),
      empl_num(10).
*-----
```

```
empl_num = '0000654321'.
```

Using the SHIFT statement, then, the 4 zeros which begin this character string will be removed, and the rest moved across to the left. Type the statement SHIFT, followed by the field name. Define that it is to be shifted to the left, deleting leading zeros (don't forget the help screen can be used to view similar additions which can be added to this statement). Then include a WRITE statement so that the result of the SHIFT statement can be output. To the right of the number here, there will be four spaces, which have replaced the leading zeros:

```
SHIFT empl_num LEFT DELETING LEADING '0'.
WRITE empl_num.
```

```
654321
```

If no addition to the SHIFT statement is specified, the system will by default move everything just one character to the left, leaving one space to the right:

```
SHIFT empl_num.
WRITE empl_num.
```

```
000654321
```

The **CIRCULAR** addition to the SHIFT statement will cause, by default, everything to move one space to the left again, but this time the character which is displaced at the beginning of the statement will reappear at the end, rather than leaving a blank space:

```
SHIFT empl_num CIRCULAR.
WRITE empl_num.
```

```
0006543210
```

Split

The SPLIT statement is used to separate the contents of a field into two or more fields. Observe the code below:

```
* SPLIT Statement - Splitting Character Strings
DATA: mystring(30) TYPE c,
      a1(10)       TYPE c,
      a2(10)       TYPE c,
      a3(10)       TYPE c,
      sep2(2)      TYPE c VALUE '**'.

mystring = ' 1234** ABCD **6789'.
*mystring = ' 1234** ABCD **6789**WXYZ'.
WRITE mystring.
SKIP.

SPLIT mystring AT sep2 INTO a1 a2 a3.

WRITE / a1.
WRITE / a2.
WRITE / a3.
```

The first section contains several DATA statements, "**mystring**", "**a1**", "**a2**", "**a3**" and "**sep2**", along with their lengths and types. "Sep2" here is a separator field, with a value of '**'.

"**mystring**" is then given a value of ' **1234** ABCD **6789**', followed by a comment line (which the program will ignore), then a WRITE statement, so that this initial value appears in the output followed by a blank line, using the SKIP statement.

The SPLIT statement appears, followed by the name of the string which is to be split. The **AT** addition appears next, telling the program that, where "sep2" appears (remember the value of this is '**'), the field is to be split. Following this, the **INTO** then specifies the fields which the split field is to be written to. The slightly odd positioning of the spaces in the value of "mystring" will, when the statement is output, make clear the way that the SPLIT statement populates the fields which the data is put into. Execute the code, and this is the result:

```
 1234** ABCD **6789

 1234
 ABCD
6789
```

You can see that the initial field has been split into a1, a2 and a3 exactly where the ** appeared, leaving a leading space in the first two fields, but not in the third. Additionally, on closer inspection there are blank spaces following the numbers in each field up to its defined length, which is 10.

This next example shows the initial value of "mystring" now is made into a comment line, and the comment line becomes part of the code:

```
*mystring = ' 1234** ABCD **6789'.
mystring = ' 1234** ABCD **6789**WXYZ'.
WRITE mystring.
SKIP.

SPLIT mystring AT sep2 INTO a1 a2 a3.

WRITE / a1.
WRITE / a2.
WRITE / a3.
```

'mystring' now contains the original contents plus a further set of characters. While the contents are still to be split into 3 fields, the data suggests it should be split into 4. In this case, with less fields than those defined, the system will include the remainder of the string in the final field. Note that if this field is not long enough for the remainder, the result would be truncated.

```
 1234** ABCD **6789**WXYZ

 1234
 ABCD
6789**WXYZ
```

SubFields

Within ABAP, you have the option of referring to specific characters within a field. This is referred to as processing ***subfields***, whereby a specific character's position within its field is referenced. Again, observe the code below:

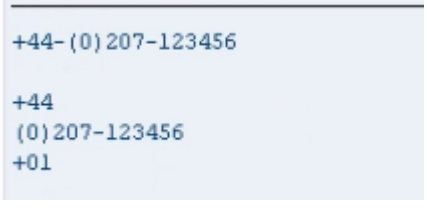
```
DATA: int_telephone_num(17) TYPE c,
      country_code(3)   TYPE c,
      telephone_num(14) TYPE c.

int_telephone_num = '+44-(0)207-123456'.
WRITE int_telephone_num.
SKIP.

country_code = int_telephone_num(3).
telephone_num = int_telephone_num+4(13).
WRITE / country_code.
WRITE / telephone_num.

country_code+1(2) = '01'.

WRITE / country_code.
```

To start with, new DATA variables are declared, "**int_telephone_num**", "**country_code**" and "**telephone_num**", along with lengths and types. Following this, a character string is assigned to int_telephone_num, a WRITE statement for this string and a blank line.

Next, the subfield processing appears. The first line states the *country_code* field is to be filled with the first 3 characters of the *int_telephone_num* field, indicated by the number in brackets.

Then, the field *telephone_num* is to be filled with **13** characters of the *int_telephone_num* field, starting after the 4th character. The **+4** part of the code here refers to where the field is to begin. Then we have WRITE statements for both of the fields.

This last example indicates that the specific characters of *int_telephone_num* moved to the *country_code* field will be replaced, after the first character, by the literal, 2-character value '**01**', showing that a subfield can itself be edited and updated without changing the initial field. The results should look like this:



Subfields are regularly used in SAP to save time on creating unnecessary variables in memory. It is just as easy to use the subfield syntax.

Chapter 6 – Debugging Programs

This chapter will introduce the ABAP debugger, and will introduce some of the tools which can be used to ensure that the programs you create function as intended. It will also show ways to highlight logic bugs in programs that cannot be identified by the syntax checker.

The first step here is to load a program which has been used previously, and which accesses the database table which has been created regarding employee records. If you have been following along with instructions, load program “Z_Employee_List_01” into the ABAP Editor.

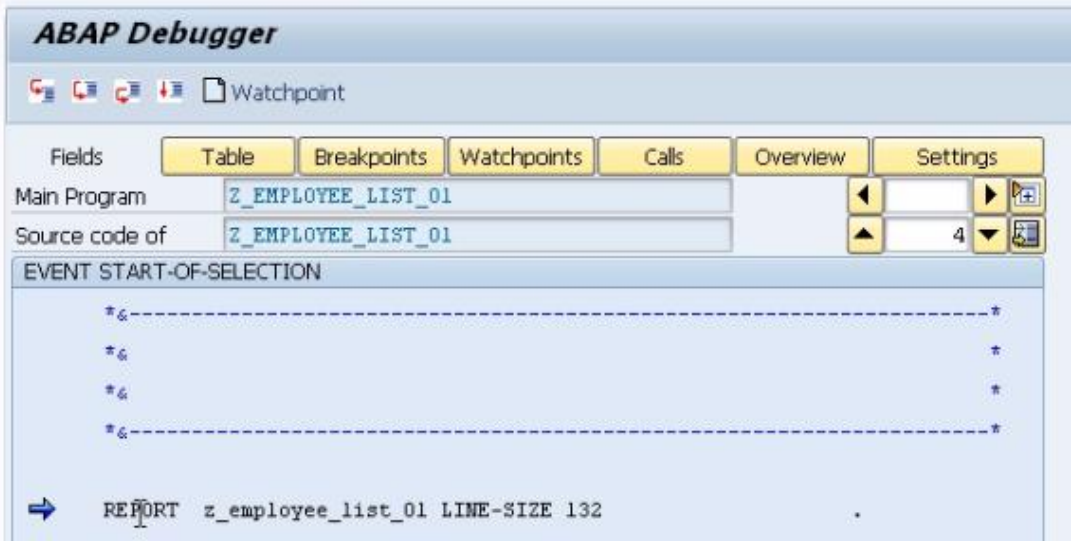
The program contains a number of SELECT loops, which in turn write the contents of the table being read to the output screen in several ways, separated by ULINE statements:

```
REPORT  z_employee_list_01 LINE-SIZE 132                    .

TABLES zemployees.

*****************************************
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.

ULINE.

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

Having examined the code, return to the front screen of the ABAP editor.

Firstly, on this screen you will notice there is a 'Debugging' button in the toolbar (also accessible with SHIFT+F5):

Click this with the program name in the program input text box to start a new debugging session. When this opens, a blue arrow should be visible, pointing at the first line of code in the program:



An alternative way of starting a debugging session is to display the code itself from the initial screen, select a line of code and set a breakpoint. This is done by, having selected a line, clicking the Stop icon: 

This sets a breakpoint for that line. When the program is then executed the execution will pause highlighting the line that has the Breakpoint set entering the debugging session. Usually, this is the easiest method to use, as one will often have a good idea of where the

issues in a program are allowing you to focus on specific areas of code straight away, rather than starting from the very beginning of a program as the previous method does:

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

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

There are two types of breakpoint which can be set in a program. Static (which will be examined later) and dynamic. A dynamic breakpoint is the kind which was used above, and these are only valid for the current session. If one leaves the SAP GUI and returns later, any dynamic breakpoints set will no longer exist. A breakpoint can also be set by double-clicking any statement within the debugging session itself. To then remove these in the session, simple double-click the stop icon appearing adjacent to them.

You will notice that a number of buttons appear at the top of the debugging screen:

These buttons allow for different modes of the ABAP debugger to be entered. The default mode here is *Fields*.

The 'Single step' button, the first on the left in the row above the modes, also accessible with F5, allows one to go through the code within the debugger line-by-line, or indeed as its name would suggest, single steps. As one presses the button, the blue arrow on the left of the code will move down one line at a time.

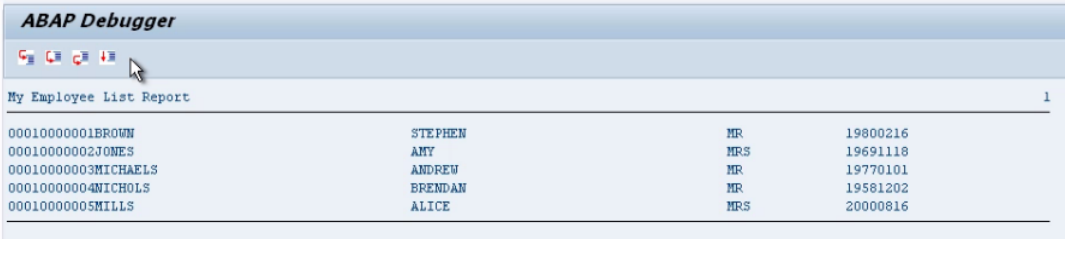
The next button along is the 'Execute' button, with a shortcut of F6. This allows for independent sections of code to be executed, such as function modules or forms. This can be very useful. If a program includes existing sections of code already created in an SAP sys-

tem which are known to be correct, there is no need to debug them. These can then be executed independently, while other parts are debugged to find specific problems.

The next button is the 'Return' function (F7). This can be very useful if one forgets to use the 'Execute' function. If one goes through the lines of a program step-by-step, using the F5 key to step into a working function module, which may contain many lines of code, it is likely the case that it does not need to be debugged (because you know this function module already exists). Pressing the F5 key endlessly to go through the lines of code here is unnecessary when one wants to step out of this function module and access the parts which require debugging. Using the 'Return' button, all of the code within a specific function can be executed, returning to the line of code which calls that function.

The fourth in the row is the 'Continue' option (F8). This allows one to continue the program without going through step-by-step, line-by-line. When this button is pressed, the program executes and the output screen is shown. This button can also be used to just access a selected line of code, where the cursor is positioned. If one positions the cursor in a line of code and presses continue, the blue arrow in the debugger will appear directly next to that line. If you then press continue again, the program will be executed.

The next option in this row of the toolbar is 'Display list', accessible with CTRL+F12. This takes you to the output screen as it currently stands within the debug session. Here, the code has been executed to output the result of the first SELECT statement in the program:

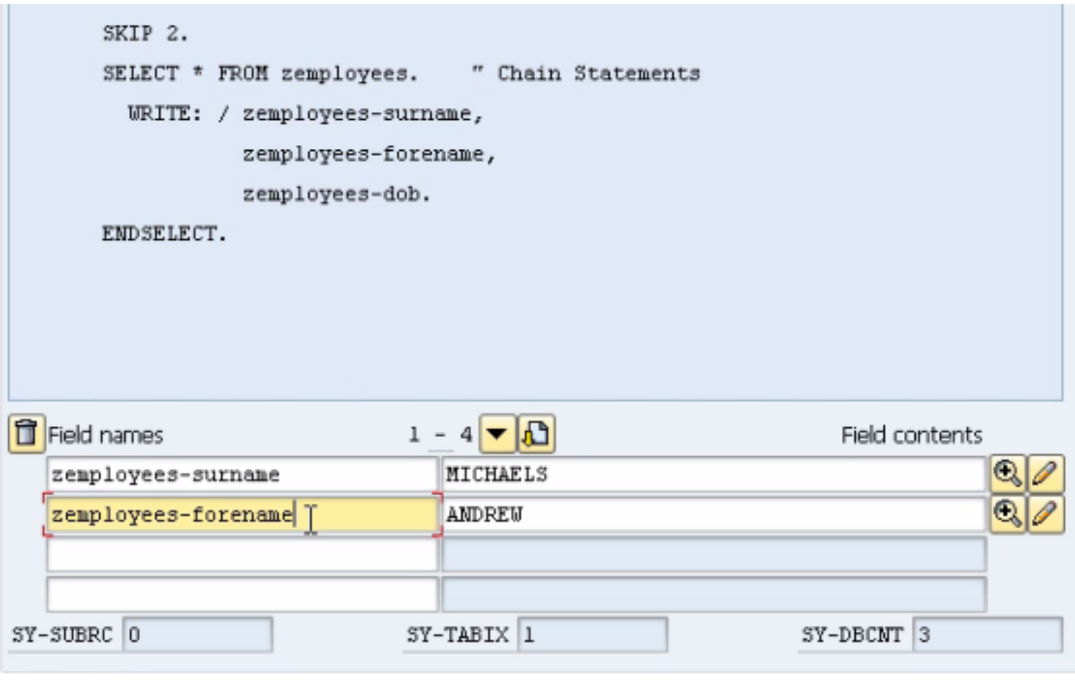


This function allows you to see the results of the reports whilst the program is in mid-flow.

The last option here is 'Create watchpoint' (SHIFT + F8). Watchpoints will be returned to soon.

Fields mode

The 'Fields' mode of the ABAP debugger allows the contents of fields to be checked and modified as the program is debugged. This can be accessed either by double-clicking the field name within the code itself, or entering it into the 'Field names' section below the code:

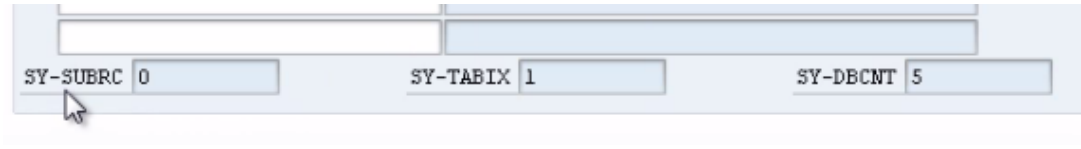


Note that, since here a table is involved, in the field name section the name of the table must first be specified, followed by a -, then the name of the field. The field contents will be filled in automatically. As you step through code line-by-line in the SELECT loop, the text held in each field will change as each loop completes and moves onto the next record in the table. This section allows for 8 fields to be monitored at any time. Fields 5 - 8 can be made visible via the navigation buttons in the middle (*to the right of the numbers 1 – 4*).

Often when debugging a program, you may want to manually change the contents of fields. This can be achieved by replacing the text in the field contents area, then clicking the 'Change field contents' icon, marked with a pencil. Doing this can save a lot of time, avoiding having to exit the debugging session multiple times to enter new values into fields elsewhere:

System Variables

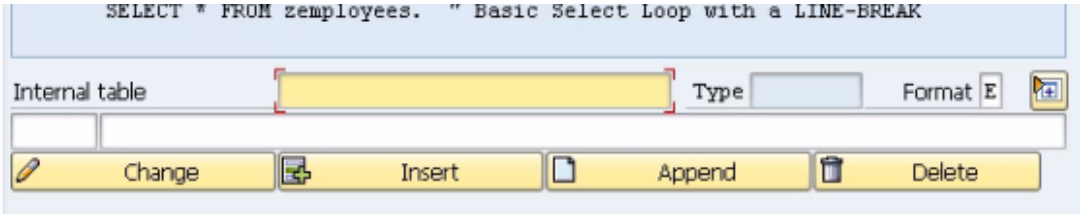
At the bottom of the debugger screen, are 3 fields, named 'SY-SUBRC', 'SY-TABIX' and 'SY-DBCNT':



Note that the value boxes here are greyed-out, meaning that they cannot be changed manually. These are system fields, belonging to a table called SYST. This system table includes many system fields which are filled in at runtime. These system fields are filled in automatically while the program is executed. Most statements within ABAP will cause these system fields to be filled with 0 when executed successfully. It is important to remember that these fields are completely statement-dependent, meaning that they will contain different values depending on which statement is executed. These system codes and variables will be looked at in greater depth later.

Table Mode

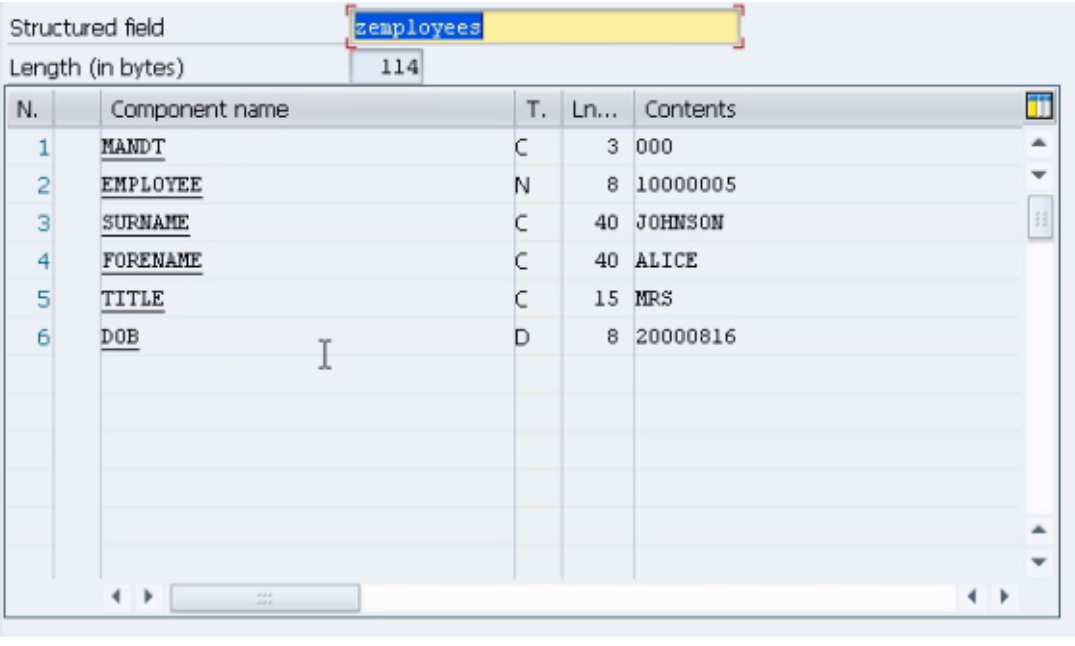
The second mode along from the Fields button on the left of the screen is Table mode. Click this button and the code remains, but the bottom section changes to include an 'Internal table' entry, and a single row:



Internal tables have not yet been covered in depth, but, put simply; an internal table is a table of records which is stored in memory while the program is running. Table mode allows one to interrogate the records and fields of each record in an internal table. As with

Fields mode, the internal table can either be double-clicked in the code, or manually entered into the 'Internal table' box.

If one does this for "zemployees", then, a new window appears, displaying the table name, its individual fields and their contents:

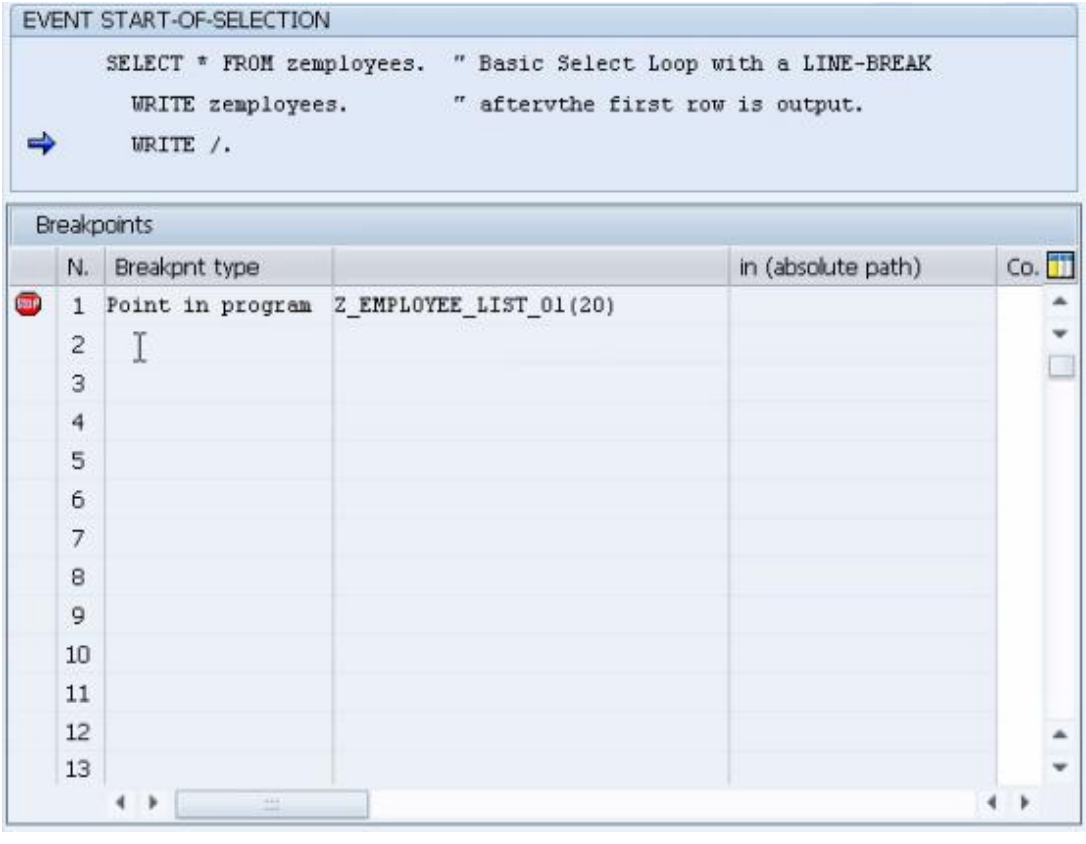


Things do look slightly different to normal here, as a table structure is being shown, rather than an actual internal table. This results in the debugger showing the table structure as above, listing the individual fields numbered 1 – 6 and their contents. When viewing an internal table in this mode, one will see a number of records for each internal table with their contents. These records can then be double-clicked to move to the above layout, showing the individual fields for each record. This will be returned to later.

In this screen, the code remains, but the area in which it is displayed is very small. One can continue to interrogate the code line-by-line as before still, but this may prove difficult. It is usually simpler to check Table mode for the information required, and then click back to Fields mode to continue the debug session.

Breakpoints

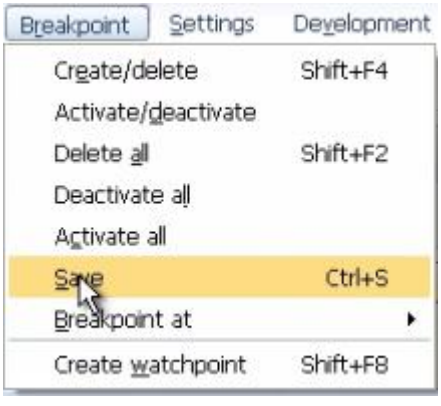
Click the Breakpoint mode button in the ABAP debugger screen. This allows you to see a list of the individual breakpoints which have been set. Double-clicking any breakpoint in the Breakpoints table will remove that breakpoint from the list:



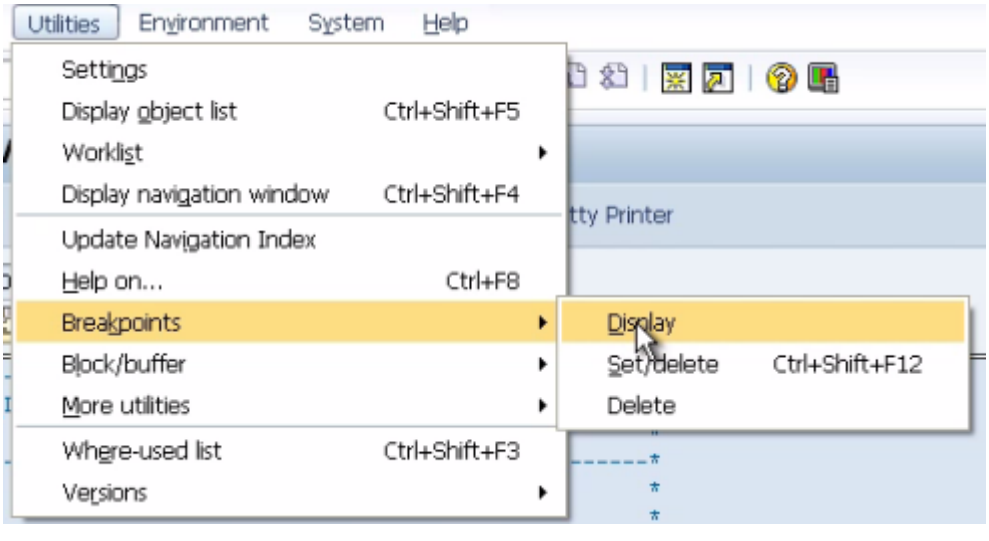
This breakpoint table can be very useful, particularly when one is in a large program with many breakpoints set. It allows one to review the breakpoint, and allows for the removal of breakpoints which are no longer desired.

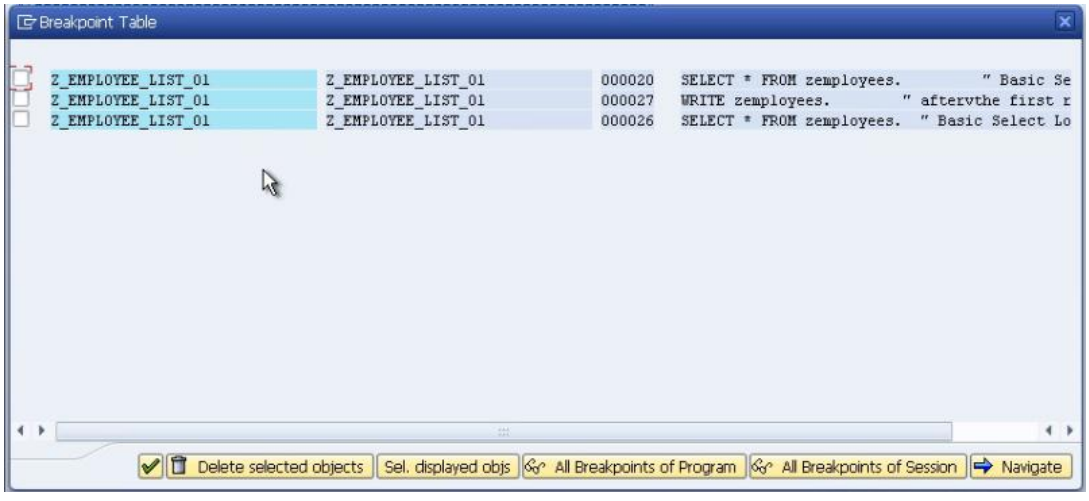
It is important to remember that breakpoints (and indeed Watchpoints) are only valid for the length of the current debug session. When you exit your session, the breakpoints will be deleted. However, an option does exist allowing you to save breakpoints (and, again, Watchpoints) before closing a debug session, keeping them active for the next time the program is to be debugged, saving the hassle of recreating them. This is done by entering

the 'Breakpoint' menu in the top toolbar and choosing 'Save'. All of the breakpoints saved will then remain until they are manually removed, or until the end of your SAP session.



If one is in the ABAP editor, it is possible to see an overview of all the dynamic breakpoints set in the program by accessing the following menu option: Utilities → Breakpoints → Display:



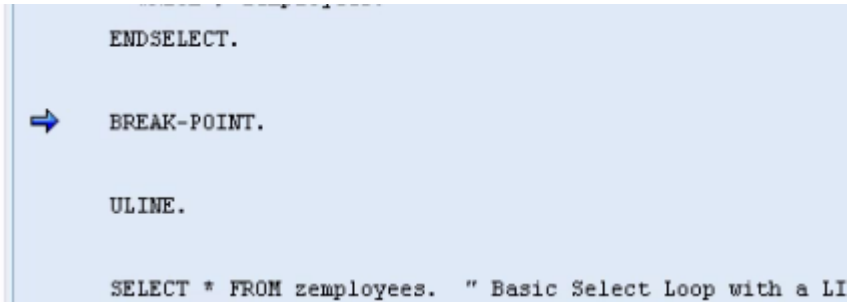


The options at the bottom of this breakpoint table allow one to delete selected breakpoints without entering the debugger and breakpoints can be navigated to in the program itself (within the ABAP editor) by double clicking them in this table.

Static Breakpoints

Static breakpoints were briefly alluded to earlier. These refer to a line of code written into a program which forces the program to enter debug mode at the specific line chosen. To do this, the statement **BREAK-POINT** is used. When the code is executed, the debug session will start with the usual blue arrow cursor pointing at the location of the static breakpoint.

```
  WRITE / zemployees.
ENDSELECT.

BREAK-POINT.

ULINE.

SELECT * FROM zemployees.  " Basic Select Loop with a I
```



Once this statement is embedded in a program, it is active for all users. This is largely undesirable, as others running the program, who do not want to debug the code, would be faced with the breakpoint set by an individual user. Be careful not to leave this statement line in programs which will be transported to other systems.

Watchpoints

Click the Watchpoints button in the ABAP debugger. The program code will be visible above the Watchpoints table in the lower half of the screen. Breakpoints have previously been discussed, and can be very useful, but are not always the ideal tool to use to pause code execution, interrogate the contents of individual fields and internal tables and analyse the program's logic.

Imagine the program was processing a table containing 1000 records, and one wanted to debug the logic only when a certain condition occurs. This condition is dependent upon the data held in the records being processed. By using breakpoints, one would have to debug each individual record, obviously taking a huge amount of time. Here, Watchpoints become useful. Using these, one can tell the program to stop in the same manner that it would for a breakpoint, but instead of stopping at a specific line of code, it would stop based on the value in a field. In this example then, if this value occurred only in the 200th line of the table, a watchpoint would allow the first 199 records to be skipped over.

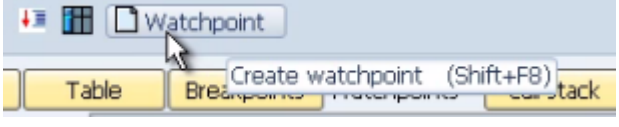
A watchpoint is created with the 'Create watchpoint' button, seen above the list of modes in the Watchpoint mode screen, or with SHIFT + F8.

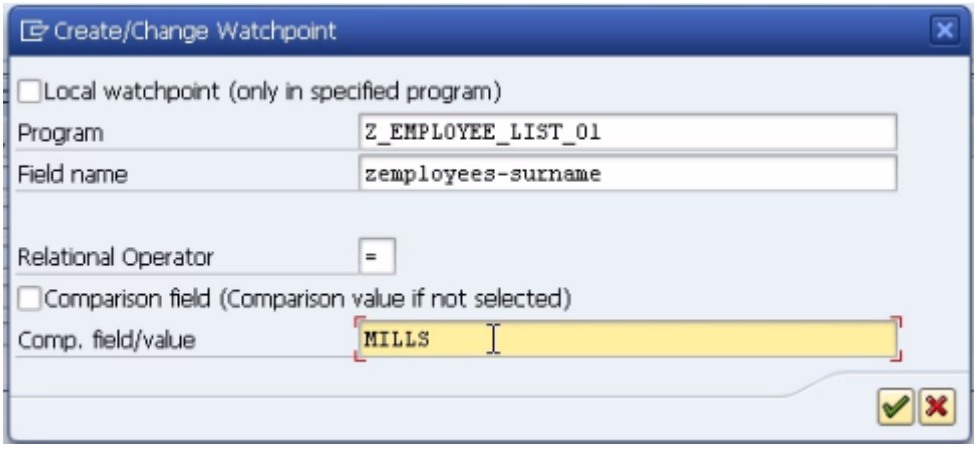
Once this is done, a dialogue box will appear, with the program name filled in automatically. Here you need to enter the name of the field to be watched. In the Z_EMPLOYEE_LIST_01 example here, we will enter the *surname* field. The format is *TABLE_NAME-FIELD_NAME*. Next, the relational operator is to be set. In this example, a sur-

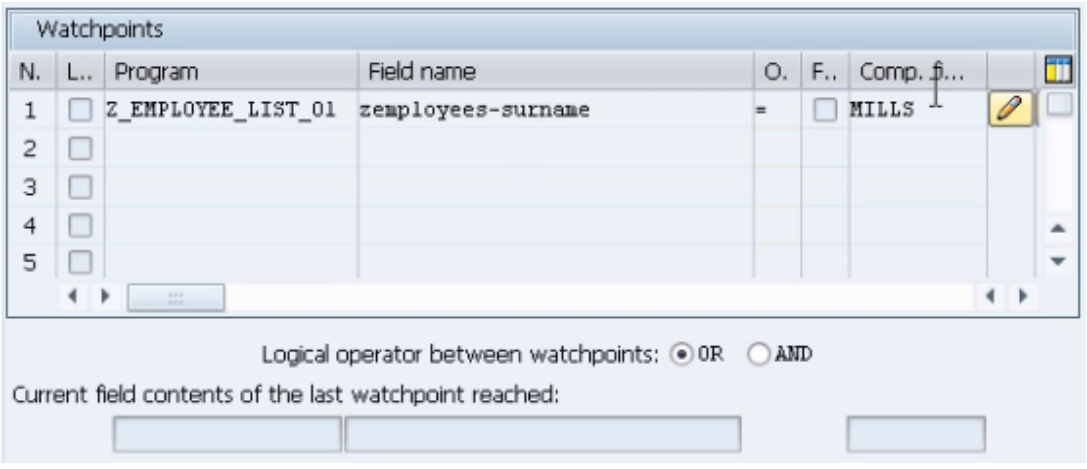
name with the value “Mills” will be sought, so the operator here is an =. This can be selected from a drop-down menu, where one can also view other potential relational operators. The bottom field, then, should be filled in with the value to be watched for.

Note that one does not have to use a specific value in the bottom field, but can get a watchpoint to compare a field against another field within the program. To do this the ‘Comparison field’ box should be checked, and the field name typed into the box rather than a specific value.

Click the green tick to continue and create the watchpoint, and the entry will have been added to the list at the bottom of the screen:

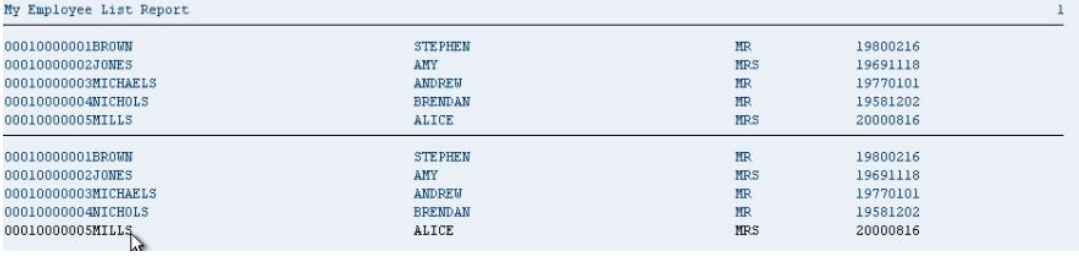




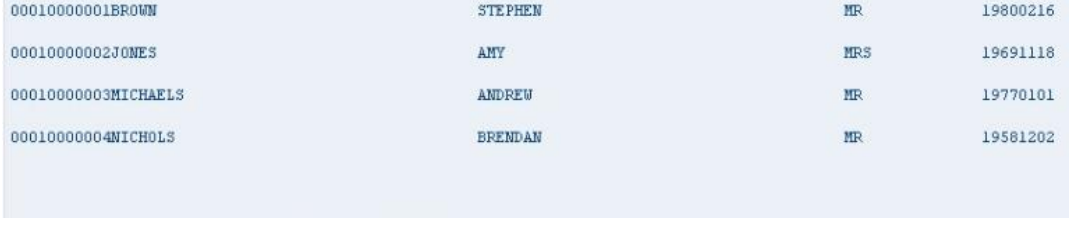


Observe the boxes below the Watchpoints list here. They are currently empty, but when the program is executed, it will pause once a value of 'Mills' is reached in the 'surname' field and this will be included in the box.

The output before the program is executed looks like this:



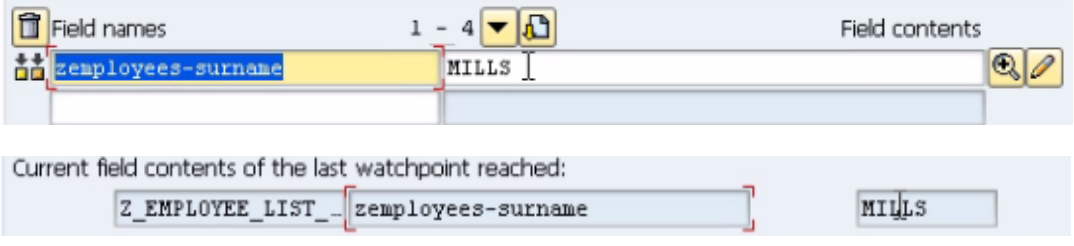
Note that the surname Mills appears in the fifth row down. When the program is executed with the 'Mills' watchpoint set, the first four records will be written to the screen before pausing at the fifth, when Mills is displayed.



You will see that the blue arrow cursor has paused at the SELECT loop in the code.

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

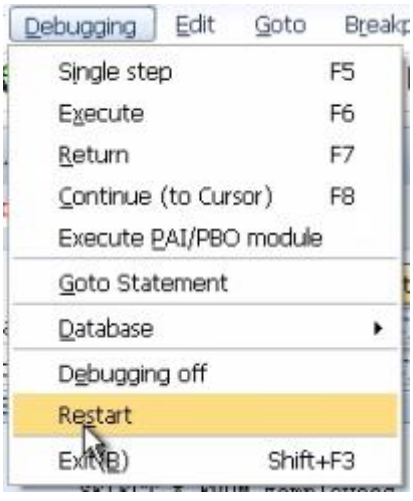
Enter zemployees-surname in the Fields mode of the debugger to view the contents of the field. You will see the field contains “MILLS”. Also in the Watchpoints mode, the bottom field will now be filled:



Ending a Debug Session

There are two ways to stop debugging a program. The first is to use the F8 key to run the program all the way through to the end. Keep in mind though, that if any break or Watch-points are set, the execution will likely pause and have to be started again, perhaps multi-ple times. Also this method depends entirely upon the program executing successfully. If any runtime errors are caused, the debug session will terminate and return you to the SAP menu screen.

The alternative way of stopping the debugger is to enter the ‘Debugging’ menu and choose ‘Restart’. This way, no more of the program will be executed, and you can return to the ABAP Editor’s initial screen:

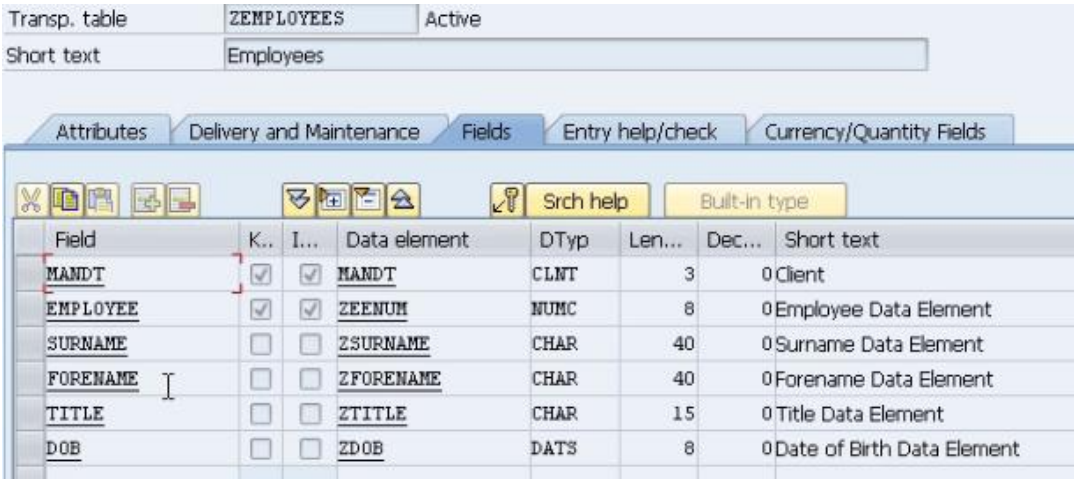

Debugging
Edit
Goto
Breakp
Single step F5
Execute F6
Return F7
Continue (to Cursor) F8
Execute PAI/PBO module
Goto Statement
Database
Debugging off
Restart
Exit(B) Shift+F3

Chapter 7: Working with Database Tables

Making a Copy of a Table

This chapter will look at ways in which one can change the transparent tables created earlier. It is important to know how to do this, and the implications of adding and taking away fields for the underlying data in a database table.

Let's take a look at the ZEMPLOYEES table created in Chapter 2. In the SAP GUI, key in transaction code SE11 to access the ABAP dictionary, then display the table:



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
FORENAME	☐	☐	ZFORENAME	CHAR	40	0	Forename Data Element
TITLE	☐	☐	ZTITLE	CHAR	15	0	Title Data Element
DOB	☐	☐	ZDOB	DATS	8	0	Date of Birth Data Element

It is important to realise that whenever one wants to change a database table, there is a risk of losing data, especially where key fields in the table are being affected. The database system itself will try to determine whether adjustments can be made by deleting and creating new items which change the underlying database catalogue, or whether what has already defined has to be re-implemented.

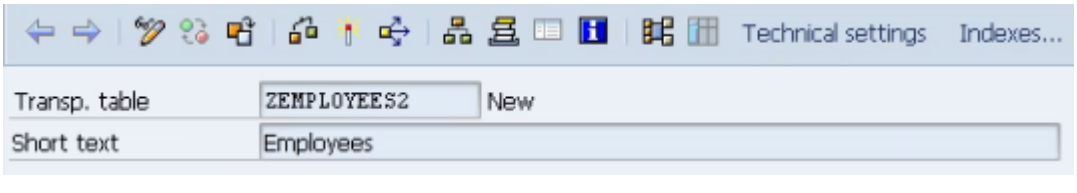
Quite often, when working with large tables, one has to manage the manipulation of the data oneself, so as to be sure that data is not lost. Deleting fields is quite a simple task, the table structure and its contents can add certain complications. Before starting any database change tasks, it is important to mitigate against as many risks as possible, and start

by using a copy of the database table, allowing one to test out any changes one may want to make, without affecting the initial table and its underlying data.

When you copy a database table, it is only the structure itself which is copied, meaning only its properties - fields and so on, not the actual data.

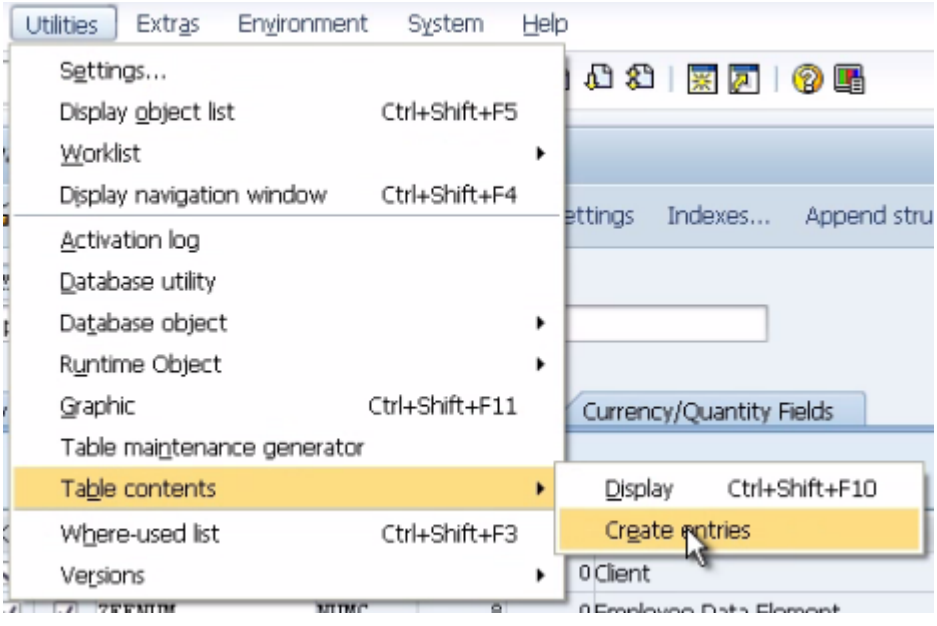
Step back to the initial SE11 screen. With ZEMPLOYEES in the Database table field, click the Copy button, then give the new table of **ZEMPLOYEES2**. The 'Create Object Directory Entry' box will appear and as before, select 'Local Object':

A copy of the table has now been created. Choose display at the SE11 screen and the copy will appear. The table's status will read as 'New'. It must be activated, so click the 'Change' button (the Pencil icon in the toolbar), and then Activate:

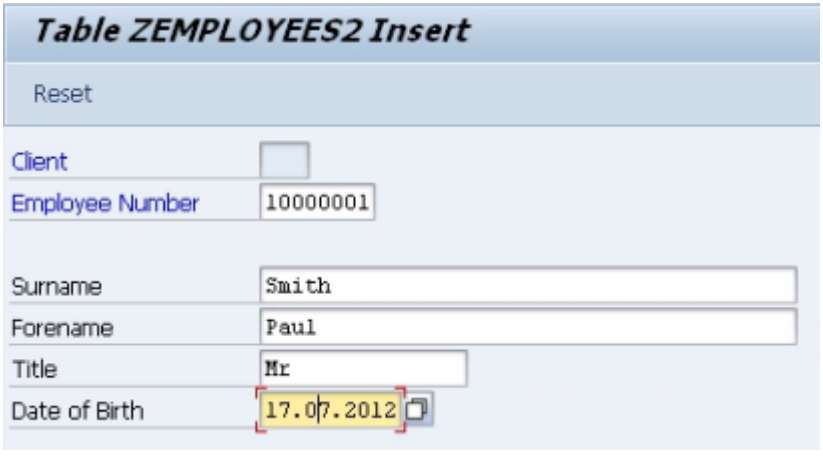


Note that all of the fields in the table, since they have been copied, are already active. This is why it is only the table itself which has to be activated here. If you try to look at the ta-

ble, you will find there are no contents, because only the structure was copied, not the underlying data. To create records, from the 'Utilities' menu, select 'Table Contents' and then 'Create Entries' to display the screen where the records for the table can be created as before.



Insert some records, click the Contents button, and then view the new table:



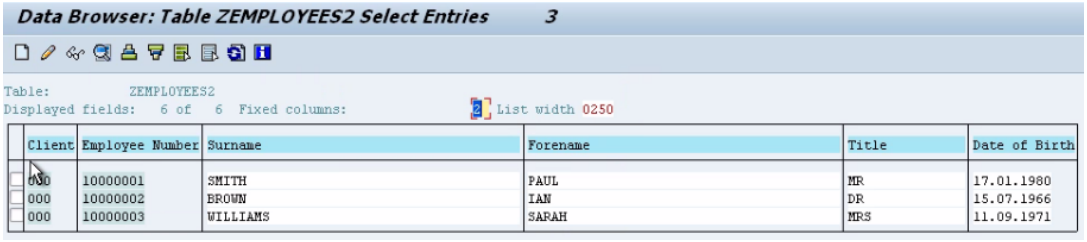

Data Browser: Table ZEMPLOYEES2 Select Entries 3

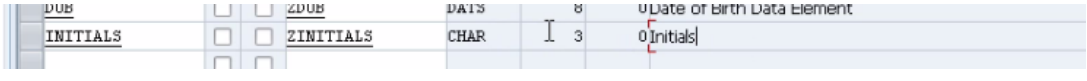
Table: ZEMPLOYEES2
Displayed fields: 6 of 6 Fixed columns: 2 List width 0250

Client	Employee Number	Surname	Forename	Title	Date of Birth
000	10000001	SMITH	PAUL	MR	17.01.1980
000	10000002	BROWN	IAN	DR	15.07.1966
000	10000003	WILLIAMS	SARAH	MRS	11.09.1971

Add New Fields

Next, a new field will be added. This will be a non-key field and will be called **INITIALS**.

Create a new Data element for this named ZINITIALS using forward navigation. For the data element, set the short text to 'Initials' and set the domain to CHAR03 (a character string of length 3). In the Field label boxes type 'Initials', then activate the Data element. The table should now have a new field like this:



Create another 3 more new fields and configure them as follows:

- Field Name 'GENDER'
    - Set the Data element to 'ZGENDER'. Configure the data element as follows:
        - Short text: 'Gender'
        - Domain: 'CHAR01'
        - Field labels set to 'Gender'
- SALARY
    - Set the Data element to ZSALARY
        - Short text: 'Salary'
        - Domain: 'CURR9' (*This has a length of 9, with 2 decimal places*)
        - Field labels set to 'Salary'.

One thing to note about the Salary field is that, because it is a currency, another field for this currency must be created and attached to ZSALARY to indicate what currency the salary is in. If you try to activate the table without doing this, an error message will appear asking for a reference field to specify the currency.

Create a new field called ECURRENCY. Currency fields should already exist in the system, so the Data element here will be a pre-existing one named **CURCY**. Type this, press enter and the remaining fields should fill in automatically, leaving the new section of the table looking like this:

DOB	☐	☐	ZDOB	DATS	8	0 Date of Birth Data Element
INITIALS	☐	☐	ZINITIALS	CHAR	3	0 Initials
GENDER	☐	☐	ZGENDER	CHAR	1	0 Gender
SALARY	☐	☐	ZSALARY	CURR	9	2 Salary
ECURRENCY	☐	☐	CURCY	CUKY	5	0 Currency Key

Next, the system must be told that the Salary field is referencing the Currency field. Above the table will be able to see a tab labelled 'Currency/Quantity Fields'. Click this and the table will be shown with two boxes to be filled in for the Salary field, since it has already been specified that the domain for this field is Currency. In the 'Reference table' column enter the name of the table, 'ZEMPLOYEES2' and in the 'Reference field' column, the name of the new Currency Key, 'ECURRENCY'. Now the table can be activated error free.

Attributes | Delivery and Maintenance | Fields | Entry help/check | Currency/Quantity Fields

Search help 1 / 10

Field	Data element	DTyp	Reference table	Ref. field	Short text
MANDT	MANDT	CLNT			Client
EMPLOYEE	ZEENUM	NUMC			Employee Data Element
SURNAME	ZSURNAME	CHAR			Surname Data Element
FORENAME	ZFORENAME	CHAR			Forename Data Element
TITLE	ZTITLE	CHAR			Title Data Element
DOB	ZDOB	DATS			Date of Birth Data Element
INITIALS	ZINITIALS	CHAR			Initials
GENDER	ZGENDER	CHAR			Gender
SALARY	ZSALARY	CURR	ZEMPLOYEES2	ECURRENCY	Salary
ECURRENCY	CURCY	CUKY			Currency Key

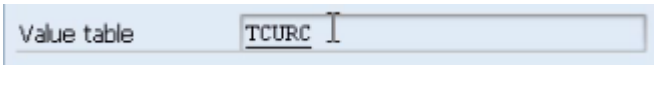
Foreign Keys

As shown earlier enter a new record. You will see that the currency key does not offer any kind of drop-down menu, here for this example, type GBP, indicating Great British Pounds:

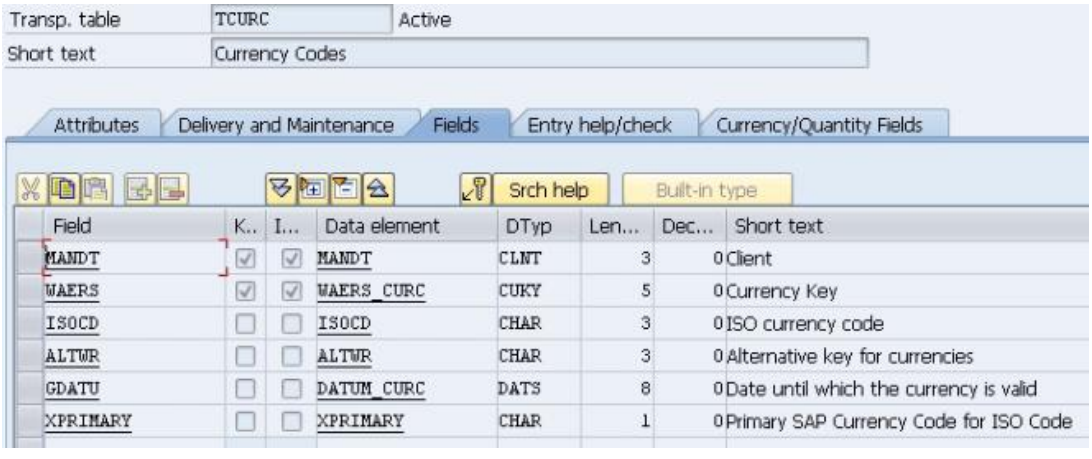
Save the record, and then return to the design of the table, where we can now add some error-checking to ensure that valid entries are made in the Currency key field.

To enable error-checking on the currency key field, we need to make use of a Foreign Key. These are used to ensure that only valid values can be entered into a field. Use forward navigation on the CURCY data element. Look at the Data type tab and you will see that the data element refers to a standard SAP domain, WAERS:

Double-click the WAERS domain to use forward navigation again. Look at the 'Value range' tab in this window, a 'Value table' box is visible at the bottom, labelled TCURC:



A Value table can be used to determine the entries that can be made in the field based on this domain. Double-click TCURC to again use forward navigation and this value table will be displayed.



Transp. table TCURC Active

Short text Currency Codes

Attributes | Delivery and Maintenance | Fields | Entry help/check | Currency/Quantity Fields

Srch help | Built-in type

Field	K..	I...	Data element	DTyp	Len...	Dec...	Short text
MANDT	☑	☑	MANDT	CLNT	3	0	Client
WAERS	☑	☑	WAERS_CURC	CUKY	5	0	Currency Key
ISOCD	☐	☐	ISOCD	CHAR	3	0	ISO currency code
ALTWR	☐	☐	ALTWR	CHAR	3	0	Alternative key for currencies
GDATU	☐	☐	DATUM_CURC	DATS	8	0	Date until which the currency is valid
XPRIMARY	☐	☐	XPRIMARY	CHAR	1	0	Primary SAP Currency Code for ISO Code

Use the data browser to look at the data in this table. If you scroll down, the GBP value from before can be found, among a number of others. This table can be used to ensure that, in future, only entries found in this table can be entered into our new table ZEMPLOYEES2

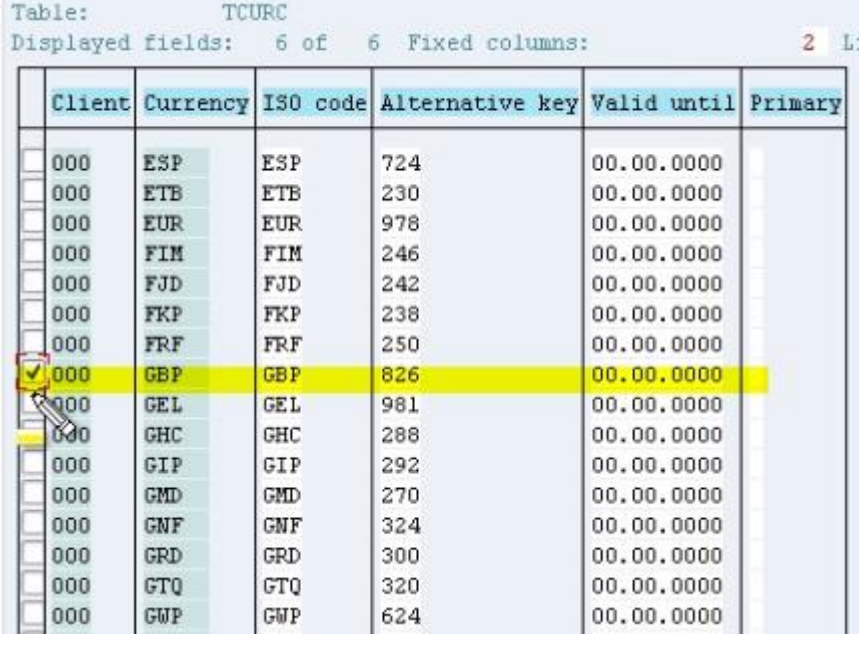


Table: TCURC

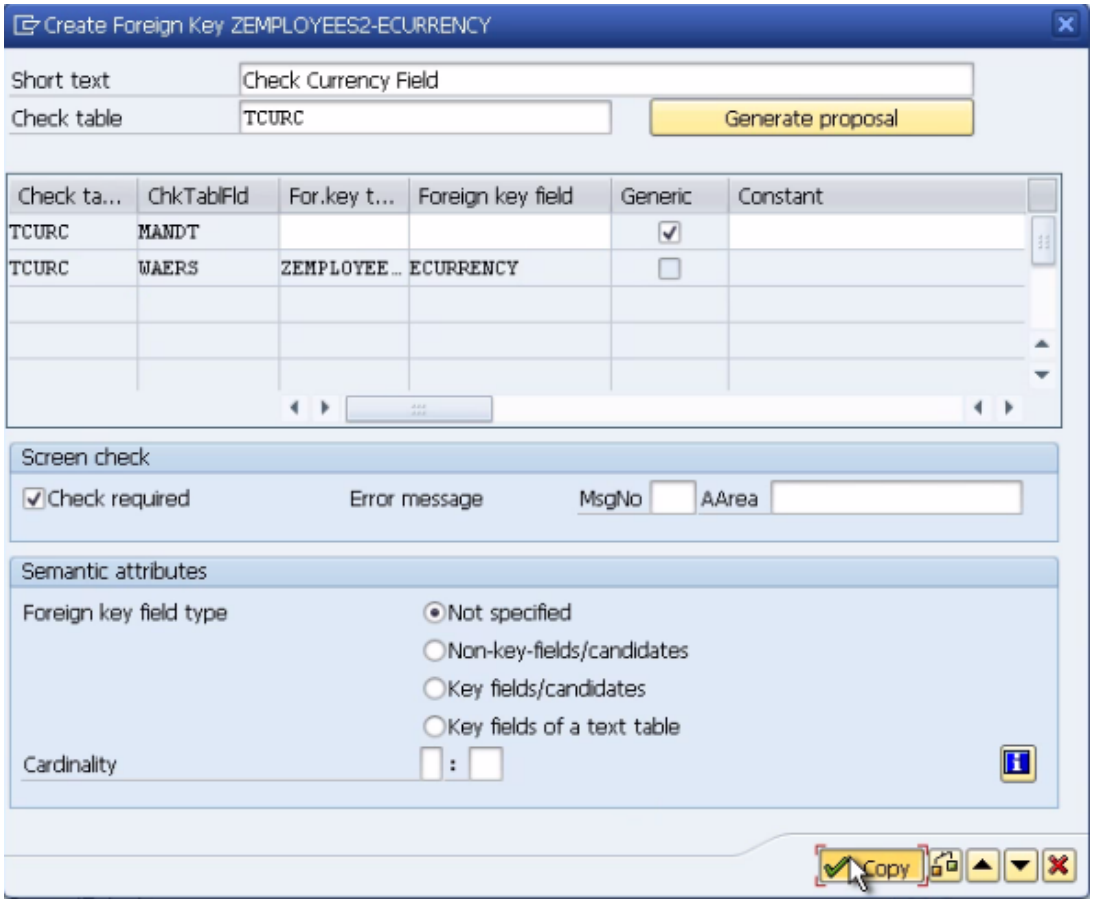
Displayed fields: 6 of 6 Fixed columns: 2

Client	Currency	ISO code	Alternative key	Valid until	Primary
000	ESP	ESP	724	00.00.0000	
000	ETB	ETB	230	00.00.0000	
000	EUR	EUR	978	00.00.0000	
000	FIM	FIM	246	00.00.0000	
000	FJD	FJD	242	00.00.0000	
000	FKP	FKP	238	00.00.0000	
000	FRF	FRF	250	00.00.0000	
000	GBP	GBP	826	00.00.0000	
000	GEL	GEL	981	00.00.0000	
000	GHC	GHC	288	00.00.0000	
000	GIP	GIP	292	00.00.0000	
000	GMD	GMD	270	00.00.0000	
000	GNF	GNF	324	00.00.0000	
000	GRD	GRD	300	00.00.0000	
000	GTQ	GTQ	320	00.00.0000	
000	GWP	GWP	624	00.00.0000	

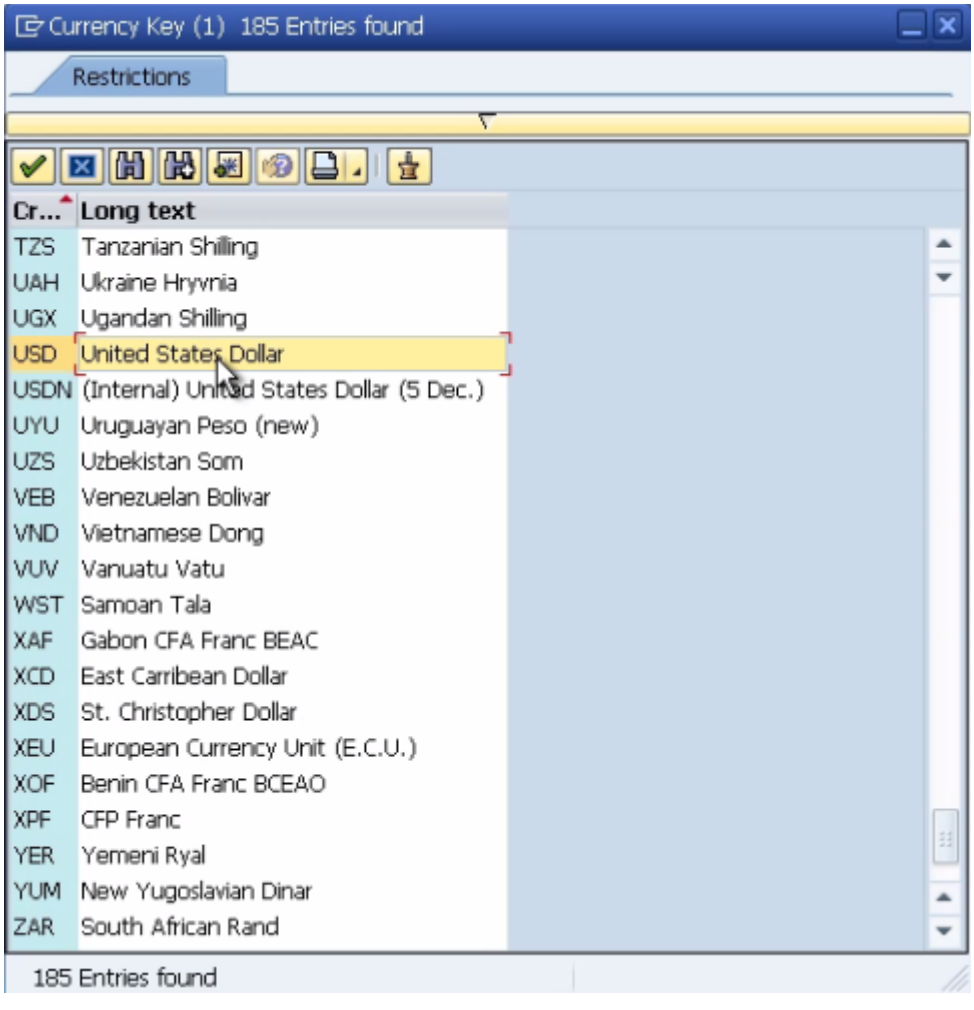
Return to the 'Maintain table' screen for ZEMPLOYEES2, highlight the ECURRENCY field, and click the Foreign key button visible in the toolbar above: 

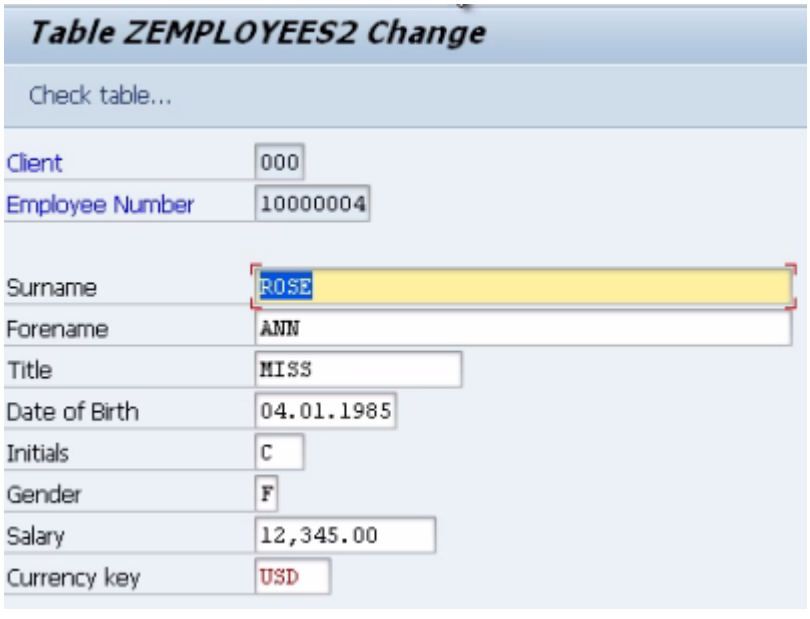
Choose 'Yes' in the box which appears and a 'Create Foreign Key' window will emerge. Type the short text 'Check Currency Field'. A small table is visible, detailing the two key fields from the TCURC table and the ZEMPLOYEES2 table. The option is available to ensure that the foreign key matches both fields, so that when the user is allowed to select an entry, the records returned will only be valid for the Client which is being worked in.

Here though, the Client is not to be chosen as part of the key, so select the Check-box 'Generic' for the top row, which refers to the Client, and remove the text from the two boxes on this row where this is possible. Then click the 'Copy' button. The foreign key will be created:



Activate the table, and then browse the data. Now, select the currency key and either press the F4 key or select the drop-down box that appears, displaying all valid entries for this field. If you were in record change mode you will then be able to select a value from the table and see it update your zemployees 2 record. Try it out and select USD (US Dollar).



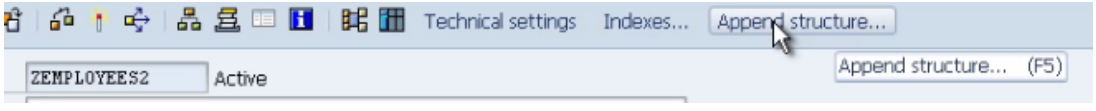


Append Structures

Having looked at foreign keys, the next thing to look at are Append structures. These can be used to add additional fields. This is the preferred method for maintaining SAP delivered tables and quite often for customer-specific tables. If one does not use Append structures, problems can arise if, for example, a new version of SAP is used which does not correspond with aspects of the tables already created, resulting in serious errors.

Append structures give a safe way to enhance tables. When these are used, the initial table remains unchanged, removing any risk of changes being overwritten later if a different version of SAP is used. Quite often, a table may have multiple Append structures applied to it, because different development needs have arisen as time has gone by and people have wanted to add further fields to the standard SAP tables.

In the SE11 Maintain Table screen, go to the 'Append structure' button on the right of the top toolbar:

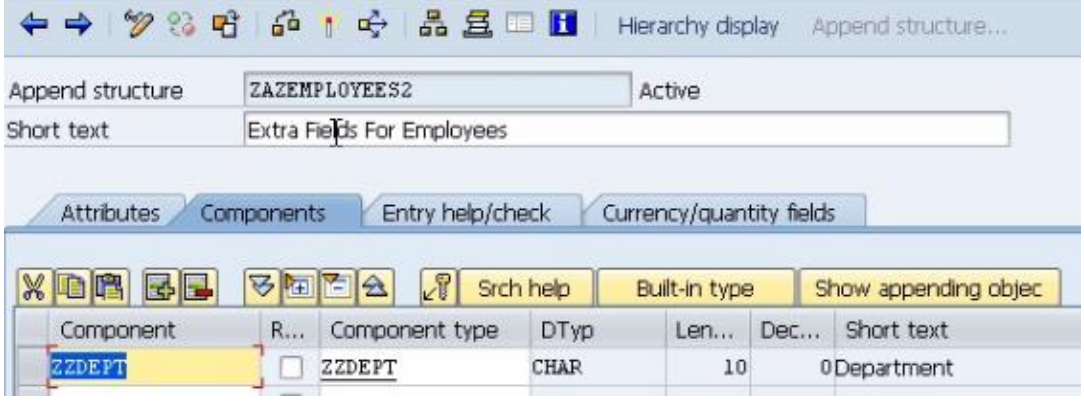


Click this, and the system will suggest a name, ZAZEMPLOYEES2 (note that this, again, must begin with a Z). Accept this and you will be presented with what looks like an empty table structure. Enter the Short text “Extra Fields For Employees”, and then move down to the table.

Note that the first field now is called ‘Component’. This is where new fields are created. However, it may be useful to differentiate between fields created in the main table, and the new components created here in the Append structure. Since both must comply with the customer name rules, where Z was used in the main table, here use ZZ.

For the first component, a ‘Department’ field will be created. Type in the ‘Component’ box ‘ZZDEPT’ and the same again in ‘Component type’. For this Component type, use forward navigation in the same way that it was used for the Data element before, double-clicking to create. Save the Append structure as a local object when prompted, and then select to create a Data element when prompted subsequently.

The familiar data element screen will now appear. Type ‘Department’ for the short text, use CHAR10 for the domain and ‘Department’ again for the Field labels, then activate the data element. Step back to the Append structure screen, then Activate:



Return to the main table screen, where a new row displaying the Append structure will have been created. To then access this structure, simply double-click the row. In Change mode only the ‘.APPEND’ line will be visible by default, but in Display mode the fields created within this will appear below:

SALARY	☐	☐	ZSALARY	CURR	9	2 Salary
ECURRENCY	☐	☐	CURCY	CUKY	5	0 Currency Key
.APPEND	☐	☐	ZAZEMPLOYEES2		0	0 Extra Fields For Employees

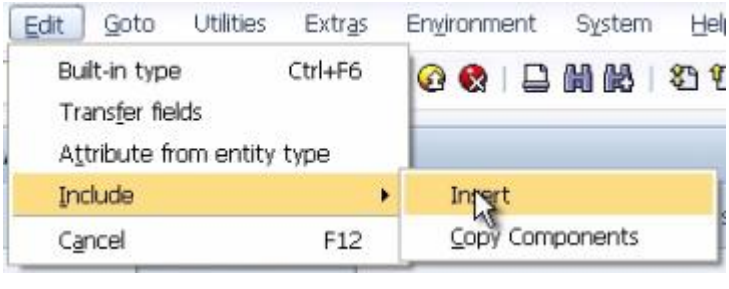
This is a very useful way to add new fields to a table without affecting the structure of the table itself. If one then browses the data as normal, a new column will have been called 'Department'. Data can then be entered into this field just like it can for any other:

Salary	Currency key	Department
0.00		
0.00		
0.00		
12,345.00	USD	

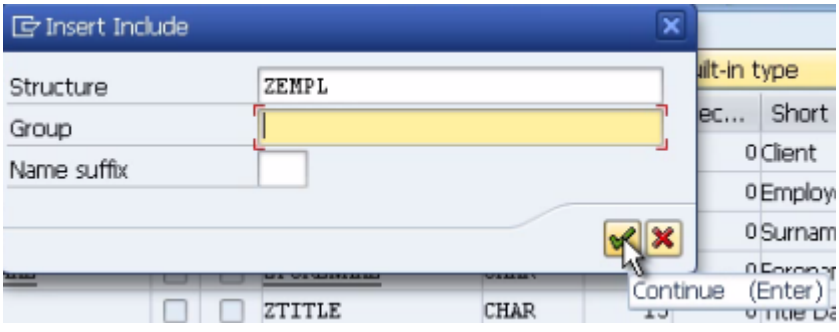
Include Structures

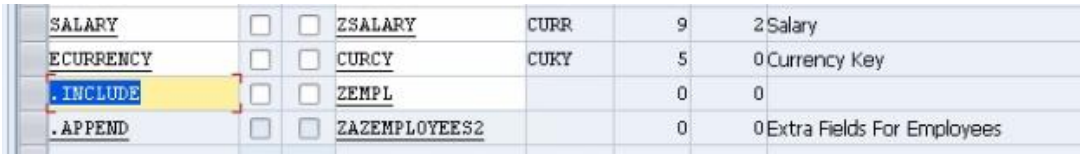
Include structures are similar to Append structures, with the main difference being that they are re-usable objects and can be linked to many other tables, ABAP programs, dialogue programs and structures. It is important to keep in mind that Include structures must be flat structures, meaning that they cannot hold any additional structure within them, and that the maximum length of the fields within an include structure is 16 characters.

There is no Include structure button in the way that there is an Append structure button. To create one, first ensure Change mode is selected. Where the cursor is placed is important here, as wherever the cursor is when the Include structure is created, it will be created one row above. If you want the Include structure to be part of the table key, it must appear at the top, because all table fields used as a table key need to be grouped together at the top. In this instance though, it will just be inserted above the Append structure. Place the cursor on the '.APPEND' row, select the 'Edit' menu, then 'Include' and 'Insert'.



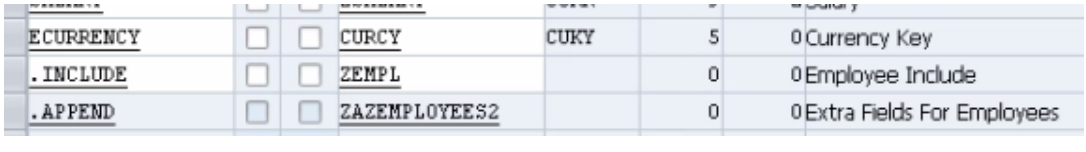
In the window that appears, enter ‘ZEMPL’ in the ‘Structure’ field and click the continue button. A warning box will appear stating that this is not yet active, dismiss this, and the Include structure should now appear in the table:



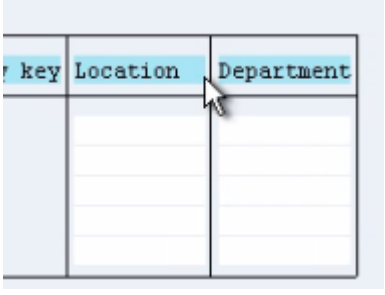


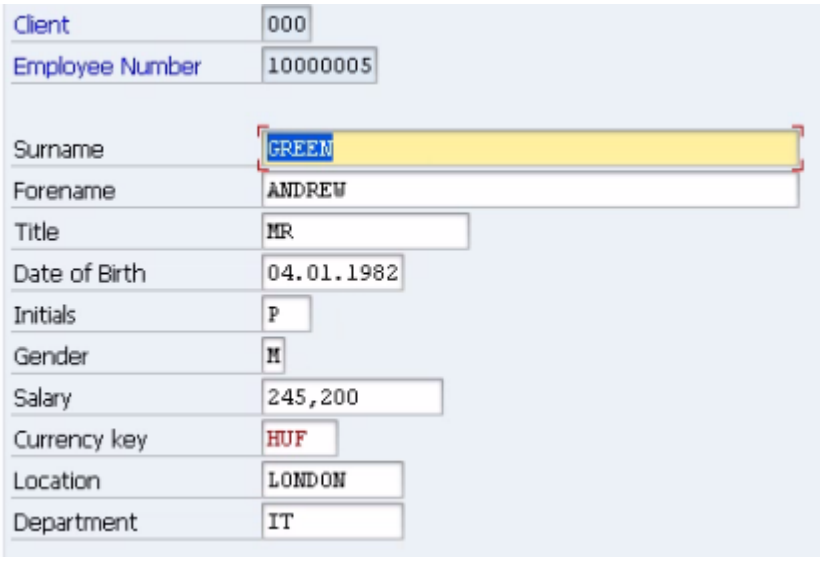
To add a field to this, use forward navigation as before, double-clicking where ‘.INCLUDE ZEMPL’ appears, save and choose ‘Yes’ to create the structure. The screen which then appears is very similar to the Append structure screen.

Type the Short text “Employee Include” and begin to create a field (the boxes are, like in the Append structure, labelled ‘Component’), this time for location, called ZZLOCAT, and use ZLOCAT for the ‘Component type’. Use forward navigation again to create this Data element with Short text ‘Location’, the domain CHAR10 and ‘Location’ again for the Field labels, then Activate this as usual. Activate the Include structure once the field has been created and return to the main table to see the Include structure located just where we wanted it, above the Append structure:

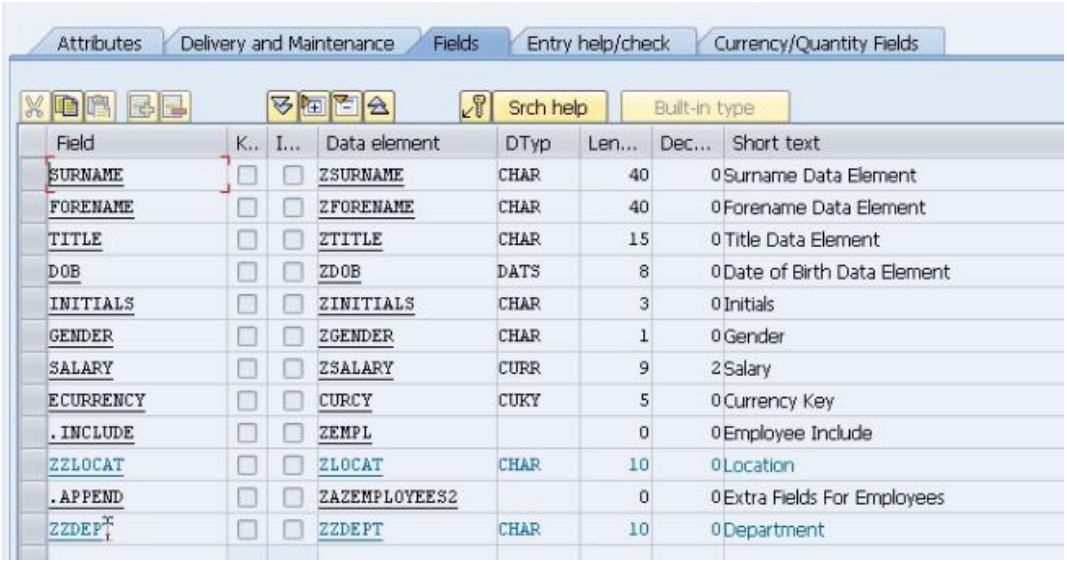


Activate the table now, and view the contents. The Location column should now be visible, and these records can now be edited and created like any other:



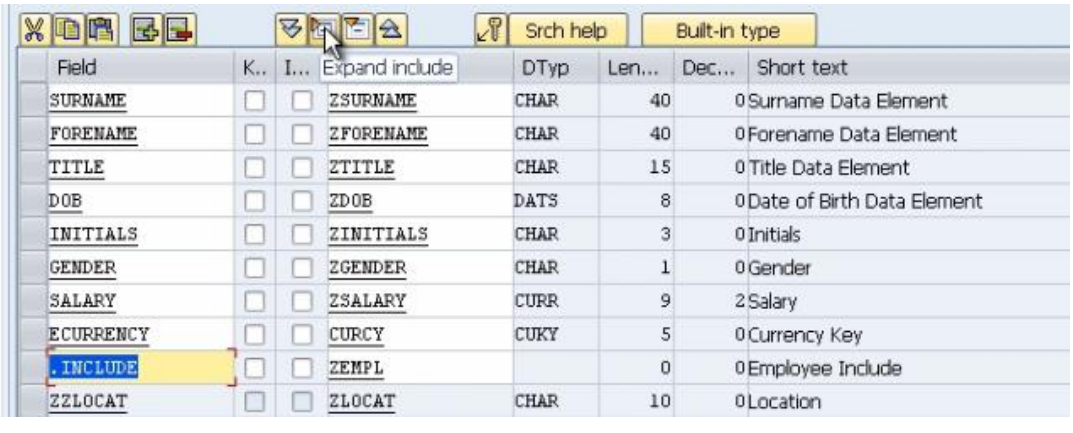


If one switches to Display mode, the field created in the Include structure can be seen in the context of the main table, albeit in a different colour:



Field	K..	I...	Data element	DTyp	Len...	Dec...	Short text
SURNAME	☐	☐	ZSURNAME	CHAR	40	0	Surname Data Element
FORENAME	☐	☐	ZFORENAME	CHAR	40	0	Forename Data Element
TITLE	☐	☐	ZTITLE	CHAR	15	0	Title Data Element
DOB	☐	☐	ZDOB	DATS	8	0	Date of Birth Data Element
INITIALS	☐	☐	ZINITIALS	CHAR	3	0	Initials
GENDER	☐	☐	ZGENDER	CHAR	1	0	Gender
SALARY	☐	☐	ZSALARY	CURR	9	2	Salary
ECURRENCY	☐	☐	CURCY	CUKY	5	0	Currency Key
.INCLUDE	☐	☐	ZEMPL		0	0	Employee Include
ZZLOCAT	☐	☐	ZLOCAT	CHAR	10	0	Location
.APPEND	☐	☐	ZAZEMPLOYEES2		0	0	Extra Fields For Employees
ZZDEPT	☐	☐	ZZDEPT	CHAR	10	0	Department

In Change mode, these fields can be seen by selecting the '.INCLUDE' row and clicking the 'Expand include' icon (the same works for the Append structure also):



Field	K..	I...	DTyp	Len...	Dec...	Short text
SURNAME	☐	☐	ZSURNAME	CHAR	40	0 Surname Data Element
FORENAME	☐	☐	ZFORENAME	CHAR	40	0 Forename Data Element
TITLE	☐	☐	ZTITLE	CHAR	15	0 Title Data Element
DOB	☐	☐	ZDOB	DATS	8	0 Date of Birth Data Element
INITIALS	☐	☐	ZINITIALS	CHAR	3	0 Initials
GENDER	☐	☐	ZGENDER	CHAR	1	0 Gender
SALARY	☐	☐	ZSALARY	CURR	9	2 Salary
ECURRENCY	☐	☐	CURCY	CUKY	5	0 Currency Key
.INCLUDE	☐	☐	ZEMPL		0	0 Employee Include
ZZLOCAT	☐	☐	ZLOCAT	CHAR	10	0 Location

Key Fields

If you want to add or remove fields which are designated key fields, then it is important to take into consideration what will be going on in the database itself. All of the new elements which have been created for this table have their features applied by the system to the ABAP dictionary, not the underlying database. When any key field is adjusted, the system has to apply changes to the underlying database itself. If there is data in the table, and key fields are changed, this can have unintended consequences.

If you introduce a new key field, this will probably not have a large effect. However, if one makes a key field no longer a key field, this will require consideration, because if there is a lot of data in the underlying database, by taking away a key field, duplicate records could be introduced. Corrupt data or records being deleted from the table can also happen here.

Let's see how we can add, remove and alter fields without these hazards.

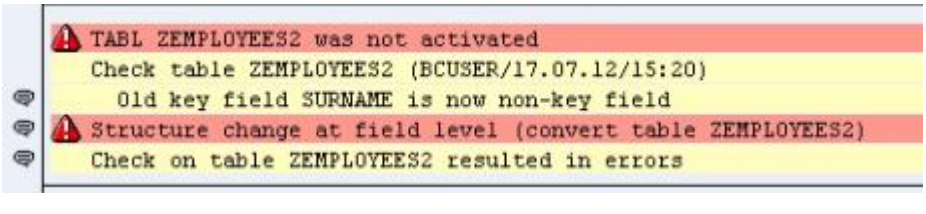
Open the full ZEMPLOYEES2 table in the ABAP Dictionary 'Maintain Table' screen. Let's change the 'Surname' field by turning it into a key field.

Check the two boxes (key and Index) by 'SURNAME' and Activate the table. When you now view the table contents, the surname column will be a darker colour, indicating that it is now a key field. Beyond this though, it appears very little has changed:

Displayed fields: 12 of 12 Fixed columns: List width 0250

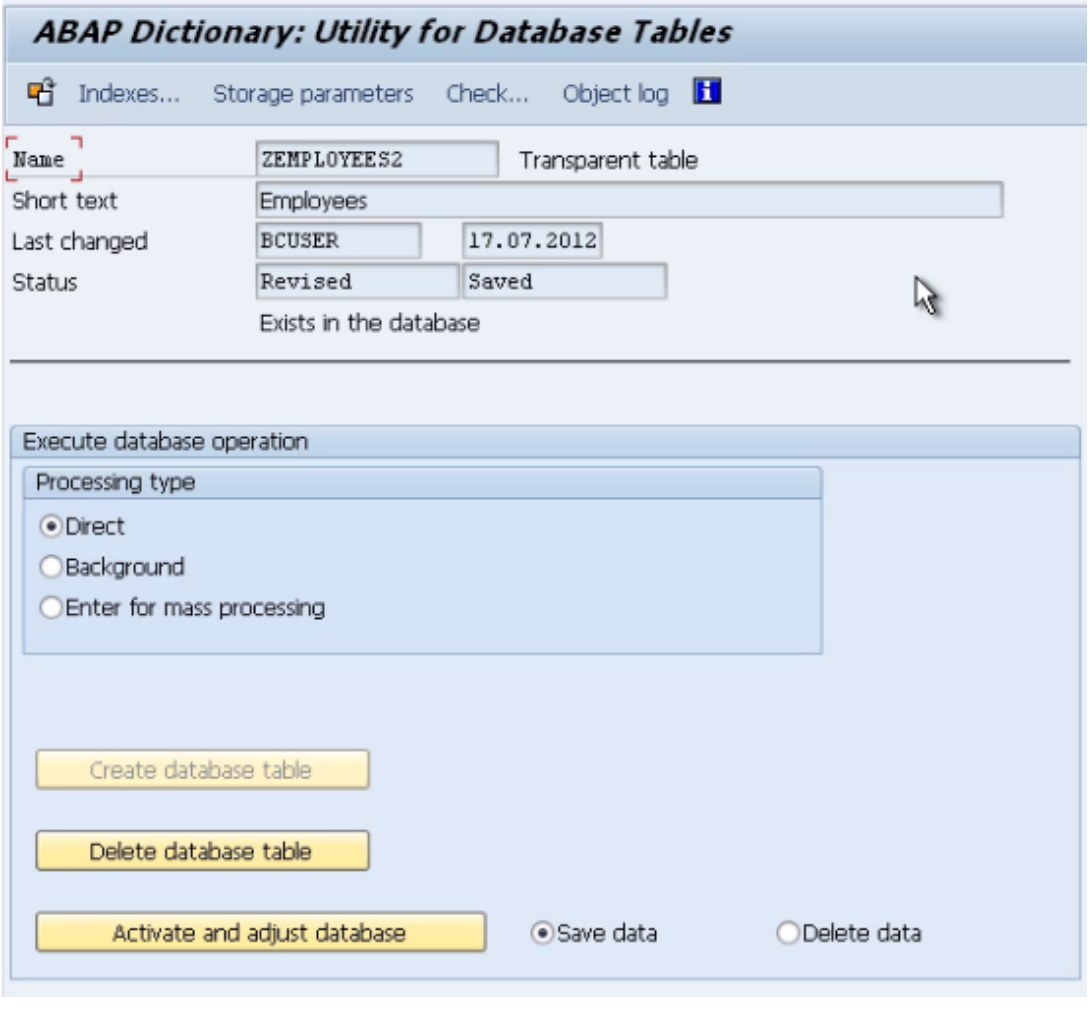
Client	Employee Number	Surname	Forename
000	10000001	SMITH	PAUL
000	10000002	BROWN	IAN
000	10000003	WILLIAMS	SARAH
000	10000004	ROSE	ANN
000	10000005	GREEN	ANDREW

Now, uncheck the boxes on the 'Maintain Table' screen, to make it no longer a key field. When you try to activate the table an error message appears, refusing to activate the table as data may be lost with the removal of a key field:



To activate the table against what seem to be the wishes of the system (after all, one knows the data will be fine as the surname field has not been operating as a key field at any point previously), a different transaction must be used.

From the 'Utilities' menu, select 'Database utility', or use transaction code SE14. A new screen will appear:



This transaction lets us automatically adjust the data held in our table when making adjustments to the database table structure. Environments where tables are being worked on may contain a huge number of records. With this in mind, this transaction can be executed as a background process. However, for our example the 'Direct' option is the option

to choose because we know we have very few records in our database table. Select this, and then click 'Activate and adjust database' with 'Save data' radio button selected. Say 'Yes' when the box asks "Request: 'Adjust'" and notice the status bar should indicate the success of this execution. Then, step back to the 'Maintain Table' screen and you will see the table should be Active with the surname field no longer key.

To insert a new field as part of the table key, you must be able to adjust the location of fields on the screen. For example, if you wanted to create a new field above the surname field, you would highlight the row and then click the 'Insert row' icon in the toolbar. *This toolbar also includes 'Cut', 'Copy' and 'Paste' options, allowing for rows to be moved up and down if there is a need to do this*:

Deleting Fields

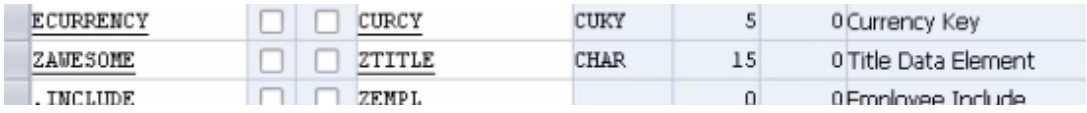
While infrequent, occasionally there may be a need to remove a field from a table. When doing this, it is important to take special care, as data can be lost in the process. Certainly in the case of key fields.

If, for example, the Currency key field was removed from our table, the foreign key relationship to the TCURC table would be removed. As the SALARY field has to have a related Currency Key this would cause the table to no longer continue working, and likely make the ZEMPLOYEES2 table become inactive.

When deleting fields it is important to ask oneself whether the data being held in the table is being used elsewhere, and whether its deletion will have further consequences. If you do try to delete fields which are being used elsewhere, the SAP system should try to prevent this, or at least issue a stern warning. This is not necessarily to be relied upon though, so always ensure to check manually what the effects of deletion are likely to be. Also, if

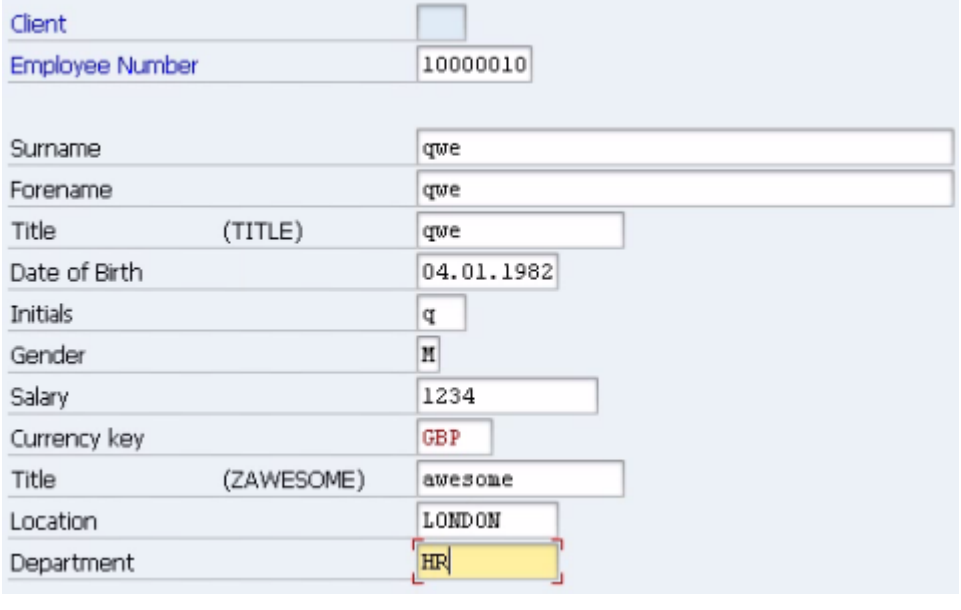
you do delete fields, the table will have to be adjusted via the SE14 transaction to be activated again.

Create a new field, above '.INCLUDE', named 'ZAWESOME'. Use a previously created Data element, here ZTITLE just to save time, and activate the table:

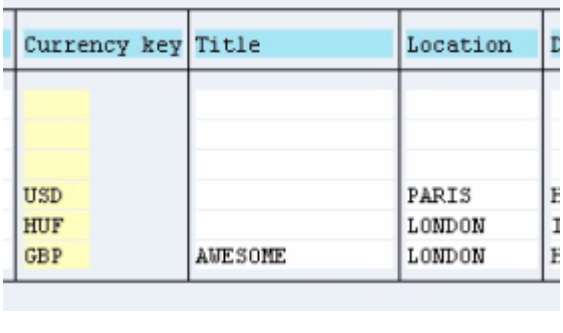


ECURRENCY	☐	☐	CURCY	CUKY	5	0	Currency Key
ZAWESOME	☐	☐	ZTITLE	CHAR	15	0	Title Data Element
.INCLUDE	☐	☐	ZEMPL		0	0	Employee Include

Create a new record in the table. The data here is not important and will be deleted, so the content can be anything:

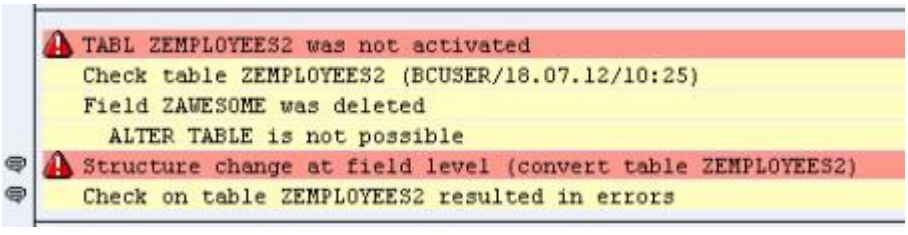


Client		
Employee Number		10000010
Surname		qwe
Forename		qwe
Title	(TITLE)	qwe
Date of Birth		04.01.1982
Initials		q
Gender		M
Salary		1234
Currency key		GBP
Title	(ZAWESOME)	awesome
Location		LONDON
Department		HR

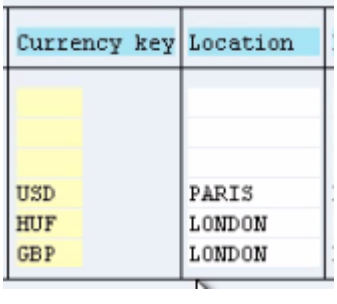


Currency key	Title	Location
USD		PARIS
HUF		LONDON
GBP	AWESOME	LONDON

Now, to delete the field, highlight it in the 'Maintain Table' screen, and click the 'Remove row' icon, in the toolbar next to 'Insert row'. The row will disappear, but when you try to activate the table, an error message will appear:

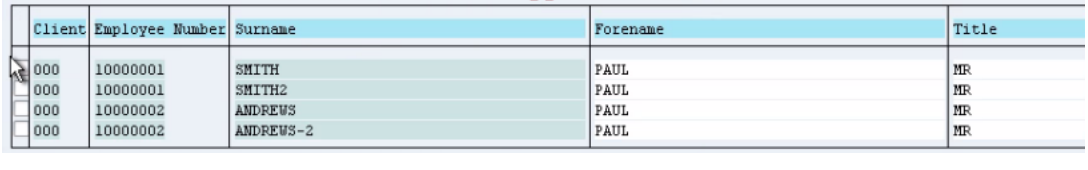


Transaction SE14 must again be used to adjust the table so the change can be applied. Follow the same steps as in the previous section to perform this task. Once this is complete, view the table again. The column has disappeared, and the data which was contained within it lost:



Currency key	Location
USD	PARIS
HUF	LONDON
GBP	LONDON

To see what happens when a key field is deleted, return to the ABAP Dictionary initial screen and make a copy of ZEMPLOYEES2, called, unsurprisingly, ZEMPLOYEES3. *Doing this will allow the ZEMPLOYEES2 table to not be damaged in this risky procedure*. Activate the new table (which, don't forget, will be empty of records). As before, again make the Surname field a key field. Now create some records for this table:



Client	Employee Number	Surname	Forename	Title
000	10000001	SMITH	PAUL	MR
000	10000001	SMITH2	PAUL	MR
000	10000002	ANDREWS	PAUL	MR
000	10000002	ANDREWS-2	PAUL	MR

To save time creating new records, the same data was replicated here, with only slight changes to the key fields. Remember that it is only one key field per entry which must be unique for that particularly record to be unique itself.

Now, the surname field will be deleted, and the effects of deleting this key field observed. By removing this key field, the only unique data which will be held for each record will be the Employee Number and Client. Since SMITH and SMITH2, and ANDREWS and ANDREWS-2 have the same Employee Number and Client, these will no longer hold unique key field data, leaving duplicate records, which the system will not allow.

Remove the Surname field; try to activate the table, and error messages will appear. Go through SE14 to adjust the table for activation. When you now view the table, the Surname field is gone, and two records have been lost, leaving only one of the two records for each of the two Employee Numbers used:

Client	Employee Number	Forename	Title	Date
000	10000001	PAUL	MR	01.01
000	10000002	PAUL	MR	01.01

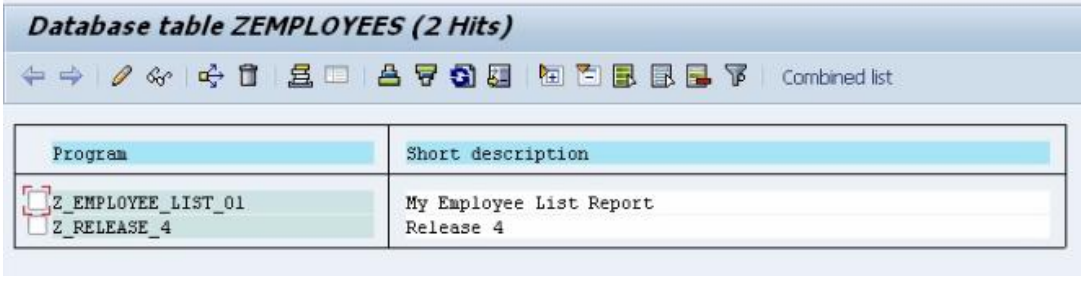
Deleting Tables

One will not often have to delete an entire database table, for largely the same reasons as were outlined above for fields. If this does have to be done it is important to remember that one's own customer-specific tables are the only ones which can be deleted, SAP delivered tables cannot be deleted. Because ZEMPLOYEES3 has only just been created, and nothing else depends on this table, it can be deleted without consequences.

To check whether a table can be deleted without causing unintended consequences elsewhere in the system, return to the ABAP Dictionary's initial screen. Because the original ZEMPLOYEES table was used in the programs which have been created, use this as a test.

Insert this into the Database table field on the screen and then click the 'Where-used list' icon from the toolbar.

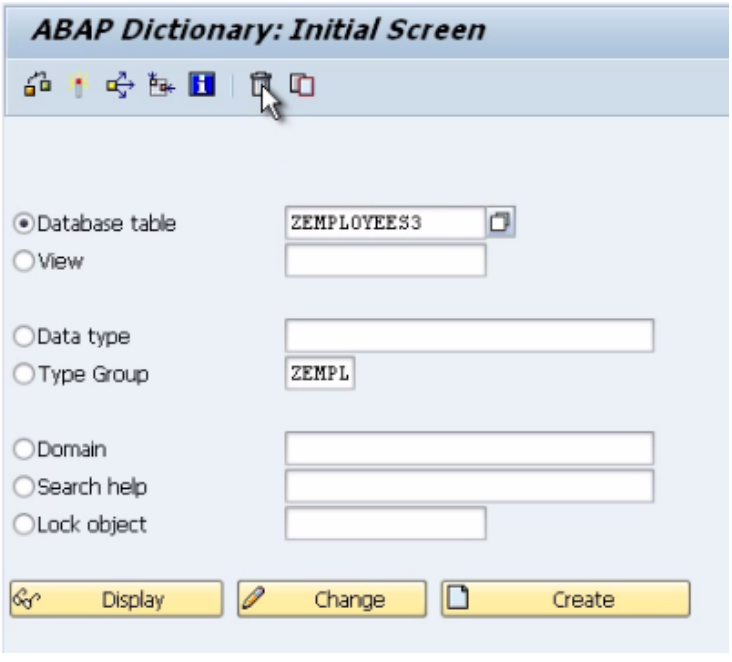
Once this is clicked, a dialogue box will appear offering a list of check-boxes. This will then search all of the different areas of the SAP system selected for references to the table ZEMPLOYEES. To execute this search click the Continue icon. Choose 'Yes' to the pop-up box, and wait while the system compiles the search results, which here show that this table is being used currently by 2 programs:



Having done this, one now knows that if the ZEMPLOYEES table were to be deleted, these programs would become inactive. By double-clicking these entries, one can see the code in the program where ZEMPLOYEES is referred to, and if you double-click on any line of the program, it will open the program at that line of code in the ABAP Editor. The Where-used button is a very useful tool, which can be invaluable not just when deleting programs, but in many other scenarios.

If you were to try to delete ZEMPLOYEES, the system would not allow this course of action and would prevent it from happening until all the programs that are dependent upon it were either edited to remove references or deleted altogether themselves.

Since nothing depends upon ZEMPLOYEES3, this can be deleted. With the correct name in the 'Database table' field, click the 'Delete' button in the toolbar:



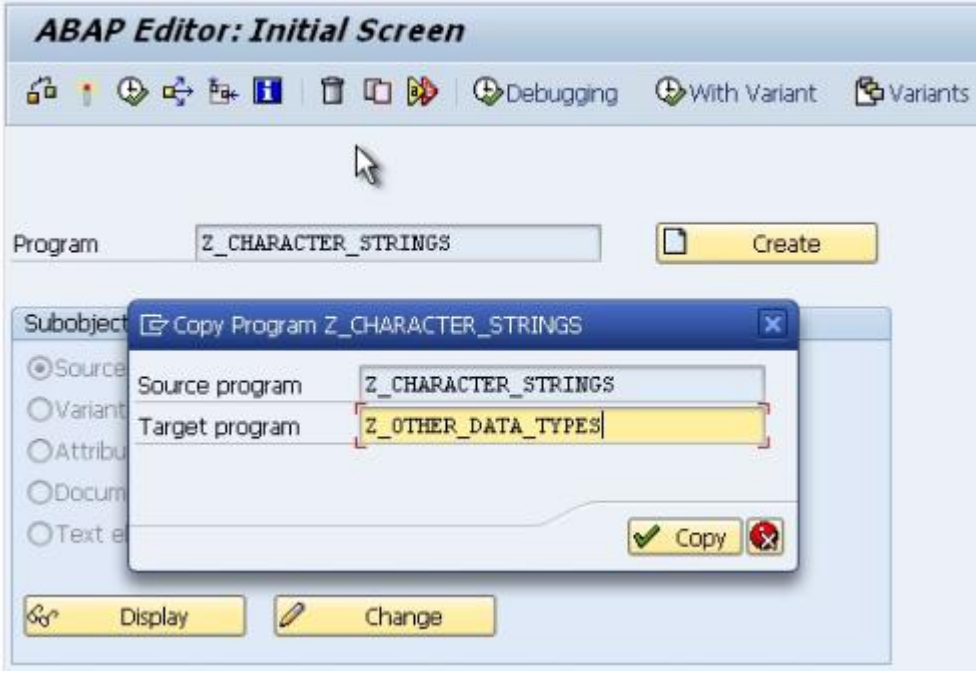
A box appears stating that the data contained in the table would also be deleted. If you click the green tick icon this time, the system would return to the main screen with the table still intact. If the middle button, illustrated with the trashcan icon is clicked, this will proceed with the deletion. Once this is done, the status bar should confirm the action. If you try to display the table now, it does not exist. Once the deletion is completed, it cannot be undone:

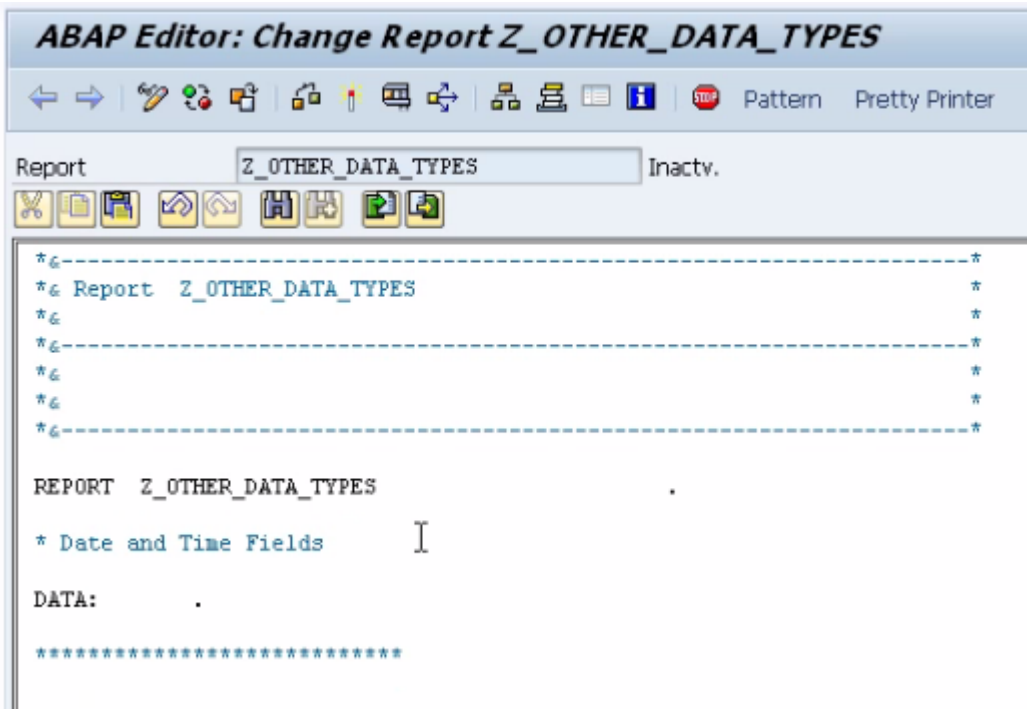
Chapter 8 – Working with Other Data Types

Date and Time Fields

This section will look at some other data types which can be used in ABAP. So far, numeric fields have been used for performing calculations, and character strings have been examined along with the ways these can be manipulated with ABAP statements. Now, date and time fields will be looked at.

Enter the ABAP editor (with transaction SE38) and make a copy of the previous program, alter the comment sections, and remove most of the code:





Date and time fields are not stored as numeric data types, but instead as character data types. Effectively, they are character strings which can be used in calculations. This is made possible by the inbuilt automatic data type conversions which have previously been discussed. Just like any other data type, the DATA statement is used to declare these fields.

For a date field, the data type is referred to with 'd', and is limited to 8 characters. The first 4 of these represent the year, the next 2 the month, and the final 2 the day. The VALUE addition is used to specify this, and if it is not used then the value, by default, is assigned as 8 zeros. In the example below, the date is the 1st of January, 2012:

```
REPORT  z_other_data_types                        .

* Date and Time Fields

* Date fields format: YYYYMMDD with initial value of '00000000'
DATA my_date TYPE d VALUE '20120101'.

*****************************
```

The LIKE statement, of course, can also be used. SY-DATUM is a system variable, which always holds the value of the system's date. Below, "my_date2" is defined in the same way as this system variable:

```
DATA my_date2 LIKE SY-DATUM.
```

Time fields work similarly, but this time are limited to 6 characters. The first 2 refer to the hour, the second 2 the minute, and the final 2 the second. Again, the default value will be 6 zeros. The data type this time is 't'. Again, the LIKE statement can be used, here for the system's time field, referred to with SY-UZEIT:

```
* Time fields format: HHMMSS with initial value of '000000'
DATA my_time TYPE t VALUE '111005'.

DATA my_time2 LIKE sy-uzeit.

*****************************
```

We can then use the WRITE statement to output the field contents:

```
WRITE: my_date,
       / my_date2,
       / my_time,
       / my_time2.
uline.
*****************************
```

```
01012012
00.00.0000
111005
00:00:00
```

Note that in the first row the my_date field has reversed itself to the format DDMMYYYY. In the second, no value was assigned to the field, so the system has output the default zeros. However, as this was defined like the system's date variable, it has included periods in the formatting. This also applies to the my_time2 field, where colons have appeared between the places where the time values would ordinarily be.

Date Fields in Calculations

Some examples of performing calculations with date and time fields will now be looked at. Using these fields in calculations is common practice within programming business systems, as one will often have to, for example, find the difference between two dates to deliver invoice dates, delivery dates and so on. Here, examples will be looked at so as to find new dates, and find the difference between two dates.

Use the DATA statement to declare a start date for an employee, called “empl_sdate”, and then give this a value of ‘20090515’. Then create another field called “todays_date” and define the value of this as ‘sy-datum’, the system variable, which should then include the date on that particular day:

```
DATA empl_sdate  TYPE d.
DATA todays_date TYPE d.
```

```
empl_sdate = '20090515'.
todays_date = sy-datum.
```

Next, a calculation will be added, so as to work out this employee’s length of service. Create a new variable named “LOS”, include a DATA statement giving “LOS” a data type ‘i’ and then define LOS as the calculation ‘todays_date – empl_sdate’. Then, add a WRITE statement for this variable, which will include the employee’s length of service in the output. Once this is complete, execute the code:

```
"************************
DATA empl_sdate  TYPE d.
DATA todays_date TYPE d.
DATA LOS type i.

empl_sdate = '20090515'.
todays_date = sy-datum.
los = todays_date - empl_sdate.
WRITE / los.
```

```
1,160
```

If one wants to add, for example, 20 days to today’s date, the same value is used for todays_date (the system variable, sy-datum). Create another variable, called “days_count” with an integer value of 20, and another called “fut_date”. This variable’s value should then be defined as ‘todays_date + days_count’, then ad a WRITE statement to output the

fut_date. Don't forget also to add the data types above ('i' for days_count and 'd' for fut_date). The output should give the date 20 days on from today's date, which here is the 7th of August, 2012:

```
todays_date = sy-datum.
days_count = 20.
fut_date = todays_date  + days_count.
WRITE / fut_date.
```

```
DATA days_count TYPE i.
DATA fut_date TYPE d.
************************
```

```
07082012
```

Subfields can be used for date fields in exactly the same way as they were used before. In the next example, a date field will be changed to represent the 20th day of the current month. Copy the todays_date variable, then add a new line of code which changes the last two figures of todays_date to the value '20', and a WRITE statement. Also, output the system date so as to compare the two:

```
todays_date = sy-datum.
todays_date+6(2) = '20'.
WRITE / sy-datum.
WRITE / todays_date.
```

```
18.07.2012
20072012
```

In this next example, the last day of the previous month will be established. Use the todays_date variable again, this time using the subfield method above to change this to represent the first day of the current month. Then on a new line of code, subtract one from this, so that the todays_date variable is now the final day of the previous month:

```
todays_date = sy-datum.
todays_date+6(2) = '01'.
todays_date = todays_date - 1.
WRITE / todays_date.
```

```
30062012
```

Time Fields in Calculations

Calculations like those above can also be performed with time fields.

In the examples, employees' clocking in and out times will be used. Use DATA statements to declare the variables "clock_in" and "clock_out" as type 't', along with others seen in the image below, which will be used for calculations to work out the differences between times in seconds, minutes and hours, all of an integer type:

```
* Field for Time Calculations
DATA clock_in     TYPE t.
DATA clock_out    TYPE t.
DATA seconds_diff TYPE i.
DATA minutes_diff TYPE i.
DATA hours_diff   TYPE i.
```

Assign values to clock_in and clock_out of '073000' and '160000' respectively. Then, to work out the difference between the two in seconds, use the calculation 'clock_out - clock_in' and assign this value to "seconds_diff". Then include some WRITE statements to output this information:

```
* TIME CALCULATIONS
clock_in = '073000'.
clock_out = '160000'.
seconds_diff = clock_out - clock_in.

WRITE: / 'clock in: ', clock_in, '   clock out: ', clock_out.
WRITE / seconds_diff.
```

```
clock in:  073000    clock out:  160000
    30,600
```

To establish the difference in minutes, simply use the seconds_diff value, and divide this by 60, and then to establish the hour's difference, follow this by dividing minutes_diff by 60:

```
minutes_diff = seconds_diff / 60.
WRITE: / 'difference in minutes: ', minutes_diff.

hours_diff = minutes_diff / 60.
WRITE: / 'difference in hours: ', hours_diff.
```

```
difference in minutes:          510
difference in hourss:           9
```

Note that here, the 510 minutes do not, in fact, equal 9 hours exactly, the system has rounded the number. This is because the hours_diff variable was declared as an integer. If the data type for this is changed to a packed decimal, the value would have been established as the more accurate 8.5 hours:

```
DATA minutes_diff TYPE i.
DATA hours_diff   TYPE p decimals 2.
```

```
difference in hours:                8.50
```

Quantity and Currency Fields in Calculations

Now, a look will be taken at using quantity and currency fields in calculations. In ABAP, these are treated the same as packed number fields. Currency fields must be declared as data type 'p', bearing in mind how many decimal places are required. This is important, as having the right number of decimal places can have a large impact on the accuracy of calculations.

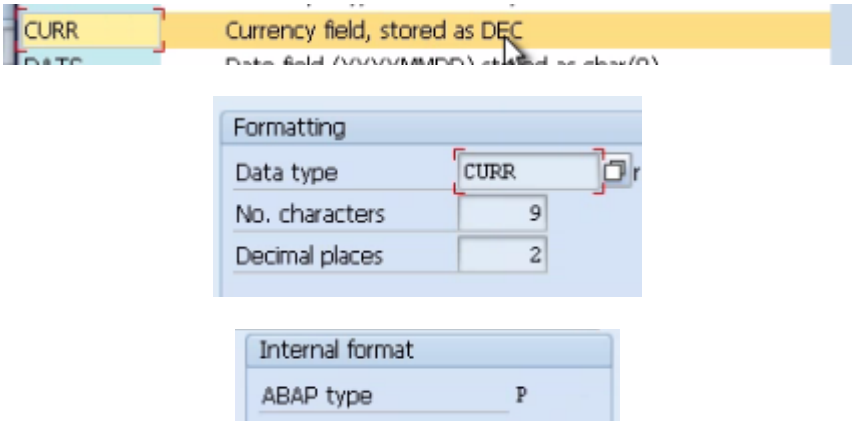
Quite often in a program, one wants to create one's own variables for quantity and currency fields. It is usually better, however, to associate these fields with the data types of those in a table created in the ABAP dictionary. This is because the ABAP dictionary will already have defined the correct field length and number of decimal places for these. For example, the Salary field in the table created previously had defined two decimal places. If a currency field in a program is declared to match this field but the data type in the program is set manually to 2 decimal places and the number of decimal places in the table was to change, the program would no longer operate properly here. For this reason, it is usually preferable to use the LIKE statement for these fields.

In this example a new variable named "my_salary" has been declared using the LIKE statement:

```
* Field for Currency Calculations
DATA my_salary LIKE zemployees2-salary.
```

Because this field in the program is linked to the field in the table, the system will ensure these data types are kept in sync. There are two aspects to this process, the number of

decimal places, and the associated currency (or quantity) keys. If you look at the CURR data type in the ABAP dictionary, you will see that this is stored as a decimal - 9 characters and 2 decimal places. You can also see that its internal format is ABAP type p, packed decimal:



Additionally, don't forget that the salary field and its currency data type always refer to the currency key field, in the table called ECURRENCY. Ultimately, then, when one is declaring fields in ABAP, it is important to reference these to the associated fields in a table, and when working with currencies, the currency key field will always be there and should be taken into account. The same applies to quantity fields. The only difference is their data type is QUAN, and rather than a currency key, will always have a UNIT associated with them.

Now, using calculations from the currency field, an employee's tax and net pay amounts will be established, so declare two more DATA statements for these fields, again referencing the salary field in the table. Also add a tax percentage variable, of type p with 2 decimals:

```
* Field for Currency Calculations
DATA my_salary  LIKE zemployees2-salary.
DATA my_tax_amt LIKE zemployees2-salary.
DATA my_net_pay LIKE zemployees2-salary.
DATA tax_perc   TYPE p decimals 2.
```

Add a TABLES statement so that the program knows to refer to the ZEMPLOYEES2 table, then observe the calculations in the code below:

```
REPORT  z_other_data_types

TABLES: zemployees2.
```

```
tax_perc = 0.20.
SELECT * FROM zemployees2.
WRITE: / zemployees2-surname, zemployees2-salary, zemployees2-ecurrency.
  my_tax_amt = tax_perc * zemployees2-salary.
  my_net_pay = zemployees2-salary - my_tax_amt.
  WRITE: / my_tax_amt, zemployees2-ecurrency,
           my_net_pay, zemployees2-ecurrency.
ENDSELECT.
```

First, the tax percentage is established. This is in this example 20%, so for the means of the calculations is written as 0.20. Then the code will select records from the ZEMPLOYEES2 table, and write the surnames, salaries and currencies for these. Next, the tax amount is established, by multiplying the tax percentage by the salary. Net pay is equal to the salary, minus the tax amount. Then add a WRITE statement to output the results the end of the SELECT loop. The output should look like this (where salaries and currencies are not present in the table, go back and edit the records in your table to put some values):

```
SMITH                                      1,111.00  ATS
     222.20  ATS        888.80  ATS
BROWN                                      2,222.00  BDT
     444.40  BDT      1,777.60  BDT
WILLIAMS                                   6,423.00  FJD
   1,284.60  FJD      5,138.40  FJD
ROSE                                      12,345.00  USD
   2,469.00  USD      9,876.00  USD
GREEN                                      2,452.00  HUF
     490.40  HUF      1,961.60  HUF
QWE                                        1,234.00  GBP
     246.80  GBP        987.20  GBP
```

The surname, salary and currency for each record are written on the first line, followed by the tax amount and net pay on the following line. To make this look tidier, descriptive text can be added to the WRITE statements in the code:

```
SMITH                                        1,111.00  ATS
tax amount:       222.20  ATS   net amount:       888.80  ATS
BROWN                                        2,222.00  BDT
tax amount:       444.40  BDT   net amount:     1,777.60  BDT
WILLIAMS                                     6,423.00  FJD
tax amount:     1,284.60  FJD   net amount:     5,138.40  FJD
ROSE                                        12,345.00  USD
tax amount:     2,469.00  USD   net amount:     9,876.00  USD
GREEN                                        2,452.00  HUF
tax amount:       490.40  HUF   net amount:     1,961.60  HUF
QWE                                          1,234.00  GBP
tax amount:       246.80  GBP   net amount:       987.20  GBP
```

Chapter 9 – 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 profiles 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 accesses 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 tables 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 records.

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 statements, 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.

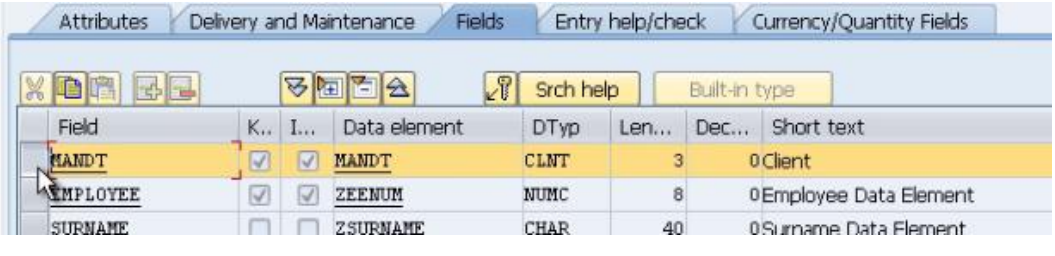
When the lock object is created, the system automatically creates two function modules, which will be discussed later. These function modules are simply modularised ABAP programs that can be called from other programs. The first of these has the action of setting a lock, and the second releases this 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:



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:

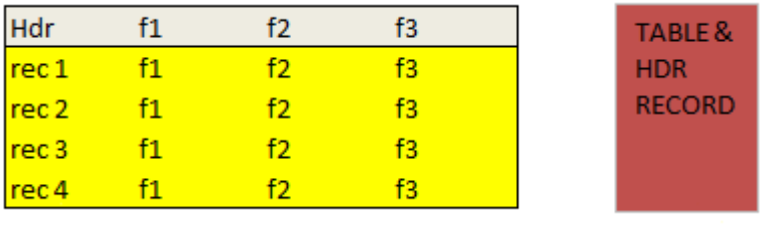
```
INSERT zemployees.
```

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.

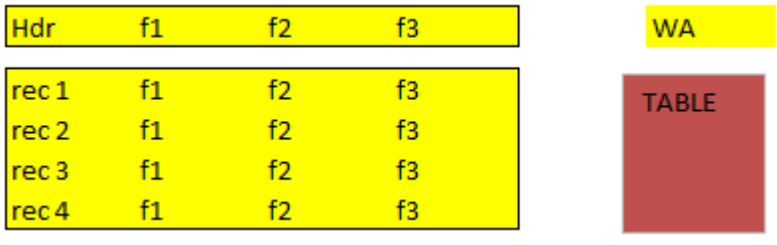
In the above statement, nothing is specified to be inserted. This is where the concept of the work area enters. The statement here expects a work area to exist which has been created when an internal table was declared. This type of work area is often referred to as a header record:



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 commonly 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

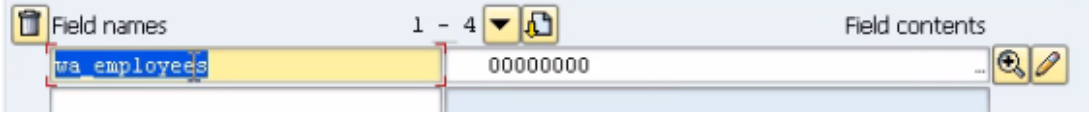
For practice use the ABAP debugger to execute the code step-by-step. First, delete the record from the table in the ABAP dictionary and put a breakpoint in the code at the beginning of the record entry to the work area:

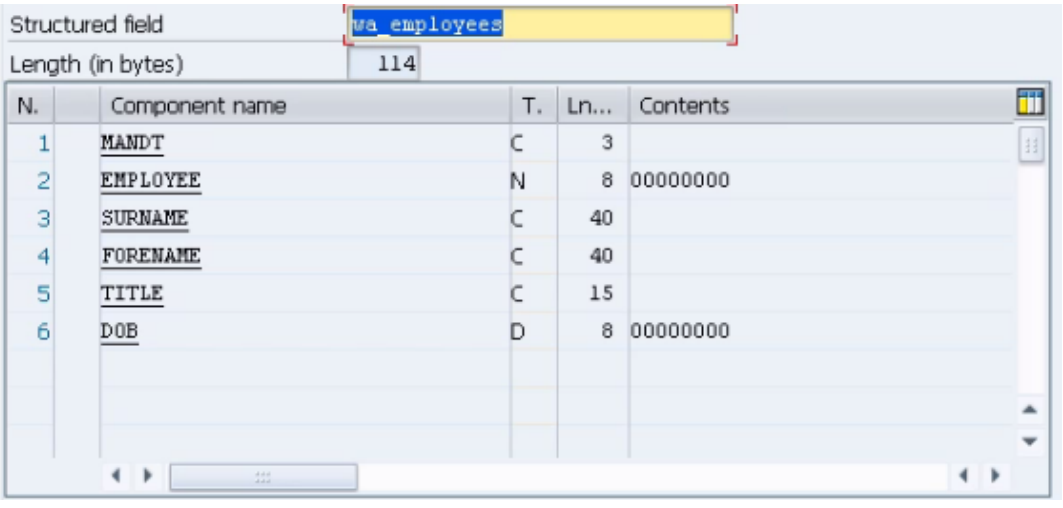
```
DATA wa_employees LIKE zemployees.

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:

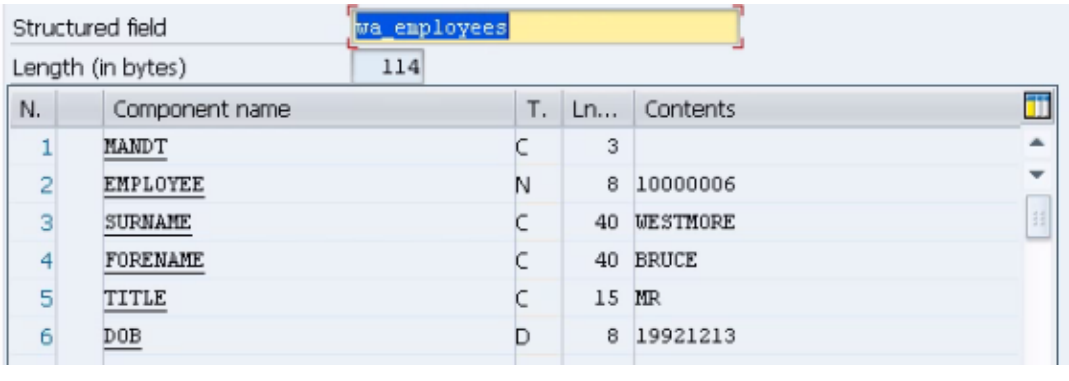
```
DATA wa_employees LIKE zemployees.

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:





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:



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.

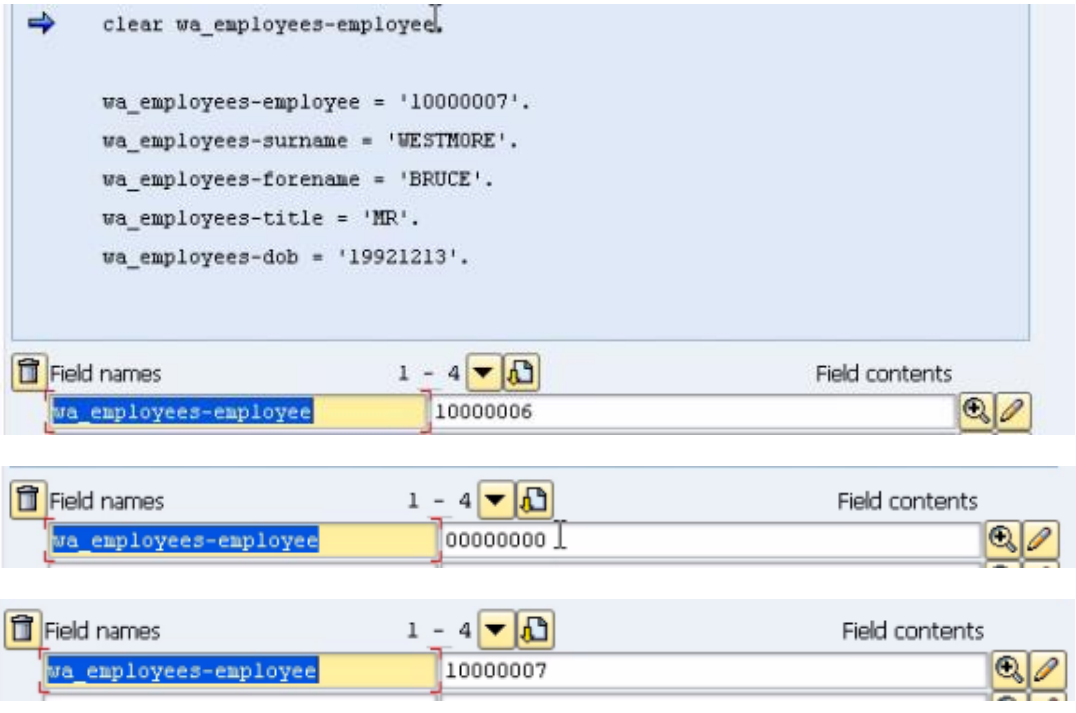
To do this, the new line of code would read “CLEAR wa_employees.”

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:



Update Statement

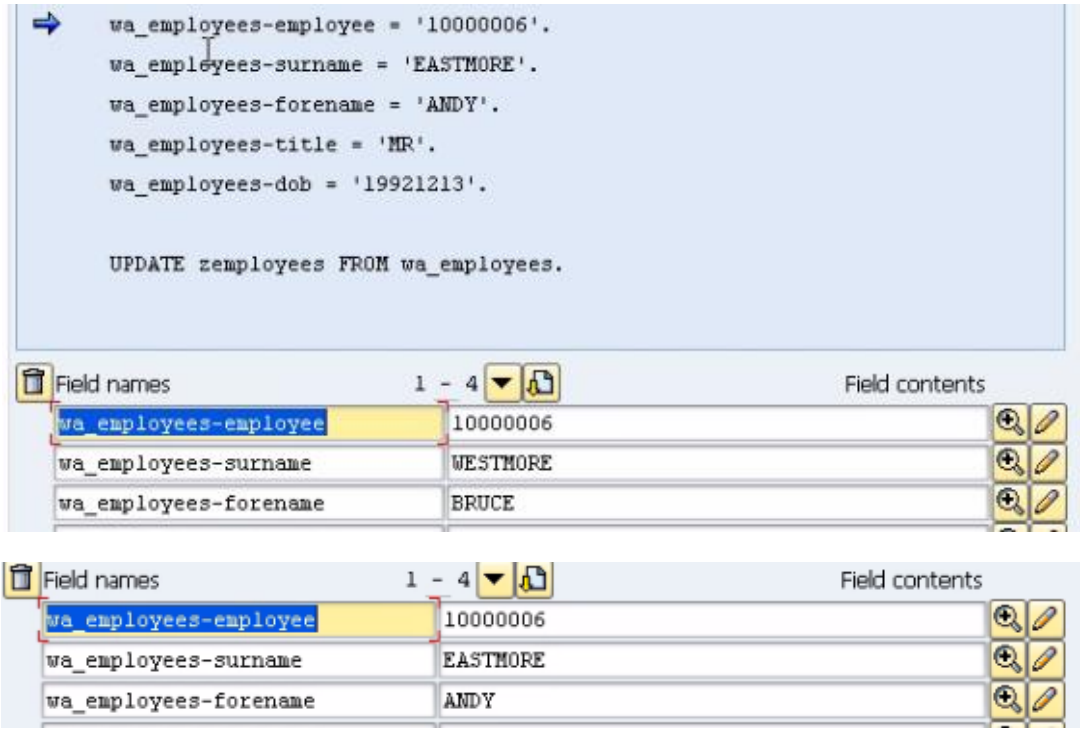
The UPDATE statement allows one or more existing records in a table to be modified at the same time. In this example it will just be applied to one, but for more the same principles generally apply.

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 previously 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:



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:

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. Generally, 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 appropriate, for example of one is writing code where a record must be inserted or updated depending 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 another 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 record 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 deleted. 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 syntax 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 appear 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

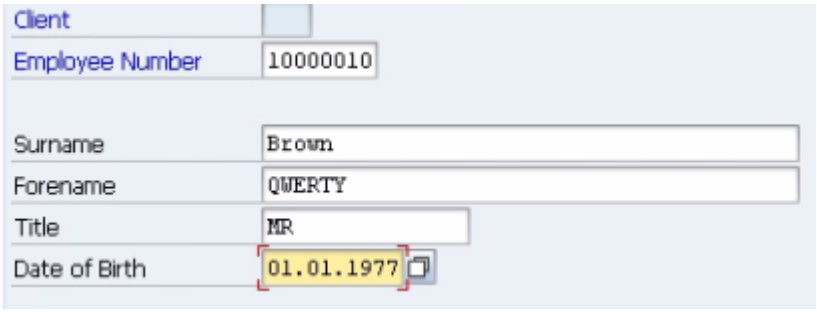
```
Record Deleted Correctly
```

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:



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:

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:

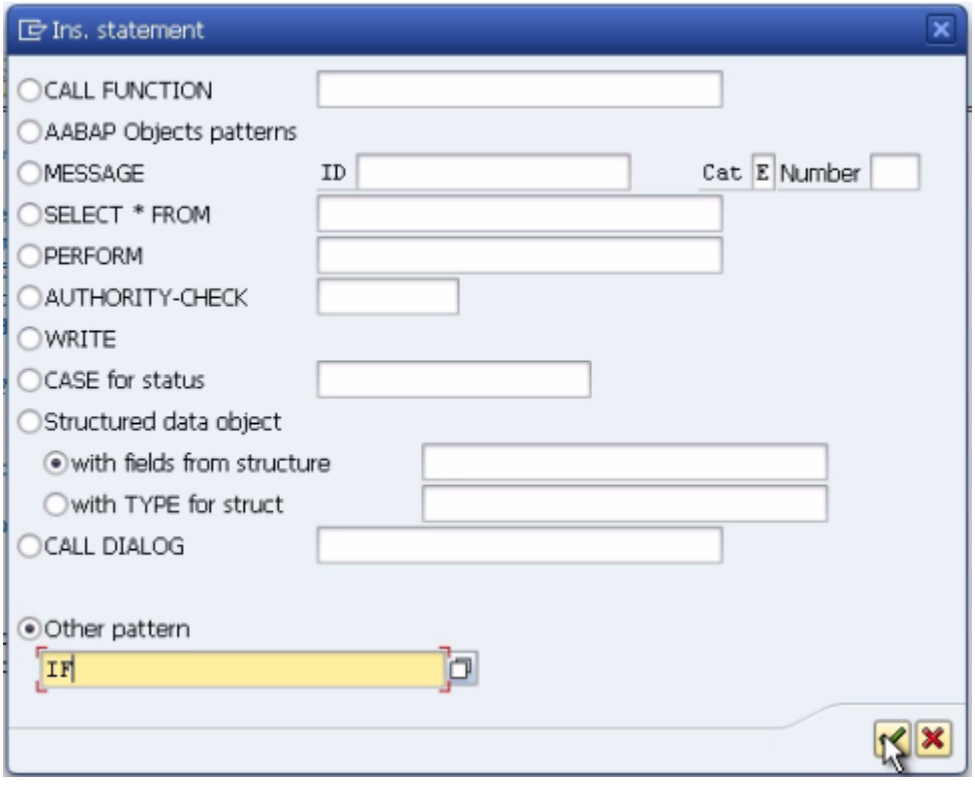
```
IF sy-subrc = 0.
  WRITE 'Record Inserted Correctly'.
ELSE.
  WRITE: 'We have a return code of ', sy-subrc.
ENDIF.
```

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:




```
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 statements. 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.
```

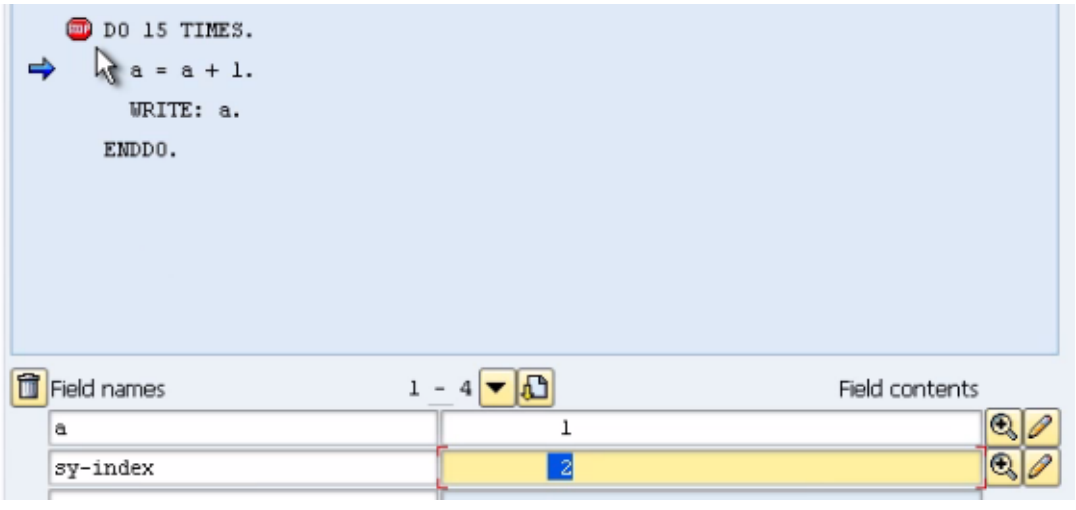
```
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

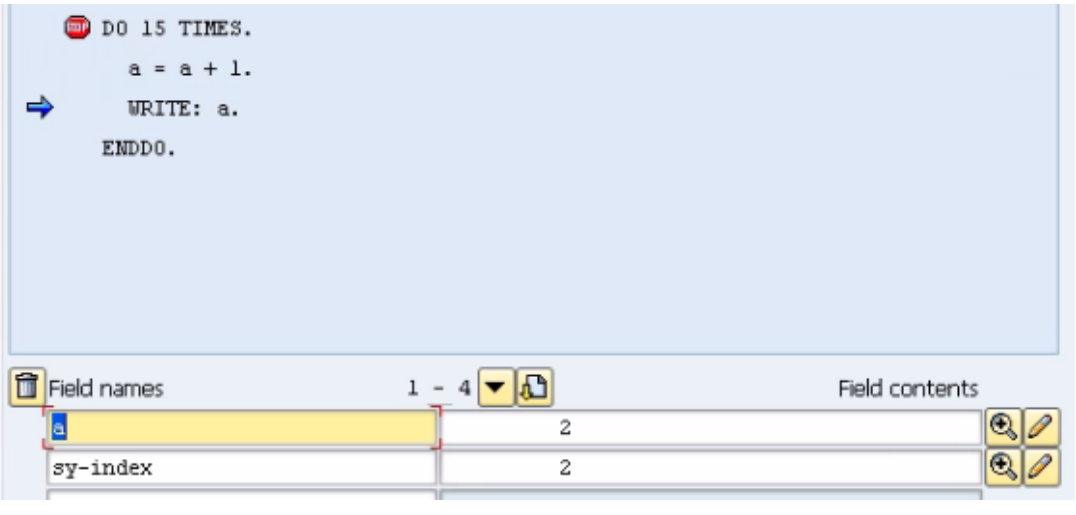
The DO statement is useful for repeating a particular task a specific number of times. Just remember to always include the TIMES addition.

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”.

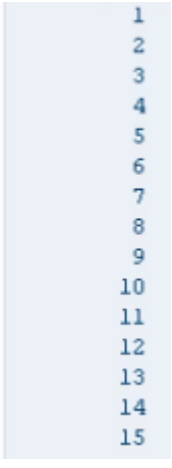
```
DATA: a TYPE i.

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:





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:



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.

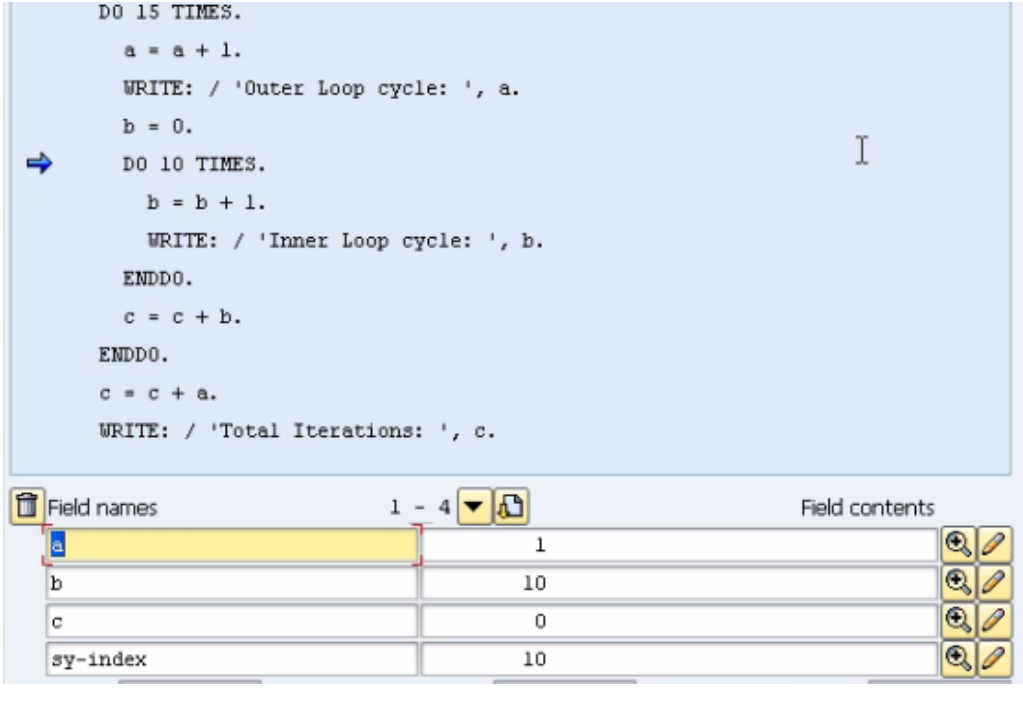
Add to the WRITE statement from the previous section a small amount of text reading 'Outer Loop cycle:' before outputting the value of 'a'. This will allow 'a' to be monitored.

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:

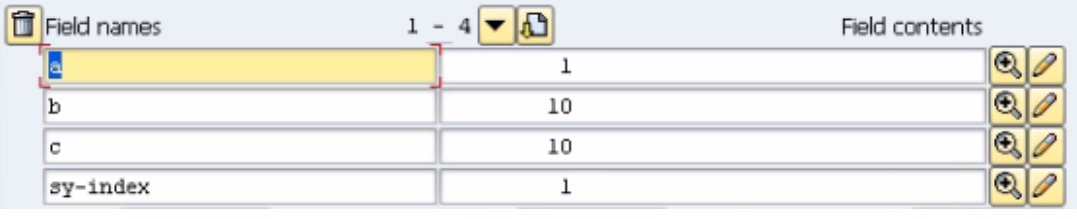
```
DATA: a TYPE i,
      b TYPE i,
      c TYPE i.

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 begin 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').

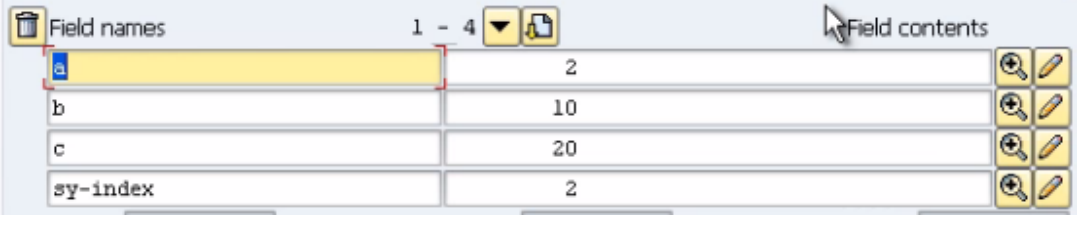


Then the full loop has completed once, the sy-index field displays 1 and the ‘c’ field has been filled in:

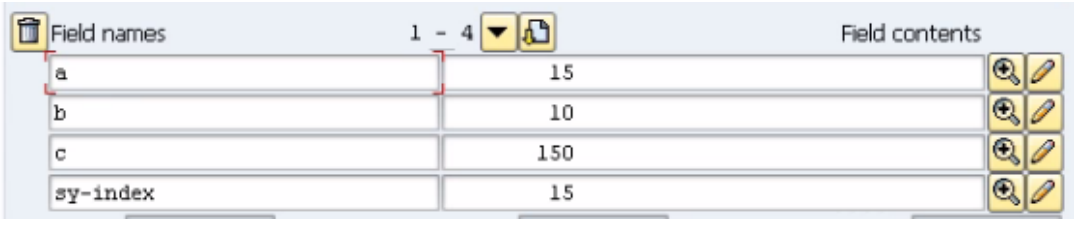
```
DO 15 TIMES.
  a = a + 1.
  WRITE: / 'Outer Loop cycle: ', a.
  b = 0.
  DO 10 TIMES.
    b = b + 1.
    WRITE: / 'Inner Loop cycle: ', b.
  ENDDO.
  c = c + b.
ENDDO.
c = c + a.
WRITE: / 'Total Iterations: ', c.
```



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:



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



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

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'.
```

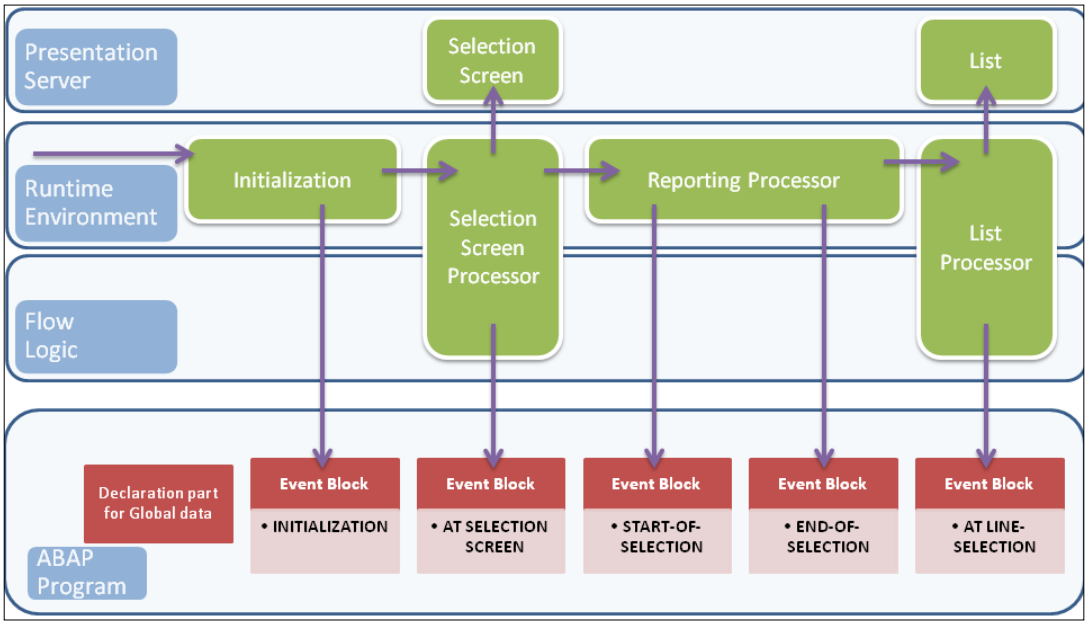
```
Outer Loop cycle:           1
Outer Loop cycle:           2
Filler
Filler
```

Chapter 11 – 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 displayed again until correct values are entered. Error messages can be included so that the user then knows where corrections must be made.

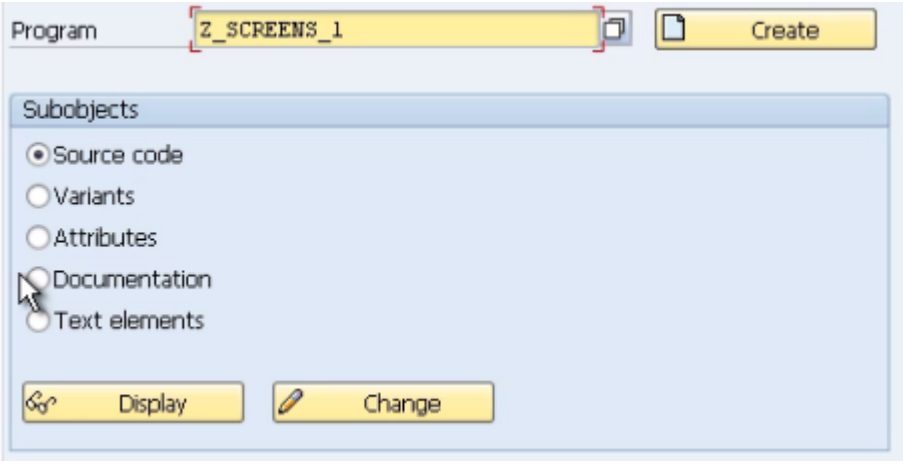
The 'start of selection' event block then takes control once the selection screen is filled correctly. This can contain code for, for example, setting up the values of internal tables or fields. There are other event blocks, which are visible in the diagram and there could be a number of others. The 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 processor, which will output the report to the screen for the user to see. The list screen occasionally 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 interface.

Intro to Selection Screens

ABAP reports have 2 types of screens, selection screens and list output screens. The output 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 selection 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 creates 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 system 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 inside 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 allow 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.

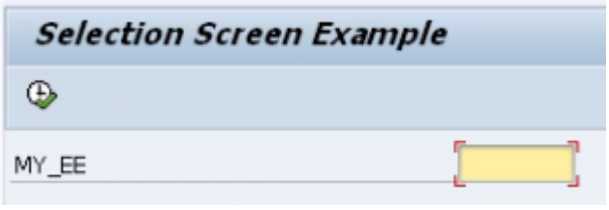
After this, an IF statement can be written, displaying an error message if the parameter value my_ee entered by the user is greater than the value held in wa_employee, the last employee number used:

```
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 properties 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.
```

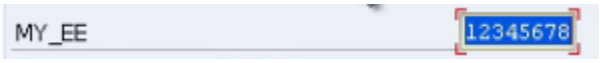
The DOB parameter takes on the same attributes as the *DOB* field in the table, to the extent that it will even offer a drop-down box to select a date. The *my_numbr* parameter is not related to another field as has been declared as an integer type parameter. Additionally, note that parameter names are 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.

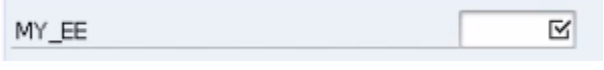
```
PARAMETERS: my_ee LIKE zemployees-employee DEFAULT '12345678'.
```

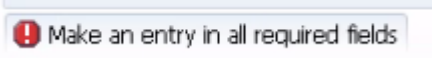


OBLIGATORY

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.
```



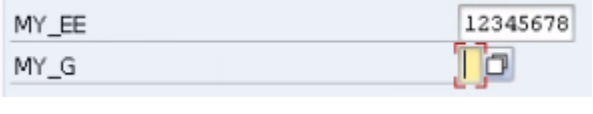


Automatic Generation of Drop-Down fields

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:

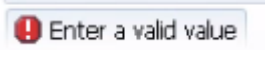
```
TABLES: zemployees, ZEMPLOYEES2.

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:



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.
```



(*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.
```

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.

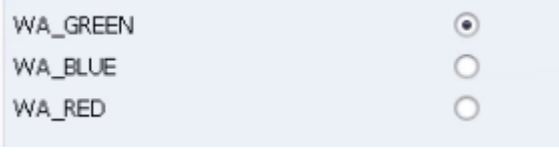
Define a new parameter called my_box1. Since this is type c, the type does not have to be declared. The field name is then followed by "**as checkbox**". Note that the output differs slightly from other parameters by seeing the box on the left and the text to its right:

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

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.
```



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 setting 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.

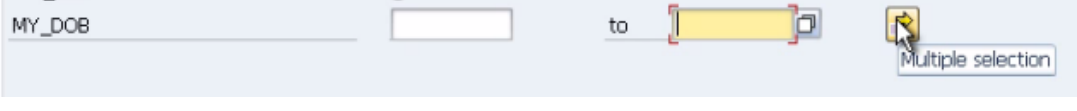
The “low” field holds the lower limit for a range of values a user can enter, while the “high” field is the upper limit. The type and length of these will be the same as those for the database table to which the selection criteria are linked.

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 selection 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:

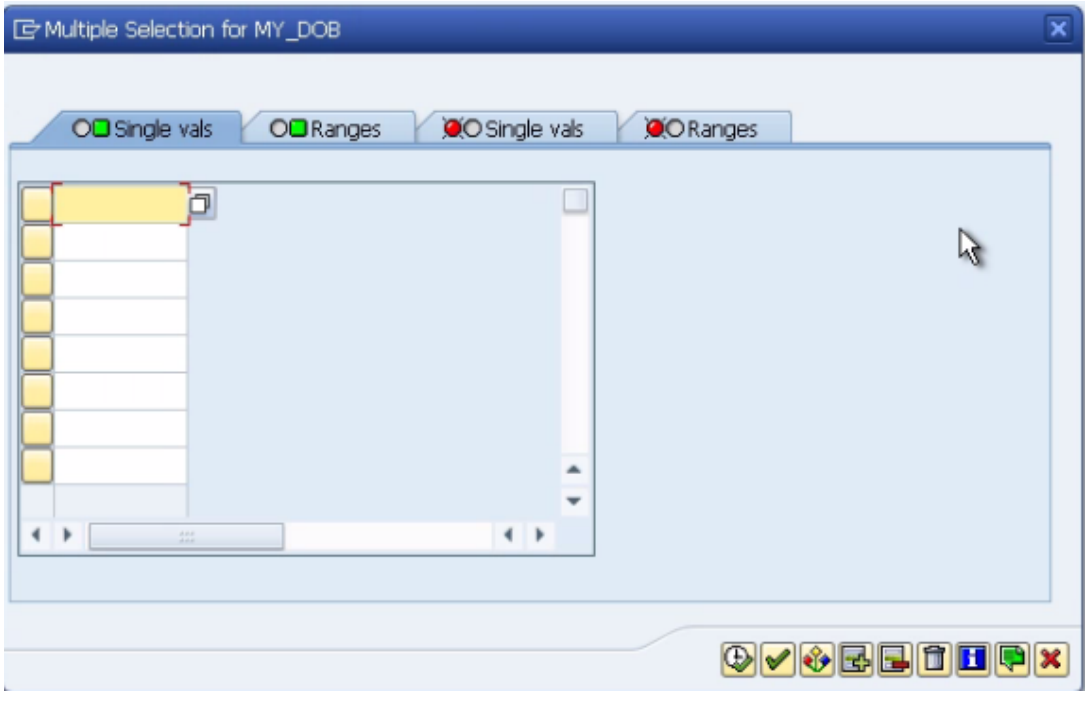
```
SELECT-OPTIONS my_dob FOR zemployees-dob.
```

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

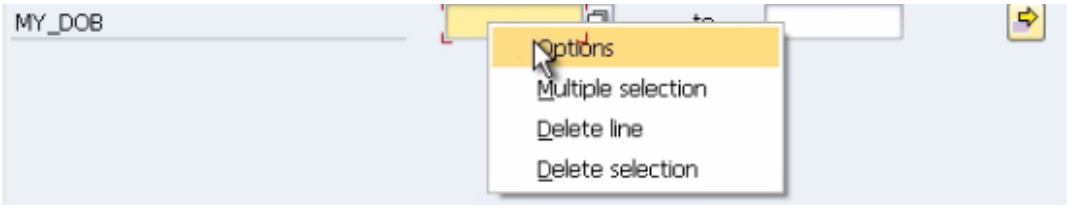


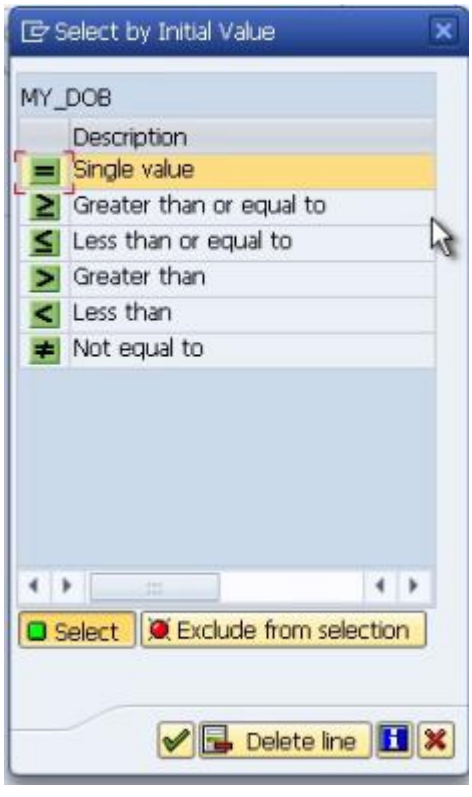
A value range can be selected by entering the low value into the left field and the high value in the right field. These two fields both include calendar drop down menus, making

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. Additionally 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

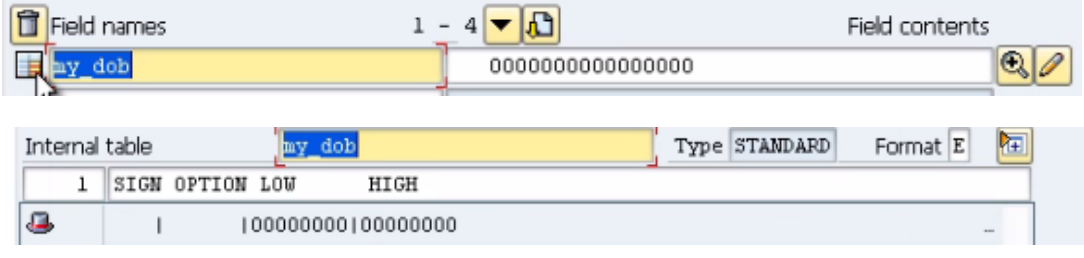
With the select-options defined, some code will now be added.

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:

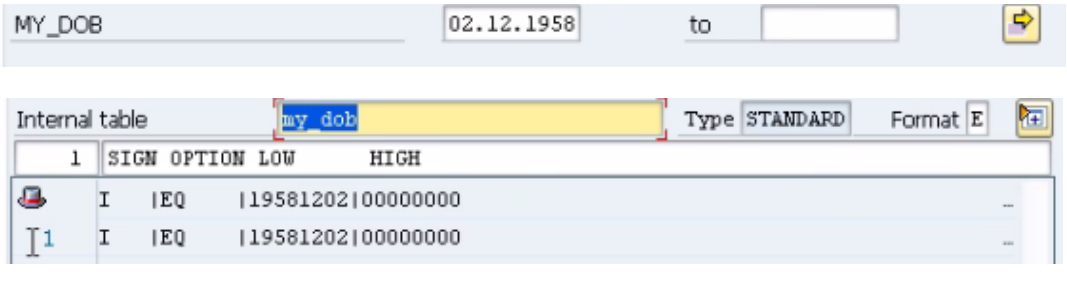
```
SELECT * FROM zemployees.
  IF zemployees-dob IN my_dob.
    WRITE: / zemployees.
  ENDIF.

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 appear. 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 values were inserted:

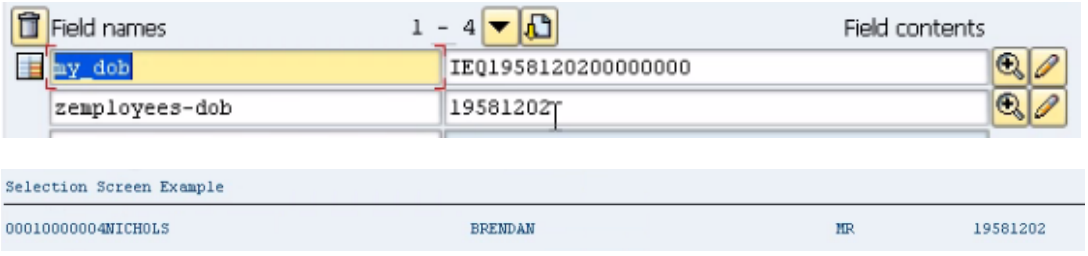


Run through the code and all of the records from the table should be written to the output screen, as no specific selection criteria were set.

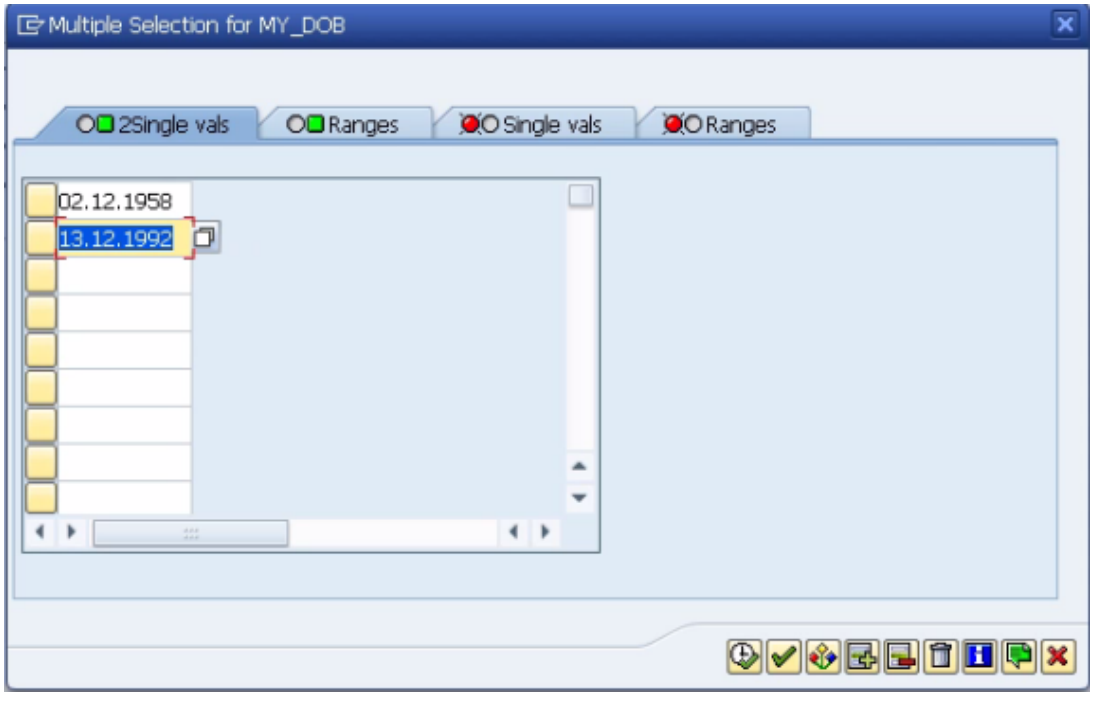
Run the program again but this time include a value in the DOB field of the selection screen. This one corresponds to one of the records in the table:

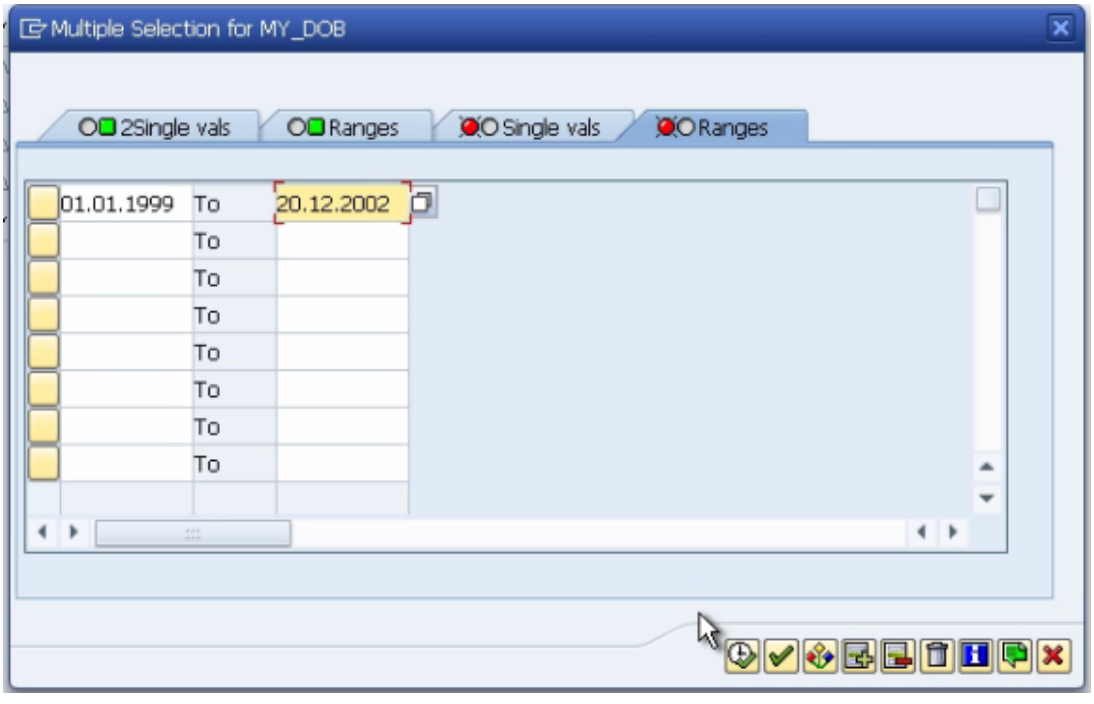


As the select loop is processed, eventually a matching record will be found. When this occurs, rather than skip back to the beginning of the loop, the WRITE statement is executed:

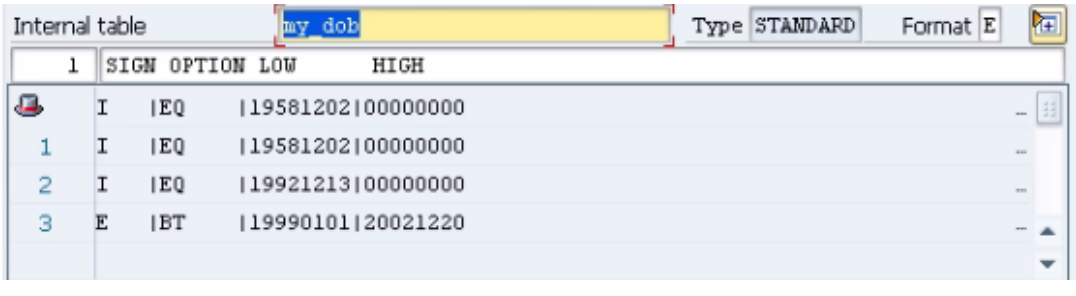


Run the program again but this time try using the multiple selection tool to select several values for the DOB field, as well as excluding some:

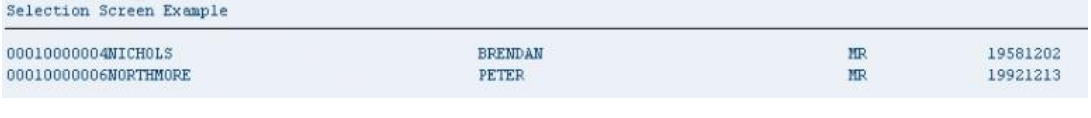




The internal table now contains several entries for values to search for and to exclude from its search:



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 selections window. Once the program is fully executed the output window then appears like this:



Select-Option Additions

As with most statements, there are a number of additions which can be appended to SELECT-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 declared 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 Elements.

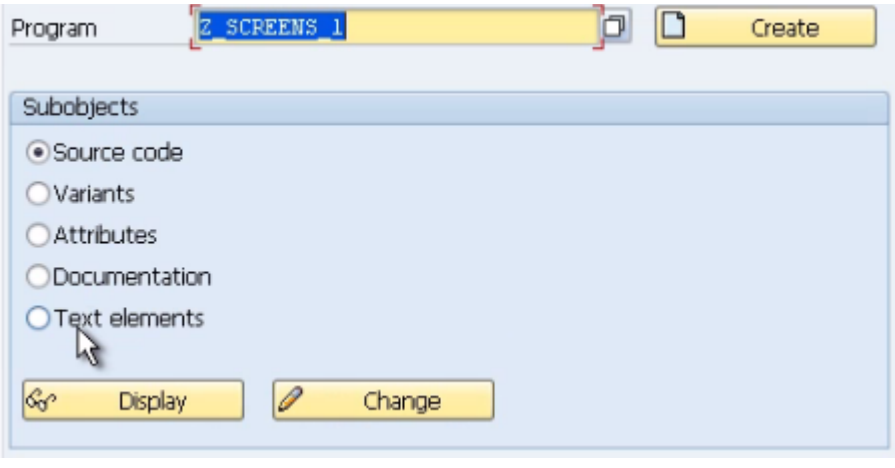
Every ABAP program is made up of sub-objects, like text elements. When one copies a program, the list of options offered asks which parts of the program one wants to copy. The source code and text elements here are mandatory, these are the elements which are essential to the program.

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.

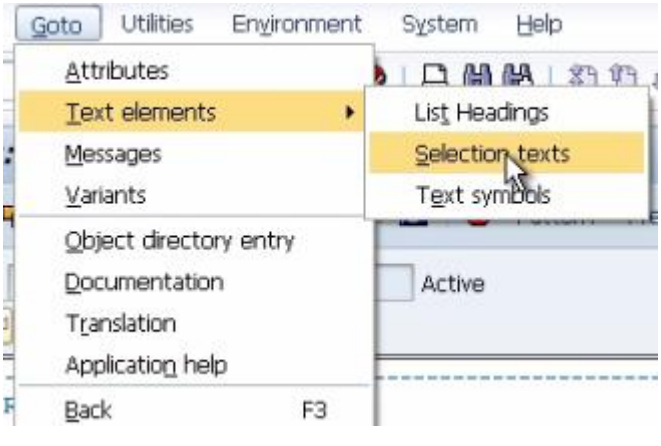
There are three kinds of text elements which can be used in a program, selection texts, mentioned above, are one. The other two are text symbols and list headings. Text symbols can be created for a program so that one does not have to hard code literals into the source code. List headings, as the name indicates, refer to the headings used when creating a report. By using these instead of hard coding them into the program, one can be certain 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.

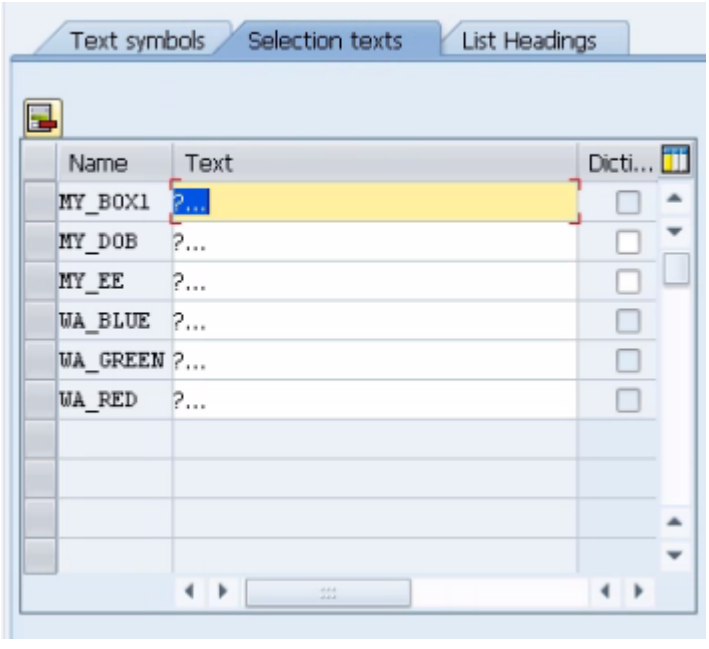
Selection texts allow text elements to be displayed on the screen so that the user does not have to see the technical names for fields and the like. There are several ways to navigate to the screen where these can be created and changed. At the initial ABAP editor screen, there is in fact an option for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



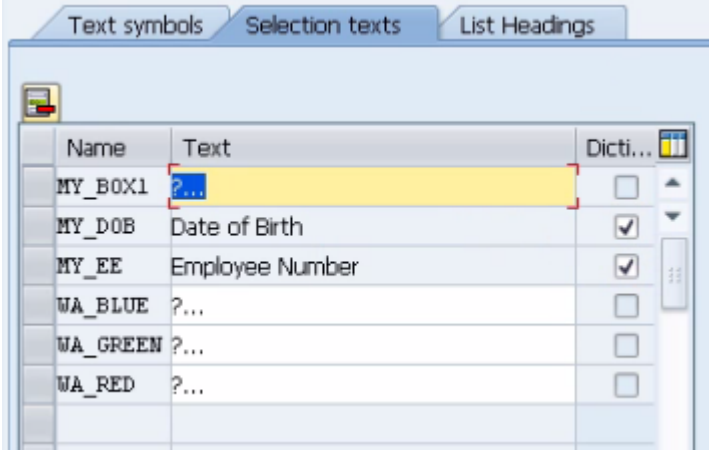
Alternatively, if one is already inside the program, this can be reached through the 'Goto' menu, 'Text elements' and select 'Selection texts':



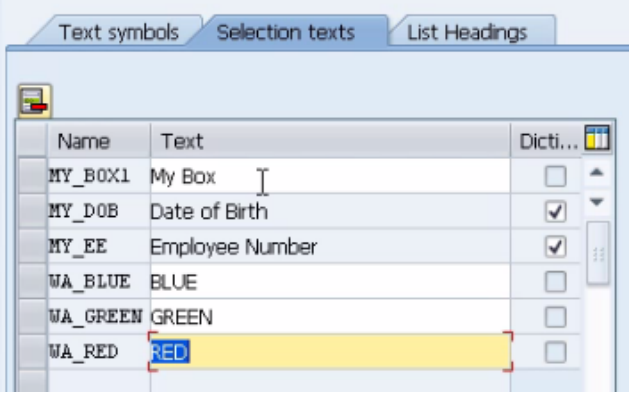
If this is clicked, a screen will appear where selection texts can be created for all of the technical field names which appear at the selection screen:



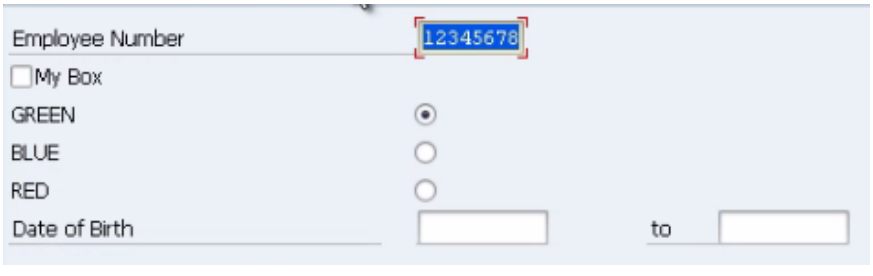
The third column here is for 'Dictionary reference', which recognises that some of these fields are linked to fields already created in the ABAP dictionary. If one checks this box and clicks save, the field names from the initial fields and the ABAP dictionary automatically appear. You can of course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



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

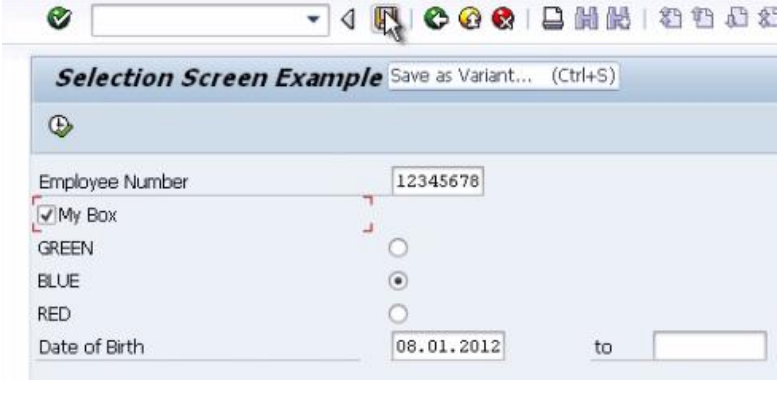


Text Elements must then be activated and once this is done, they are automatically saved and will appear on the selection screen in place of the technical names. The output screen will now look like this:

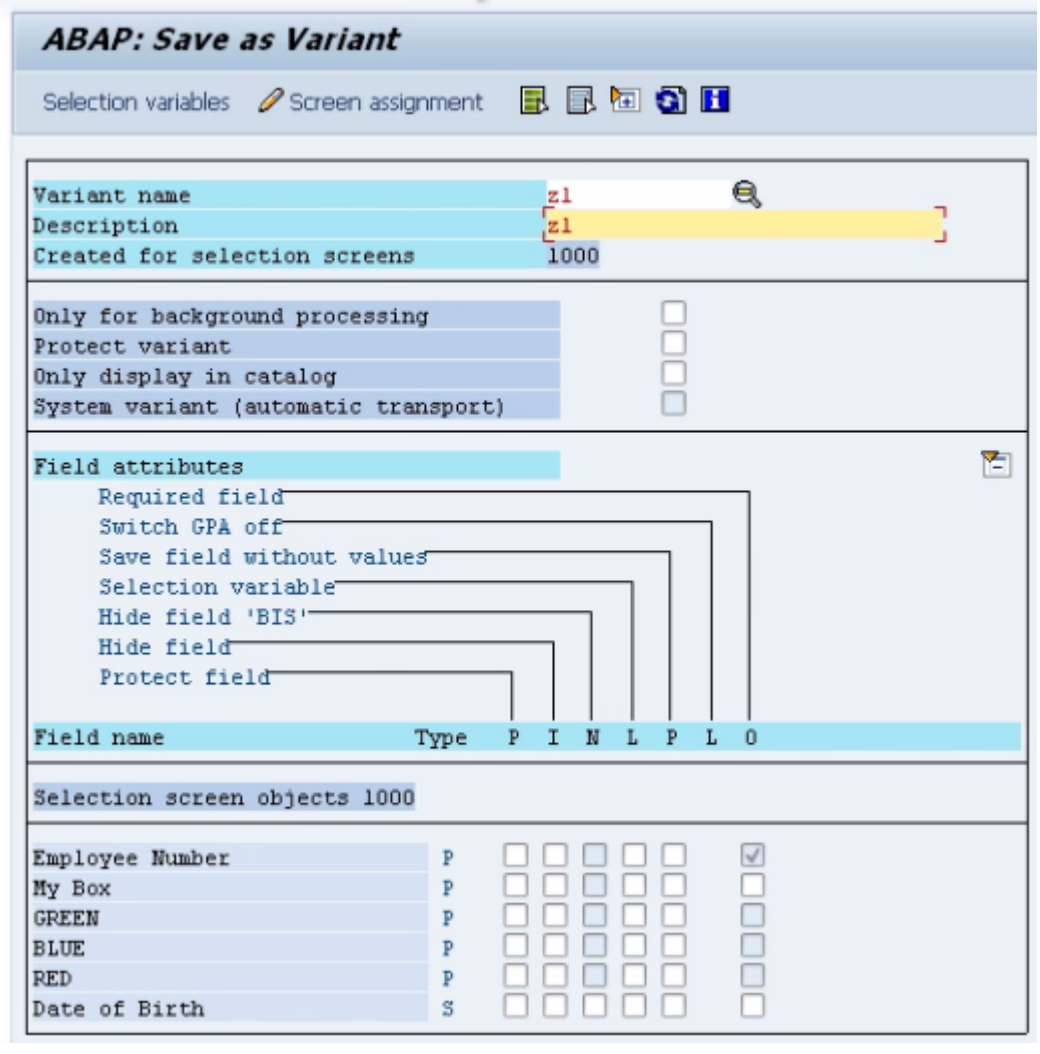


Variants

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



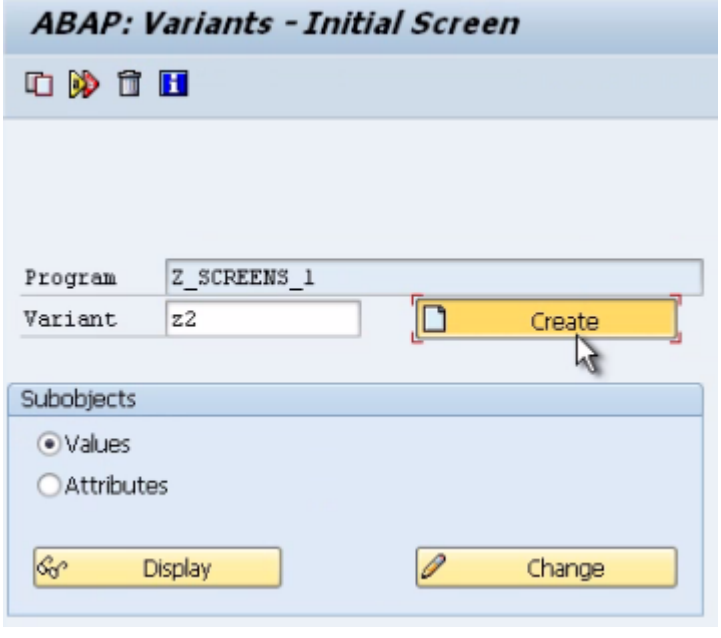
Once this is done, a new screen appears. As long as a name and description are given, this can be saved for use later on:



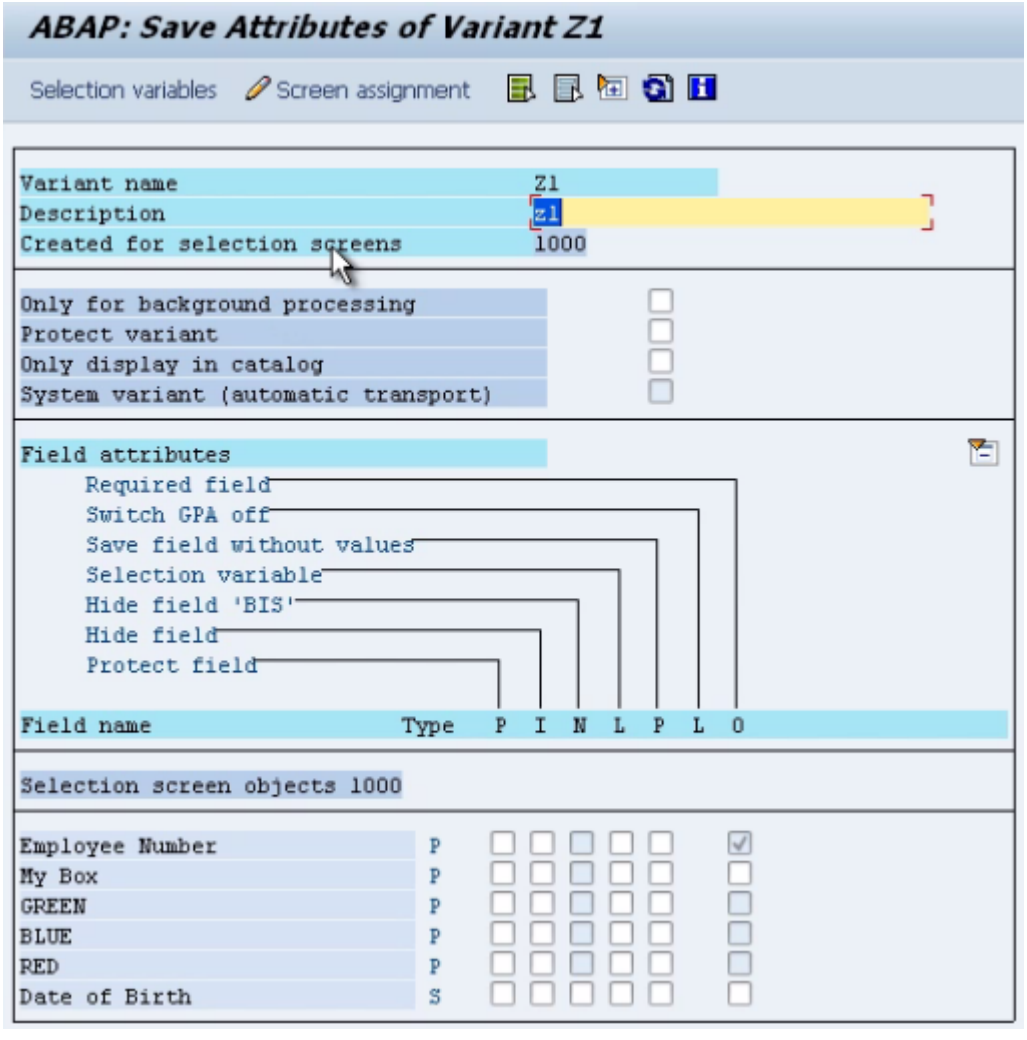
Once saved a new button appears on the selection screen next to the execute button, named '**Get variant**' allowing the variant entry to be recalled.

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:



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):



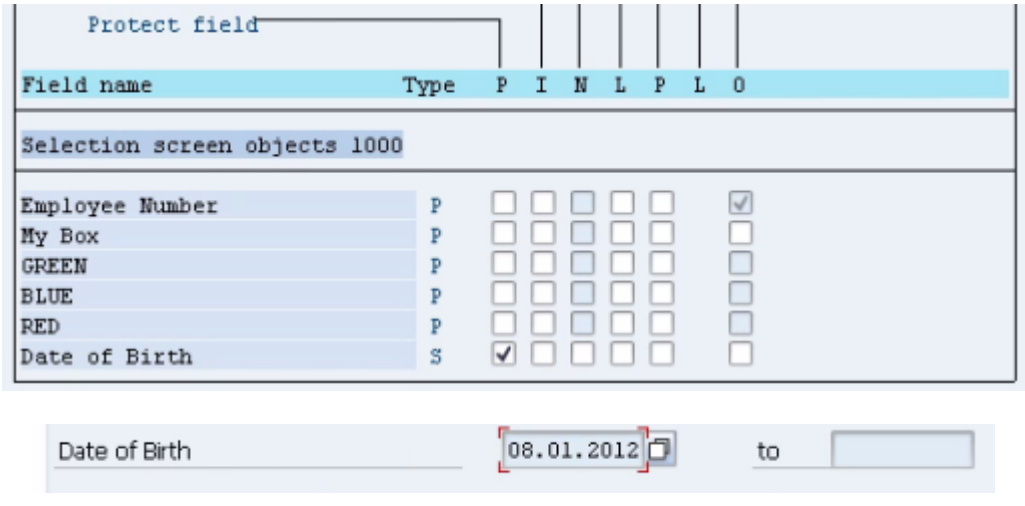
The '**Only for background processing'** check box allows you to tell the system to only use this variant as part of a background job. Here, a job can be scheduled to run overnight so the program does not in fact have to be monitored.

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 'Required 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.

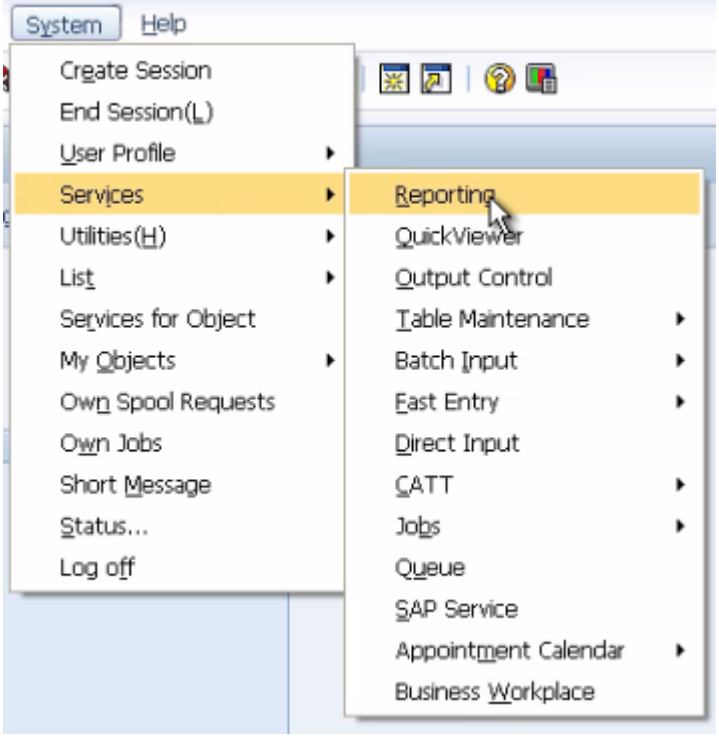
Choose '**Protect field**' for the Date of Birth field; it will no longer be possible to change the value set until such time as this box is un-checked. In the image below you can see this field has been greyed out and cannot be changed:



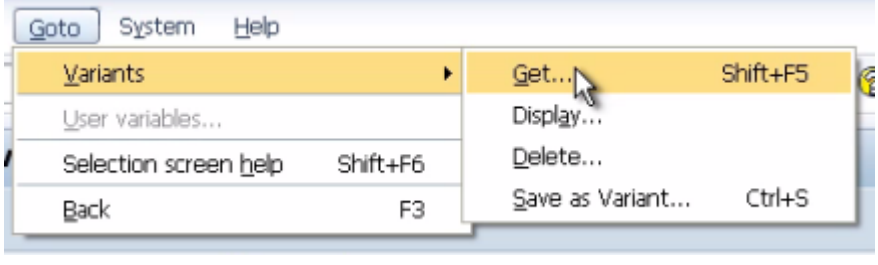
When large selection screens are created, users will regularly create variants so that, if necessary, the same data can be used repeatedly when running reports, saving the time it would take to fill in the information again and again. Unnecessary fields, or fields which will always hold the same value 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 executed 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

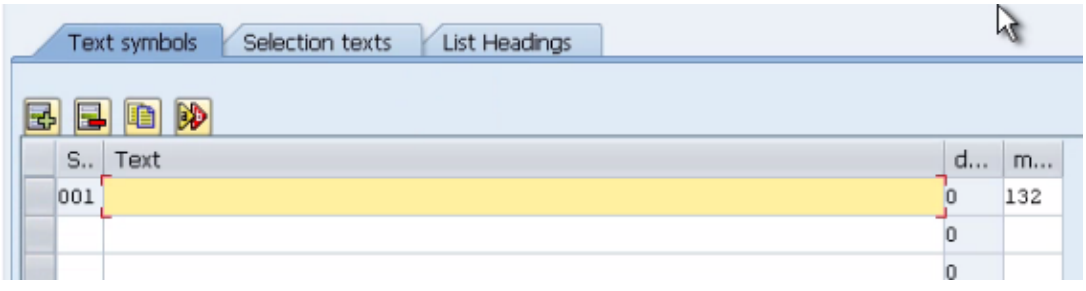
We will now take a look at other text objects starting with Text Symbols. These are used to replace literals in a program. For example, when the WRITE statement is used, one can choose to use text symbols to reuse text which has already been set up. This also gives the added functionality of being able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

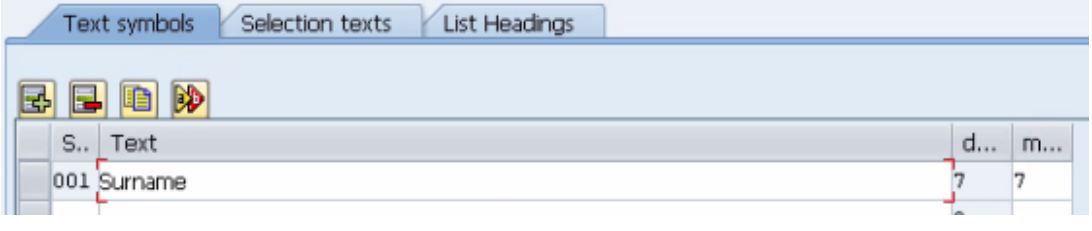
```
WRITE: / 'Surname'.

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.



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:



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:

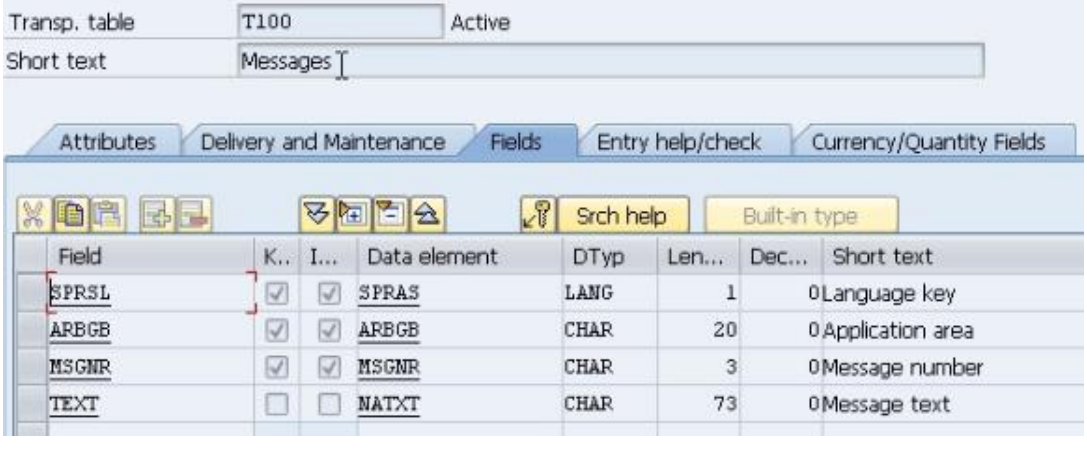
```
Surname
Surname
```

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

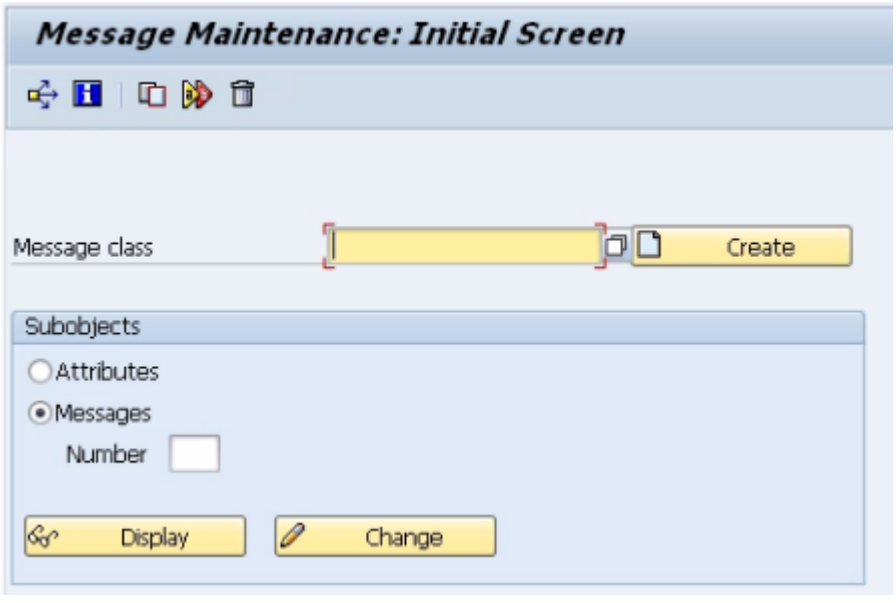
The next thing to be examined here is messages. When one wants to give feedback to the user, literals can be used, but as stated above, this is to be avoided as far as possible. To use messages then, these must first be stored in a message class, which is in turn stored in a database table called T100.

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



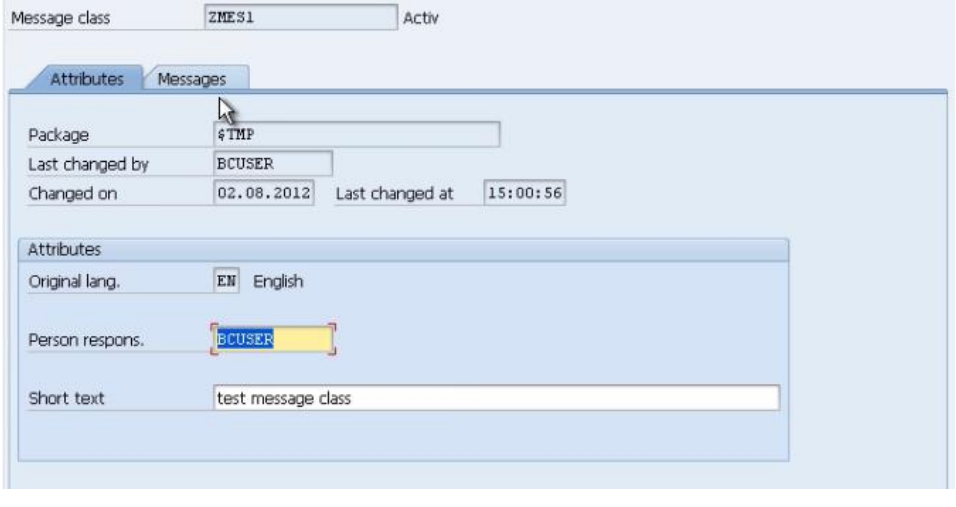
If one views the contents of this, one can see the four fields displayed. One for language (here D, referring to German), one for the application area, one for the message code and one for the message text:

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

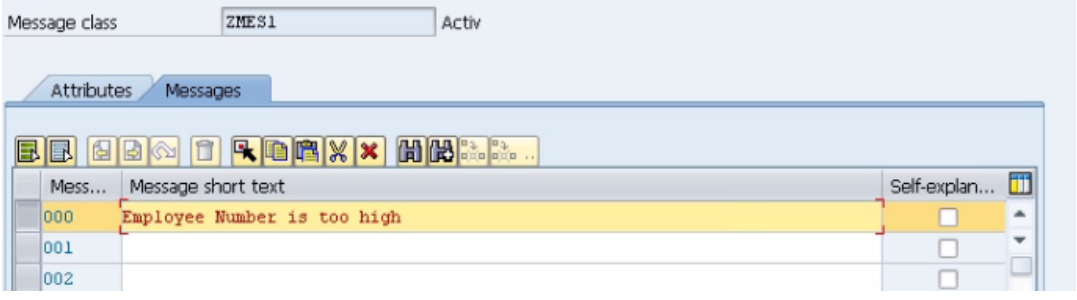


First, create a message class. These must again follow the customer name space rules, here beginning with the letter Z. Let's call this **ZMES1** and choose Create. Messages are distinct from text elements as they are not themselves part of the program created. They exist independently. They are instead 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:



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:



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.

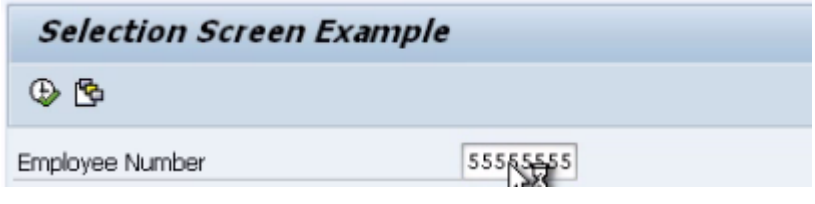
For this example, type E, an error message, will be used. Depending on where this type of message is used, it will have a different effect. Here, it will be used at the “at selection-screen” and the program’s execution will pause, the error message will be displayed and

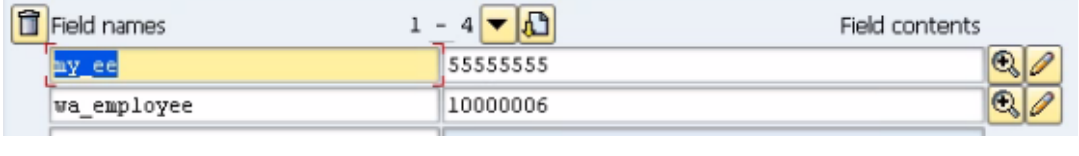
the user will be allowed to amend their entry. When the new entry appears, the event will begin again. If an error message is used elsewhere, outside of an event in the main body of the code, when this is triggered the program will terminate entirely.

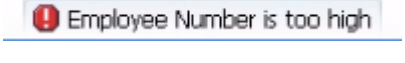
To include the newly created message in the code, then, the syntax is “MESSAGE e000(ZMES1).” The ‘e’ refers to the error message type, the ‘000’ to the number assigned to the message in the message class, and then ‘ZMES1’ to the class itself:

```
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 displayed in the status bar:



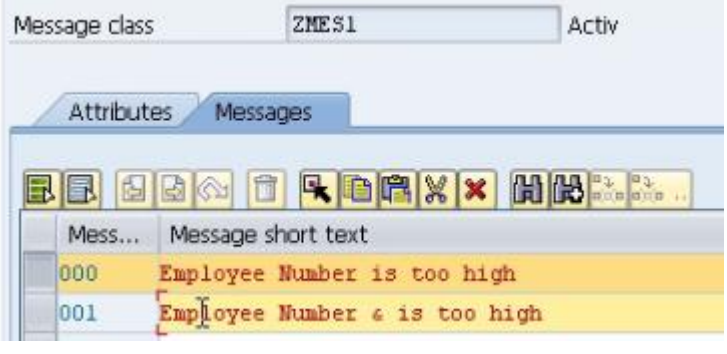




Once a legitimate, lower value is entered, the program will continue as normal without triggering the error message.

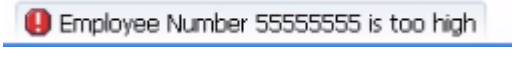
An addition which can be used with the MESSAGES statement is **WITH**. Here, a field can be specified, for example to display the invalid value which was entered by the user in the message itself. The WITH addition allows up to 4 parameters to be included in the error message. To do this, one must ensure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



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.
```



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.

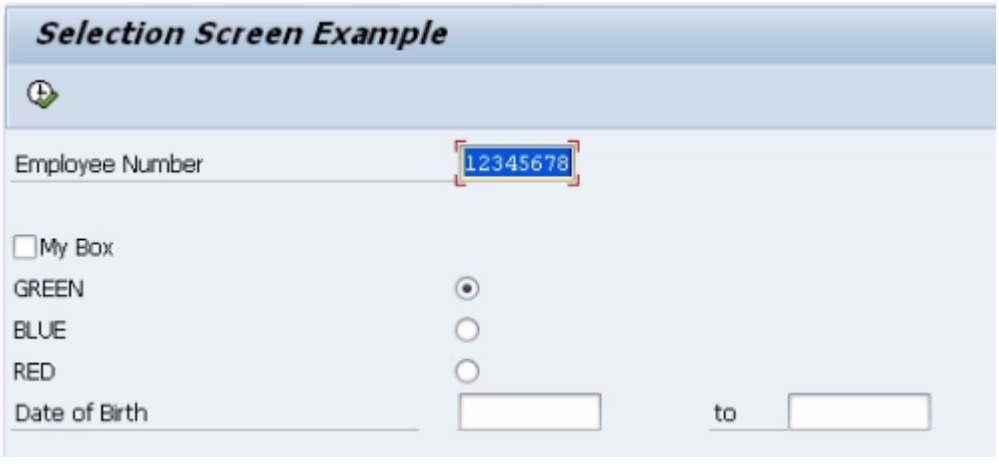
Skip Lines and Underline

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 formatting to be done. One must locate where in the code the screen layout begins to be referred 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.

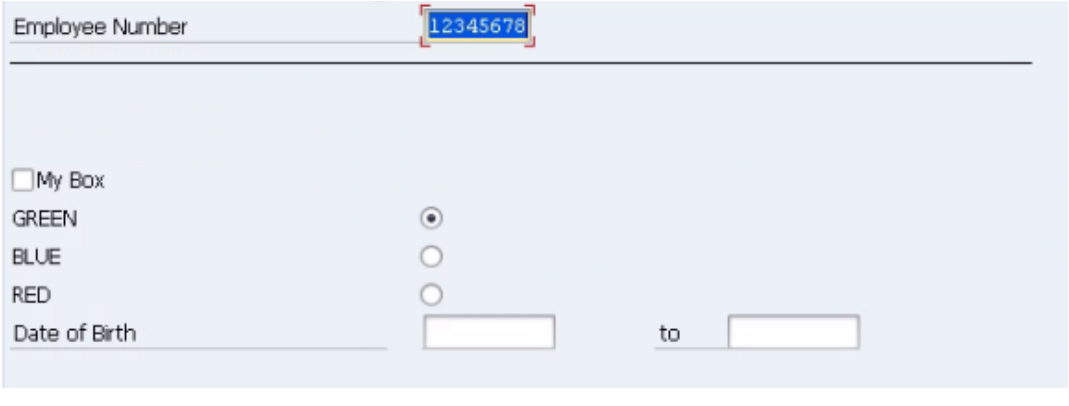
First, to add blank lines you can use the SKIP addition, followed by the number of lines to be skipped. If you only want to skip 1 line then the number can be omitted as this is the default values. This line of code must then be moved to the place where you want the line to be skipped. Place it under 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.
```



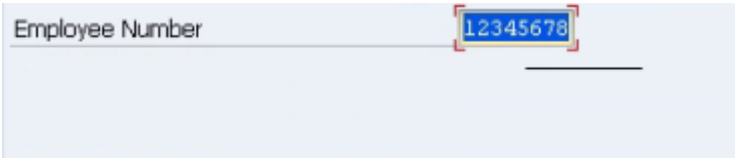
To add a horizontal line, the ULINE addition can be used:

```
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.
```



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:

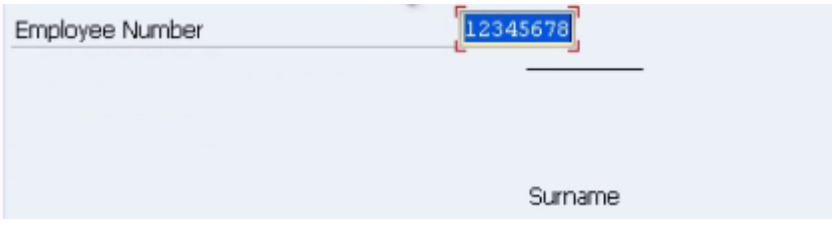
```
PARAMETERS: my_ee LIKE zemployees-employee
                          DEFAULT '12345678' OBLIGATORY.
SELECTION-SCREEN ULINE /40(8).
SELECTION-SCREEN SKIP 2.
```



Comments

Comments allow text to be placed on screen without creating new fields. The SELECTION-SCREEN statement is again used, with the addition COMMENT. Similar additions to ULINE can be used to set the position and length of the comment. This is then followed by either a text element which has already been set, or a field name. This is not declared with a DATA statement, but is determined by the length which the comment is set. Here, the text element **text001** is used, which reads 'Surname', and this will appear 40 characters from the left:

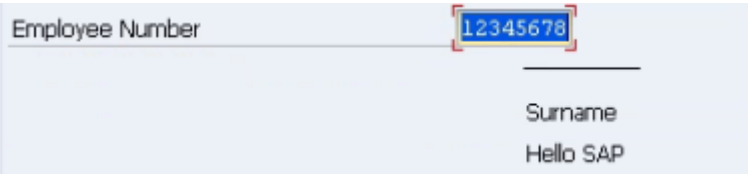
```
PARAMETERS: my_ee LIKE zemployees-employee
                          DEFAULT '12345678' OBLIGATORY.
SELECTION-SCREEN ULINE /40(8).
SELECTION-SCREEN SKIP 2.

SELECTION-SCREEN COMMENT /40(15) text-001.
```



If you do not want to use a text element, a new field can be created here. Copy the initial SELECTION-SCREEN statement and add the new variable "comm1". This variable is currently empty and must be given a value. This must be added in the INITIALIZATION part of the code, so that it is initialised when the program starts. Here, write "**comm1 = 'Hello SAP'.":**

```
PARAMETERS: my_ee LIKE zemployees-employee
                      DEFAULT '12345678' OBLIGATORY.
SELECTION-SCREEN ULINE /40(8).
SELECTION-SCREEN COMMENT /40(15) text-001.
SELECTION-SCREEN COMMENT /40(15) comm1.
```

```
INITIALIZATION.

  comm1 = 'Hello SAP'.
```

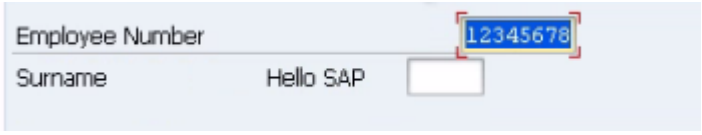


Format a Line and Position

Now, let's take a look at how to format a single line on the selection screen. When individual lines for the selection screen are defined, the start and end of these lines must be declared, and in between these lines the parameters and select-options appear.

Above the formatting code already created, type "**SELECTION-SCREEN BEGIN OF LINE.**", and then underneath "**SELECTION-SCREEN END OF LINE.**" Anything appearing between these statements will now all appear on the same line. Then, alter the formatting code slightly so that this will work, removing the ULINE statement, moving the text001 comment (which reads 'Surname') to the first space on the line, and the comm1 comment (reading 'Hello SAP') to the 20th space and change the length of this to 10 characters. Also, remove the /n which put these on a new line. Finally, add a new PARAMETERS statement beneath the second comment in the code, named 'ABC', with a length of 5:

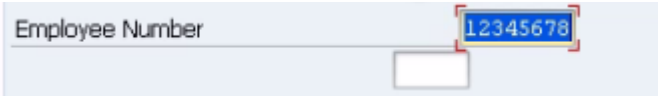
```
PARAMETERS: my_ee LIKE zemployees-employee
                         DEFAULT '12345678' OBLIGATORY.

SELECTION-SCREEN BEGIN OF LINE.
SELECTION-SCREEN COMMENT 1(15) text-001.
SELECTION-SCREEN COMMENT 20(10) comm1.
PARAMETER ABC(5).
SELECTION-SCREEN END OF LINE.
```



You can now see that the code between the BEGIN and END OF LINE statements now all appears on one line; its formatting determined by the positions and lengths assigned to each statement. Note that here the parameter was not automatically given a description (the technical name of the field) as others have been. This is because specific comments have been used on the same line. When you are formatting a line in this way, he comments can be used to act as descriptions for the field.

Another addition, which can only be used within BEGIN and END OF LINE, is POSITION. This is not commonly used because this can effectively be set by alternate methods, as above. However, if one desires, the position of the next element can be set separately. Here, the parameter will appear 30 spaces into the new line:

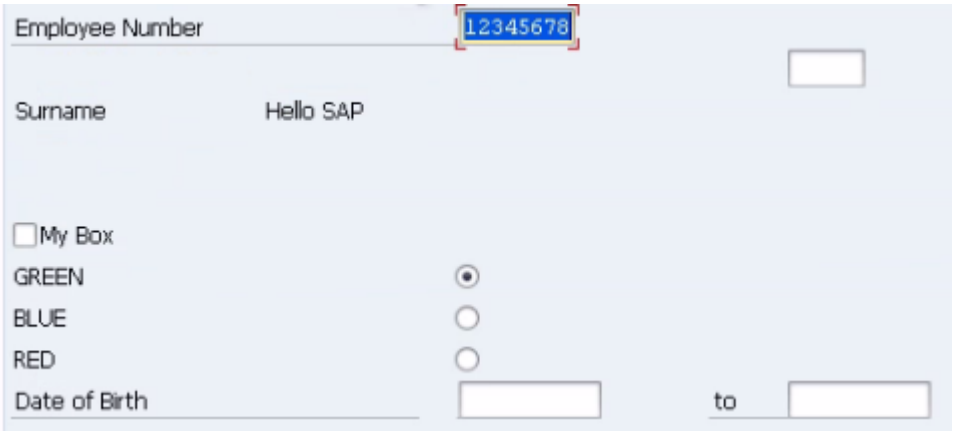
```
PARAMETERS: my_ee LIKE zemployees-employee
                          DEFAULT '12345678' OBLIGATORY.

SELECTION-SCREEN BEGIN OF LINE.
SELECTION-SCREEN POSITION 30.
PARAMETER abc(5).


SELECTION-SCREEN END OF LINE.
```



Note that here the technical name still does not appear, as the parameter is still between the BEGIN and END OF LINE statements.

There is also a further option which can be included with the POSITION addition. The default positions of parameters and select-options on the screen are referred to as '**position low**' for the left hand side, where standard parameters and the low end of value ranges appear, and '**position high**' for the right hand side, where the upper end of a value range would appear. These default positions can be used with the POSITION addition. To place a parameter in the 'position high' position, you would include **pos_high** at the end of the statement:

```
SELECTION-SCREEN BEGIN OF LINE.
SELECTION-SCREEN POSITION pos_high.
PARAMETER abc(5).


SELECTION-SCREEN END OF LINE.
```



You can see that the parameter now matches up with the 'position high' default value when compared to the upper end of the Date of Birth value range. Unsurprisingly, this is replaced with **pos_low** to make it correspond to the default 'position low' column.

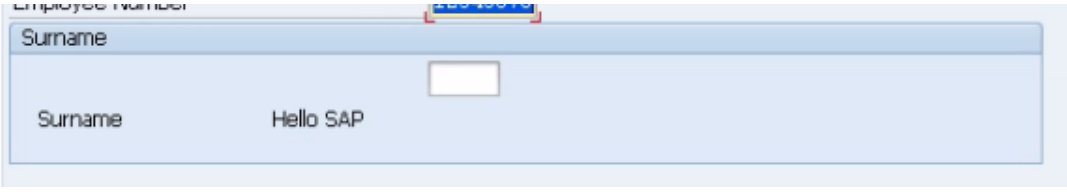
Element Blocks

When you are creating selection screens, it is common practice to group certain fields together. You can make use of these element blocks, which will draw frames around the certain groups of fields which are designated. These frames can then be given frame labels. Bear in mind when looking at these it is possible to nest element blocks within other element blocks, allowing individual sections of the selection screen to be subdivided.

The syntax for this is very similar to that of BEGIN and END OF LINE. Above where these statements were tested before, add the code "**SELECTION-SCREEN BEGIN OF BLOCK**", followed by a name for this block, here "myblock1". To then add a frame to the block, the **WITH FRAME** addition is then used. The frame can then be given a title using, like comments, either a text element or separately defined variable. This is done after the WITH FRAME addition, adding **TITLE** and then, here 'text-001', which as before contain the values 'Surname'.

Having done all of this, you must remember to then use END OF BLOCK followed by the block name so that the system knows which block is ending:

```
SELECTION-SCREEN BEGIN OF BLOCK myblock1 WITH FRAME TITLE text-001.

SELECTION-SCREEN BEGIN OF LINE.
SELECTION-SCREEN POSITION pos_low.
PARAMETER abc(5).
SELECTION-SCREEN END OF LINE.
SELECTION-SCREEN COMMENT 1(15) text-001.
SELECTION-SCREEN COMMENT 20(10) comm1.

SELECTION-SCREEN END OF BLOCK myblock1.
```



Element blocks, when used correctly, add context to the selection screen, making it easier for the end user to understand the screen entry requirements.

Chapter 12 – Internal Tables

Introduction

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

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

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

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

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

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

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

Types of Internal Tables

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

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

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

Standard Tables

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

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

Sorted Tables

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

Hashed Table

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

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

Internal Tables - Best Practice Guidelines

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

Creating Standard and Sorted Tables

Create a new program in the ABAP editor called Z_EMPLOYEES_LIST_03 to use for the creation of internal tables. To begin to declare an internal table, the DATA statement is used. The program must be told where the table begins and ends, so use the BEGIN OF statement, then declare the table name, here '**itab01**' (itab is a commonly used shorthand when creating temporary tables in SAP). After this, the **OCCURS** addition is used, followed by a number, here 0. OCCURS tells SAP that an internal table is being created, and the 0 here states that it will not contain any records initially. It will then expand as it is filled with data:

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

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

```
DATA: BEGIN OF itab01 OCCURS 0,
        surname LIKE zemployees-surname,
        dob     LIKE zemployees-dob,
      END OF itab01.
```

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

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

Create an Internal Table with Separate Work Area

Instead of using the DATA statement, this time start by defining a **line type**, using the **TYPES** statement. Following this, the **BEGIN OF** statement is used, followed by a name, here '**line01_typ**'. Below this, the surname and dob fields from above can be created as before. Then the **END OF** statement is used to end the line type definition:

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

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

Once the line has been defined, next you define the **table type**. Again, use the TYPES statement, followed this time by the table, here '**itab02_typ**' (*note the **_typ** addition to the end as it is only the table type being defined, not the table itself*). Follow this with "**TYPE STANDARD TABLE OF line01_typ.**"; telling the system it will be a standard table containing the structure of the line-type defined above:

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

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

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

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

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

```
DATA itab02 TYPE itab02_typ.
```

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

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

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

```
DATA wa_itab02 TYPE line01_typ.
```

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

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

Filling an Internal Table with Header Line

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

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

```
TABLES: zemployees.

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

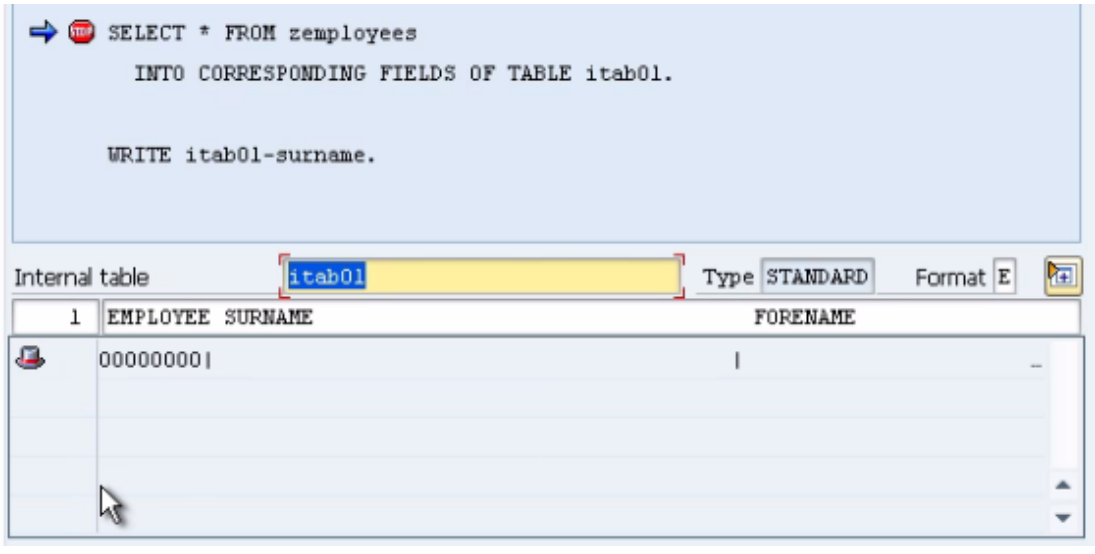
The fields should broadly be familiar. The only new one here is '**los**', representing '*length of service*', an integer type with a default value of 3.

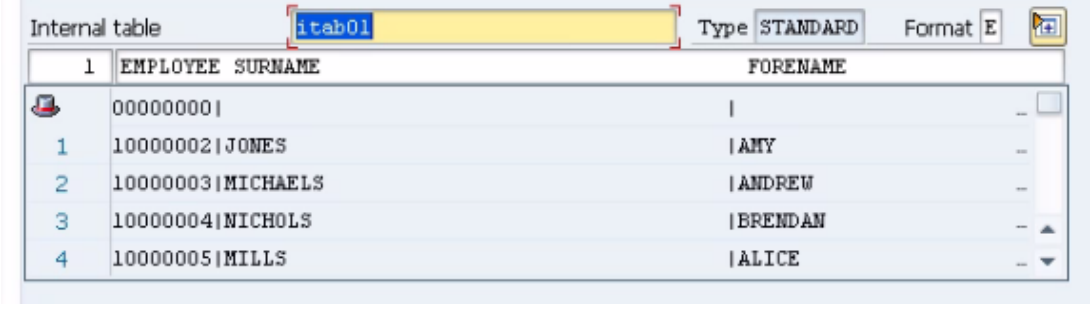
To start to fill this table, you can use a SELECT statement to select all of the records from the **zemployees** table and then use "**INTO CORRESPONDING FIELDS OF TABLE itab01.**", which will move the records from the original table into the new internal table into the fields where the names correspond.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

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

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

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

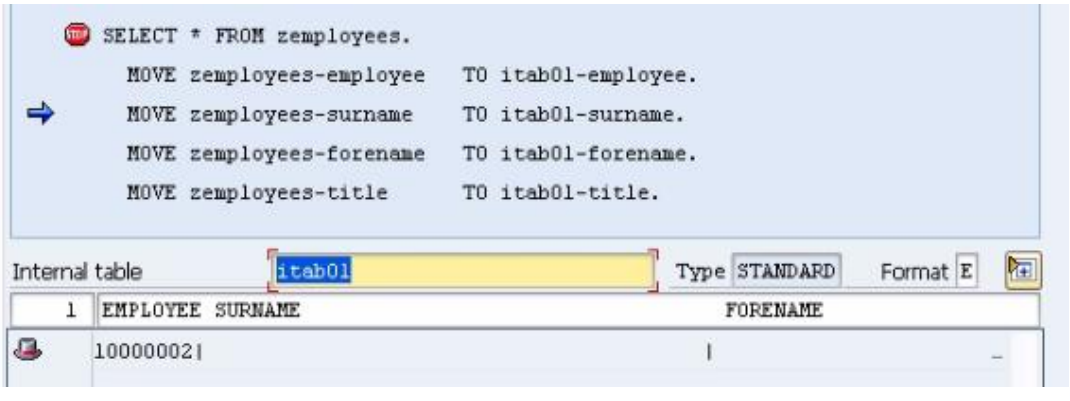




A different way of filling the table would be with the code below, this time with a select loop filling each field one at a time, using the MOVE statement to move the data from one table's field to the other. Note that los is not present here since it does not have a field in the zemployees table.

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

ENDSELECT.

WRITE itab01-surname.
```

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



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

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

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

  APPEND itab01.
ENDSELECT.
```

```
    APPEND itab01.
ENDSELECT.

WRITE itab01-surname.
```

Internal table itab01 Type STANDARD Format E

1	EMPLOYEE SURNAME	FORENAME
	10000002 JONES	AMY
1	10000002 JONES	AMY

Move-Corresponding

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

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

  APPEND itab01.
ENDSELECT.
```

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

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

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

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

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

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

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

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

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

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

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

Filling Internal Tables with a Work Area

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

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

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

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

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

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

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

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

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

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

Using Internal Tables One Line at a Time

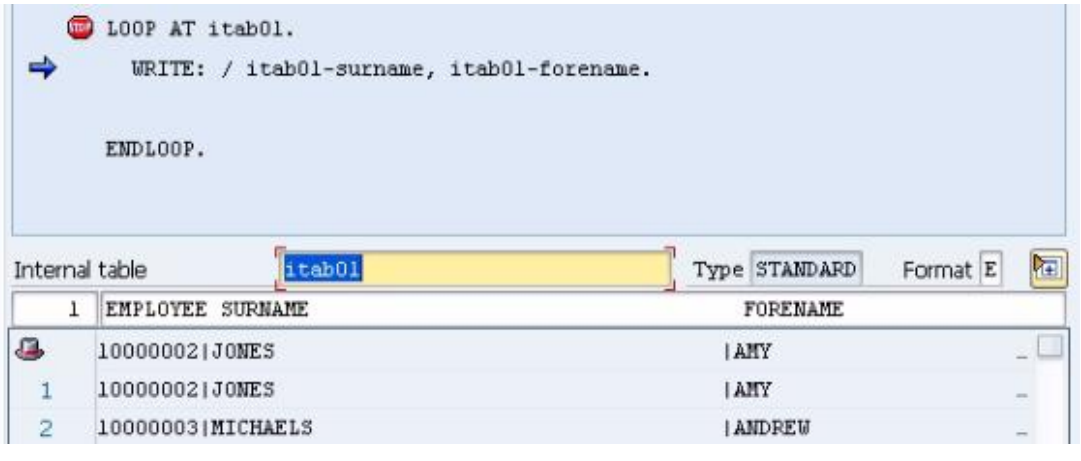
Now you know how to fill internal tables with data, a look will be taken at how to use the data in them line-by-line.

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

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

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

ENDLOOP.
```

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



Modify

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

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


ENDLOOP.
```

Describe and Insert

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

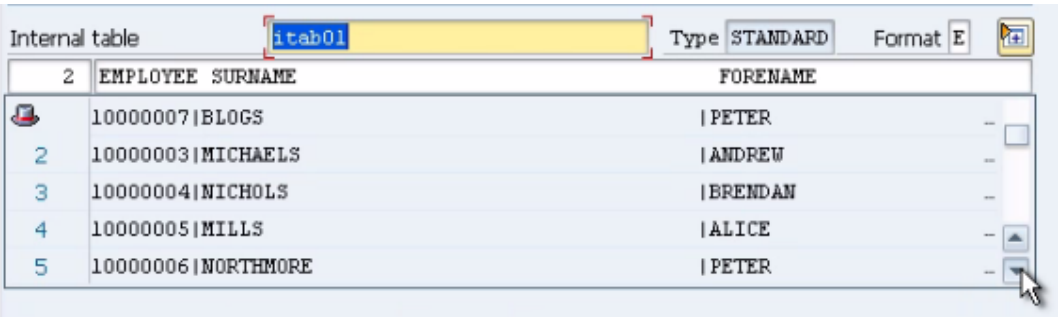
Beneath the ENDIF, add the line of code "**DESCRIBE TABLE itab01 LINES line_cnt.**" The **LINES** part of this statement is used to request the value of the number of lines contained in the internal table, and '**line_cnt**" is a new variable (of type i) set up to hold this value.

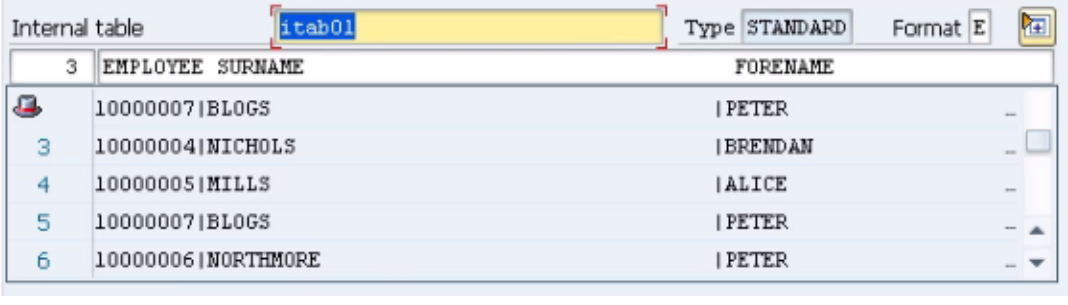
Up until now, the APPEND statement has been used to add records to the table. This automatically inserts the new record at the end of the table. If you want to add a record somewhere in the middle, the **INSERT** statement should be used, along with the table *index number*, to specify the position where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

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

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


DESCRIBE TABLE itab01 LINES line_cnt.

INSERT itab01 INDEX line_cnt.
```

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





Read

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

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

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

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

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

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

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

Delete Records

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

```
DELETE itab01 INDEX 5.
```

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

This statement can also be used inside a loop:

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

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

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

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

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

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

Sort Records

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

```
SORT itab01.
```

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

```
SORT itab01 BY surname.
```

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

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

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

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

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

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

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

Work Area Differences

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

Loops

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

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

In this example the records will be read one record at a time into the work area **wa_itab02**, then the contents of the surname field will be written to the output screen.

Modify

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

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

Insert

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

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

Read

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

```
READ TABLE itab02 INDEX 5 INTO wa_itab02.
```

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

Delete

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

Delete a Table with a Header Line

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

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

CLEAR

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

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

REFRESH

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

```
REFRESH itab01.
```

FREE

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

```
FREE itab01.
```

Delete a Table with a Work Area

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

The CLEAR statement above, when used on a table without a header line, will clear the whole contents of the table without needing to add the **[]**. Remember that another CLEAR statement must be used to empty the work area. The same applies to the REFRESH and FREE statements. The syntax above will work, and a further CLEAR statement must be used to empty the work area. In the examples below, assume itab01 and wa_itab01 refer to the newer style internal table and its work area:

```
CLEAR itab01.
CLEAR wa_itab01.

REFRESH itab01.
CLEAR wa_itab01.

FREE itab01.
CLEAR wa_itab01.
```

Chapter 13 – Modularizing Programs

Introduction

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

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

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

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

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

Includes

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

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

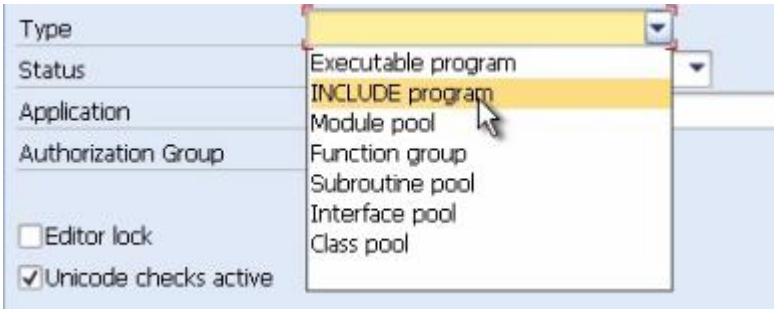
```
REPORT  z_mod_1                                 .

TABLES: zemployees.

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

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

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

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

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

SELECT * FROM zemployees
  INTO CORRESPONDING FIELDS OF TABLE itab02.

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

CLEAR: itab02, wa_itab02.

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

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

READ TABLE itab02 INDEX 5 INTO wa_itab02.

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

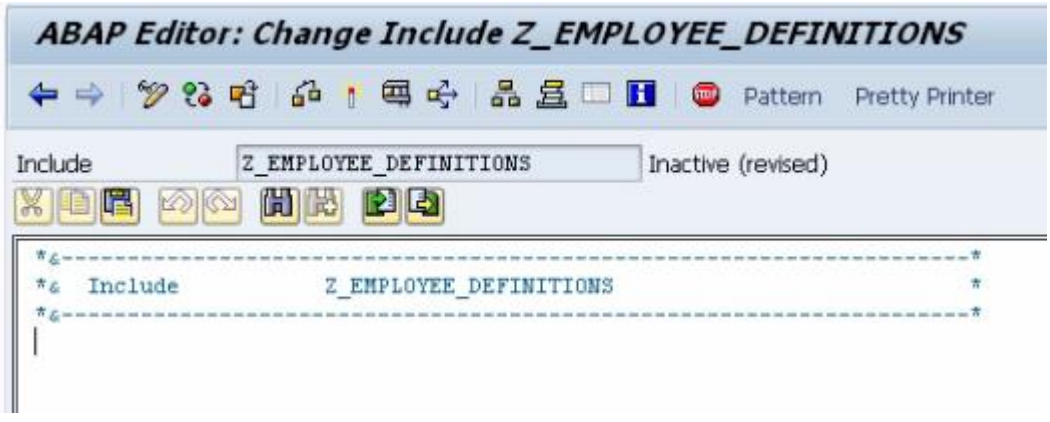
First, we will look at INCLUDE programs. INCLUDE's are made available globally within an SAP system and their sole purpose is modularizing code. They are simple to define and accept no parameters. Below the REPORT statement, fill in the statement for declaring an include. Type **INCLUDE** and then define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

```
REPORT  z_mod_1

INCLUDE Z_EMPLOYEE_DEFINITIONS.

TABLES: zemployees.
```

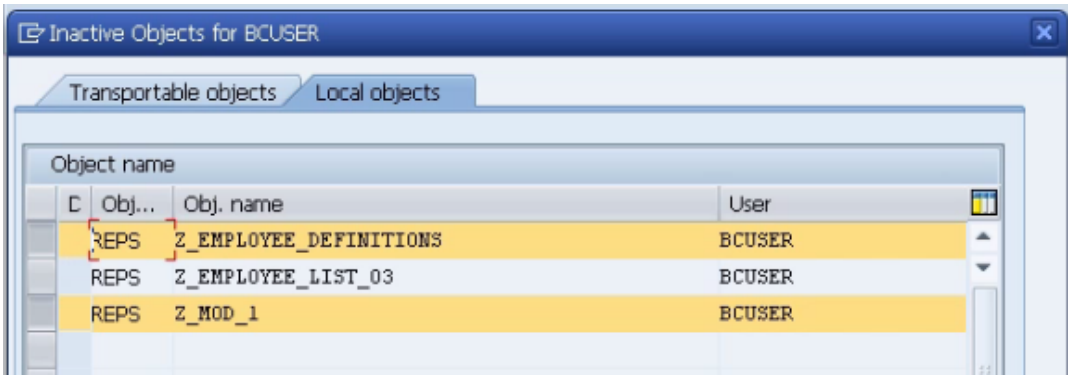
This statement is telling the program to *include* the INCLUDE program within our original program. There are two ways of creating this new INCLUDE program. You can type the name into the ABAP editor's initial screen and select the 'Attributes' radio button, followed by 'Create'. Then, when the window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



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



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



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

```
INCLUDE Z_EMPLOYEE_DEFINITIONS.

TABLES: zemployees.

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

```
*&---------------------------------------------------------------------*
*&  Include           Z_EMPLOYEE_DEFINITIONS
*&---------------------------------------------------------------------*

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

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

Procedures

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

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

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

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

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

Sub-Routines

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

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

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

Above the commented-out section, use the statement **PERFORM**. This statement is used to perform a sub-routine. Then a name for the sub-routine is added. Here, since this code fills the itab02 internal table, call the sub-routine "**itab02_fill**" as shown:

```
PERFORM itab02_fill.

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

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

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

SELECT * FROM zemployees
  INTO CORRESPONDING FIELDS OF TABLE itab02.

endform.                    " itab02_fill
```

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

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

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

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

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

Create code for a second sub-routine, called "**itab_02_fill_again**" and above this create 2 new DATA fields, as shown in the example below, telling the new sub-routine to use the new data fields. Then use forward navigation to create this sub-routine:

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

*****************************************

PERFORM itab02_fill.

perform itab02_fill_again USING z_field1 z_field2.
```

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

endform.                    " itab02_fill_again
```

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

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

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

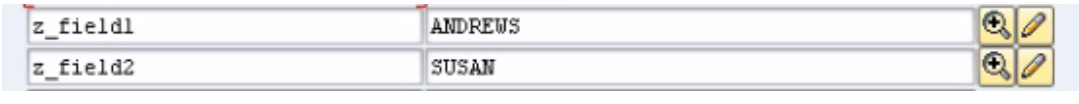
```
form itab02_fill_again  using    p_zsurname
                                 p_zforename.

  write / p_zsurname.
  write / p_zforename.

  p_zsurname = 'abcde'.

endform.                    " itab02_fill_again
```

Ensure these fields hold some data by giving z_field1 and z_field2 values in the main program:

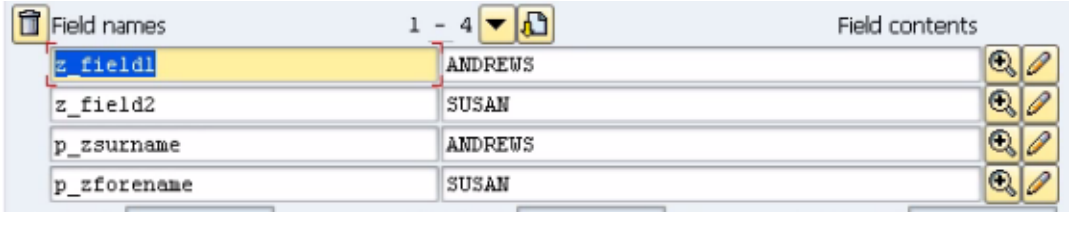
```
z_field1 = 'ANDREWS'.
z_field2 = 'SUSAN'.

perform itab02_fill_again USING z_field1 z_field2.
```

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

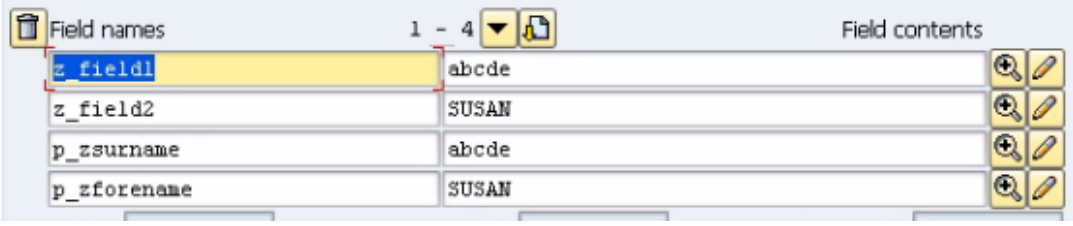
You can see z_field1 and z_field2 are filled with their initial values:



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



The two WRITE statements are then executed, followed by the change in value for p_zsurname. Because the field is used in the interface, the global variable, z_field1's value also changes:



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

Passing Tables

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

Create a new sub-routine called **itab02_write**. Then, use the **TABLES** addition to specify the table to be passed, here **itab02**:

```
PERFORM itab02_write TABLES itab02.
```

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

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

endform.                    " itab02_write
```

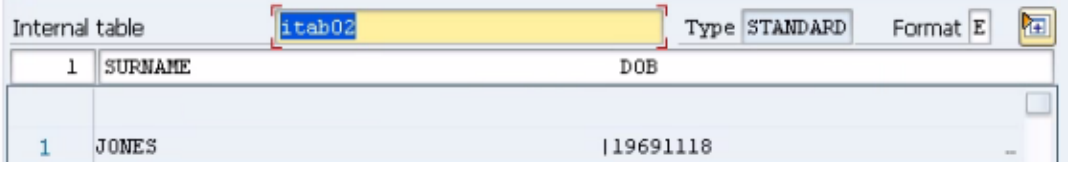
Using the TABLES addition, the program ensures that the contents of the internal table are transferred to the subroutine and stored in the internal table **p_itab_02**. Once this subroutine is processed, the contents of the local internal table are then passed back to the global internal table.

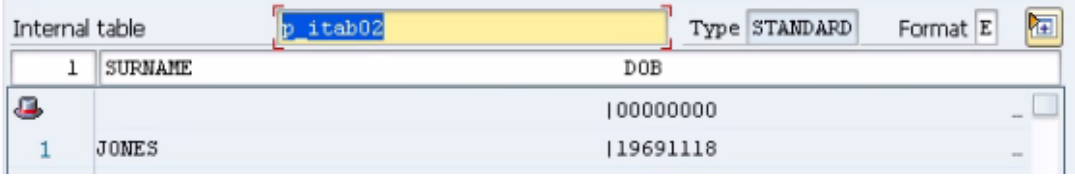
Note that this method is for a table without a header line. If this code was used with an old-style internal table, only the header line would be passed to the table. To pass the full table, you need to add **[]** at the end of the statement.

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

```
FORM itab02_write  TABLES   p_itab02.

  DATA wa_tmp TYPE line01_typ.

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

ENDFORM.                    " itab02 write
```

When analysed in debug mode, the itab02 table does not have a header record, but p_itab02 does:





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

Passing Tables and Fields Together

Now, a combination of fields and tables will be passed into a subroutine at the same time. Create another PERFORM statement, called **itab02_multi**. Retain the TABLES statement, but then add the USING statement afterwards:

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

Use forward navigation to generate the form.

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

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

Sub-Routines - External Programs

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

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

If you want to call a sub-routine called **'sub_1**', held in a program called 'zemployee_hire', the code would look like this. Note that additions can still be used with this method:

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

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

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

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

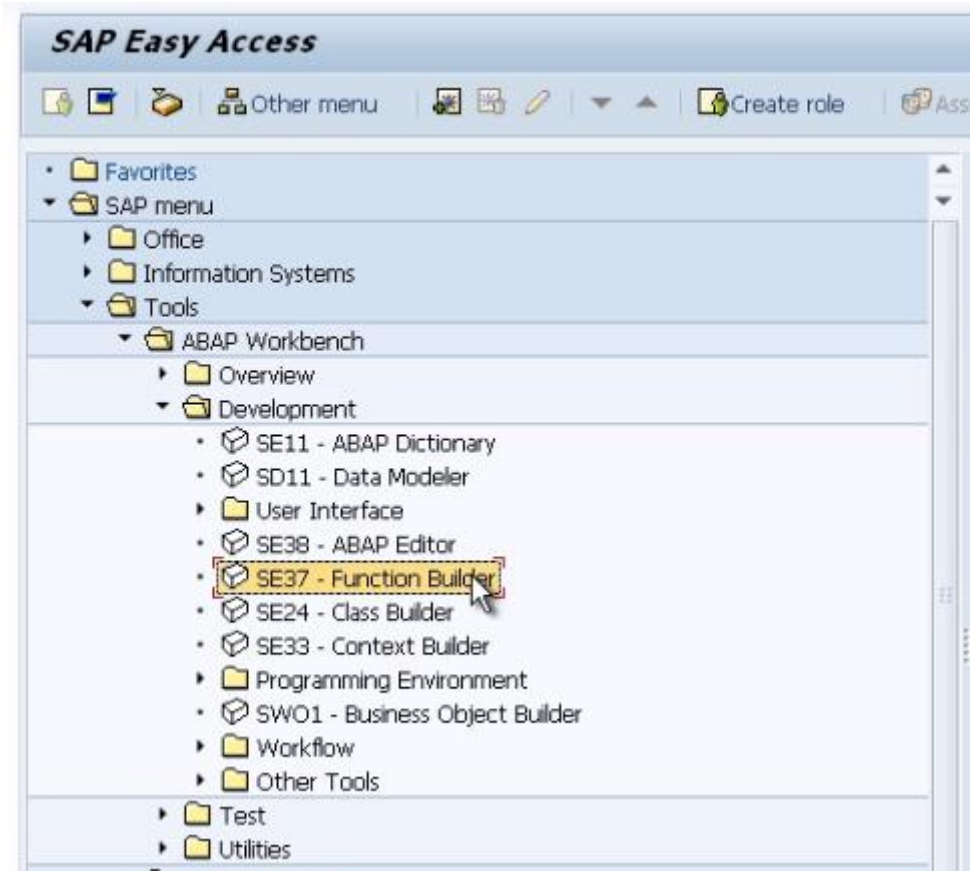
Calling external sub-routines is not common practice, sub-routines tend to stay internal to the program and where you want to call sub-routines in external programs, this is usually done via function modules.

Function Modules

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

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

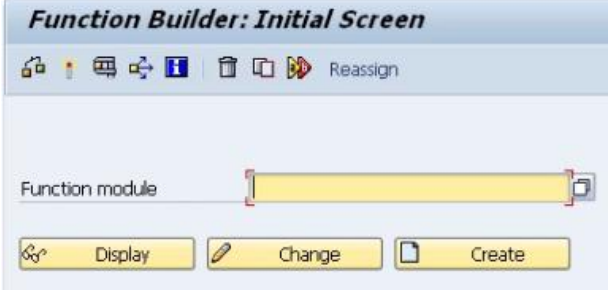
To look at how to create function modules, the function builder must be looked at. This is found via the menu at the very beginning of the system, via the SAP menu → Tools → ABAP Workbench → Development. There one will find the function builder, with transaction code SE37:



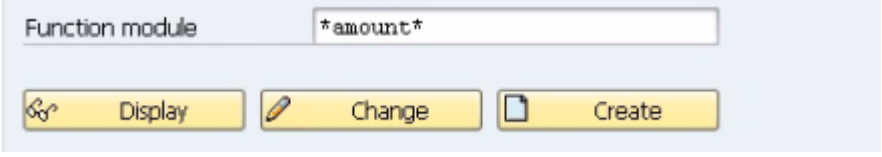
Before diving into an example of how to use a function module we need look at how function modules are put together, so as to understand how to use them in a program.

Function Modules – Components

The initial screen of the function builder appears like this:



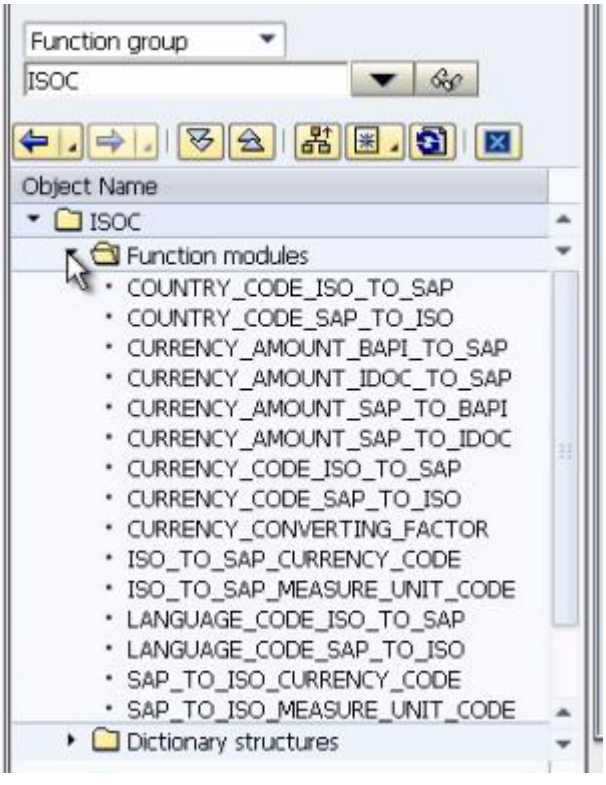
Rather than typing the full name here, part of a function module name will be typed with a wild card character to demonstrate the way function modules can be searched for. Type *amount* and then press the F4 key. The results of the search will then be displayed in a new window:



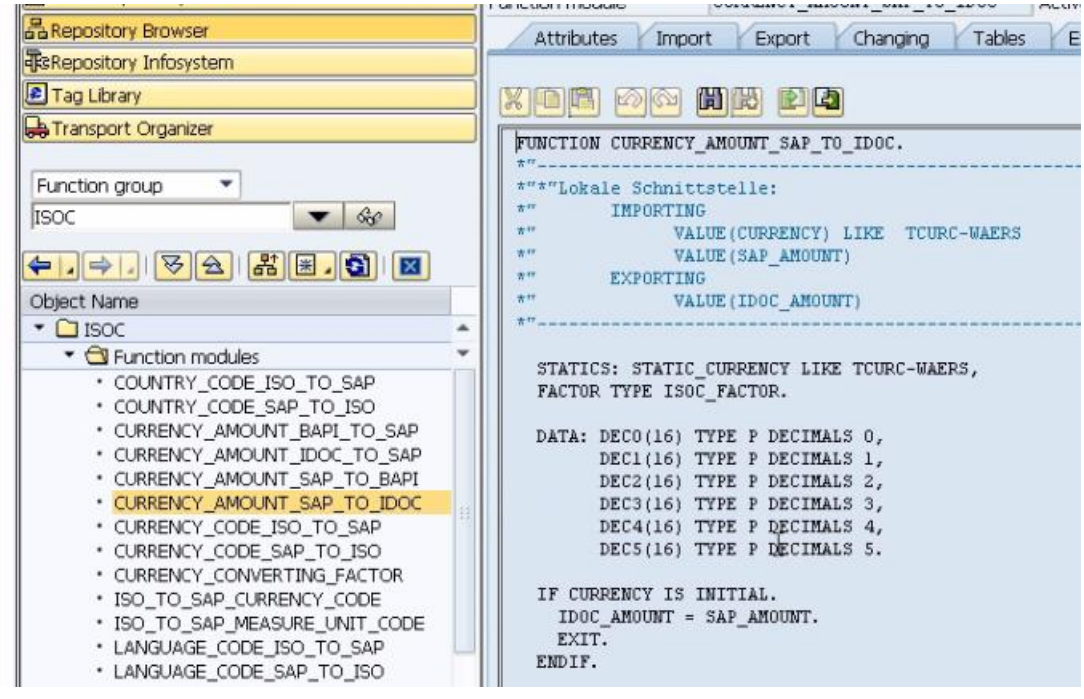
Repository Info System: Function modules Find (12 Hits)

Function group	Function group short text
Name of function module	Short text for function module
F017	Conversion of amounts to words utility
SPELL_AMOUNT	Convert numbers and figures in words
FF01	
FIMA_COND_DETAIL_AMOUNT_CHECK	
FIMA_COND_DETAIL_AMOUNT_PAI	
FIMA_COND_DETAIL_AMOUNT_PBO	
FF05	
FIMA_AMOUNT_DISCOUNT	
ISOC	
CURRENCY_AMOUNT_BAPI_TO_SAP	
CURRENCY_AMOUNT_IDOC_TO_SAP	
CURRENCY_AMOUNT_SAP_TO_BAPI	
CURRENCY_AMOUNT_SAP_TO_IDOC	
RHALE_CONVERT	
RH_ALE_CURR_AMOUNT_IDOC_TO_SAP	
RH_ALE_CURR_AMOUNT_SAP_TO_IDOC	
SFCC	
ALV_CORRECT_CURR_AMOUNTS	

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

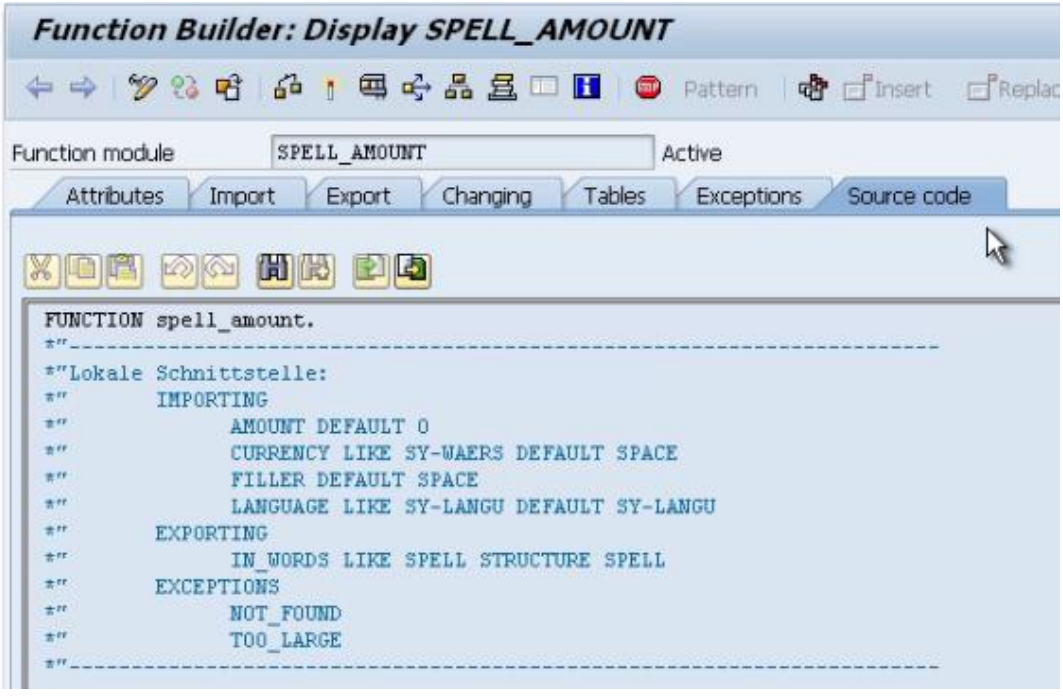


The four which showed up in the *amount* search are present, along with a number of others. If double-click any of these function modules, the code for that function module will appear on screen to the right of the menu:



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

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

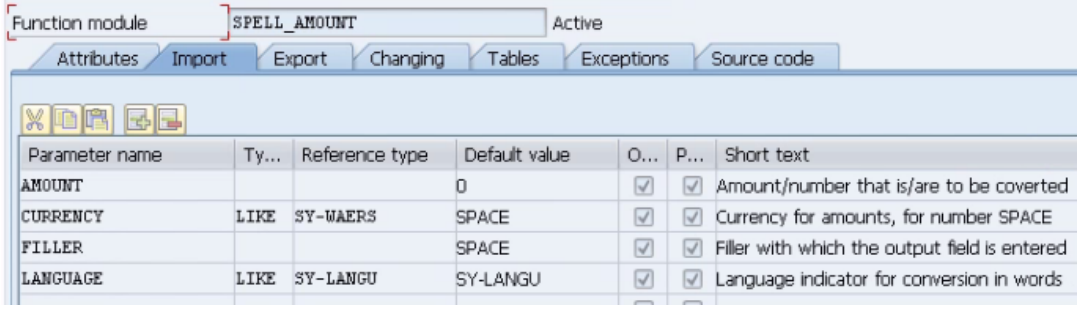


Attributes Tab

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

Import Tab

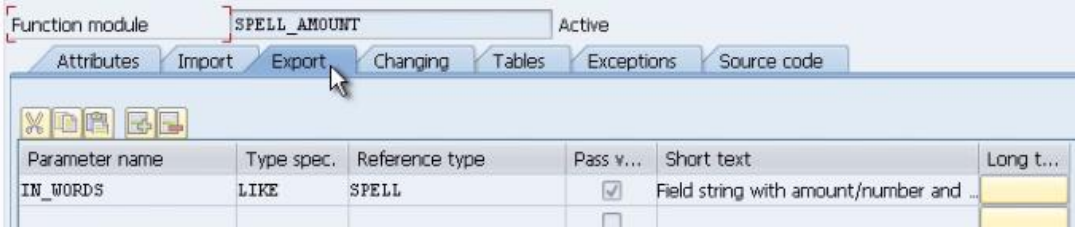
This lists the fields which will be used in the data interface which are passed into the function module from the calling program. These fields are then used by the function module code:



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

Export Tab

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



Changing Tab

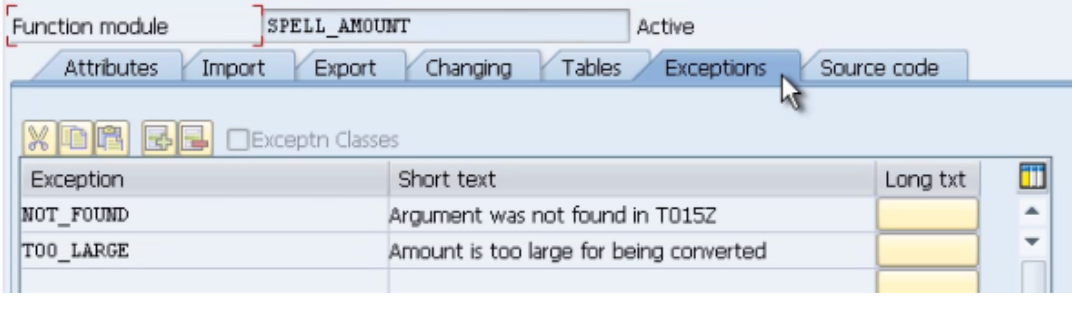
This lists fields which can be *changed* by the function module.

Tables Tab

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

Exceptions Tab

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



Source Code Tab

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

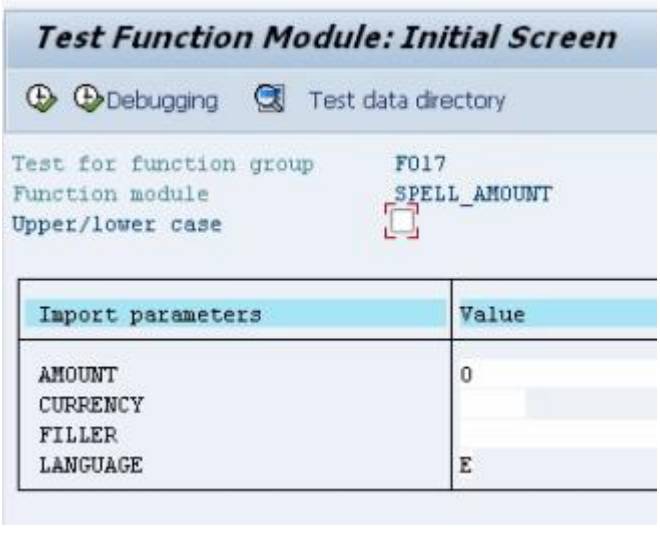
With pre-existing function modules you generally do not even have to look at this, as you should know what data the function module is supposed to send back.

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

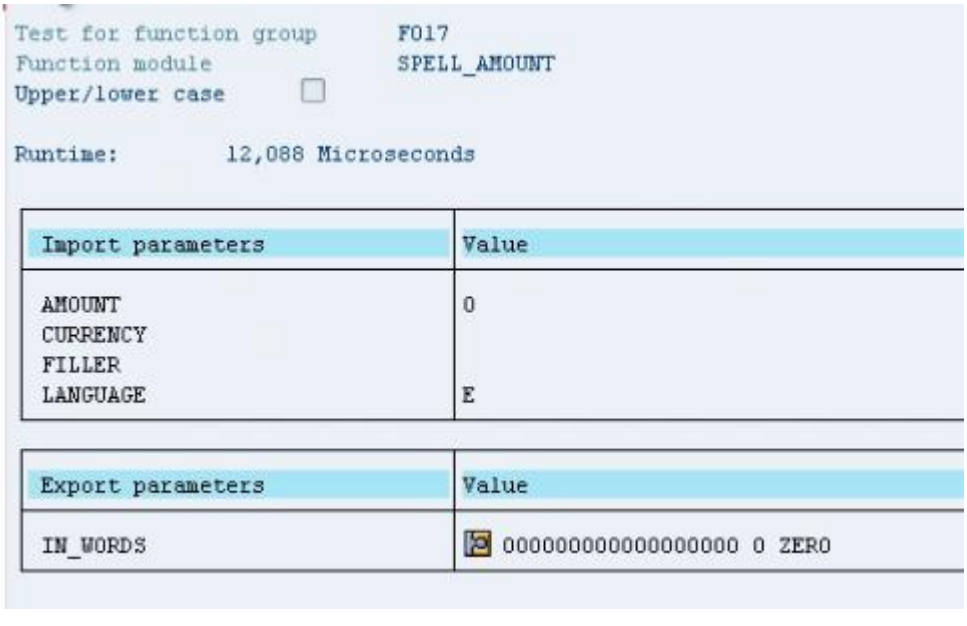
Function Module Testing

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

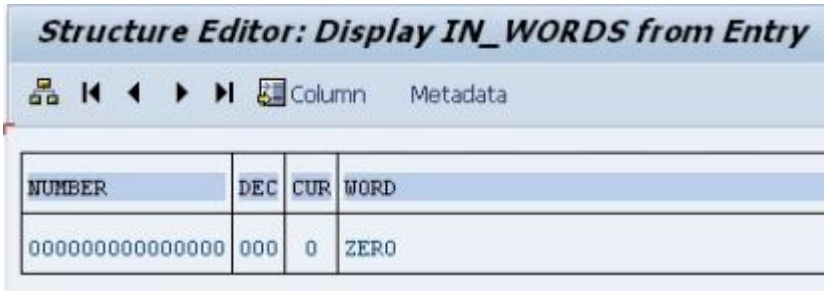
Test out the function module using the Test button as shown above.



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



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



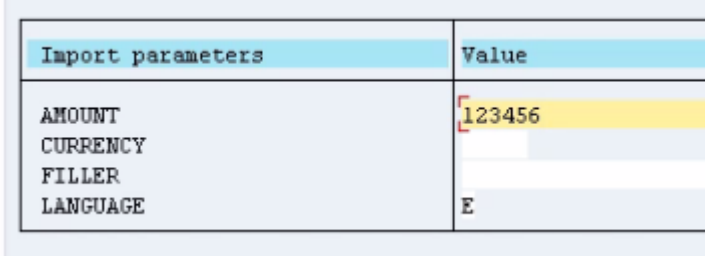
Structure Editor: Display IN_WORDS from Entry

Column Metadata

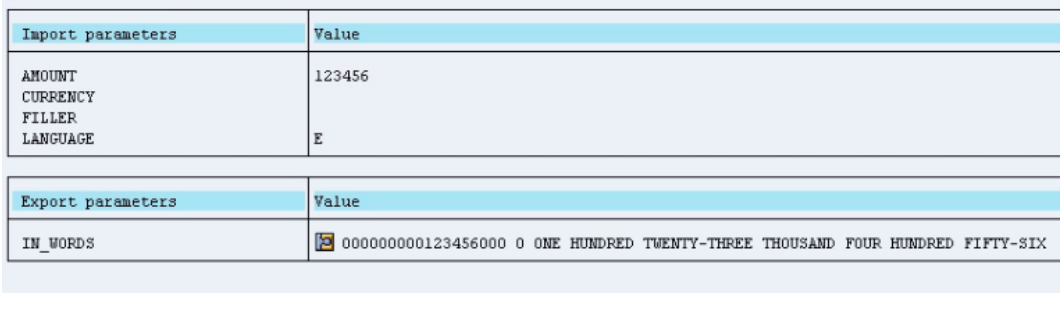
NUMBER	DEC	CUR	WORD
000000000000000	000	0	ZERO

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

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

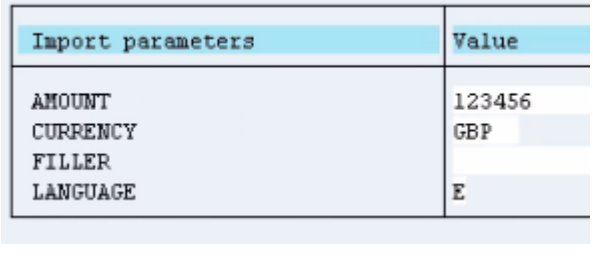


Import parameters	Value
AMOUNT	123456
CURRENCY	
FILLER	
LANGUAGE	E



Import parameters	Value
AMOUNT	123456
CURRENCY	
FILLER	
LANGUAGE	E

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



Import parameters	Value
AMOUNT	123456
CURRENCY	GBP
FILLER	
LANGUAGE	E

Export parameters	Value
IN_WORDS	000000000001234560 2 ONE THOUSAND TWO HUNDRED THIRTY-FOUR

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

NUMBER	DEC	CUR	WORD
000000000001234	560	2	ONE THOUSAND TWO HUNDRED THIRTY-FOUR

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

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

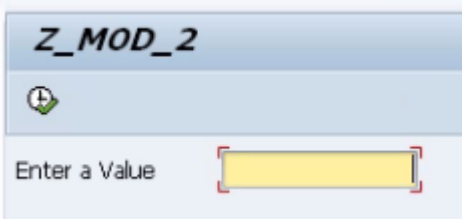
Function Modules - Coding

Now we have successfully tested the function module and know what it does, let's see how we would call it from an ABAP program.

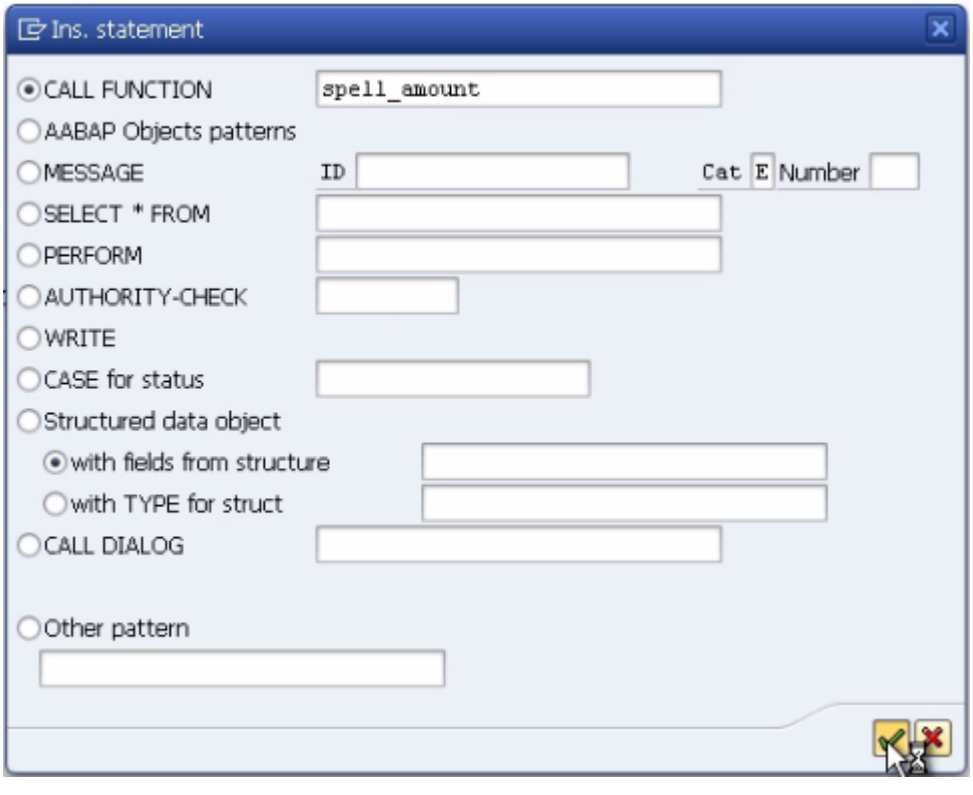
In SE38, create a new program called **Z_MOD_2**. Enter some code so that a parameter can be set up where a value can be entered:

```
REPORT  Z_MOD_2                                 .

SELECTION-SCREEN BEGIN OF LINE.
SELECTION-SCREEN COMMENT 1(15) text-001.
PARAMETER mynum   Type i.
SELECTION-SCREEN END OF LINE.
```

(*The text element text-001 here reads 'Enter a Value'*)



Now a value can be entered into the selection screen, the value can be passed on to the function module. To write the code for this, the 'Pattern' button can be used (also CTRL+F6). It is advisable to always use this as it returns all the variables you need to use automatically. Once this is clicked, a window appears where CALL FUNCTION is the first option in a list. In the text box, enter "**spell_amount**", the function module's name, and click the continue button:



ABAP code is then generated automatically:

```
CALL FUNCTION 'SPELL_AMOUNT'
* EXPORTING
*   AMOUNT          = 0
*   CURRENCY        = ' '
*   FILLER          = ' '
*   LANGUAGE        = SY-LANGU
* IMPORTING
*   IN_WORDS        =
* EXCEPTIONS
*   NOT_FOUND       = 1
*   TOO_LARGE       = 2
*   OTHERS          = 3
          .
IF sy-subrc <> 0.
* MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO
*         WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
ENDIF.
```

Note that a large amount of the code is initially commented out. This is because all of the fields in the function module **spell_amount** are optional. Mandatory fields would not be commented out. The comment character should be removed if that field is then to be used. Here, only the AMOUNT field will be imported to the function module.

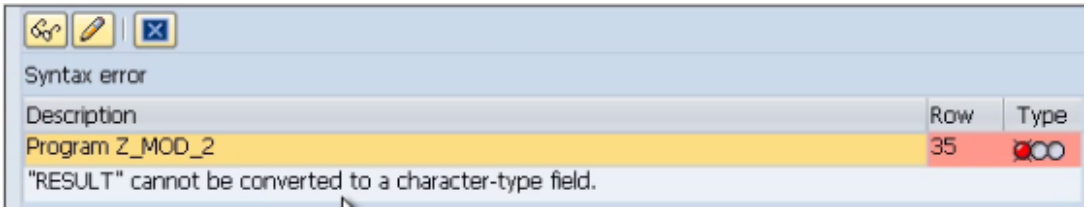
Note the position of the period between the CALL FUNCTION statement, its additions and the IF statement below. It appears entirely on its own line. It appears there as the system does not know how much of the initially commented out code is to be used.

The IF statement was also included automatically so as to follow best practices expecting the programmer to and check whether the function module was executed successfully. As has been discussed before, if **sy-subrc** does not equal zero, there is generally a problem of some kind, so a message can be included to indicate this. Here, depending on the problem, it will be filled with one of the numbers defined in the EXCEPTIONS part of the function module.

Enhance the IF statement to include a code to WRITE a message to the screen to say "**'The function module returned a value of: ', sy-subrc.**", then add the **ELSE** addition, so as to write the correct result out when the function module is successful. This should then read "**WRITE: 'The amount in words is: '**" Here, a new variable must be set up to hold the value returned from the function module. Call this "**result**".

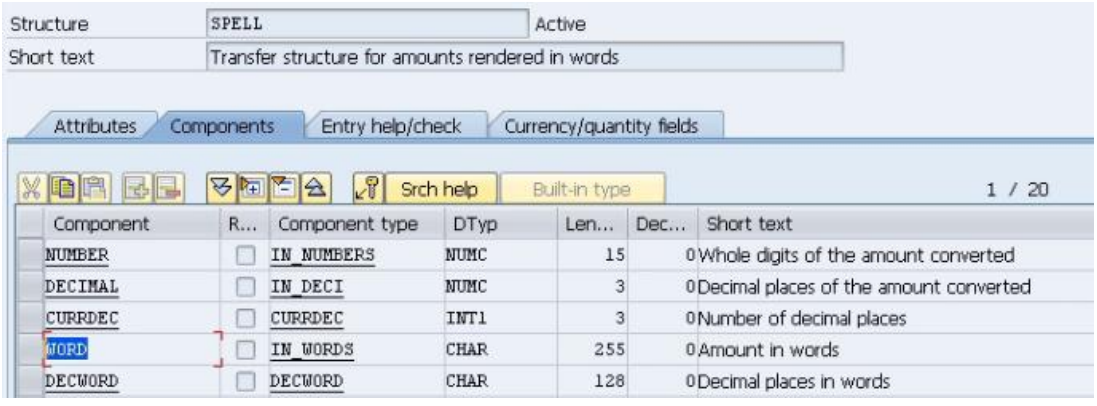
The variable which the function module returns is called **IN_WORDS**, so set up the corresponding variable in the program called result. Use forward navigation to check the data type and length of IN_WORDS so that result can be set up to match. When you do this you will see that IN_WORDS is defined using the LIKE statement to refer to a structure called SPELL.

The same can then be used for the new “result” variable in the main program. Use a DATA statement to declare “result” LIKE the spell structure. As result is defined LIKE a structure with a mix of data types, it is unlikely that the WRITE statement will process it correctly. If you try to use this a syntax error will appear:



Remember that when the function module was tested? What was returned was not in fact only a character field, but a series of numbers as well. If you scroll to the far right of the test results screen you would see several more fields were present.

This means you need to look at the structure of SPELL, and find the component which applies to IN_WORDS:

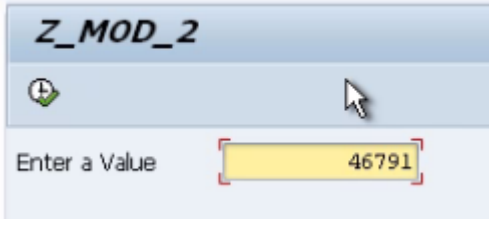


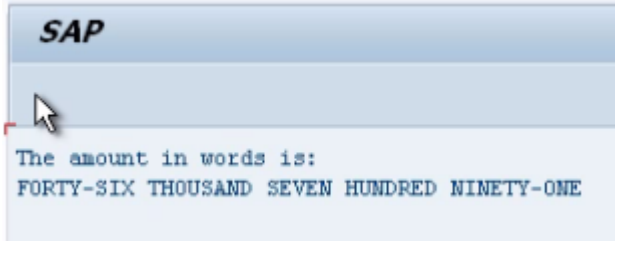
The component is **WORD**, so all you need to do is add WORD to the end of the WRITE statement for result.

The final thing to be changed is to feed in the correct value which the function module will then import. The **AMOUNT** field in the code currently equals 0. Change this to the **mynum** variable used for the selection screen.

The final code should appear like this and can be tested:

```
REPORT  Z_MOD_2                                 .

data result like SPELL.

SELECTION-SCREEN BEGIN OF LINE.
SELECTION-SCREEN COMMENT 1(15) text-001.
PARAMETER mynum   Type i.
SELECTION-SCREEN END OF LINE.

CALL FUNCTION 'SPELL_AMOUNT'
 EXPORTING
   AMOUNT           = mynum
*   CURRENCY         = ' '
*   FILLER           = ' '
*   LANGUAGE         = SY-LANGU
 IMPORTING
   IN_WORDS         = result
* EXCEPTIONS
*   NOT_FOUND        = 1
*   TOO_LARGE        = 2
*   OTHERS           = 3
          .

IF sy-subrc <> 0.
  write: 'The function Module returned a value of: ', sy-subrc.
else.
  write: 'The amount in words is: ', result-word.
ENDIF.
```





Having completed this example, these guidelines can be followed for practically any function module in SAP and, once you have got to grips with how function modules work, it should not be too big a leap to then create your own when necessary.

You have reached the end of this book but don’t stop learning about SAP.

Visit http://www.saptraininghq.com/book-purchase-thankyou to find out about the special bonus offer for all readers of the book as well as read all the additional training material on the website.