



# BC402 ABAP Programming Techniques

BC402

Since the introduction of **ABAP Objects**, there is now a system called the **RTTI** concept (**R**un **T**ime **T**ype **I**nformation) that you can use to find out type attributes at runtime. It is based on system classes. The concept includes all ABAP types, and so covers all of the functions of the statements **DESCRIBE FIELD** and **DESCRIBE TABLE**.

Release 46B 09.01.2003



# BC402

## ABAP Programming Techniques

- System R/3
- Release 4.6A
- September 1999
- Material number: 5003 4163

**Copyright 2001 SAP AG. All rights reserved.**

**Neither this training manual nor any part thereof may be copied or reproduced in any form or by any means, or translated into another language, without the prior consent of SAP AG. The information contained in this document is subject to change and supplement without prior notice.**

**All rights reserved.**

© SAP AG 2001

■ **Trademarks:**

- Microsoft®, Windows®, NT®, PowerPoint®, WinWord®, Excel®, Project®, SQL-Server®, Multimedia Viewer®, Video for Windows®, Internet Explorer®, NetShow®, and HTML Help® are registered trademarks of Microsoft Corporation.
- Lotus ScreenCam® is a registered trademark of Lotus Development Corporation.
- Vivo® and VivoActive® are registered trademarks of RealNetworks, Inc.
- ARIS Toolset® is a registered Trademark of IDS Prof. Scheer GmbH, Saarbrücken
- Adobe® and Acrobat® are registered trademarks of Adobe Systems Inc.
- TouchSend Index® is a registered trademark of TouchSend Corporation.
- Visio® is a registered trademark of Visio Corporation.
- IBM®, OS/2®, DB2/6000® and AIX® are a registered trademark of IBM Corporation.
- Indeo® is a registered trademark of Intel Corporation.
- Netscape Navigator®, and Netscape Communicator® are registered trademarks of Netscape Communications, Inc.
- OSF/Motif® is a registered trademark of Open Software Foundation.
- ORACLE® is a registered trademark of ORACLE Corporation, California, USA.
- INFORMIX®-OnLine for SAP is a registered trademark of Informix Software Incorporated.
- UNIX® and X/Open® are registered trademarks of SCO Santa Cruz Operation.
- ADABAS® is a registered trademark of Software AG
- The following are trademarks or registered trademarks of SAP AG; ABAP™, InterSAP, RIVA, R/2, R/3, R/3 Retail, SAP (Word), SAPaccess, SAPfile, SAPfind, SAPmail, SAPoffice, SAPscript, SAPtime, SAPtronic, SAP-EDI, SAP EarlyWatch, SAP ArchiveLink, SAP Business Workflow, and ALE/WEB. The

SAP logo and all other SAP products, services, logos, or brand names included herein are also trademarks or registered trademarks of SAP AG.

- Other products, services, logos, or brand names included herein are trademarks or registered trademarks of their respective owners.

# ABAP Workbench



## Level 2

**BC400** 5 days

ABAP Workbench:  
Concepts and Tools

**MBC40** 2 days

Managing ABAP  
Development Projects

## Level 3

**BC402** 3 days

ABAP Programming  
Techniques

**BC404** 3 days

ABAP Objects: Object-  
Oriented Programming  
in R/3

**BC405** 3 days

Techniques of List  
Processing and SAP Query

**BC410** 5 days

Programming  
User Dialogs

**BC420** 5 days

Data Transfer

**BC430** 2 days

ABAP Dictionary

**BC460** 3 days

SAPscript: Forms Design  
and Text Management

**CA610** 2 days

CATT: Test Workbench and  
Computer Aided Test Tool

**BC414** 2 days

Programming  
Database Updates

**BC415** 2 days

Communication  
Interfaces in ABAP

**BC425** 3 days

Enhancements  
and Modifications

**BC412** 2 days

Dialog Programming  
using EnjoySAP Controls

**BC440** 5 days

Developing  
Internet Applications

**BC490** 3 days

ABAP Performance  
Tuning

Recommended supplementary  
courses are:  
Business Process Technologies  
**CA925, CA926, CA927**  
**BC095** (Business Integration  
Technology)  
**BC619** (ALE), **BC620, BC621**

- **SAP 50 (Basis Technology)**
- **BC400 (ABAP Workbench: Concepts and Tools)**
- **Recommended:**
  - **BC430 (ABAP Dictionary)**
  - **A little ABAP programming experience**

- **Audience:**
  - **Programmers**
  - **Consultants**
- **Duration: 3 days**



© SAP AG 1999

### ■ Notes to the participant

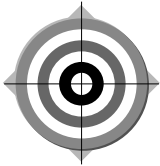
This course material is not a **self-teach program**. It is **only complete in conjunction with the instructor's explanations**. There is space in the course book for you to write your own notes.

## Contents

- **Course Goal**
- **Course Objectives**
- **Course Content**
- **Course Overview Diagram**
- **Main Business Scenario**

© SAP AG 1999





### **This course will enable you to:**

- Understand the basic principles of the ABAP programming language
- Choose the appropriate ABAP programming technique to solve a particular problem
- Write and maintain your own ABAP programs



**At the conclusion of this course, you will be able to:**

- Describe how the ABAP runtime system works
- Use ABAP statements and their syntax variants
- Create, test, compare, and classify ABAP programs
- Design, write, and modify your own ABAP programs
- Evaluate alternative programming solutions based on their assurance of data consistency, possible runtime errors, and the effectiveness of the modularization techniques they use

## Preface

---

Unit 1	<b>Course Overview</b>	Unit 6	<b>Subroutines</b>
Unit 2	<b>ABAP Runtime Environment</b>	Unit 7	<b>Function Groups and Function Modules</b>
Unit 3	<b>Data Types and Objects</b>	Unit 8	<b>Introduction to ABAP Objects</b>
Unit 4	<b>Statements</b>	Unit 9	<b>Calling Programs and Passing Data</b>
Unit 5	<b>Internal Table Operations</b>		

---

The exercises and solutions are at the end of the relevant unit.

## Appendix

Preface



**Course Overview**

ABAP Runtime Environment

Data Types and Data Objects

Statements

Internal Table Operations

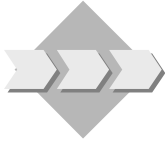
Subroutines

Function Groups and Function Modules

Introduction to ABAP Objects

Calling Programs and Passing Data

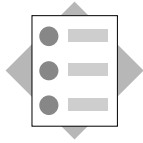
© SAP AG 1999



- You will use the ABAP Workbench to develop simple business applications. These are based on the tables, data, and ABAP Dictionary objects of the flight data model. In this course, you will only read the flight data. You will process the data in your programs, but the actual data in the database will not be changed.
- You will also use, analyze, copy, and modify existing Repository objects.

## Contents

- **Components of an ABAP program**
- **Processors within a work process**
- **ABAP programs: Types and execution methods**



**At the conclusion of this unit, you will be able to:**

- **Name the components of an ABAP program**
- **Describe how an ABAP program is organized**
- **Take technical aspects of program execution into account when you write an ABAP program**

Preface

Course Overview



**ABAP Runtime Environment**

Data Types and Data Objects

Statements

Internal Table Operations

Subroutines

Function Groups and Function Modules

Introduction to ABAP Objects

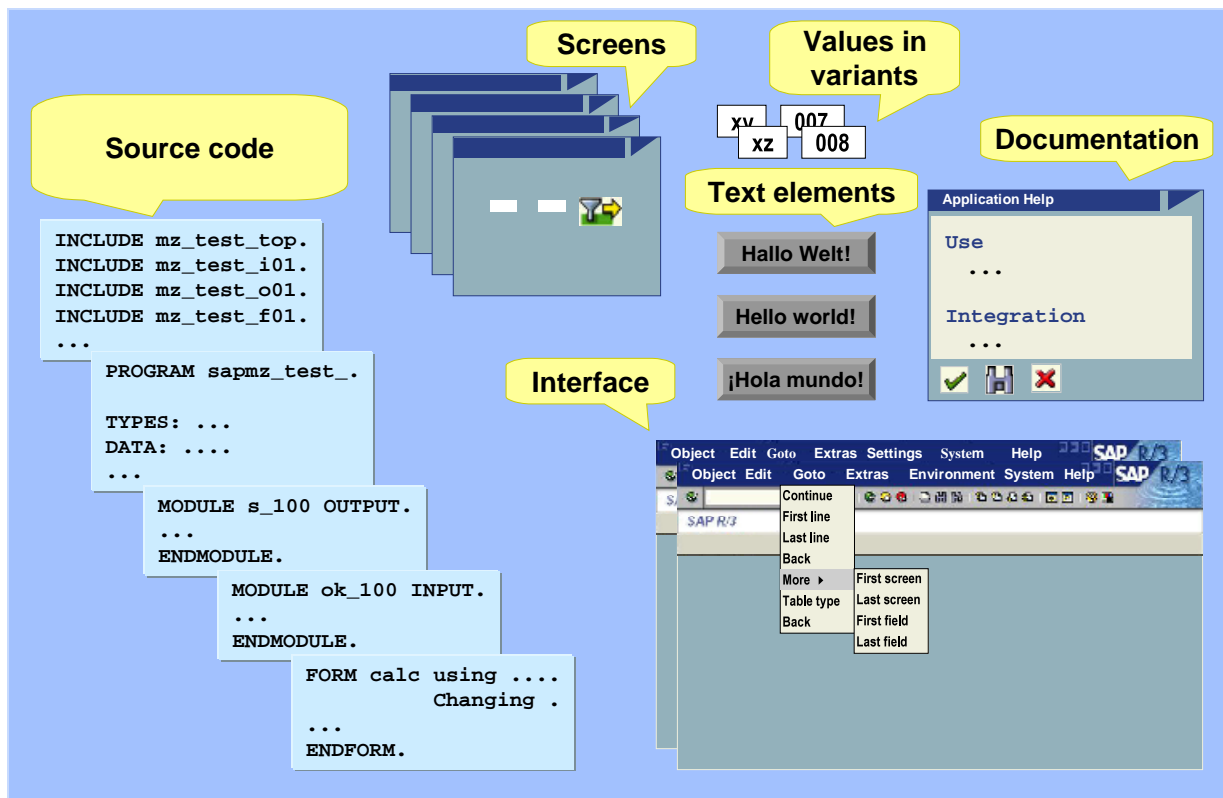
Calling Programs and Passing Data

© SAP AG 1999



# Components of an ABAP Program

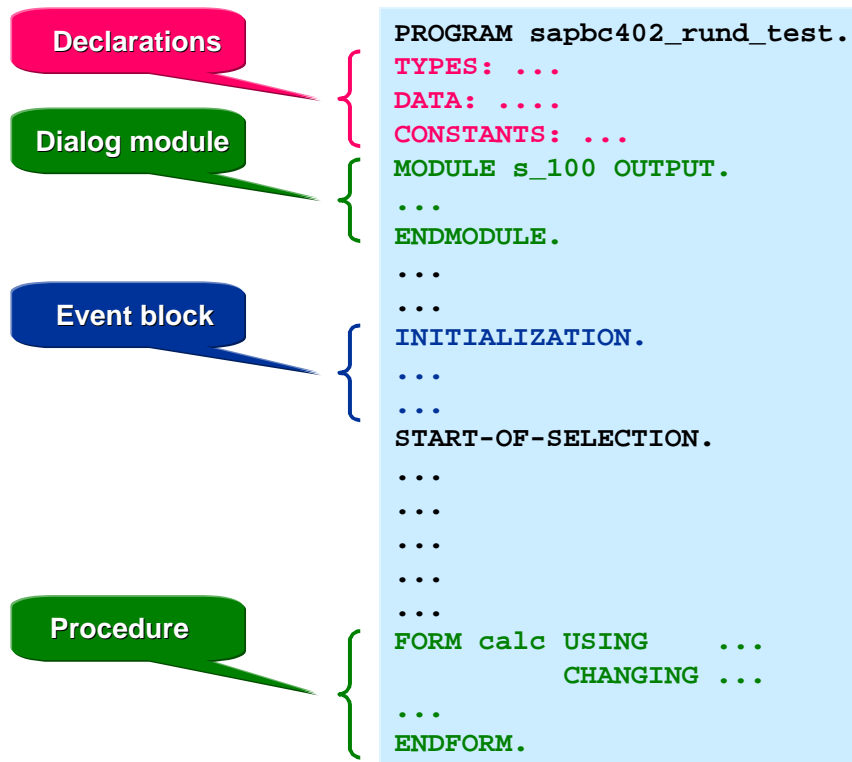
SAP



© SAP AG 1999

An ABAP program contains the following components:

- **Source code**  
...containing the ABAP statements
- **Screens**  
... consist of the screen **layout** and associated **flow logic**. You normally create the layout of a screen using the Screen Painter. However, there are special kinds of screens, called selection screens and lists, whose layout and flow logic are designed exclusively using ABAP statements.
- **Interface**  
...contains all of the entries in the menus, the standard toolbar, the application toolbar, and function key settings. It contains titles and statuses. A status is a collection of function key settings and menus.
- **Text elements**  
... are language-specific. They can be translated either directly from the text element maintenance tool, or using a special translation tool.
- **Documentation**  
... is also language-specific. Always write documentation from the user's point of view. If you want to document the programming techniques you have used, use comments in the program code instead.
- **Variants**  
... allow you to predefine the values of input fields on the selection screen of a program.



© SAP AG 1999

ABAP is an **event-driven** programming language, and as such is suited to processing user dialogs. The source code of an ABAP program consists of **two** parts:

## ■ Declarations

Declarations include the statements for global data types and objects, selection screens, and (in ABAP Objects) local classes and interfaces within the program.

## ■ Processing Blocks (indivisible program units)

Each processing block must be programmed as a single entity. There are two basic kinds of processing blocks:

### Event Blocks:

Event blocks are introduced by an event keyword. They are not concluded explicitly, but end when the next processing block starts.

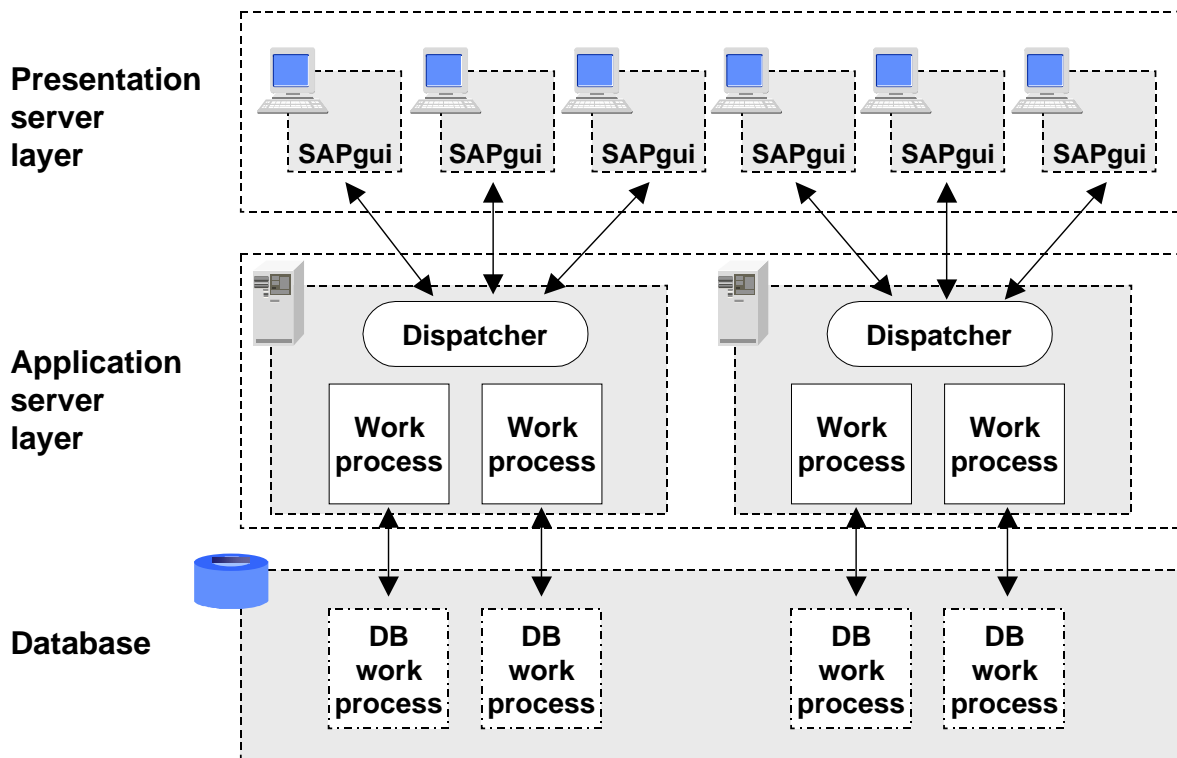
### Dialog Modules and Procedures:

Dialog modules and procedures are introduced **and concluded** using keywords.

The contents of all processing blocks form the processing logic.

When you **generate** the program, these parts are **compiled** to form the load version. This is **interpreted** at runtime.

## The Three-Tier Client/Server Architecture of the R/3 System

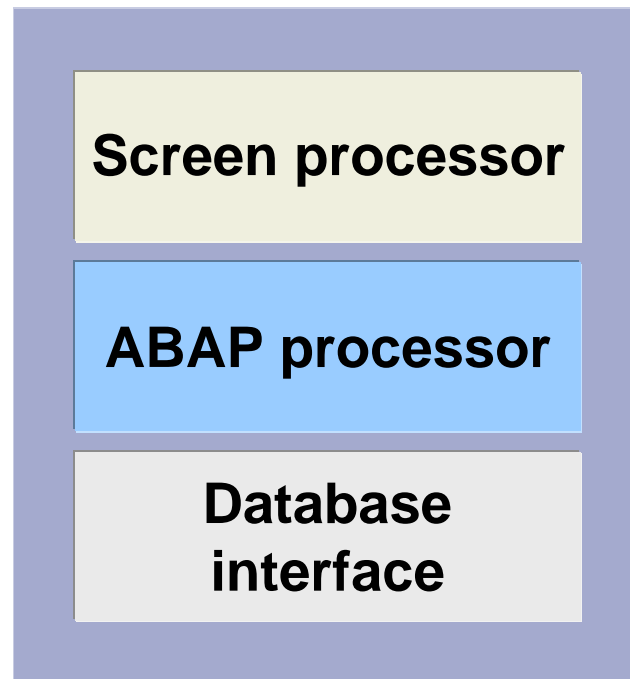


© SAP AG 1999

The R/3 System is based on a client/server architecture with the three tiers database server, application server, and presentation server. It allows a large number of users with **inexpensive** and relatively **slow** machines to take advantage of a smaller number of **faster, expensive** application servers by occupying work processes on them. Each work process on an application server is assigned to a work process on the (expensive, even more powerful) database server.

**User dispatching** is the process by which the individual clients at presentation server level are assigned to a work process for a particular length of time. The work process in turn is linked to a work process in the database. Once the user input from a dialog step has been processed, the user and program context is "rolled out" of the work process so that the work process can be used for another dialog step from another user while the first user is making entries on the next screen. This makes the best possible use of the resources available on the application server.

The three-tier architecture makes the system easily scalable. To add extra users, you merely have to install more inexpensive presentation servers. You can also increase the efficiency of the whole system by adding extra application servers with their associated work processes.



© SAP AG 1999

The work processes in the middle layer - often called the *application server* - are software components that are responsible for processing dialog steps. They are implemented as "virtual machines". This ensures that ABAP programs can run independently of the hardware platform on which the R/3 System is installed.

Work processes contain other software components that are responsible for various tasks within a dialog step:

■ **Screen processor**

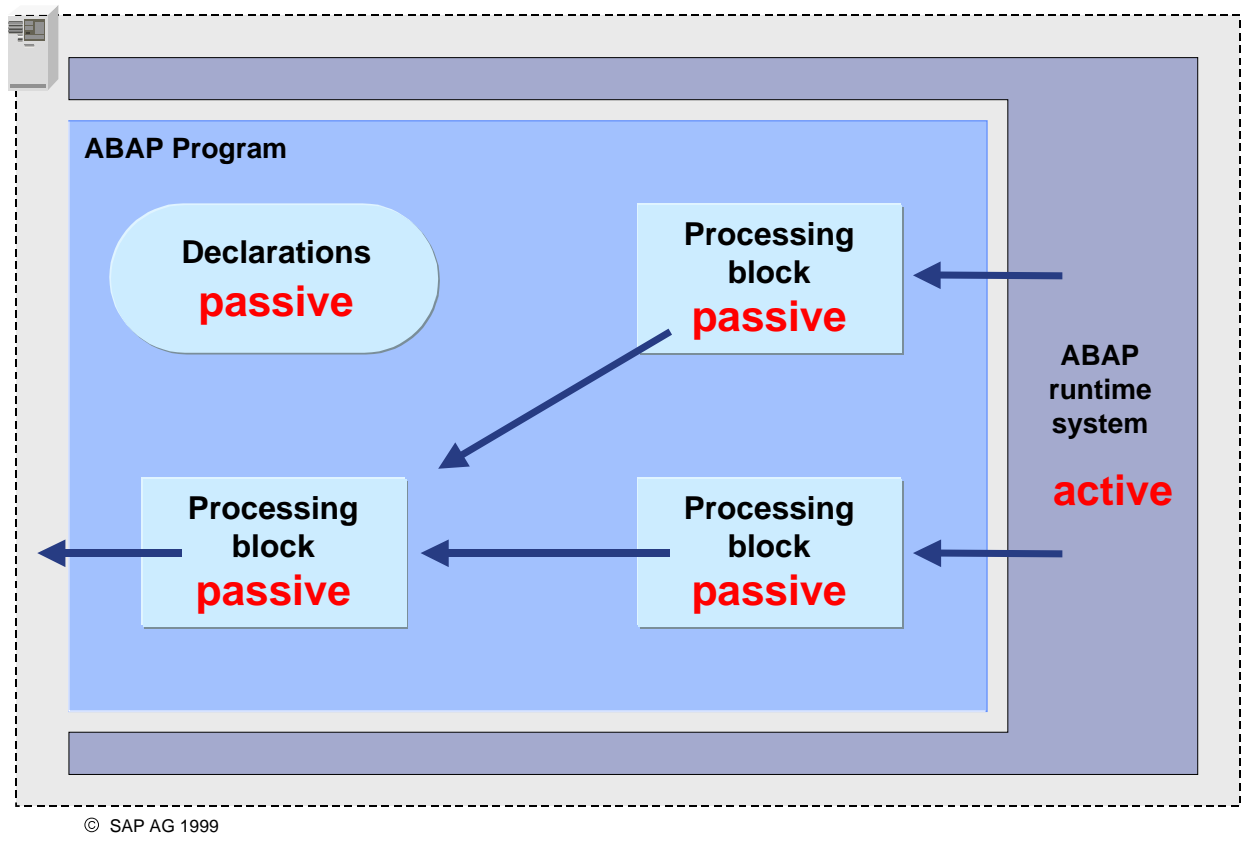
The screen processor is responsible for communication between the SAPgui and the work process (via the dispatcher). It processes the screen flow logic and passes field contents to the processing logic in the program.

■ **ABAP processor**

The ABAP processor executes the processing logic in the ABAP program and communicates with the database interface. The screen processor tells the ABAP processor which part of the program (module) needs to be processed (according to the screen flow logic).

■ **Database interface**

The database interface is responsible for the communication with the database. It allows access to tables and Repository objects (including ABAP Dictionary objects), controls transaction execution (COMMIT and ROLLBACK), and administers the table buffer on the application server.



The individual processing blocks are called in a predetermined sequence at runtime, regardless of the position in which they occur in the program. Once a processing block has been called, the statements within it are processed sequentially.

## ■ Event block

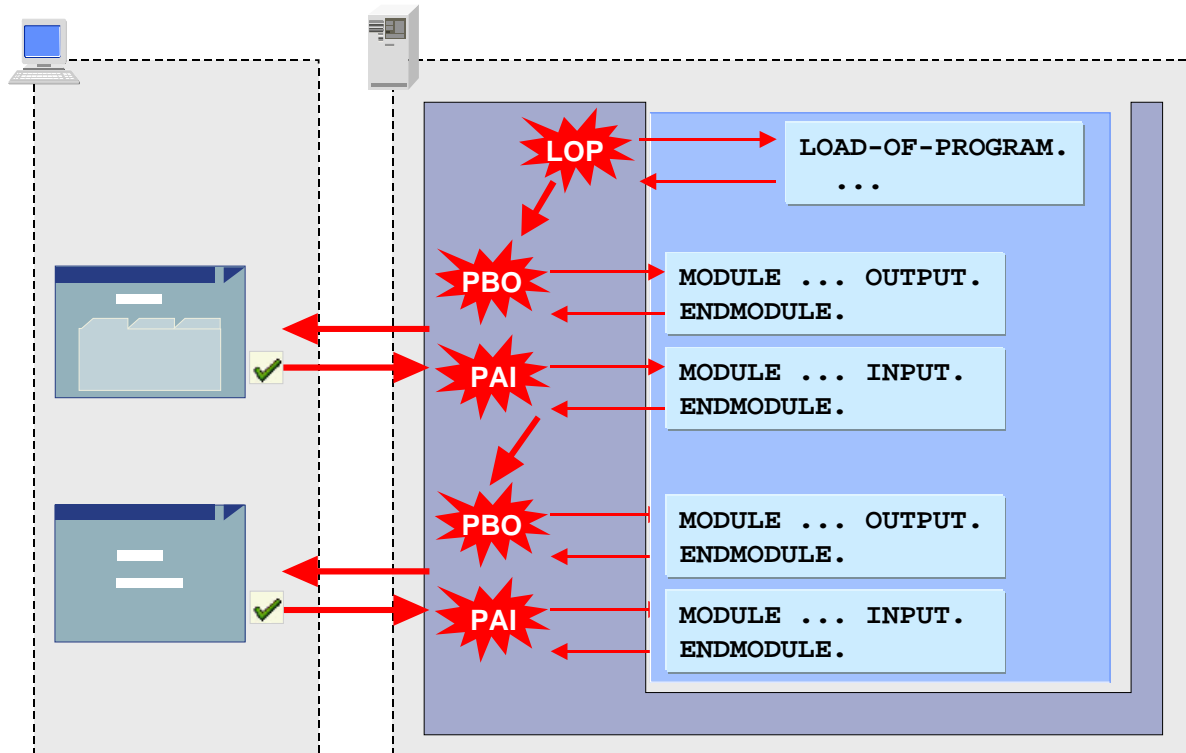
If the system program or a user triggers an event for which the corresponding event block has been written in the processing logic, that event block is processed. The program flow is controlled either by the system or the user.

## ■ Modularization unit

When the system encounters a modularization unit call within a processing block, it calls the corresponding processing block. In this case, the program flow is controlled by the programmer.

### Assigning transaction codes

To allow a module pool to be executed, you **must** assign a transaction code to it. You **can** (but do not have to) assign a transaction code to an executable (type 1) program.



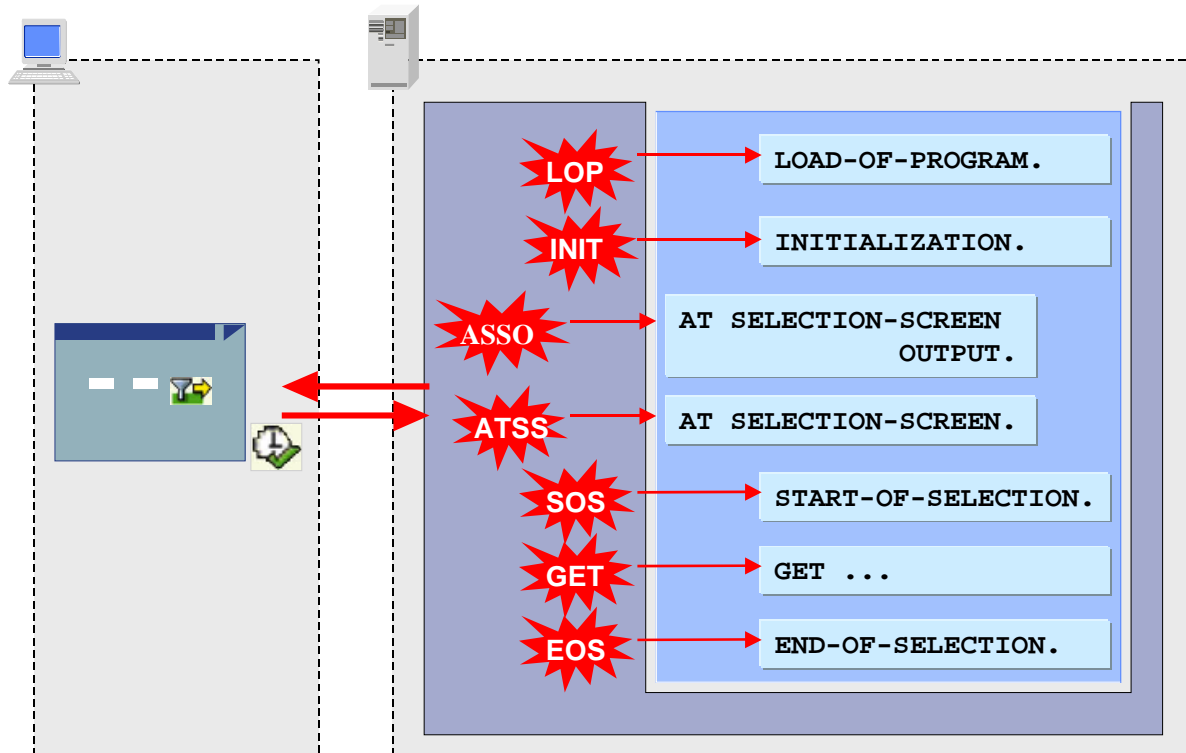
© SAP AG 1999

You assign a **dialog transaction** to a module pool. The following steps occur when you run a dialog transaction:

- First, the **LOAD-OF-PROGRAM** event is triggered. Once this event block has been executed, the ABAP processor passes control to the screen processor. For an example of how to use this new event, refer to the example in the **Function Groups and Function Modules** unit.
- The screen processor processes the initial screen specified in the transaction definition. The initial screen can be a selection screen (regardless of the program type). The **PROCESS BEFORE OUTPUT** event is triggered and control passes to the ABAP processor, which processes the first PBO module.
- The ABAP processor executes the processing block and returns control to the screen processor. Once all PBO modules have been processed, the contents of any ABAP fields with identically-named corresponding fields on the screen are transported to the relevant screen fields. Then the screen is displayed (screen contents, **active** title, **active** status).
- Once the user has chosen a dialog function (such as ENTER), the contents of the screen fields are transported back to the corresponding identically-named fields in the ABAP program, and the processing blocks that belong to the **PROCESS AFTER INPUT** event are processed. The system then continues by processing the next screen.

The only processing logic that is processed in a dialog transaction are the statements belonging to the **LOAD-OF-PROGRAM** event and those occurring in the various modules.

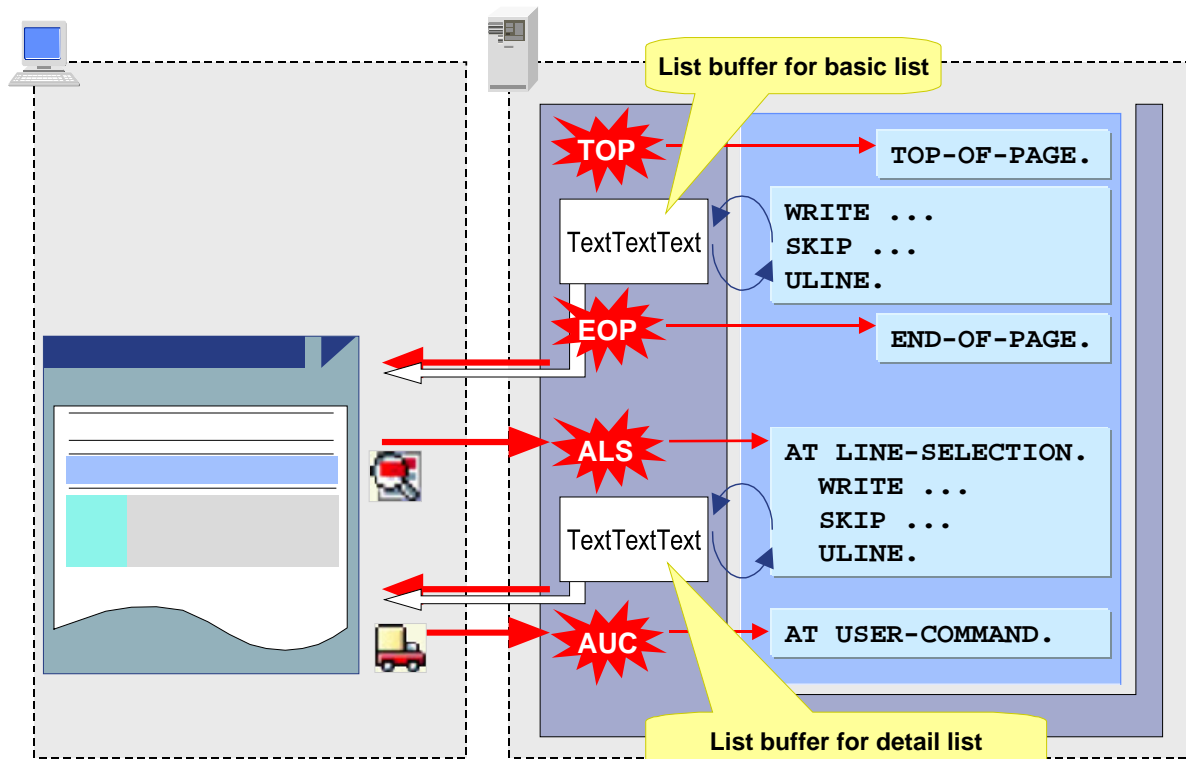
However, you can also use the statement **LEAVE TO LIST-PROCESSING**. This makes all of the list processing events available to you.



© SAP AG 1999

You can **only** assign a **report transaction** to an executable (type 1) program. In a report transaction, the system calls particular events in a fixed sequence, and calls a series of standard screens. The following steps occur when you run a report transaction:

- First, the **LOAD-OF-PROGRAM** event is triggered.
- Then the **INITIALIZATION** event is triggered.
- Next, the standard selection screen is called (if you have declared one), and its corresponding events are triggered: **AT SELECTION-SCREEN OUTPUT** and **AT SELECTION-SCREEN**.
- Next, the **START-OF-SELECTION** event is triggered. (This is the **default event block**. If you omit this event keyword, all statements that are not assigned to another processing block are treated as though they belong to it.)
- If you have attached a logical database to your program, the system triggers the **GET <node>** and **GET <node> LATE** events.
- Then the **END-OF-SELECTION** event is triggered.
- You can also include screen processing (as in module pools) by using the **CALL SCREEN** statement. You can start executable (type 1) programs without using a transaction code. You can also run them in the background.



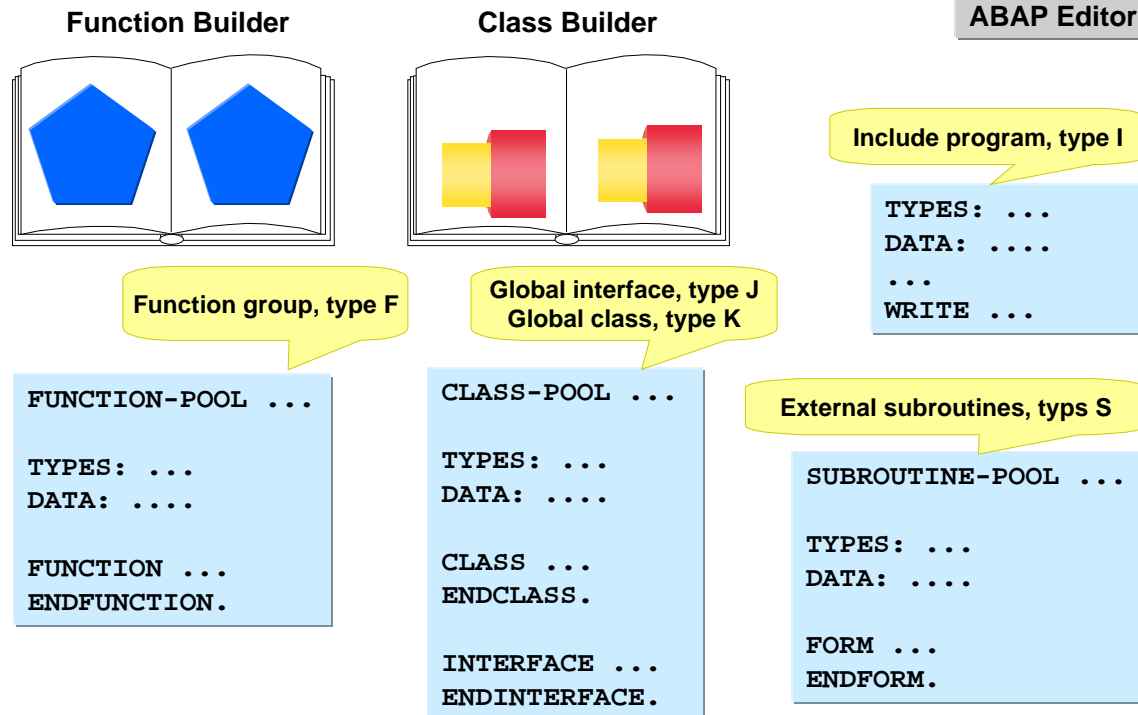
© SAP AG 1999

If you fill the list buffer of the basic list (using the **WRITE**, **SKIP**, and **ULINE** statements), two further events are triggered: At the beginning of each new page, the **TOP-OF-PAGE** event is triggered, and the **END-OF-PAGE** event is triggered at the end of each page.

Once the **END-OF-SELECTION** event block has been processed, **interactive list processing** starts. The system displays the formatted **basic list**. The **user** can now trigger further events.

- If the user double-clicks a line or triggers the function code **PICK** in some other way, the **AT LINE-SELECTION** event is triggered. In the standard list status, this function code is always assigned to function key <F2>. In turn, <F2> is always assigned for a mouse double-click.
- If you fill the list buffer of the detail list (of which you may have up to twenty) using the **WRITE**, **SKIP**, and **ULINE** statements, two further events are triggered:  
At the beginning of each new page, the **TOP-OF-PAGE DURING LINE-SELECTION** event is triggered, and the **END-OF-PAGE DURING LINE-SELECTION** event is triggered at the end of each page. (These events are not displayed in the graphic.) **Interactive list processing** is started again. The system displays the formatted **detail list** (screen 120).
- Any other function codes that have not been "trapped" by the system trigger the event **AT USER-COMMAND**.



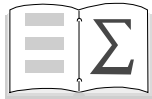


© SAP AG 1999

The following types of programs **cannot** be executed directly. Instead, they serve as containers for modularization units that you call from other programs. When you call one of these modularization units, the system always loads its entire container program.

Further information about this is provided later on in the course.

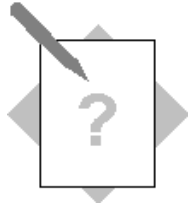
- **Function group** (type F)  
A function group can contain function modules, local data declarations for the program, and screens.  
For further information, refer to the **Function Groups and Function Modules** unit.
- **Include program** (type I)  
An include program can contain any ABAP statements.  
For further information, refer to the **Program Organization** section of this unit.
- **Global interface** (type J)  
An interface pool can contain global interfaces and local data declarations.  
For further information, refer to the **Introduction to ABAP Objects** unit.
- **Global class** (type K)  
A class pool can contain global classes and local data declarations.  
For further information, refer to the **Introduction to ABAP Objects** unit.
- **Subroutine pool** (type S) (external subroutines)  
A subroutine pool can contain subroutines and local data declarations.  
**Caution! Type S programs are obsolete and have been replaced by function groups.**



**You are now able to:**

- **Name the components of an ABAP program**
- **Describe how an ABAP program is organized**
- **Take technical aspects of program execution into account when you write an ABAP program**

## ABAP Runtime Environment: Exercises

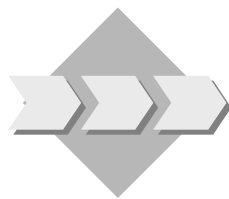


**Unit:** ABAP Runtime Environment  
**Topic:** Creating Repository Objects



At the conclusion of these exercises, you will be able to:

- Create development classes
- Create programs



You are a programmer for an airline consortium, and it is your job to write analysis programs for several airlines.

1. Log onto the operating system and then to the R/3 training system (your instructor will tell you its name) with the user name **BC402-##**. Enter a new password.  
**## is your two-digit group number.**

2. You need to create a development class as a container for your Repository objects. The development class must be assigned to a change request. You also need to create two programs.  
**## is your two-digit group number.**

Model solutions:

BC402

SAPBC402\_TYPS\_COUNTERLIST1

SAPBC402\_TYPS\_FLIGHTLIST1

2-1 Create the development class **Z##\_BC402**.

2-2 Create an executable (type 1) program **Z##\_BC402\_COUNTERLIST1**  
**without** a TOP include.

2-3 Create an executable program **Z##\_BC402\_FLIGHTLIST1**  
**without** a TOP include.



From this point onwards, you should always work with the **Object Navigator**. This provides you with an overview of all of the Repository objects in your development class. From here, you can select objects you want to work on.





Unit: ABAP Runtime Environment  
Topic: Creating Repository objects

### 2-2 Model solution SAPBC402\_TYPS\_COUNTERLIST1

```
*&-----*
*& Report SAPBC402_TYPS_COUNTERLIST1          *
*&                                           *
*&-----*
*& solution to exercise 1 data types and data objects .    *
*&                                           *
*&-----*
```

**REPORT sapbc402\_typs\_counterlist1.**

### 2-3 Model solution SAPBC402\_TYPS\_FLIGHTLIST1

```
*&-----*
*& Report SAPBC402_TYPS_FLIGHTLIST1          *
*&                                           *
*&-----*
*& solution of exercise 2 datatypes and dataobjects      *
*&                                           *
*&-----*
```

**REPORT sapbc402\_typs\_flightlist1.**

## Contents

- Kinds of data types
- Defining data types
- Kinds of data objects and how to declare them
- Field symbols and references



**At the conclusion of this unit, you will be able to:**

- **Differentiate between the various kinds of data types and data objects**
- **Define data types and declare data objects**
- **Use field symbols and references**

Preface

Course Overview

ABAP Runtime Environment



**Data Types and Data Objects**

Statements

Internal Table Operations

Subroutines

Function Groups and Function Modules

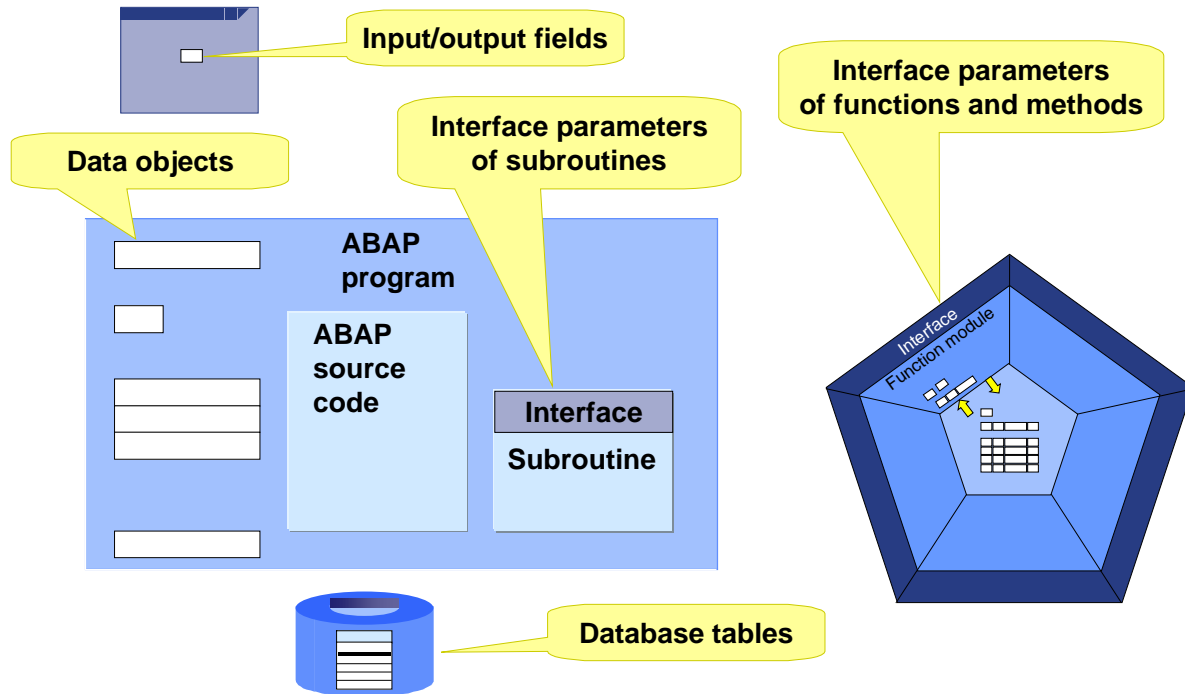
Introduction to ABAP Objects

Calling Programs and Passing Data

© SAP AG 1999



## Types describe the attributes of...



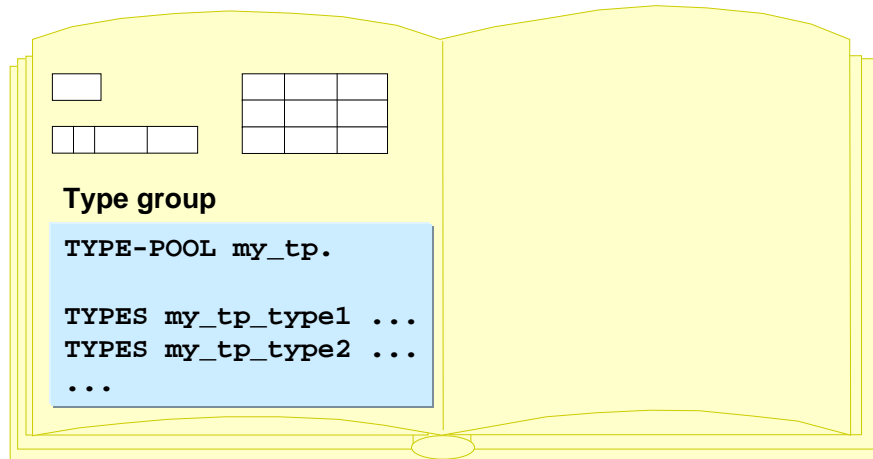
© SAP AG 1999

To work with data at runtime, you need to store it in the program at runtime and address it from your program. The system needs to know the **type** of the data (for example, character string, whole number, table consisting of name, currency amount, and date, and so on). A **data object** is a named memory area that is structured according to a particular **data type**. The type normally specifies all of the attributes of the data object. Using the name, you can access the contents, that is the data, directly. The name may be a compound name consisting of more than one single name.

You could regard a data type as being similar to the construction plans for a building. The plans could be used for more than one building, which would all have the same type, but you would still be able to tell them apart. Suppose the buildings were used for storage. You would find a particular item using the address of the building, and knowing on which floor, in which room, and in which shelf or bin it was stored. You would have to consider carefully when drawing up your plans the kinds of things you would want to store in your buildings.

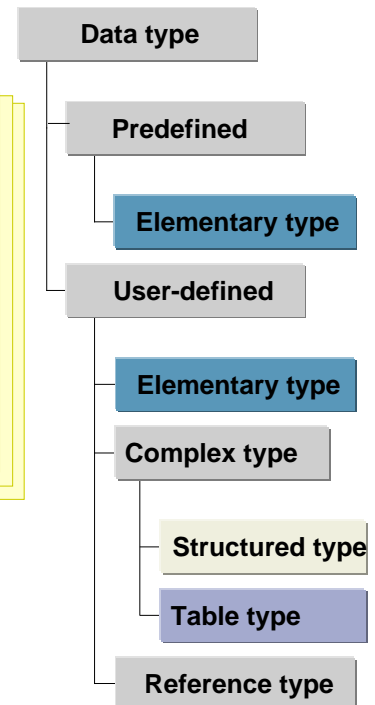
The ABAP language is very flexible. Some of the attributes of a data type do not have to be specified until you use it to declare a data object, or, in some cases, not until runtime. It also allows you to use data objects that you have already declared or ABAP Dictionary objects as the basis for new types or data objects.

## ABAP Dictionary (global type)



## ABAP program (local type in program)

```
...
TYPES local_type ...
...
```



© SAP AG 1999

There are various places in the ABAP Workbench in which you can store and define data types:

### ■ ABAP Dictionary

The ABAP Dictionary contains 23 predefined data types, which serve as a basis for all other ABAP Dictionary objects (such as domains, data elements, data types, and so on). These ABAP Dictionary types are available for use globally throughout the system.

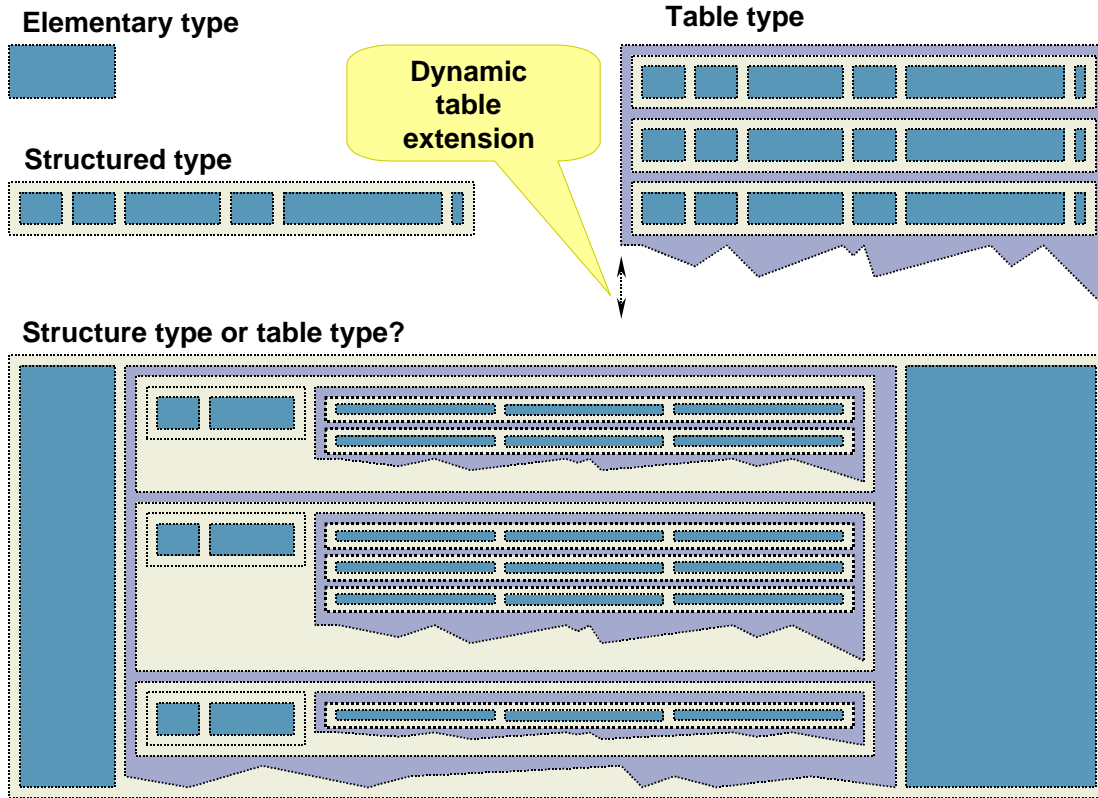
As well as the Dictionary objects used to access tables (tables, views, search helps, and so on), you can also (from Release 4.5) **create global data types** in the ABAP Dictionary.

Previously, the only way to define global data types was to use a **type group**. Type groups are still supported, but the concept is actually obsolete now that it is possible to define global data types in the ABAP Dictionary.

### ■ ABAP programs

Data types that you define in an ABAP program are **local**, that is, only valid within that program. You use the ten predefined ABAP data types as a basis for your own types.

Both global and local data types fit into the schematic diagram above. The names used above should make it easier for you to understand the following slides and the online documentation.



© SAP AG 1999

- The technical attributes of an elementary field are defined by an **elementary type**.
- A structure type consists of **components**.
- A table type consists of a **line type**, **access type**, **key definition**, and **key type**.
- In certain exceptional cases, types only describe part of the attributes of a data object. For example, a table type does not specify how many lines the table will have. This attribute is not set until runtime, and only affects that one data object.
- You can nest types "deeply" to any level. That means:  
A structured type can have components that are themselves structured or table types. This enables you to construct very complex data types. However, the smallest indivisible unit is always an elementary type.

## Predefined ABAP Dictionary Types

SAP

<b>ACCP</b> .....	Accounting period YYYYMM
<b>CHAR</b> .....	Character string
<b>CLNT</b> .....	Client
<b>CUKY</b> .....	Currency key, referenced by a <b>CURR</b> field
<b>CURR</b> .....	Currency field, stored as <b>DEC</b>
<b>DATS</b> .....	Date field (YYYYMMDD), stored as <b>CHAR(8)</b>
<b>DEC</b> .....	Calculation or amount field, with plus or minus sign
<b>FLTP</b> .....	Floating point number with eight-byte accuracy
<b>INT1</b> .....	1 byte integer. Whole number <= 255
<b>INT2</b> .....	2 byte integer. Only for length field before <b>LCHR</b> or <b>LRAW</b>
<b>INT4</b> .....	4 byte integer. Whole number with plus or minus sign
<b>LANG</b> .....	Language key
<b>LCHR</b> .....	Long character string. Must be preceded by an <b>INT2</b> field.
<b>LRAW</b> .....	Long byte string. Must be preceded by an <b>INT2</b> field.
<b>NUMC</b> .....	Character string containing only digits
<b>PREC</b> .....	Accuracy of a <b>QUAN</b> field.
<b>QUAN</b> .....	Quantity field. Points to a units field with the type <b>UNIT</b> .
<b>RAW</b> .....	Uninterpreted byte sequence
<b>TIMS</b> .....	Time field (HHMMSS), stored as <b>CHAR(6)</b>
<b>VARC</b> .....	Long character string (not supported after Release 3.0)
<b>STRING</b> .....	Character string of variable length
<b>RAWSTRING</b> ...	Byte sequence of variable length
<b>UNIT</b> .....	Unit key for a <b>QUAN</b> field

Not  
elementary  
types

© SAP AG 1999

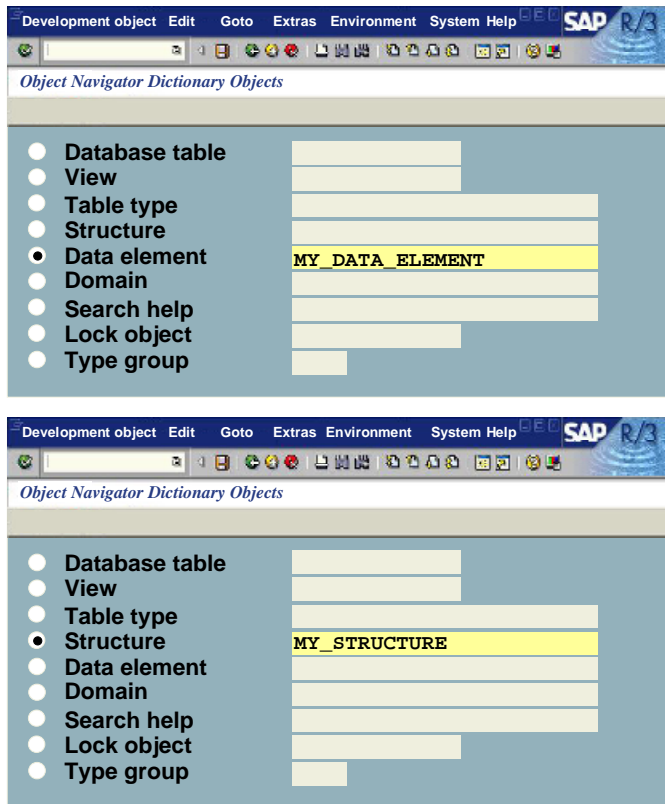
The ABAP Dictionary contains a series of predefined data types to represent the external data types of the various database systems.

- When you define a field with type **CURR** in the ABAP Dictionary, you must always link to a currency. You do this by specifying a field with the type **CUKY**. (When you create a list, you use the **CURRENCY** addition in the **WRITE** statement). The same applies to type **QUAN**, which must always link to a field with type **UNIT**.
- Type **FLTP** is useful for calculations involving very large or very small numbers. This usually only occurs in scientific applications or when making estimates.
- For business calculations, you should always use type **DEC** or **QUAN**. The arithmetic is the same as that to which you are used "on paper" - the system calculates precisely to the last decimal place.
- A typical use for type **NUMC** is for postal code fields - fields in which only digits should be allowed, but with which you do not want to perform calculations. (It is, however, possible to use conversions and calculate with alpha-numeric data.) For further details about arithmetic and conversions, refer to the **Statements** unit.
- Based on their underlying data type, some data objects are displayed according to **formatting options** (for example, country-specific date formats). Each user defines these formats in their user defaults.

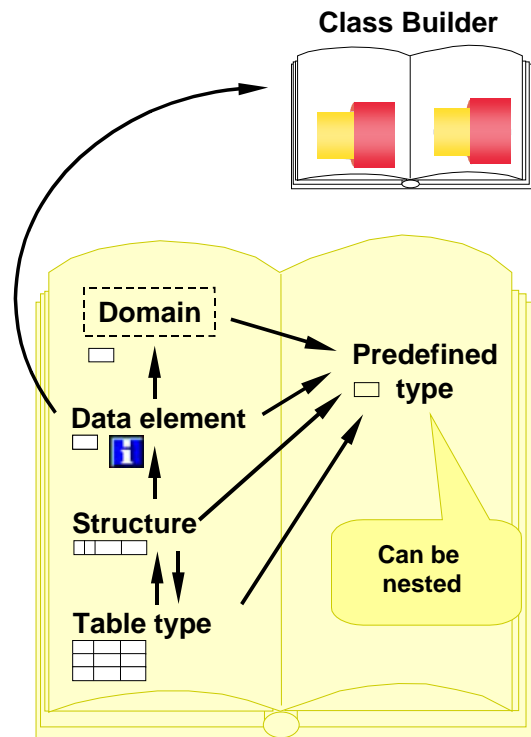
All of these data types apart from string and rawstring are elementary types. For technical reasons, these are classified as nested types. This has consequences for certain uses, such as the **INTO** clause of a **SELECT** statement.

# Data Elements and Structures in the ABAP Dictionary

SAP



© SAP AG 1999



## ■ Data element

Data elements have a business meaning (field label, help text, and so on). Up to and including Release 4.0, it was only possible to specify the technical attributes of a data element by specifying a domain. Each domain had to have a predefined Dictionary type assigned to it. This is still possible. However, it is now possible to enter a predefined Dictionary type directly. If you want to ensure that the technical attributes of a group of data elements can only be changed centrally, you should continue to use domains.

As part of **ABAP Objects**, you can now designate a **data element a reference type** and declare global types for references to global classes or interfaces. Note that, in this case, the type of the data element is no longer elementary, but nested. The same applies when you use the predefined types **string** and **rawstring**.

## ■ Structure

Each component of a structure must have a name so that it can be addressed directly. For the type of a component you may specify a predefined Dictionary type, a data element, a structured type, or a table type. This allows you to construct nested data types. Note the consequences we have already mentioned with particular kinds of access. For example, if a structure contains a component with the type reference or **string**, you cannot use **INTO CORRESPONDING FIELDS OF** in a **SELECT** statement. Instead, you must list the components in the **INTO** clause.

CARRID CONNID DISTANCE			Line type
			Key definition
AA	0017	2,572	
LH	0400	6,162	
<del>LH</del>	<del>0400</del>	<del>7,273</del>	Key type
QF	0005	10,000	Access type
SQ	0866	1,625	
UA	0007	2,572	

Index access

5

UA 0007

Key access

© SAP AG 1999

The data type of an internal table is fully specified by its:

## ■ Line type

The line type defines the attributes of the individual fields. You can specify **any** ABAP data type.

## ■ Key definition

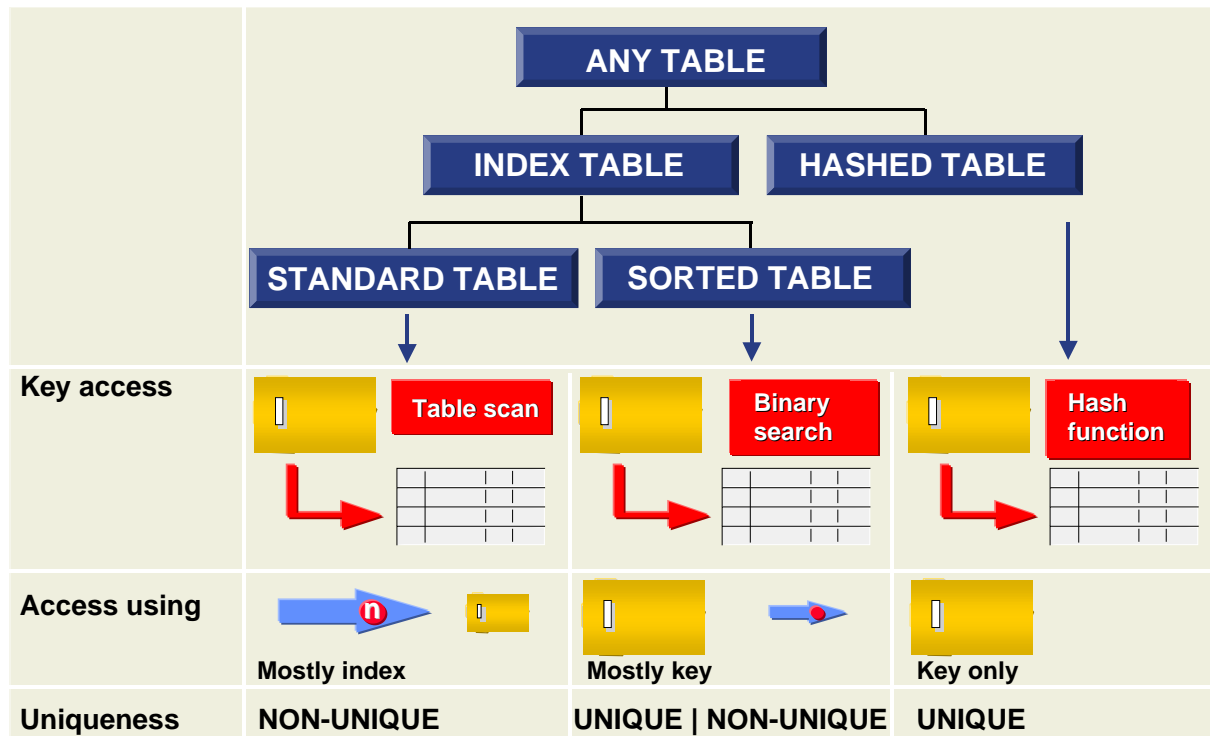
The key fields and their sequence determine the criteria by which the system identifies table lines.

## ■ Key type

You can define the key as either **unique** or **non-unique**. The uniqueness of the key must be compatible with the access type you have chosen for the table. If the key is unique, there can be no duplicate entries in the table.

## ■ Access type

Unlike database tables, the system assigns line numbers to certain kinds of internal tables. This means that you can use the **index** to access lines as well as the **key**. We sometimes use the term "**table type**" to refer to this.



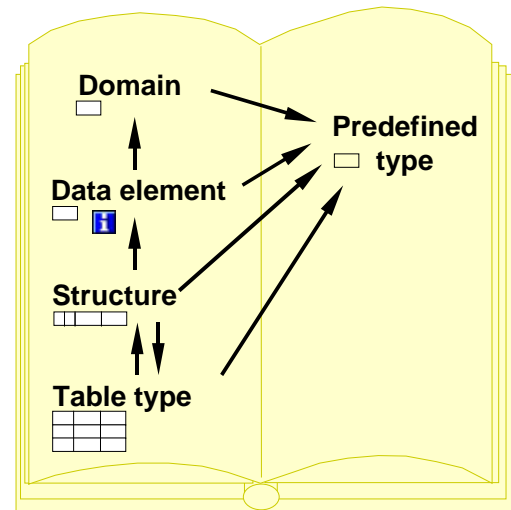
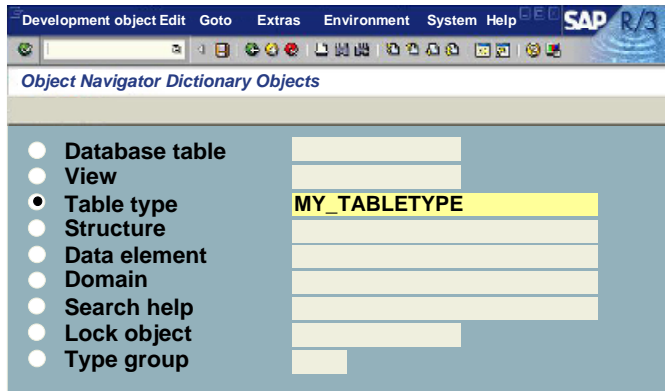
© SAP AG 1999

We can also divide up internal table types into three kinds by their access type:

- **Standard tables.** In a standard table, you can access data using either the table index or the key. Since the key of a standard table always has to be non-unique for compatibility reasons, the system searches the whole table each time you access it using the key. Consequently, you should always use the index to access a standard table whenever possible.
- **Sorted tables.** In a sorted table, the system **automatically** stores the entries and inserts new entries sorted by the table key. The system uses a binary search on the table when you access it using the key. You can specify the key of a sorted table as unique. You will often use the key to access a sorted table, but it is also possible to use the index. Standard tables and sorted tables are generically known as **index tables**.
- **Hashed tables.** You can only access a hashed table using the key. There are certain conditions under which you can considerably reduce the access times to large tables by using a hashed table. The key of a hashed table must always be unique.

You do not have to specify the **access type** fully. You can either omit it altogether, or specify it partially (index table). The table type is then **generic**, and, by omitting certain attributes, we can use it to specify the types of interface parameters.

To find out the access type of an internal table at runtime, use the statement **DESCRIBE TABLE <itab> KIND <charfield>**.



© SAP AG 1999

The **line type** specifies the semantic and technical attributes of the individual fields in a line. As already mentioned, you can specify either another table type, a structured type, or an elementary type. If you only use an elementary type, the internal table will have a single column with no component name (**unstructured table**).

## Key definition

- The **default key** consists of all character (alphanumeric) components of the line type that are not themselves table types. In this case, it would be empty (only possible with standard tables).
- It is particularly useful to name the **line type**, that is, the whole line, as the key if the table type is unstructured.
- You can also name **key components** and their **sequence** explicitly.
- A final possibility is **not to specify the key**, leaving it generic instead.

## Key type

As well as defining the key as **unique** and **non-unique**, you can specify a generic key type by omitting the specification.

For further information about choosing the right table type attributes, refer to the **Internal Table Operations** unit.



## Predefined ABAP Types



	Data type	Meaning	Initial value	Possible values	
				Default length (chars)	Maximum length (chars)
Numeric	<b>I</b>	Whole number	0	[-2147483648 ; 2147483647]	
	<b>F</b>	Floating point number	0.0...E+000	[2.2...E-308 ; 1.7...E+308]	
	<b>P</b>	Packed number	0	15	31
Alpha-numeric	<b>N</b>	Numeric string	00 ... 0	1	65535
	<b>C</b>	Character string	_ _ ... _	1	65535
	<b>STRING</b>	Character string	empty	0	any
	<b>D</b>	Date YYYYMMDD	00000000	8	8
	<b>T</b>	Time HHMMSS	000000	6	6
	<b>X</b>	Hexadecimal code	X'00'	1	65535
	<b>XSTRING</b>	Hexadecimal code	empty	0	any

© SAP AG 1999

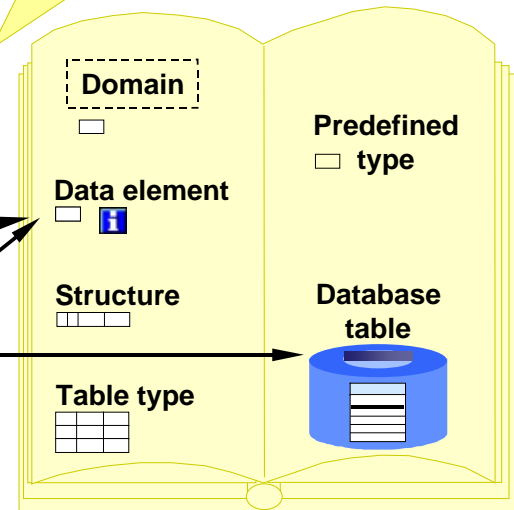
- Type **F** is useful for calculations involving very large or very small numbers. This usually only occurs in scientific applications or when making estimates.
- For business calculations, you should always use type **P**. The arithmetic is the same as that to which you are used "on paper" - the system calculates precisely to the last decimal place.
- A typical use for type **N** is for postal code fields - fields in which only digits should be allowed, but with which you do not want to perform calculations. (It is, however, possible to use conversions and calculate with alpha-numeric data.) For further details about arithmetic and conversions, refer to the **Statements** unit.
- Unlike type **C**, **N**, or **X** fields, the length of a string or hexadecimal string is not statically defined. Instead, it is variable, and, at runtime, will always take the length of its current contents. The memory is managed dynamically by the system. Strings and hexadecimal strings can have any length.
- You cannot currently use **STRING** or **XSTRING** to specify the type of a screen field.

```
TYPES <type> [TYPE <type>|LIKE <dataobject>].
```

```
TYPES:
  t_char,
  t_name(8) TYPE c.
DATA:
  d_value(5) TYPE p DECIMALS 2.

TYPES:
  t_namenew TYPE t_name,
  t_valnew  LIKE d_value,
  t_mydate  TYPE dats,
  t_myfield TYPE zmy_data_element,
  t_mycarr  TYPE spfli-carriid.
```

Conversion



Field type

© SAP AG 1999

You can only define a new data type based on an existing type. Use the **TYPE** addition to refer to **data types**, that is, predefined ABAP types, user-defined local types, predefined ABAP Dictionary types, user-defined ABAP Dictionary types, or fields or entire lines from **database** tables. If you refer to an ABAP Dictionary type, changes to the global type are automatically passed on to your type. This ensures that your type is always compatible with the corresponding ABAP Dictionary object. Types that refer to the ABAP Dictionary also have the advantages of formatting options, field help, and possible entries help. The underlying ABAP Dictionary data type is converted into the corresponding ABAP data type when the program is generated. For further information, refer to the ABAP syntax documentation for the **TABLES** statement.

If a global and a local data type both have the same name, the system uses the **local type**.

Use the **LIKE** addition to refer to the type of a **data object** that you have already declared. This also applies to the next slides.

## Elementary types

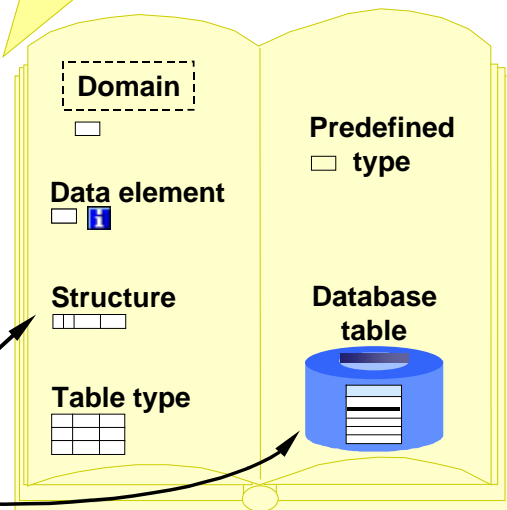
The length specification after the type name for ABAP data types **C**, **N**, and **X** specifies the number of characters in the type. For type **P** fields, you can also set the number of decimal places. If you omit these specifications, the system uses the default values (refer to the chart under Predefined ABAP Types).

```
TYPES:
  BEGIN OF <structype>,
  * ...components...
  END OF <structype>.
```

```
TYPES:
  BEGIN OF t_linetype,
    name TYPE t_name,
    val LIKE d_value,
    mydate TYPE dats,
    myfield TYPE zmy_data_element,
    mytab TYPE zmy_tabletype,
  END OF t_linetype.
```

```
TYPES:
  t_linenew TYPE t_linetype,
  t_mystruc TYPE zmy_structure,
  t_mysflight TYPE sflight.
```

Conversion



Line type

© SAP AG 1999

## Structured types

Use the statements

**TYPES BEGIN OF** <structype>.

and

**TYPES END OF** <structype>.

to enclose the list of **components** in your structure. Any type definitions may appear between the two statements. You can also construct nested data types.

To refer to the **line type** of a table type or an internal table, use the additions **TYPE LINE OF** <itabtype> or **LIKE LINE OF** <itab> respectively.

```
TYPES <itabtype>
  {TYPE|LIKE} {[STANDARD]|SORTED|HASHED|INDEX|ANY}
  TABLE OF {<structype>|<strucdataobject>}
  [WITH {[NON-UNIQUE]|UNIQUE}
    {KEY {<f1> ... <fn>|TABLE LINE} |DEFAULT KEY}]
  [INITIAL SIZE <n>].
```

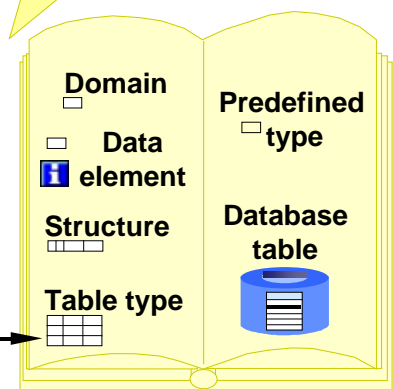
```
TYPES t_simpltab
  TYPE STANDARD TABLE OF t_linetype
  WITH DEFAULT KEY.

TYPES t_sophtab
  TYPE SORTED TABLE OF t_linenew
  WITH UNIQUE KEY myfield name. "order!"

TYPES t_mytabnew TYPE t_sophtab.

TYPES t_othertab TYPE zmy_tabletype.
```

## Conversion



© SAP AG 1999

## Table types

Similarly to when you create table types in the ABAP Dictionary, you must specify various attributes here:

- The **line type** after ... **TABLE OF** (as always, if you refer to a data type, use **TYPE**, if you refer to a data object that has already been declared, used **LIKE**);
- The **access type** before **TABLE OF ...** (If you omit this, the system uses the default access type, which is standard. You can also specify a generic table type using **INDEX** or **ANY**.);
- The **key definition** after the key type (to specify the default key, use the **DEFAULT KEY** addition); You can also specify fields from the (flat) line type and specify the sequence explicitly. If the table is unstructured, you can use the **TABLE LINE** addition);
- The key type after ...**WITH (UNIQUE or NON-UNIQUE)**.

If you omit the key specification entirely, the system uses the **non-unique default key**.

- For information about the optional **INITIAL SIZE <n>** addition, refer to the page **Declaring Internal Tables**.

We have not yet introduced **reference types**. These will be discussed in conjunction with field symbols and references.

```
DATA <dataobject> [TYPE <type>|LIKE <dataobject>]
                  [VALUE <value>].
```

```
DATA:
  BEGIN OF <strucdataobject>,
*   ... components ...
  END OF <strucdataobject>.
```

```
...
DATA: d_char, d_name(8) TYPE c VALUE 'SAP AG'.
DATA:
  d_valnew LIKE d_value,
  d_line1 TYPE t_linetype,

  BEGIN OF d_line2,
    flag TYPE t_char,
    name TYPE t_name,
    mycarr TYPE spfli-carrid,
  END OF d_line2.
```

Starting value

© SAP AG 1999

Similarly to when you define data types, you must specify a type when you declare data objects. You can do this in one of two ways:

- Either by referring to a data type (using the **TYPE** addition), or a data object in the program that has already been declared (using the **LIKE** addition). You can use exactly the same syntax variants in the DATA statement as when you declare local data types using the TYPES statement.
- You can also construct a field, structure, or internal table directly in a DATA statement, **without** having to define your own data type first.

In most cases, you will want to change the value of data objects at runtime. They are therefore also known as **variables**. You can assign a starting value to a data object using the **VALUE** addition. If you do not, the system assigns it the initial value appropriate to its type (see the table under Predefined ABAP Types).

There are two further statements that you can use to declare **special** data objects:

- **STATICS** declares local variables in a subroutine whose values are retained in subsequent subroutine calls instead of being initialized again. For further information, refer to the **Subroutines** unit.
- **CLASS-DATA**, an **ABAP Objects** statement, allows you to declare static class attributes.

```
DATA <itabdataobject>
  {TYPE|LIKE} { [STANDARD] | SORTED | HASHED | INDEX | ANY }
  TABLE OF {<structype> | <strucdataobject>}
  [WITH { [NON-UNIQUE] | UNIQUE }
    {KEY {<f1> ... <fn> | TABLE LINE} | DEFAULT KEY}]
  [INITIAL SIZE <n>]
  [WITH HEADER LINE].
```

```
DATA d_simpltab TYPE TABLE OF t_linetype.

DATA d_sophtab
  TYPE SORTED TABLE OF t_linenew
  WITH UNIQUE KEY myfield name. "order!"

DATA d_mytabnew LIKE d_sophtab.

DATA d_othertab TYPE zmy_tabletype.
```

Dynamic  
table  
extension !!!

© SAP AG 1999

With the exception of the **WITH HEADER LINE** addition, the syntax for declaring internal table objects is exactly the same as that used to define table types or other kinds of data objects. The addition allows you to create an **internal table with a header line**. However, this is an obsolete programming technique, and you should consequently no longer use it. For more information about header lines, along with general information about internal tables, refer to the **Internal Table Operations** unit.

## Dynamic table extension

Unlike arrays in other programming languages, the number of lines in an internal table is **increased automatically** by the ABAP runtime environment as required. You therefore do not have to worry about managing the size of the table, but only about inserting, reading, or deleting lines. This makes chained lists redundant in ABAP.

## INITIAL SIZE addition

When you create an internal table, the system allocates 256 bytes to it. The system then allocates a block of 8 KB to the table when you first add data, followed by further 8KB blocks as required. If you are only expecting to place a few lines in your table, or are using nested internal tables, it may be worth restricting the first automatic extension using the addition **INITIAL SIZE <n>**. You may do this either in the data object definition or in the type definition. <n> is the maximum number of lines that you are expecting to put in the table. When the system first allocates memory, it allocates the product of <n> and the length of the line. In the second step, it allocates twice that amount, and then in subsequent steps, it allocates between 12 and 16 KB.

```
PARAMETERS <parameter> [TYPE <type>|LIKE <dataobject>][...].  
  
SELECT-OPTIONS <selection> FOR <dataobject> [...].  
  
DATA <set> {TYPE RANGE OF <type>|LIKE RANGE OF <dataobject>}.
```

```
PARAMETERS pa_carr TYPE spfli-carriid DEFAULT 'LH'.  
  
DATA d_conn TYPE spfli-connid.  
SELECT-OPTIONS so_conn FOR d_conn OBLIGATORY.  
DATA set_connection LIKE RANGE OF d_conn.  
  
MOVE so_conn TO set_connection.  
SELECT ... FROM spfli  
        INTO ...  
        WHERE carriid EQ pa_carr  
        AND   connid IN set_connection.
```

© SAP AG 1999

Selection screens are a special kind of screen whose layout you program directly in the processing logic using **ABAP statements**. In an executable (type 1) program, there is a standard selection screen (screen number 1000). The definition of the standard selection screen does not require the statements that normally mark the beginning and end of a selection screen definition, neither does it require an explicit call. The following statements allow you to easily create screens on which the user can enter data.

- **PARAMETERS** creates an input field on the selection screen with the type you specify **and** a variable in the program with the same name. You cannot use **f**, **string**, **xstring**, or references to specify the type.
- **SELECT-OPTIONS** creates a pair of "from - to" fields on the screen, in which it is possible to enter sets of complex selections **for** a specified variable. The values that the user enters are stored in an internal table that the system creates automatically. The internal table has four fields **sign**, **option**, **low**, and **high**.
- You can also create this kind of table using ...{**TYPE|LIKE**} **RANGE OF** ... . However, tables declared in this way are not linked to the selection screen.

For further information about these statements, refer to the courses **BC405 (Techniques of List Processing and ABAP Query)** and **BC410 (Programming User Dialogs)**.

```
CONSTANTS <constant> {TYPE <datatype>|LIKE <dataobject>}
                        VALUE {<literal>|IS INITIAL}.
```

```
DATA factor TYPE f.
```

```
CONSTANTS: c_number1 TYPE i VALUE 123456789,
            c_number2 TYPE i VALUE '1234567890',
            c_eurofak(4) TYPE p DECIMALS 5 VALUE '1.95583',
            c_pi LIKE factor VALUE '3.1415E01',
            c_factor TYPE f VALUE IS INITIAL,
            c_clause(20) TYPE c VALUE 'John''s bike is red.',
            c_our_carr TYPE scarr-carrid VALUE 'LH'.
```

*\* bad programming:*

```
SELECT SINGLE * FROM spfli INTO wa_spfli
              WHERE carrid = 'LH' AND connid = pa_conn.
```

*\* good programming:*

```
SELECT SINGLE * FROM spfli INTO wa_spfli
              WHERE carrid = c_our_carr AND connid = pa_conn.
```

© SAP AG 1999

Constants and literals are **fixed** data objects - you **cannot** change their values at runtime.

- You define **constants** using the ABAP keyword **CONSTANTS**. In it, you **must** use the **VALUE** addition to assign a value to your constant.

#### Recommendation:

Avoid using literals wherever possible. Use constants instead. Your programs will then be easier to maintain.

- Literals allow you to specify a value directly in an ABAP statement. There are two kinds of literals - numeric literals and text literals. Text literals must always be enclosed in single quotes. Integers (including a minus sign if appropriate) can be represented as numeric literals. They are mapped to the data types **i** and **p** (based on the interval that each data type can represent).

#### Example:

```
DATA: result1 TYPE i, result2 LIKE result1.
```

```
result1 = -1000000000 / 300 * 3. "result1: 999.999-
```

```
result2 = -10000000000 / 300 * 3. "result2: 10.000.000-
```

A numeric literal can contain up to 31 digits.

All other values (decimal and floating point numbers, strings, and so on) **must** be given as text literals.

The system converts the data type if necessary.

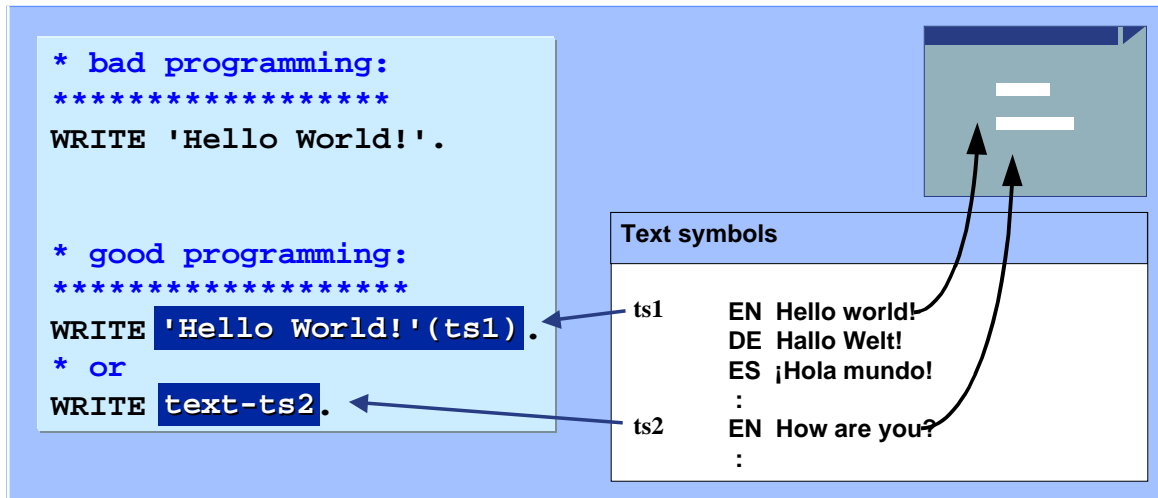
A text literal can contain up to 255 characters.

If you want a single quote to appear in a text literal, you must use two single quotes in order for it to be interpreted as part of the literal and not the closing single quote.



```
<output_statement> '<default_text>'(<tsk>).
```

```
<output_statement> text-<tsk>.
```



© SAP AG 1999

**Text symbols** are a special form of text literals.

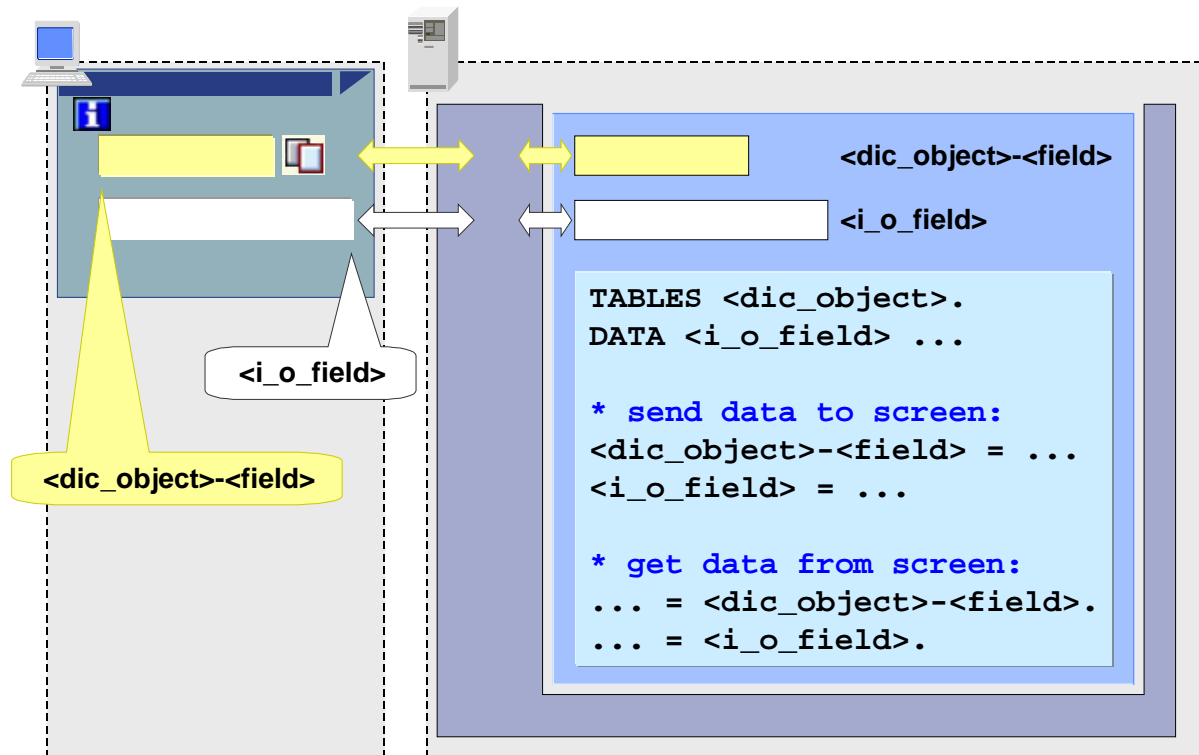
You can create a set of text symbols for any program. These can be used for output in various ways. The advantage of text symbols over normal text literals is that they can be translated. Furthermore, text elements are stored separately from the program source code, making your program easier to understand.

Text symbols are often used to create lists that are not language-specific. You can also use them to assign texts **dynamically** to screen objects. (Static text elements for screen objects are a special case, and can be translated).

You can display text symbols in two different ways using the **WRITE** statement:

- **WRITE text-<ts1>**. (where <ts1> can be any three-character ID).
- **WRITE '<default text>'(<ts2>)**. (where <ts2> can be any three-character ID).

In this case, <ts2> is displayed if there is a text for it in the current logon language. If there is not, the default text is displayed.



© SAP AG 1999

When you use screens, the system automatically transports field contents from the processing logic to the screen and back, but **only where screen fields and ABAP fields have the same names**.

## Restriction:

If you use screen fields with a reference to the ABAP Dictionary (*Get from Dictionary* function in the Screen Painter), you **must** use the **TABLES** statement to declare a data object with the same name as the ABAP Dictionary object in order for the field transport to work. Structures you declare like this are often referred to as **work areas**.

There are numerous advantages to using an ABAP Dictionary reference: Dictionary objects normally include foreign key checks, field help, possible entries, and the necessary error dialogs. Consequently you can catch inconsistent data as soon as the user enters it and before you even leave the screen.

If you program your own field checks, the field contents must already have been transported to the program. If you forget to reset the field when a check fails, an unwanted value may remain in the work area. You also face the same danger if you are not sure whether work areas are shared by more than one program.

To avoid these dangers, you should regard **TABLES** work areas as an interface between the screen and program, and only use them in this context. They provide data for the screen at the end of the PBO event, and receive it again when the values are transported from the screen.

```
NODES <node> [TYPE <type>].
```

```
REPORT ...
```

```
NODES:
```

```
    spfli, sflight.
```

```
START-OF-SELECTION.
```

```
    GET spfli.
```

```
        WRITE: / spfli-carriid, spfli-connid.
```

```
    GET spfli LATE.
```

```
        WRITE: / 'all flights of this connection:'(afc).
```

```
        SKIP. ULINE.
```

```
    GET sflight.
```

```
        WRITE: /10 sflight-fldate.
```

© SAP AG 1999

Logical databases are special ABAP programs that you can attach to an executable (type 1) program. They read data from the database and pass it to the executable program. Because the task of reading the data has been passed to the logical database, your own ABAP program becomes considerably simpler.

The logical database passes the data to your program using interface work areas that you declare using the **NODES** <node> statement. The statement creates a variable <node> that refers to the data type in the ABAP Dictionary with the same name.

The data is passed to your program record by record. Each time the logical database makes a record available to your program, the corresponding **GET** <node> or **GET** <node> **LATE** event is triggered. In your program, you can code the relevant event blocks.

You can determine the type of the data record returned by the logical database using the **TYPE** addition. However, you are restricted to types that are supported by the logical database. For further information about this statement, refer to the online documentation or course **BC405 (Techniques of List Processing and ABAP Query)**.

```
...  
  
SET PF-STATUS space.  
  
...  
  
WRITE:  
/ sy-tcode, "current transaction code  
  sy-mandt, "current client  
  sy-uname, "current user  
  sy-datum, "current date  
  sy-langu, "current language  
  sy-subrc. "return code of ABAP statements
```

© SAP AG 1999

The data object **SPACE** is a constant with type **C** and length 1. It contains a single space.

The system automatically creates a structure called **sy** for each program, based on the ABAP Dictionary structure **syst**. The individual components of the structure are known as **system fields**. They contain values that inform you about the current state of the system. The values are updated automatically by the ABAP runtime environment.

You can access individual system fields using the notation **sy-<system field>**.

System fields are variables, so you can change them in your programs. However, you should only do this in cases where it is explicitly recommended in the documentation (for example, navigating between list levels by manipulating **sy-lsind**). In all other cases, you should only read the contents of system fields, since by changing them you might overwrite information that is important for subsequent steps in the program.

The online documentation contains a list of all system fields with notes on their use. You can also display the structure **syst** in the ABAP Dictionary.

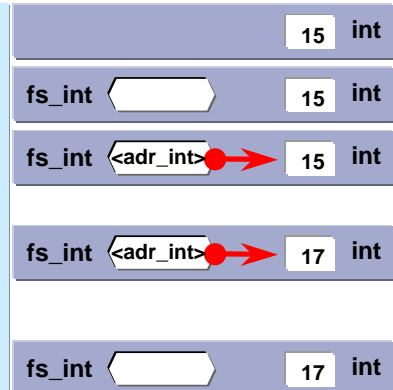
```
FIELD-SYMBOLS <<fs>> {{TYPE|LIKE} ... |TYPE ANY}.
ASSIGN ... <dataobject> TO <<fs>> [CASTING [TYPE <type>|...] ].
UNASSIGN <<fs>>.
... <<fs>> IS ASSIGNED ...
```

```
DATA int TYPE i VALUE 15.
FIELD-SYMBOLS <fs_int> TYPE i.

ASSIGN int TO <fs_int>.
WRITE: / int, <fs_int>.

<fs_int> = 17.
WRITE: / int, <fs_int>.

UNASSIGN <fs_int>.
IF <fs_int> IS ASSIGNED.
    WRITE: / int, <fs_int>.
ELSE.
    WRITE: / 'fieldsymbol not assigned'(fna).
ENDIF.
```



© SAP AG 1999

- You declare field symbols using the **FIELD-SYMBOLS** <<fs>> statement. The brackets (<>) are part of the syntax.  
Field symbols allow you symbolic access to an existing data object. All of the changes that you make to the field symbol are applied **to the data object assigned to it**. If the field symbol is not typed (**TYPE ANY**), it adopts the type of the data object. By specifying a type for the field symbol, you can ensure that only compatible objects are assigned to it.  
Field symbols are similar to **dereferenced pointers**.
- You use the **ASSIGN** statement to assign a data object to the field symbol <<fs>>. To lift a type restriction, use the **CASTING** addition. The data object is then interpreted **as though** it had the data type of the field symbol. You can also do this with untyped field symbols using the **CASTING TYPE** <type> addition.
- Use the expression <<fs>> **IS ASSIGNED** to find out whether the field symbol <<fs>> is assigned to a field.
- The statement **UNASSIGN** <<fs>>. sets the field symbol <<fs>> so that it points to nothing. The logical expression <<fs>> **IS ASSIGNED** is then false.  
An untyped field symbol that does not have a data object assigned to it behaves (for compatibility reasons) like a constant with type **C** and length 1.

```

TYPES <reftype>    TYPE REF TO data.
DATA <reference> TYPE REF TO data.

GET REFERENCE OF <dataobject> INTO <reference>.

ASSIGN <reference>->* TO <<fs>> [CASTING [TYPE <type>|...] ].

CREATE DATA <reference> TYPE|LIKE ... .
    
```

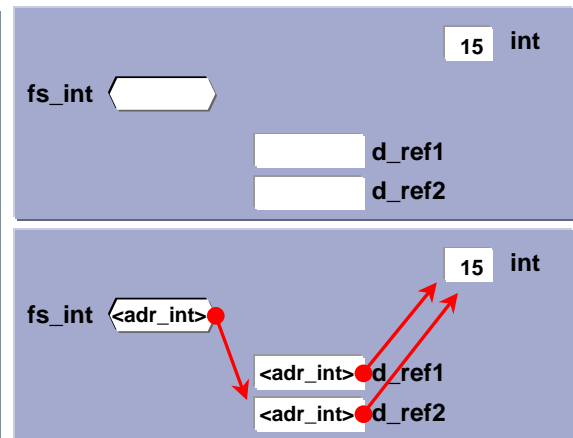
```

DATA int TYPE i VALUE 15.
FIELD-SYMBOLS <fs_int> LIKE int.
DATA d_ref1 TYPE REF TO data.
DATA d_ref2 LIKE d_ref1.

GET REFERENCE OF int INTO d_ref1.
d_ref2 = d_ref1.

ASSIGN d_ref2->* TO <fs_int>.

WRITE: / int, <fs_int>.
    
```



© SAP AG 1999

- The statement **TYPES <reftype> TYPE REF TO data.** \*) defines a reference type to a data object. **DATA...** defines the corresponding reference itself. Such a reference is a field in which an address can be stored.
- The **GET REFERENCE OF <data object> INTO <reference>** statement writes the address of the data object (already declared) into the reference variable. In other words, the reference points to the data object in memory.  
Thus ABAP uses reference semantics (changes apply to the address) as well as value semantics, as used in field symbols (where changes apply to the data objects). However, in ABAP, reference semantics is restricted to assignments.
- The dereferencing operator **->\*** in the **ASSIGN** statement allows you to assign the data object to which the reference points to a field symbol. You can then access the value of the data object.
- You can create a data object with a specified type at runtime using the **CREATE DATA <reference>** statement. This data object has no name, but the reference points to its address. (See also **GET REFERENCE OF...**)
- For further information about using references in ABAP Objects, refer to the **Introduction to ABAP Objects** unit.
- \*) **Note:** Data in this context is not a keyword, but rather a predefined name like space or p.

## Example of Dynamic Type Casting



```
PARAMETERS: pa_dbtabs TYPE dd02l-tabname.

DATA dummy TYPE i. " address of line mod 4 has to be zero !!!
DATA: line(65535).
FIELD-SYMBOLS: <fs_wa> TYPE ANY, <fs_comp> TYPE ANY.

SELECT * FROM (pa_dbtabs) INTO line.
  ASSIGN line TO <fs_wa> CASTING TYPE (pa_dbtabs).
  DO.
    ASSIGN COMPONENT sy-index OF STRUCTURE <fs_wa> TO <fs_comp>.
    IF sy-subrc NE 0.
      SKIP.
      EXIT.
    ENDIF.
    WRITE <fs_comp>.
  ENDDO.
ENDSELECT.
```

© SAP AG 1999

You can use **type casting** dynamically when you assign a data object to a field symbol. The graphic presents an example of this.

The name of the database table is not known until runtime (and consequently, neither is the line type).

Since you cannot specify a dynamic **INTO** clause in the **SELECT** statement, the system writes the data records into the long character field line.

The assignment to field symbol **<fs\_wa>** and the type casting then make it possible to access the field as though it were a flat structure. All type attributes are inherited from the database table. (You can also refer to the line type of an ABAP Dictionary object using the **TYPE** addition.)

If you knew the component names, you could display the fields directly using

**WRITE <fs\_wa>-... .**

However, you will not normally know the names of the components. In this case, you must use the **ASSIGN COMPONENT** variant, in which the components of the structure **<fs\_wa>** are assigned one-by-one to the field symbol **<fs\_comp>** and then displayed. When the loop runs out of components, the program reads the next data record.

### Problem

The address of **line** must satisfy the same address rules as a table structure (address must be divisible by four with no remainder). You can force this by declaring an integer field **dummy** directly before declaring **line**. (Integers are always stored at addresses that are divisible by four.)

## Declaring Data Objects Dynamically: Example



```
PARAMETERS: pa_dbtabs(30) DEFAULT 'SFLIGHT'.

DATA: d_ref TYPE REF TO data.
FIELD-SYMBOLS: <fs_wa> TYPE ANY, <fs_comp> TYPE ANY.

CREATE DATA d_ref TYPE (pa_dbtabs).
ASSIGN d_ref->* TO <fs_wa>.

SELECT * FROM (pa_dbtabs) INTO <fs_wa>.
  DO.
    ASSIGN COMPONENT sy-index OF STRUCTURE <fs_wa> TO <fs_comp>.
    IF sy-subrc NE 0.
      SKIP.
      EXIT.
    ENDIF.
    WRITE <fs_comp>.
  ENDDO.
ENDSELECT.
```

© SAP AG 1999

Unlike conventional data objects, you can specify the type of a data object **created at runtime** dynamically.

The above example is a slightly modified version of the example on the previous page.

This time, the idea is to create the data object for the **INTO** clause **dynamically at runtime**. In this case, the type is already known (you have entered the table name), and there are no more alignment problems. The statement **ASSIGN d\_ref->\* to <fs\_wa>** assigns the data object to the field symbol.

The data type of the table is inherited by the field symbol, so type casting is no longer necessary.

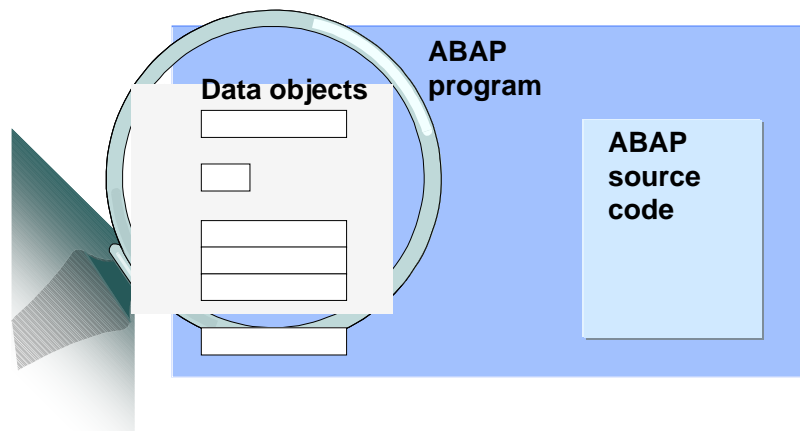
Instead of using a long character field, you can now write the data record into the data object with the same type to which the reference **d\_ref** is pointing, by using the field symbol **<fs\_wa>**.



```

DESCRIBE FIELD <field>
  LENGTH <len>                                "length
  TYPE <type> [COMPONENTS <num>]              "type [number of components]
  OUTPUT-LENGTH <len>                          "length (WRITE-statement)
  DECIMALS <num>                               "number of decimals
  ...

DESCRIBE TABLE <itab>
  LINES <num>                                  "number of filled lines
  ...
    
```



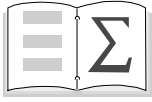
© SAP AG 1999

You will sometimes need to find out the attributes of a data object **at runtime**, especially when you use field symbols and references. The **DESCRIBE FIELD** statement returns various type attributes of variables.

**Caution:**

If you query the length of a field with type **string** or **xstring**, the system does not return the length of the string. Instead, it returns the length of the string reference, which is always eight bytes. To find out the length of the string, use the **OUTPUT-LENGTH** addition.

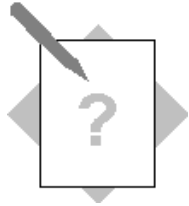
The statement **DESCRIBE TABLE <itab> LINES <n>** returns the number of lines in an internal table.



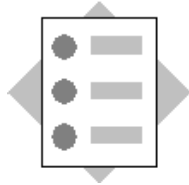
**You are now able to:**

- **Differentiate between the various kinds of data types and data objects**
- **Define data types and declare data objects**
- **Use field symbols and references**

## Data Types and Data Objects: Exercises

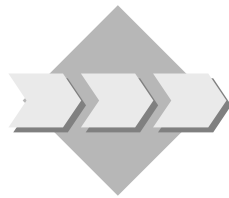


**Unit:** Data Types and Data Objects:  
**Topic:** Defining data types  
and data objects



At the conclusion of these exercises, you will be able to:

- Define data types
- Define variables and selection options
- Declare field symbols



You are a programmer for an airline consortium, and it is your job to write analysis programs for several airlines.

1. Complete the program **Z##\_BC402\_COUNTERLIST1**.  
Create a suitable internal table to allow you to buffer the names of airports and their codes. Create a second internal table for airlines, airports, and counter numbers.  
**## is your two-digit group number.**  
Model solution:  
SAPBC402\_TYPS\_COUNTERLIST1

- 1-1 Define the structure type **t\_airport**. It should have the following structure:

Component	Type
<b>id</b>	<b>sairport-id</b>
<b>name</b>	<b>sairport-name</b>

- 1-2 Define the structure type **t\_counter**. It should have the following structure:

Component	Type
<b>airport</b>	<b>scounter-airport</b>
<b>airp_name</b>	<b>sairport-name</b>
<b>carrid</b>	<b>scounter-carrid</b>
<b>countnum</b>	<b>scounter-countnum</b>

- 1-3 Declare the internal table **it\_carr\_counter** as a standard table with the line type **t\_counter** and a non-unique default key.

- 1-4     Declare the structure **wa\_counter** with the data type **t\_counter**.
- 1-5     Declare the internal table **it\_airport\_buffer** as a hashed table with the line type **t\_airport**. The unique key should contain the component **id**.
- 1-6     Declare the structure **wa\_airport** with data type **t\_airport**.
- 1-7     On the standard selection screen, declare the selection option **so\_carr** for the field **wa\_counter-carrid**.
- 1-8     Maintain a selection text.

2. Complete the program **Z##\_BC402-FLIGHTLIST1**:  
 Declare an internal table to hold flight timetable data for various airlines.  
 It should later be possible to assign one or more available aircraft types to each airline. This should be implemented as a second internal table, nested in the line type of the original table.  
 To type the inner internal table, you will first create two ABAP Dictionary objects.  
 You should also declare a selection table for the set of airlines for which the authorization check was successful.  
**## is your two-digit group number.**  
 Model solutions:  
 BC402\_TYPS\_PLANE  
 BC402\_TYPS\_PLANETAB  
 SAPBC402\_TYPS\_FLIGHTLIST1

- 2-1 In the **ABAP Dictionary**, create the global structure **Z##\_BC402\_PLANE**. It should have the following structure (remember that you must relate a currency field to the amount field):

Component	Type using data element
<b>PLANETYPE</b>	<b>S_PLANETYPE</b>
<b>SEATSMAX</b>	<b>S_SEATSMAX</b>
<b>AVG_PRICE</b>	<b>S_PRICE</b>
<b>CURRENCY</b>	<b>S_CURRCODE</b>

- 2-2 In the **ABAP Dictionary**, create the global structure **Z##\_BC402\_PLANETAB**. For the line type, use the global structure **Z##\_BC402\_PLANE**. Define the internal table as a standard table with a non-unique key consisting of the component **PLANETYPE**.

- 2-3 In the program, define the structure type **t\_flight**. It should have the following structure: (Note that the last component has a global table type.)

Component	Type
<b>carrid</b>	<b>sflight-carrid</b>
<b>connid</b>	<b>sflight-connid</b>
<b>fldate</b>	<b>sflight-fldate</b>
<b>cityfrom</b>	<b>spfli-cityfrom</b>
<b>cityto</b>	<b>spfli-cityto</b>
<b>seatsocc</b>	<b>sflight-seatsocc</b>
<b>paymentsum</b>	<b>sflight-paymentsum</b>
<b>currency</b>	<b>sflight-currency</b>
<b>it_planes</b>	<b>z##_bc402_planetab</b>

- 2-4 Define the internal table type **t\_flighttab** with line type **t\_flight**. It should be a sorted table with the unique key **carrid connid fldate**.
- 2-5 Declare a structure **wa\_flight** with the data type **t\_flight**.
- 2-6 Declare an internal table **it\_flights** with the data type **t\_flighttab**.

- 2-7 On the standard selection screen, declare the selection option **so\_carr** for the field **wa\_flight-carrid**.
- 2-8 Declare a selection table **allowed\_carriers** for fields with type **t\_flight-carrid**.



... TYPE RANGE OF ...

- 2-9 Declare a work area **wa\_allowed\_carr** for the selection table **allowed\_carriers**.
- 2-10 Maintain the selection texts.

## Data Types and Data Objects: Solutions



Unit: Data Types and Data Objects:  
Topic: Defining data types  
and data objects

### 1 Model solution SAPBC402\_TYPS\_COUNTERLIST1

```
*&-----*  
*& Report SAPBC402_TYPS_COUNTERLIST1 *  
*& *  
*&-----*  
*& solution of exercise 1 data types and data objects *  
*& *  
*&-----*
```

REPORT sapbc402\_typs\_counterlist1.

#### TYPES:

```
BEGIN OF t_airport,  
  id TYPE sairport-id,  
  name TYPE sairport-name,  
END OF t_airport,
```

```
BEGIN OF t_counter,  
  airport TYPE scounter-airport,  
  airp_name TYPE sairport-name,  
  carrid TYPE scounter-carrid,  
  countnum TYPE scounter-countnum,  
END OF t_counter.
```

#### DATA:

```
it_carr_counter TYPE STANDARD TABLE OF t_counter,  
  
wa_counter TYPE t_counter,  
  
it_airport_buffer TYPE HASHED TABLE OF t_airport  
  WITH UNIQUE KEY id,  
  
wa_airport TYPE t_airport.
```

SELECT-OPTIONS so\_carr FOR wa\_counter-carrid.

## 2 Model solution SAPBC402\_TYPS\_FLIGHTLIST1

```
*&-----*
*& Report SAPBC402_TYPS_FLIGHTLIST1 *
*& *
*&-----*
*& solution of exercise 2 data types and data objects *
*& *
*&-----*
```

REPORT sapbc402\_typs\_flightlist1.

### TYPES:

```
BEGIN OF t_flight,
  carrid TYPE spfli-carrid,
  connid TYPE spfli-connid,
  fldate TYPE sflight-fldate,
  cityfrom TYPE spfli-cityfrom,
  cityto TYPE spfli-cityto,
  seatsocc TYPE sflight-seatsocc,
  paymentsum TYPE sflight-paymentsum,
  currency TYPE sflight-currency,
  it_planes TYPE bc402_typs_planetab,
END OF t_flight,
```

```
t_flighttab TYPE SORTED TABLE OF t_flight
  WITH UNIQUE KEY carrid connid fldate.
```

### DATA:

```
wa_flight TYPE t_flight,
it_flights TYPE t_flighttab.
```

SELECT-OPTIONS so\_carr FOR wa\_flight-carrid.

\* for authority-check:  
\*\*\*\*\*

### DATA:

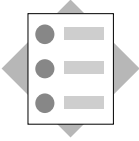
```
allowed_carriers TYPE RANGE OF t_flight-carrid,
wa_allowed_carr LIKE LINE OF allowed_carriers.
```



## Contents

- **Assigning values**
- **Processing strings and parts of fields**
- **Numeric operations**
- **Program flow control**

© SAP AG 1999



**At the conclusion of this unit, you will be able to:**

- **Write arithmetic expressions and perform calculations**
- **Process strings and parts of fields**
- **Assign values between compatible and non-compatible fields**
- **Write logical expressions and control the program flow**
- **Catch runtime errors**

Preface

Course Overview

ABAP Runtime Environment

Data Types and Data Objects



**Statements**

Internal Table Operations

Subroutines

Function Groups and Function Modules

Introduction to ABAP Objects

Calling Programs and Passing Data

© SAP AG 1999

```
char,  
name,  
length,  
line,  
itab.
```

The diagram illustrates the layout of a C program in memory. It shows four rows of memory locations:

- char**: A single byte memory location.
- name**: A string memory location, represented by a box with a dashed line underneath.
- length**: An integer memory location, represented by a box containing eight zeros (00000000) with a vertical dashed line in the center.
- line**: A line of text memory location, represented by a box with a dashed line underneath.

Below these memory locations, the same variables are shown as they appear in the **itab** (symbol table):

- flag**: A small box with a dashed line underneath.
- name**: A larger box with a dashed line underneath.
- mycarr**: A box with a dashed line underneath.

The **itab** section shows the variables **flag**, **name**, and **mycarr** as pointers, each represented by a box with a dashed line underneath, indicating their location in memory.

---

4-4

## Assigning Values

SAP

```
MOVE <source> TO <target>.
MOVE-CORRESPONDING <s_struct> TO <t_struct>.
WRITE <source> TO <target>.
```

name

SAP\_AG\_\_

DATA:

```
linenew LIKE line,
itabnew LIKE itab,
wa_scustom TYPE scustom,
time TYPE tims VALUE '083045'.
```

```
MOVE 'X' TO line-flag.
MOVE name TO line-name.
line-mycarr = 'YZ'.
```

```
linenew = line.
MOVE itab TO itabnew.
```

```
MOVE-CORRESPONDING wa_scustom TO line.
```

```
WRITE time TO charfield.
```

line

flag	name	mycarr
X	SAP_AG__	YZ_

linenew

flag	name	mycarr
X	SAP_AG__	YZ_

wa\_scustom

mandt	id	name	form ...
400	123	Scheer	...

line

flag	name	carr
X	Scheer	YZ_

time

083045

charfield

08:30:45

© SAP AG 1999

Use **MOVE** to copy the contents of one data object to another variable data object.




With complex objects, you can either address the components individually or use a "deep copy". If the source and target objects are compatible (see next page), the system copies the objects component by component or line by line.

If they are not compatible, the system **converts** the objects as long as there is an appropriate conversion rule.

If you are copying between two structures, and only want to copy values between **identically-named** fields, you can use the **MOVE-CORRESPONDING** statement.

The conversion mechanisms explained on the following pages apply not only to the **MOVE** and **MOVE-CORRESPONDING** statements, but also to calculations and value comparisons.

Unlike the **MOVE** statement, when you use **WRITE... TO...** to assign values, the target field is always regarded as a character field, irrespective of its actual type. **WRITE...TO** behaves in the same way as when you write output to a list. This allows you to use formatting options when you copy data objects (for example, country-specific date formatting).

- **When are two types compatible?**
  - Two elementary types are compatible when they have exactly the same **type** and **length** (and, for type P fields, the same number of **decimal places**). 
  - Two structured types are compatible when they have exactly the same **structure** and their **components** are compatible. 
  - Two table types are compatible when their **line types** are **compatible** and their **key sequences**, **uniqueness attributes** and **table types** are the same. 
- **Compatible types can be assigned to each other without conversion.**
- **Non-compatible types can be converted if a conversion rule exists.**

© SAP AG 1999

If two data types are not compatible but there is a conversion rule, the system converts the source object into the type of the target object when you assign values, perform calculations, or compare values. The following pages contain the basic forms of the conversion rules, and examples for the most frequent cases. For a full list of all conversion rules, refer to the ABAP syntax documentation for the **MOVE** statement.

If there is no conversion rule defined for a particular assignment, the way in which the system reacts depends on the context in which you programmed the assignment.

- If the types of the objects involved are defined **statically**, a **syntax error** occurs.

**Example:**

**DATA:** date TYPE d VALUE '19991231', time TYPE t.

**FIELD-SYMBOLS:** <fs\_date> TYPE d, <fs\_time> TYPE t.

**ASSIGN:** date TO <fs\_date>, time TO <fs\_time>.

<fs\_time> = <fs\_date>.

- If the types of the objects involved are defined **dynamically**, a **runtime error** occurs, because the system cannot tell in the syntax check whether they are convertible or not.

**Example** (remainder as above):

...

**FIELD-SYMBOLS:** <fs\_date> TYPE ANY, <fs\_time> TYPE ANY.

...

- Character-type fields are filled from the left if they contain characters, and from the right if they contain digits
- Numeric fields are filled from the right and filled with leading zeros if required
- The system must be able to interpret the contents of the source field according to the data type of the target field

Source field			Target field		
Type	Length	Value	Type	Length	Value
C	1	A	C	4	A _ _ _
C	4	A B C D	C	2	A B
C	7	- 4 7 1 1 0 _	P	3	4 7 1 1 0 -
P	3	1 2 3 4 5 -	C	7	_ 1 2 3 4 5 -

© SAP AG 1999

In general, there is a rule for converting every predefined ABAP data type into any other predefined type.

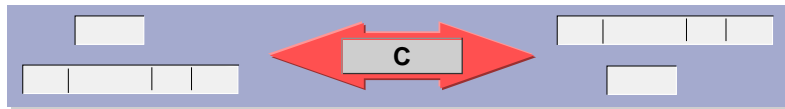
## Special cases:

- There are **no** rules for converting from type **D** to type **T** or vice versa, or for converting ABAP Objects data types (object reference to object reference, object reference to interface reference). Assignments or comparisons of this type cause syntax errors (where it is possible for the system to detect them).
- When you assign a type **C** field to a type **P** field, you may only use digits, spaces, a decimal point, and a plus or minus sign. The target field must also be large enough.
- When you convert a packed number to a type **C** field, the leading zeros are converted into spaces.

For full information about conversion rules for elementary types, refer to the online documentation in the ABAP Editor for the **MOVE** statement.

## Conversion Rules for Structured Types

SAP



```
DATA: BEGIN OF rec1,
      text1(3) TYPE c          VALUE 'AAA',
      text2(4) TYPE c          VALUE 'BBCC',
      pack      TYPE p DECIMALS 2 VALUE '2.26',
      text3(10) TYPE c         VALUE 'DD',
    END OF rec1,
    BEGIN OF rec2,
      text1(5) TYPE c          VALUE 'YYYYY',
      pack      TYPE p DECIMALS 2 VALUE '72.34',
      text3      TYPE c          VALUE 'Z',
    END OF rec2.
MOVE rec1 TO rec2.
```

rec1	text1	text2	pack	text3
	AAA	BBCC	0000000000002.26	DD _ _ _ _ _

rec2	text1	pack	text3
	YYYYY	00000000000072.34	Z

© SAP AG 1999

The ABAP runtime environment has rules for converting

- Structures to non-compatible structures
- Elementary fields to structures
- Structures to elementary fields

In each case, the system converts the source variables to character fields and fills the target structures byte by byte. The relevant conversion rules for elementary fields then apply.

Internal tables can only be converted into other internal tables. The system converts the lines types according to the relevant rule for structures.

The example above shows that copying between non-compatible types may result in the target fields containing values that cannot be interpreted properly. To avoid this problem, you should copy values field by field in these cases. This ensures that the system applies the correct conversion rule for elementary fields.

If you want to manipulate strings, it is better to use the statements expressly intended for this purpose.



		sy-subrc	sy-fdpos	u/l
SEARCH	ABAP	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
REPLACE	ABAP → BBAP	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
TRANSLATE	ABAP → a b a p	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
SHIFT	ABAP → BAP	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
CONDENSE	A P → AP	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
CONCATENATE	AB + AP → ABAP	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
OVERLAY	<div> <div>AB P</div> <div>AAA</div> </div> → ABAP	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
SPLIT	ABAP → AB AP	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

© SAP AG 1999

You can use the following statements to process strings in ABAP:

- **SEARCH** To search in a string
- **REPLACE** To replace the first occurrence of a string
- **TRANSLATE** To replace all specified characters
- **SHIFT** To shift a character at a time
- **CONDENSE** To remove spaces
- **CONCATENATE** To chain two or more strings together
- **OVERLAY** To overlay two strings
- **SPLIT** To split a string

In all string operations, the operands are treated like type **C** fields, regardless of their actual field type. They are not converted.

- All of the statements apart from **TRANSLATE** and **CONDENSE** set the system field **sy-subrc**. **SEARCH** also sets the system field **sy-fdpos** with the offset of the beginning of the string found.
- All of the statements apart from **SEARCH** distinguish between upper- and lowercase.
- To find out the occupied length of a string, use the standard function **STRLEN()**.



**REPLACE** <str1> WITH <str2> INTO <field>.

A	b	A	p			+
B	b	A	p			+

**TRANSLATE** <field> USING <str>.

A	b	A	p			+
B	b	B	p			+

**TRANSLATE** <field> TO {UPPER|LOWER} CASE.

A	b	A	p			+
A	B	A	P			+

**SHIFT** <field> [<var>] [RIGHT|CIRCULAR].

A	b	A	p			+
b	A	p				+

**CONDENSE** <field> [NO-GAPS].

A	b	A	p			+
A	b	A	p		+	

© SAP AG 1999

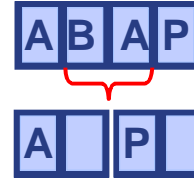
- **REPLACE** <str1> WITH <str2> INTO <field>.  
Replaces the first occurrence of <str1> in <field> with <str2>.
- **TRANSLATE** <field> USING <str>.  
Replaces all letters in <field> according to <str>. <str> contains the search and replacement characters in pairs. For the example: **TRANSLATE ... USING 'AB'**.
- **TRANSLATE** <field> TO UPPER|LOWER CASE  
Replaces all lowercase letters in <field> with uppercase (or vice versa).
- **SHIFT** <field> [<var>] [RIGHT] [CIRCULAR].  
<var> can be one of the following:  
**BY <n> PLACES** Shifts <field> by <n> characters  
**UP TO <str>** Shifts <field> up to the beginning of <str>  
 The additions have the following effect:  
**RIGHT** Shifts to the right  
**CIRCULAR** Shifts to the right - characters shifted off the right-hand edge of the field reappear on the left.
- **CONDENSE** <field> [NO-GAPS].  
Consecutive spaces are replaced by a single space or are deleted.  
**Note:**  
 You can delete leading or trailing spaces using  
**SHIFT <field> LEFT DELETING LEADING SPACE** or  
**SHIFT <field> RIGHT DELETING TRAILING SPACE.**

## Splitting and Joining Strings

SAP

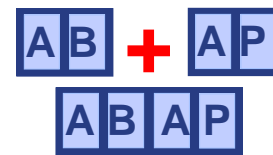
```
SPLIT <field> AT <sep> INTO {<f1> ... <fn>|TABLE <itab>}.
```

```
SPLIT 'ABAP' AT 'A' INTO f1 f2.
```



```
CONCATENATE <f1> ... <fn> INTO <field> [SEPARATED BY <sep>].
```

```
CONCATENATE 'AB' 'AP' INTO f.
```



```
OVERLAY <f1> WITH <f2> [ONLY <str>].
```

```
OVERLAY f WITH 'XBAX'.
```



© SAP AG 1999

### ■ SPLIT <field> AT <sep> INTO <f1> ... <fn>|TABLE <itab>.

Splits <field> at each occurrence of the separator string <sep> and places the parts into the fields <f1> ... <fn> or into consecutive lines of the internal table <itab>.

### ■ CONCATENATE <f1> ... <fn> INTO <f> [SEPARATED BY <separator>].

Combines the fields <f1>... <fn> in <f>. Trailing spaces are ignored in the component fields. You can use the **SEPARATED BY** <separator> addition to insert the string <separator> between the strings <f1>... <fn>.

### ■ OVERLAY <f1> WITH <f2> [ONLY <str>].

<f2> overlays <f1> at all positions where <f1> contains a space or one of the characters in <str>.

#### Note

See also Accessing Parts of Fields.

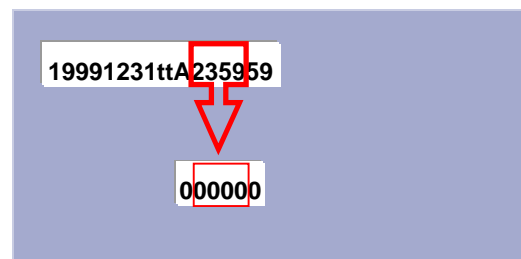
```
<statement> <field>+<off>(<len>) ...
```

Possible with any character-type field

```
REPORT ...

PARAMETERS:
  pa_str(40) LOWER CASE,
  pa_pos TYPE i,
  pa_len TYPE i.

WRITE pa_str+pa_pos(pa_len).
```



© SAP AG 1999

In any statement that operates on a character-type field, you can address part of the field or structure by specifying a starting position and a number of characters. If the field lengths are different, the system either truncates the target or fills it with initial values. The source and target fields must have the type **X**, **C**, **N**, **D**, **T**, or **STRING**. You can also use structures.

## Example

```
MOVE <field1>+<off1>(<len1>) TO <field2>+<off2>(<len2>).
```

This statement assigns <len1> characters of field <field1> starting at offset <off1> to <len2> characters of <field2> starting at offset <off2>.

## Caution

Under Unicode \*) only fields with type **C**, **X**, and **STRING** are suitable for partial access. In other cases, you should use field symbols with casting.

\*) Language- and culture-independent character set.



```
[COMPUTE] <result> = <arithmetic_expression>.

* expressions:
... <op> ( <expr1> <op> <expr2> ) <op> ...

* functions:
... <func>( <expr> ) ...

* possible operators:
... <expr1> + <expr2> ... "ADD
... <expr1> - <expr2> ... "SUBSTRACT
... <expr1> * <expr2> ... "MULTIPLY
... <expr1> / <expr2> ... "DIVIDE
... <expr1> ** <expr2> ... "power operator
... <expr1> DIV <expr2> ... "integer division
... <expr1> MOD <expr2> ... "remainder
```

© SAP AG 1999

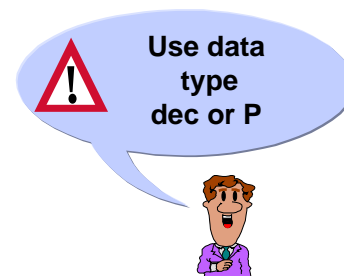
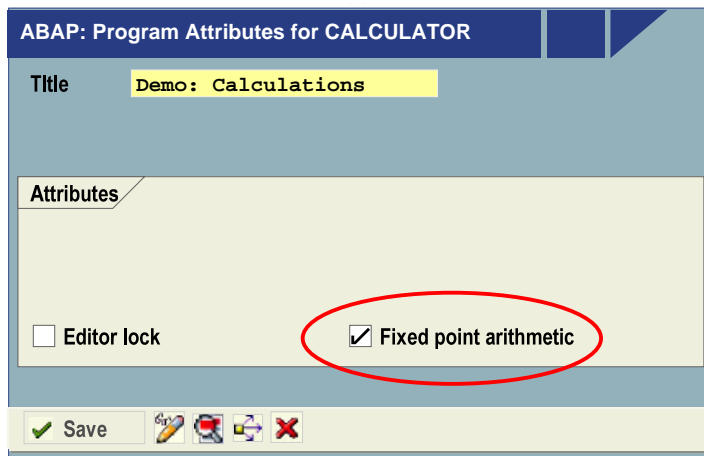
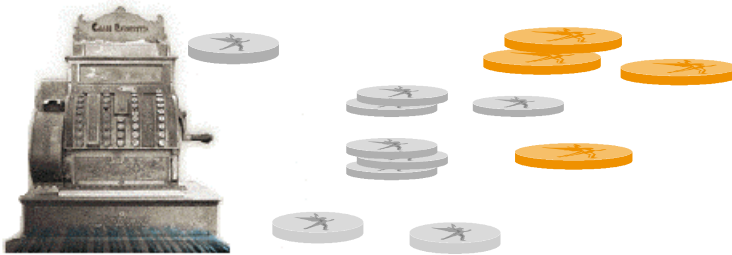
In ABAP, you can program arithmetic expressions nested to any depth. You must remember that parentheses and operators are keywords, and must therefore be preceded and followed by at least one space.

The ABAP runtime environment contains a range of functions for different data types. The opening parenthesis belongs to the functions name itself (and is therefore not separated from it by a space). The remaining elements of each expression must be separated by spaces.

The expressions are processed in normal algebraic sequence - parenthetical expressions, followed by functions, powers, multiplication and division, and finally, addition and subtraction.

A calculation may contain any data types that are convertible into each other and into the type of the result field. The system converts all of the fields into one of the three numeric data types (**I**, **P**, or **F**), depending on the data types of the operands. The ABAP runtime system contains an arithmetic for eachh of the three data types. The system then performs the calculation and converts it into the data type of the result field.

The **DIV** (integer division) and **MOD** (remainder of a division) always return whole numbers.



© SAP AG 1999

In integer and packed number arithmetic, the system always rounds to the corresponding decimal place.

So, for example:

```
DATA int TYPE i.      int = 4 / 10.  " result: 0
                    int = 5 / 10.  " result: 1
```

or

```
DATA: pack TYPE p DECIMALS 2. pack = 4 / 1000. " result: 0.00
      pack = 5 / 1000. " result: 0.01.
```

However, **intermediate results** using **packed numbers** are **always** accurate to 31 decimal places. The arithmetic used depends on how the system interprets the numeric literals:

```
DATA int TYPE i. int = 1000000000 / 300000000 * 3. " result: 9
                    int = 10000000000 / 3000000000 * 3. " result: 10
```

If you do **not** set the fixed point arithmetic option in the program attributes, the **DECIMALS** addition in the **DATA** statement **only affects the output, not the arithmetic**. In this case, all numbers are interpreted internally as integers, regardless of the position of the decimal point. You would then have to calculate the number of decimal places manually and ensure that the number was displayed correctly. Otherwise, the results would be meaningless.

```
DATA: pack TYPE p DECIMALS 2.
      pack = '5000.00' * '0.20'. " result: pack = 100000.00
```

Furthermore, the system would also round internally (integer arithmetic - see above).

**The fixed point arithmetic option is always selected by default. You should always accept this value and use packed numbers for business calculations.**

# Calculations: Floating Point Numbers and Runtime Errors

SAP



$$1.5 = 1 \cdot 2^0 + 1 \cdot 2^{-1}$$

$$= 1 + \frac{1}{2}$$



$$0.15 = 1 \cdot 2^{-3} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-10} + 1 \cdot 2^{-11} + \text{⊗}$$

$$= \frac{1}{8} + \frac{1}{64} + \frac{1}{128} + \frac{1}{1024} + \frac{1}{2048} + \text{⊗}$$

$$= 0.125 + \text{⊗}$$

$$= 0.140625 + \text{⊗}$$

$$= 0.1484375 + \text{⊗}$$

$$= 0.1494140625 + \text{⊗}$$

$$= 0.1499023437 \text{⊗}$$

Only 53 bits available



© SAP AG 1999

Calculations using data type **F** are **always**, for technical reasons, imprecise.

## Example

Suppose you want to calculate 7.72% of 73050 and display the result accurate to two decimal places. The answer should be 5310.74 ( $73050 \cdot 0.0727 = 5310.735$ ). However, the program returns the following:

**DATA: float TYPE f, pack TYPE p DECIMALS 2.**

**float = 73050 \* '0.0727'. " result: 5.3107349999999997E+03**

**pack = float. WRITE pack. " result: 5310.73**

**You should therefore only use floating point numbers for approximations. When you compare numbers, always used intervals, and always round at the end of your calculations.**

There are four general categories of runtime error that can occur in calculations:

- A field that should have been converted could not be interpreted as a number.
- A number range is too small for a conversion, value assignment, or to store intermediate results.
- You tried to divide by zero.
- You passed an invalid argument to a built-in function  
(For example: ... **log( -3 )** ... ).

For further information, refer to the ABAP syntax documentation for the **COMPUTE** statement.



```
DATA:
  diffdays    TYPE i,
  datestring  LIKE sy-datum,

  BEGIN OF daterec,
    year(4)    TYPE c,
    month(2)   TYPE c,
    day(2)     TYPE c,
  END OF daterec.

...
daterec = sy-datum.

daterec-day = '01'.           " first day of month
datestring = daterec.
datestring = datestring - 1. " last day of previous month

diffdays = sy-datum - datestring.    " difference days
```

© SAP AG 1999

- If you assign a date fields to a numeric field, the runtime system calculates the number of days that have elapsed since 01.01.0001.
- Conversely, when you assign a numeric value to a date field, the system interprets it as the number of days since 01.01.0001.
- Before any calculations are performed with dates, the value of the date field is converted into its numeric value (number of days since 01.01.0001)

The above example calculates the last day of the previous month.

When you calculate with time fields, the system uses a similar procedure, that is, it counts the number of seconds since 0:00:00.

```

* comparisons for all data types:
.. <dobj> EQ|= <literal>|<dobj>} .. "equal
.. <dobj> NE|<> <literal>|<dobj>} .. "not equal
.. <dobj> GT|> <literal>|<dobj>} .. "greater than
.. <dobj> GE|>= <literal>|<dobj>} .. "greater or equal
.. <dobj> LT|< <literal>|<dobj>} .. "less than
.. <dobj> LE|<= <literal>|<dobj>} .. "less or equal
.. <dobj> BETWEEN <lit>|<dobj> AND <lit>|<dobj>} ..
.. <dobj> IS INITIAL ..

* nesting logic expressions:
.. <nest_op> ( <expr1> <nest_op> <expr2> ) <nest_op> ..

* possible operators <nest_op>:
.. AND .. "all expressions must be true
.. OR .. "one of the expressions must be true

* negation:
.. NOT <expr> .. "true, if <expr> false
    
```

© SAP AG 1999

Comparisons between **non-numeric** data objects are interpreted differently according to their data type.

- If possible: Conversion into numbers (hexadecimal, for example, as dual number);
- Date and time: Interpreted as earlier/later, so 31.12.1999 < 01.01.2000;
- Other characters: Lexical interpretation according to character codes. The two operands are given the same length and filled with trailing spaces where necessary;
- References: System compares address and data type.  
It only makes sense to compare for equality.

When you **join** and **negate** comparisons, the usual rules for logical expressions apply:

**NOT** is stronger than **AND**, and **AND** is stronger than **OR**.

**Example**

**NOT f1 = f2 OR f3 = f4 AND f5 = f6** is the same as  
**( NOT ( f1 = f2 ) ) OR ( f3 = f4 AND f5 = f6 )**.

You should therefore enclose the components expressions of your comparisons in parentheses, even when it is not strictly necessary, to make them **easier to understand**, and for additional **safety**.

You can also considerably improve the runtime of your programs by optimizing the structure of your expressions.

```
* comparisons for character data types:
.. <str1> CO <str2> .. "contains only
.. <str1> CA <str2> .. "contains any
.. <str1> CS <str2> .. "contains string
.. <str1> CP <str2> .. "contains pattern
```

	u/l	space	sy-fdpos
CO	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
CA	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
CS	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
CP	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

© SAP AG 1999

For each of the above relational operators, there is a corresponding negative expression.

The logical expression ... <str1> <op> <str2> .. can contain an operator <op> as follows:

- **CO** <str1> only contains characters from <str2>;
- **CN** <str1> contains **not** only characters from <str2> (corresponds to **NOT** <str1> **CO** <str2>);
- **CA** <str1> contains at least one character from <str2>;
- **NA** <str1> does not contain any characters from <str2>;
- **CS** <str1> contains the string <str2>;
- **NS** <str1> does not contain the string <str2>;
- **CP** <str1> contains the pattern <str2>;
- **NP** <str1> does not contain the pattern <str2>;

The system field **sy-fdpos** contains the offset of the character that satisfies the condition, or the length of <str1>.

In the first four expressions, the system takes into account upper- and lowercase letters and the full length of the string (SPACE column).

To specify patterns, use '\*' for any string, and '+' for any character. The escape symbol is '#'.

```

CASE <dobj1>.
  WHEN <dobj2>.
    *   ... statements ...
    [WHEN <dobj3> OR <dobj4> .. OR <dobjn>.
    *   ... statements ...]
    [WHEN OTHERS.
    *   ... statements ...]
ENDCASE.

IF <logic_expr1>.
  *   ... statements ...
[ELSEIF <logic_expr2>.
  *   ... statements ...]
[ELSE.
  *   ... statements ...]
ENDIF.

CHECK <logic_expr>.
  *   ... statements ...
    
```

© SAP AG 1999

- In a **CASE - ENDCASE** structure, you test a data object for equality against various values. When a test succeeds, the corresponding statement block is executed. If all of the comparisons fail, the **OTHERS** block is executed, if you have programmed one.
- In an **IF - ENDIF** structure, you can use any logical expressions. If the condition is met, the corresponding statements are executed. If none of the conditions is true, the **ELSE** block is executed, if you have programmed one.

In both cases, the system only executes **one** statement block, namely that belonging to the **first valid case**.

If each condition tests the same data object for equality with another object, you should use a **CASE- ENDCASE** structure. It is **simpler**, and **requires less runtime**.

**Outside a loop**, you can make the execution of **all remaining statements in the current processing block** conditional using **CHECK**. If the check fails, processing resumes with the first statement in the next processing block.

```
DO [<n> TIMES] [...].
* ... statements ...
ENDDO.
```

```
WHILE <logic_expr>.
* ... statements ...
ENDWHILE.
```

```
LOOP AT ...
* ... statements ...
ENDLOOP.
```

```
SELECT ...
* ... statements ...
ENDSELECT.
```

Loop counter:  
sy-index

```
DO.
* ... statements ...
  CHECK <abort_condition>.
  EXIT.
ENDDO.
```

© SAP AG 1999

There are four loop structures. The number of the current loop pass is always available from the system field **sy-index**. If you use nested loops, the value of **sy-index** refers to the current one. You can take control over loop processing using the **CHECK** <logical expression> and **EXIT** statements. For further information, refer to Leaving Processing Blocks. The graphic shows how you can control the further processing of a loop.

## ■ Unconditional/Index-based loops

The statements between **DO** and **ENDDO** are executed until you end the loop with a termination statement. You can specify a maximum number of loop passes. If you do not, you have an **endless loop**.

## ■ Conditional loops

The statements between **WHILE** and **ENDWHILE** are repeated as long as the condition <logical expression> is met.

## ■ Multiple-line access to an internal table

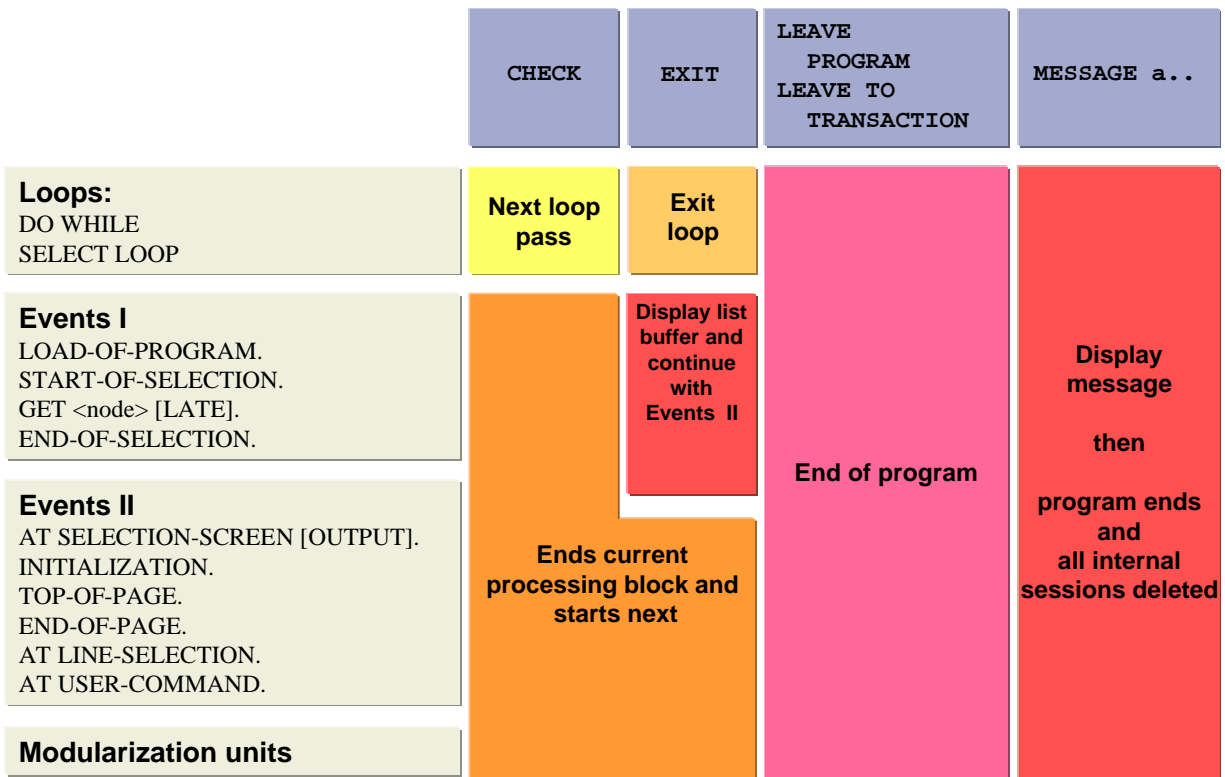
For further information, refer to the **Internal Table Operations** unit.

## ■ Multiple-record access to a database table or view

Refer to the courses **BC400** (ABAP Workbench: Concepts and Tools) and **BC405** (Techniques of List Processing and ABAP Query), and the syntax documentation for the **SELECT** statement.

## Overview: Leaving Processing Blocks

SAP



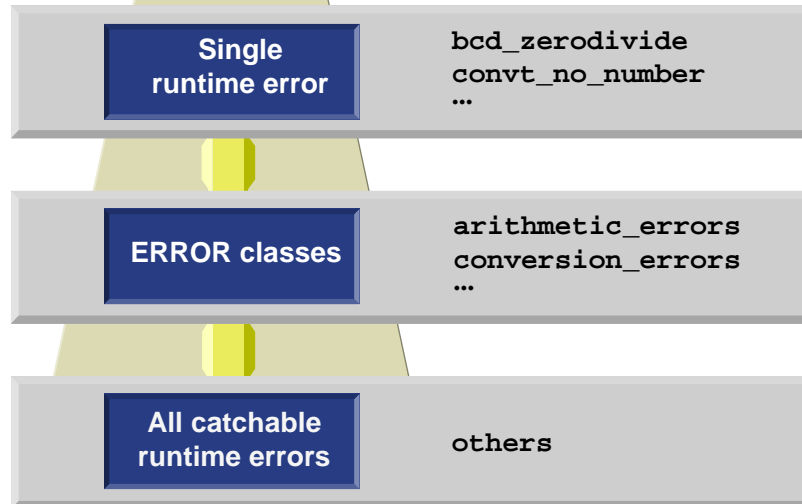
© SAP AG 1999

### Reminder

A **processing block** is an ABAP event block or modularization unit.

- You can use the ABAP statement **CHECK** <logical expression> **outside** a loop to terminate the processing block if the logical condition is **not** met. Processing continues with the first statement in the next processing block. **Within** a loop, processing resumes at the beginning of the next loop pass.
- The **EXIT** statement can behave in three different ways: **Within** a loop, it terminates the loop processing completely. Outside a loop but in one of the events listed under Events I, it makes the system display the current contents of the list buffer. Events from the other groups listed above can still be triggered. In the case of **LOAD-OF-PROGRAM**, **START-OF-SELECTION** is triggered.
- In all other cases, **EXIT** has the same effect as **CHECK**.
- The statements **LEAVE PROGRAM** and **LEAVE TO TRANSACTION** <tcode> terminate the current program.
- When you send a termination (type A) message, the current program ends, and the entire **program stack** is destroyed. For further information, refer to the unit **Program Calls and Passing Data**.

```
CATCH SYSTEM-EXCEPTIONS <excpt1> = <rc1> ... <excptn> = <rcn>.  
...  
ENDCATCH.
```



© SAP AG 1999

Within a processing block, you can use the structure **CATCH SYSTEM-EXCEPTIONS... ENDCATCH** to catch runtime errors. If the specified system exception occurs, the system leaves the statements in the block and continues processing after the **ENDCATCH** statement. This construction only catches runtime errors at the **current call level**. If you call a subroutine in which a runtime error is triggered, you must catch this error **in the subroutine itself**.

Each runtime error is assigned to an **ERROR class**. For a full list, refer to the syntax documentation for the **CATCH** statement.

You can specify one of the following as the system exception <excpt> that you want to catch:

- A single error (for example, **convt\_no\_number**);
- ERROR classes (for example, **arithmetic\_errors**);
- All catchable runtime errors.

The return code values <rc1>... <rcn> must be numeric literals.

The return code assigned to the runtime error that occurred is placed in the system field **sy-subrc**. If more than one value was assigned to it, the system uses the first. This is particularly important if you specify two different ERROR classes that contain the same runtime error.

## Example: Catching Runtime Errors



```
DATA pack(4) TYPE p DECIMALS 2 VALUE '3.14'.
...
CATCH SYSTEM-EXCEPTIONS convt_no_number = 1
                        convt_overflow  = 2.

...
CATCH SYSTEM-EXCEPTIONS convt_no_number = 3
                        bcd_zerodivide  = 4.

...
CATCH SYSTEM-EXCEPTIONS remote_call_errors = 5.
...
    pack = pack / 0.
ENDCATCH.
...
    pack = 'ABC'.
ENDCATCH.
...
    pack = '123456789.987654321'.
ENDCATCH.
...
```

© SAP AG 1999

**CATCH SYSTEM-EXCEPTIONS ... ENDCATCH** constructions can be nested to any depth. If a runtime error occurs, the system searches for an assignment in the current statement block. If it does not find one, it searches in the next-highest block, and so on. Processing resumes after the **ENDCATCH** statement of the block in which the assignment was found.

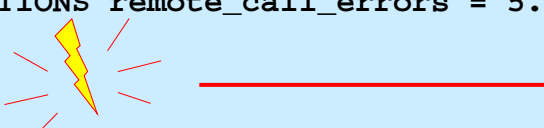
The above example nests three **CATCH SYSTEM-EXCEPTIONS ... ENDCATCH** structures. Before each **ENDCATCH** statement is a statement that causes a runtime error.

At which statement does the system set the field **sy-subrc**? With which value? At which statement does processing resume?



```
DATA pack(4) TYPE p DECIMALS 2 VALUE '3.14'.
...
CATCH SYSTEM-EXCEPTIONS convt_no_number = 1
                        convt_overflow = 2.

...
CATCH SYSTEM-EXCEPTIONS convt_no_number = 3
                        bcd_zerodivide = 4.
...
CATCH SYSTEM-EXCEPTIONS remote_call_errors = 5.
...
    pack = pack / 0.
ENDCATCH.
...
    pack = 'ABC'.
ENDCATCH.
...
    pack = '123456789.987654321'.
ENDCATCH.
...
```



© SAP AG 1999

The division by zero in the innermost block triggers the runtime error **bcd\_zerodivide**. However, there is no assignment for the error in this block.

Consequently, the system looks in the next-highest block for the error, where it is assigned. **sy-subrc** is set to 4.

```
DATA pack(4) TYPE p DECIMALS 2 VALUE '3.14'.
...
CATCH SYSTEM-EXCEPTIONS convt_no_number = 1
                        convt_overflow = 2.

...
CATCH SYSTEM-EXCEPTIONS convt_no_number = 3
                        bcd_zerodivide = 4.
...
CATCH SYSTEM-EXCEPTIONS remote_call_errors = 5.
...
    pack = pack / 0.
ENDCATCH.
...
    pack = 'ABC'.
ENDCATCH.
...
    pack = '123456789.987654321'.
ENDCATCH.
...
```

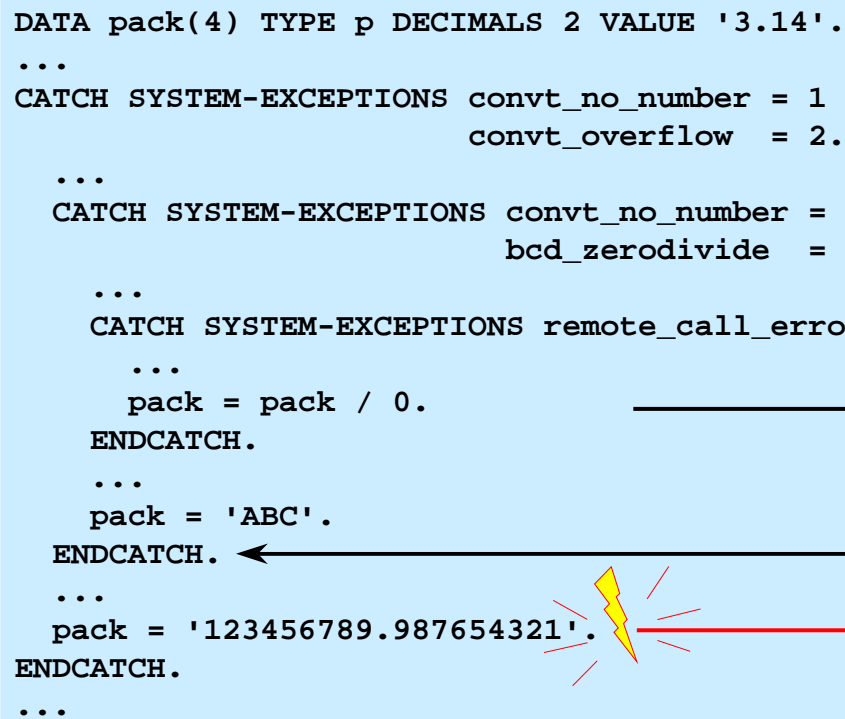
© SAP AG 1999

The system resumes processing at the first statement after the middle **ENDCATCH** statement. The assignment of a string (which cannot be interpreted as a packed number) to a packed number field is therefore not executed, even though we would have caught the ensuing runtime error **convt\_no\_number** in our program.

## Example: Solution - Part 3

SAP

```
DATA pack(4) TYPE p DECIMALS 2 VALUE '3.14'.  
...  
CATCH SYSTEM-EXCEPTIONS convt_no_number = 1  
                        convt_overflow = 2.  
...  
CATCH SYSTEM-EXCEPTIONS convt_no_number = 3  
                        bcd_zerodivide = 4.  
...  
CATCH SYSTEM-EXCEPTIONS remote_call_errors = 5.  
...  
    pack = pack / 0.  
ENDCATCH.  
...  
    pack = 'ABC'.  
ENDCATCH.  
...  
    pack = '123456789.987654321'.  
ENDCATCH.  
...
```



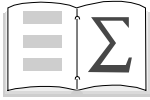
© SAP AG 1999

Finally, we trigger the runtime error **convt\_overflow** by assigning a number to the packed field that is too big for it. A return code is assigned to this error in the outer block. **sy-subrc** is set to 2.

```
DATA pack(4) TYPE p DECIMALS 2 VALUE '3.14'.
...
CATCH SYSTEM-EXCEPTIONS convt_no_number = 1
                        convt_overflow = 2.
...
CATCH SYSTEM-EXCEPTIONS convt_no_number = 3
                        bcd_zerodivide = 4.
...
CATCH SYSTEM-EXCEPTIONS remote_call_errors = 5.
...
    pack = pack / 0.
ENDCATCH.
...
    pack = 'ABC'.
ENDCATCH.
...
    pack = '123456789.987654321'.
ENDCATCH.
...
```

© SAP AG 1999

The system resumes processing after the corresponding **ENDCATCH** statement.



**You are now able to:**

- **Write arithmetic expressions and perform calculations**
- **Process strings and parts of fields**
- **Assign values between compatible and non-compatible fields**
- **Write logical expressions and control the program flow**
- **Catch runtime errors**

## Contents

- **General information**
- **Access using the index**
- **Access using the key**
- **Access using field symbols**
- **Applied example**



**At the conclusion of this unit, you will be able to:**

- **Insert data records**
- **Read data records**
- **Change data records**
- **Delete data records**
- **Recognize the syntax of internal tables with header lines**
- **Evaluate the advantages and disadvantages of using the different types of internal tables in your applications**

Preface

Course Overview

ABAP Runtime Environment

Data Types and Data Objects

Statements



**Internal Table Operations**

Subroutines

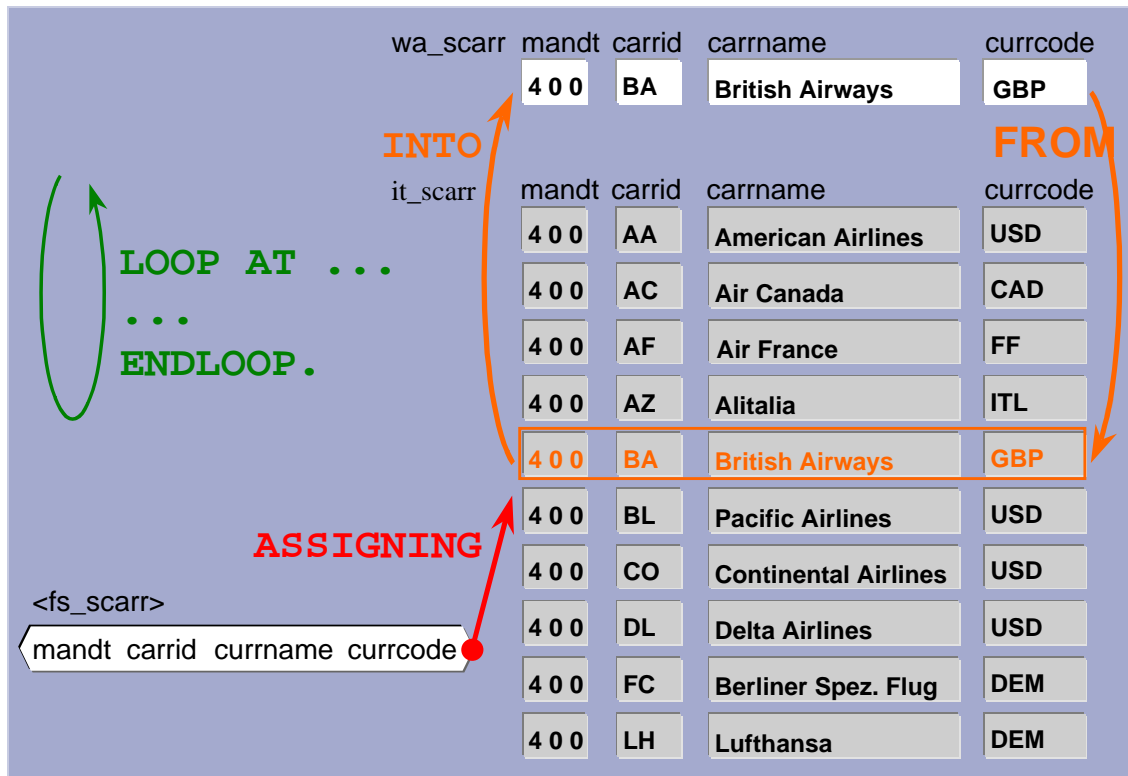
Function Groups and Function Modules

Introduction to ABAP Objects

Calling Programs and Passing Data

© SAP AG 1999




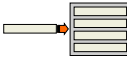




© SAP AG 1999

There are two ways of accessing the records in an internal table:

- By copying individual records into a **work area**. The work area must be compatible with the line type of the internal table.  
You can access the work area in any way, as long as the component you are trying to access is not itself an internal table. If one of the components is an internal table, you must use a further work area, whose line type is compatible with that of the nested table.  
When you change the internal table, the contents of the work area are either written back to the table or added as a new record.
  - By assigning the individual data records to an appropriate field symbol. Once the system has read an entry, you can address its components **directly** via its address. There is no copying to and from the work area. This method is particularly appropriate for accessing large or complex tables.
- If you want to read more than one record, you must use a **LOOP... ENDLOOP** structure. You can then change or delete the line that has just been read, and the system applies the change to the table body. You can also change or delete lines using a logical condition.

# Appending, Inserting, and Reading With Index Tables

	 Standard tables	Sorted tables
<b>Append</b>		
Single line	APPEND <wa> TO <itab>.	<div>Only possible if you do not violate the sort sequence</div>
Initial line	APPEND INITIAL LINE TO <itab>.	
Lines from an (index) table	APPEND LINES OF <itab1> [FROM <n1>] [TO <n2>] TO <itab2>.	
<b>Insert</b>		
Single line	INSERT <wa> INTO <itab> INDEX <n>.	<div>Only possible in a loop</div>
Initial line	INSERT INITIAL LINE INTO <itab> INDEX <n>.	
Several lines	INSERT <wa> INTO <itab>.	
Lines from an (index) table	INSERT LINES OF <itab1> [FROM <n1>] [TO <n2>] INTO <itab2>.	<div>sy- tabix contains the index of the line read</div>
<b>Read</b>		
Single line	READ TABLE <itab> INDEX <n> INTO <wa> [COMPARING ...] [TRANSPORTING ...].	

© SAP AG 1999

When you use the above statements with **sorted** tables, you must ensure that the sort sequence is maintained.

Within a loop, the **INSERT** statement adds the data record before the **current** record in the table. If you want to insert a set of lines from an internal table into another index table, you should use the **INSERT LINES OF <itab>** variant instead.

When you read single data records, you can use two further additions:

- In the **COMPARING** addition, the system compares the field contents of a data record with those in the work area for equality.
- In the **TRANSPORTING** addition, you can restrict the data transport to selected fields.

## Other statements for standard tables

- **SORT <itab> [ASCENDING|DESCENDING] [BY <f1> [ASCENDING|DESCENDING] .. <fn> [ASCENDING|DESCENDING]][AS TEXT] [STABLE].**

These statements sort the table by the table key or the specified field sequence. If you do not use an addition, the system sorts ascending. If you use the **AS TEXT** addition, character fields are sorted in culture-specific sequence. The relative order of the data records with identical sort keys **only** remain constant if you use the **STABLE** addition.

- **APPEND <wa> INTO <rank> SORTED BY <f>.**

This statement appends the work area to the **ranked list** <ranked> in descending order. The ranked list **may not be longer than the specified INITIAL SIZE**, and the work area must satisfy the sort order of the table.

## Changing, Deleting and Looping in Index Tables

SAP

	Standard tables	Sorted tables
<b>Change</b>		
Single line	<code>MODIFY &lt;itab&gt; FROM &lt;wa&gt; INDEX &lt;n&gt; [TRANSPORTING ...].</code>	✓
Several lines	<code>MODIFY &lt;itab&gt; FROM &lt;wa&gt;.</code>	✓
	Only possible in a loop	
<b>Delete</b>		
Single	<code>DELETE &lt;itab&gt; INDEX &lt;n&gt;.</code>	✓
Several	<code>DELETE &lt;itab&gt; [FROM &lt;n1&gt;] [TO &lt;n2&gt;] [WHERE &lt;logic_expr&gt;].</code>	✓
	Only possible in a loop unless you use an addition	
<b>Loops</b>		
	<code>LOOP AT &lt;itab&gt; INTO &lt;wa&gt; [FROM &lt;n1&gt;] [TO &lt;n2&gt;].</code> <code>...</code> <code>ENDLOOP.</code>	✓
	sy-tabix contains the current line index	

© SAP AG 1999

The statements listed here can be used freely with both **standard** and **sorted** tables.

When you **change** a single line, you can specify the fields that you want to change using the **TRANSPORTING** addition. Within a loop, **MODIFY** changes the **current** data record.

If you want to **delete** a set of lines from an index table, use the variant **DELETE <itab> FROM... TO..** or **WHERE...** instead of a loop. You can program almost any logical expression after **WHERE**. The only restriction is that the first field in each comparison must be a component of the line structure (see the corresponding Open SQL statements). You can pass component names dynamically.

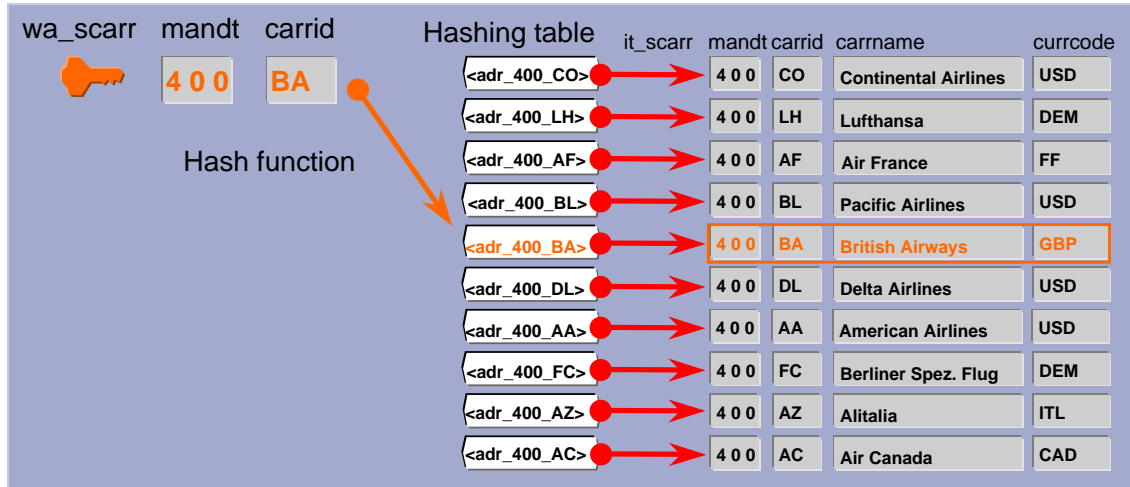
If you want to delete the **entire internal table**, use the statement **CLEAR <itab>**.

In the **LOOP AT... ENDLOOP** structure, the statements within the loop are applied to **each data record** in turn. The **INTO** addition copies entries one at a time into the work area.

The system places the index of the current loop pass in the system field **sy-tabix**. When the loop has finished, **sy-tabix** has the same value that it had **before the loop started**.

Inserting and deleting lines within a loop affects the following loop passes.

# Hashed Tables



Sort



```
SORT <itab> [ASCENDING|DESCENDING]
            [BY <f1> [ASCENDING|DESCENDING]
              ..
              <fn> [ASCENDING|DESCENDING]]
            [AS TEXT][STABLE].
```

© SAP AG 1999

Access to a hashed table is implemented using a hash algorithm. **Simplified**, this means that the data records are distributed **randomly but evenly** over a particular memory area.. The addresses are stored in a special table called the **hashing table**. There is a hash function, which determines the address at which the pointer to a data record with a certain key can be found. The function is not injective, that is, there can be several data records stored at a single address. This is implemented internally as a chained list.

Therefore, although the system still has to search sequentially within these areas, it only has to read a few data records (usually no more than three). The graphic illustrates the simplest case, that is, in which there is only one data record stored at each address.

Using a hash technique means that the access time no longer depends on the total number of entries in the table. On the contrary, it is always very fast. Hash tables are therefore particularly useful for large tables with which you use predominantly read access.

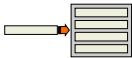
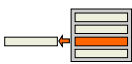
Data records are not inserted into the table in a sorted order. As with standard tables, you can sort hashed tables using the **SORT** statement:

```
SORT <itab> [ASCENDING|DESCENDING]
        [BY <f1> [ASCENDING|DESCENDING] ..
          <fn> [ASCENDING|DESCENDING]][AS TEXT].
```

Sorting the table can be useful if you later want to use a loop to access the table.

## Inserting and Reading Using Key Access

SAP

<b>Insert</b>		
Single line	<code>INSERT &lt;wa&gt; INTO TABLE &lt;itab&gt;.</code>	
Initial line	<code>INSERT INITIAL LINE INTO TABLE &lt;itab&gt;.</code>	
Lines from an (index) table	<code>INSERT LINES OF &lt;itab1&gt; [FROM &lt;n1&gt;] [TO &lt;n2&gt;] INTO TABLE &lt;itab2&gt;.</code>	
		For standard and hashed tables same as append
<b>Read</b>		
Key	<code>READ TABLE &lt;itab&gt; FROM &lt;wa1&gt; INTO &lt;wa2&gt;.</code>	
[Key] fields	<code>READ TABLE &lt;itab&gt; WITH [TABLE] KEY     &lt;k1&gt; = &lt;val1&gt; .. &lt;kn&gt; = &lt;valn&gt;     INTO &lt;wa&gt;     [COMPARING ...]     [TRANSPORTING ...].</code>	
		sy- tabix contains the index of the read line
Unstructured	<code>READ TABLE &lt;itab&gt; WITH [TABLE] KEY     table_line = &lt;val&gt;     INTO &lt;wa&gt;.</code>	
		Same as index tables

© SAP AG 1999

You can use the statements listed here with tables of **all three types**. Apart from a few special cases, you can recognize the statements from the extra keyword **TABLE**. The technical implementation of the statements varies slightly according to the table type.

As a rule, index access to an internal table is quickest. However, it sometimes makes more sense to access data using key values. A **unique** key is only possible with sorted and hashed tables. If you use the syntax displayed here, your program coding is independent of the table type (generic type specification, easier maintenance).

With a standard table, inserting an entry has the same effect as appending. With sorted tables with a non-unique key, the entry is inserted before the first (if any) entry with the same key.

To read individual data records using the first variant, all fields of <wa1> that are key fields of <itab> must be filled. <wa1> and <wa2> can be identical. If you use the **WITH TABLE KEY** addition in the second variant, you must also specify the key fully. Otherwise, the system searches according to the sequence of fields that you have specified, using a binary search where possible. You can force the system to use a binary search with a standard table using the **BINARY SEARCH** addition. In this case, you must sort the table by the corresponding fields first. The system returns the first entry that meets the selection criteria.

## Changing, Deleting, and Loop Processing With Key Access

SAP

<b>Change</b>	
Single line	<code>MODIFY TABLE &lt;itab&gt; FROM &lt;wa&gt; [TRANSPORTING ...].</code>
Several lines	<code>MODIFY &lt;itab&gt; FROM &lt;wa&gt; TRANSPORTING &lt;f1&gt; .. &lt;fn&gt; WHERE &lt;logic_expr&gt;.</code>
<b>Delete</b>	
Single	<code>DELETE TABLE &lt;itab&gt; FROM &lt;wa&gt;.</code>
Key fields	<code>DELETE TABLE &lt;itab&gt; WITH TABLE KEY &lt;k1&gt; = &lt;val1&gt; .. &lt;kn&gt; = &lt;valn&gt;.</code>
Several	<code>DELETE &lt;itab&gt; WHERE &lt;logic_expr&gt;.</code>
<b>Loop</b>	
	<code>LOOP AT &lt;itab&gt; INTO &lt;wa&gt; [WHERE &lt;logic_expr&gt; [TRANSPORTING NO FIELDS]]. ... ENDLOOP.</code>

sy- tabix contains  
the current  
line index

© SAP AG 1999

Similarly to when you read entries, when you change and delete entries using the key and a work area, you must specify all of the key fields.

You can prevent fields from being transported into the work area during loop processing by using the **TRANSPORTING NO FIELDS** addition in the **WHERE** condition. (You can use this to count the number of a particular kind of entry.)

### Other statements for all table types

- **DELETE ADJACENT DUPLICATES FROM** <itab>  
**[COMPARING <f1> .. <fn>|ALL FIEL <fn>|ALL FIELDS]].**

The system deletes all adjacent entries with the same key field contents apart from the first entry. You can prevent the system from only comparing the key field using the **COMPARING** addition. If you sort the table by the required fields beforehand, you can be sure that only unique entries will remain in the table after the **DELETE ADJACENT DUPLICATES** statement.

- Searches all lines of the table <itab> for the string . If the search is successful, the system sets the fields **sy-tabix** and **sy-fdpos**.
- **FREE** <itab>.  
Unlike **CLEAR**, which only deletes the contents of the table, **FREE** releases the memory occupied by it as well.

```
TYPES:
  BEGIN OF t_cust,
    id      TYPE scustom-id,
    name    TYPE scustom-name,
    city    TYPE scustom-city,
    reg_date LIKE sy-datum,
  END OF t_cust,

  t_cust_list TYPE STANDARD TABLE OF t_cust
               WITH NON-UNIQUE KEY id.

DATA:
  wa_cust TYPE t_cust,
  wait_list TYPE t_cust_list,

  position TYPE sy-tabix.
```

© SAP AG 1999

If you want to access your data using the index and do not need your table to be kept in sorted order or to have a unique key, that is, when the **sequence** of the entries is the most important thing, not sorting by key or having unique entries, you should use standard tables. (If you decide you need to sort the table or access it using the key or a binary search, you can always program these functions by hand.)

This example is written to manage a waiting list.

Typical functions are:

- Adding a single entry,
- Deleting individual entries according to certain criteria,
- Displaying and then deleting the first entry from the list,
- Displaying someone's position in the list.

For simplicity, the example does not encapsulate the functions in procedures.

The first thing we do in the example is to declare line and table type, from which we can then declare a work area and our internal table. We also require an elementary field for passing explicit index values.

```
*** add a waiting customer:
READ TABLE wait_list FROM wa_cust TRANSPORTING NO FIELDS.
IF sy-subrc <> 0.
    APPEND wa_cust TO wait_list.
ENDIF.

*** delete a waiting customer:
DELETE wait_list WHERE id = wa_cust-id.

*** get the first waiting customer:
READ TABLE wait_list INTO wa_cust INDEX 1.
DELETE wait_list INDEX 1.

*** get position of a waiting customer:
READ TABLE wait_list FROM wa_cust TRANSPORTING NO FIELDS.
position = sy-tabix.
```

© SAP AG 1999

This example omits the user dialogs and data transport, assuming that you understand the principles involved. We really only want to concentrate on the table access:

### ■ Adding new entries

The data record for a waiting customer is only added to the table if it does not already exist in it. If the table had a unique key, you would not have had to have programmed this check yourself.

### ■ Deleting single entries according to various criteria

The criterion is the key field. However, other criteria would be possible - for example, deleting data records older than a certain insertion date **reg\_date**.

### ■ Displaying and deleting the first entry from the list

Once a customer comes to the top of the waiting list, you can delete his or her entry. If the waiting list is empty, such an action has no effect. Consequently, you do not have to check whether there are entries in the list before attempting the deletion.

### ■ Displaying the position of a customer in the waiting list

As above, you do not need to place any data in the work area. We are only interested in the values of **sy-subrc** and **sy-tabix**. If the entry is not in the table, **sy-tabix** is set to zero.

At this stage, let us return to the special case of the restricted ranked list:

```
DATA <rank> {TYPE|LIKE} STANDARD TABLE OF ... INITIAL SIZE <n>. ...
APPEND <wa> INTO <rank> SORTED BY <f>.
```



```
DATA:
  wa_flight TYPE sflight,

  flight_list TYPE SORTED TABLE OF sflight
                WITH UNIQUE KEY carrid connid fldate.

SELECT * FROM sflight
        INTO CORRESPONDING FIELDS OF TABLE flight_list.

*** insert additional flight dates:
INSERT wa_flight INTO TABLE flight_list.

*** adjust pricing:
wa_flight-price = '1250.00'. wa_flight-currency = 'USD'.
MODIFY flight_list FROM wa_flight TRANSPORTING price currency
        WHERE carrid = wa_flight-carrid
        AND   connid = wa_flight-connid.

*** modify database table from internal table ...
```

© SAP AG 1999

When you choose to use a sorted table, it will normally be because you want to define a **unique key**. The mere fact that the table is kept in sorted order is not that significant, since you can sort any kind of internal table. However, with sorted tables (unlike hashed tables), new data records are inserted in the correct sort order. If you have a table with few entries but lots of accesses that change the contents, a sorted table may be more efficient than a hashed table in terms of runtime.

The aim of the example here is to modify the contents of a database table. The most efficient way of doing this is to create a local copy of the table in the program, make the changes to the copy, and then write all of its data back to the database table. When you are dealing with large amounts of data, this method both saves runtime and reduces the load on the database server.

Since the internal table represents a database table in this case, you should ensure that its records have unique keys. This is assured by the key definition. Automatic sorting can also bring further advantages. When you change a group of data records, only the fields **price** and **currency** are copied from the work area.

For information about changing database tables (data consistency, authorization and locking issues), refer to course **BC414 (Programming Database Updates)** and the online documentation.

```
TYPES:
  BEGIN OF t_city,
    city      TYPE sgeocity-city,
    country   TYPE sgeocity-country,
    latitude  TYPE sgeocity-latitude,
    longitude TYPE sgeocity-longitude,
  END OF t_city,

  t_city_list TYPE HASHED TABLE OF t_city
               WITH UNIQUE KEY city country.

DATA:
  wa_city  TYPE t_city,
  city_list TYPE t_city_list.

PARAMETERS:
  pa_city TYPE sgeocity-city,
  pa_ctype TYPE sgeocity-country.
```

© SAP AG 1999

The hash algorithm **calculates** the address of an entry based on the key. This means that, with larger tables, the access time is reduced significantly in comparison with a binary search. In a loop, however, the hashed table has to search the entire table (full table scan). Since the table entries are stored unsorted, it would be better to use a sorted table if you needed to run a loop through a left-justified portion of the key. It can also be worth using a hashed table but sorting it.

A typical use for hashed tables is to buffer detailed information that you need repeatedly and can identify using a unique key. You should bear in mind that you can also set up table buffering for a table in the ABAP Dictionary to cover exactly the same case. However, whether the tables are buffered on the application table depends on the size of the database table. Buffering in the program using hashed tables also allows you to restrict the dataset according to your own needs, or to buffer additional data as required. In this example, we want to allow the user to enter the name of a city, and the system to display its geographical coordinates.

```
SELECT city country latitude longitude
      FROM sgeocity
      INTO CORRESPONDING FIELDS OF TABLE city_list.

READ TABLE city_list WITH TABLE KEY city      = pa_city
                                country    = pa_ctype
                                INTO wa_city.

CHECK sy-subrc = 0.
WRITE: / wa_city-city,
        wa_city-country,
        wa_city-latitude,
        wa_city-longitude.
```

© SAP AG 1999

First, we fill our "buffer table" **city\_list** with values from the database table **sgeocity**.

Then, we read an entry from the hashed table, specifying the **full key**.

The details are displayed as a simple list.

At this point, it is worth repeating that you should only use this buffering technique if you want to keep **large amounts of data locally** in the program. You must ensure that you design your hashed table so that it is possible to specify the full key when you access it from your program.

## Internal Table With Header Line



### Work area

wa_scarr	mandt	carrid	carrname	currcode
4 0 0	CO	Continental Airlines	USD	

### Internal table

it_scarr	mandt	carrid	carrname	currcode
4 0 0	CO	Continental Airlines	USD	
4 0 0	LH	Lufthansa	DEM	
4 0 0	AF	Air France	FF	

### Header line

it_scarr	mandt	carrid	carrname	currcode
4 0 0	CO	Continental Airlines	USD	

### Internal table

it_scarr	mandt	carrid	carrname	currcode
4 0 0	CO	Continental Airlines	USD	
4 0 0	LH	Lufthansa	DEM	
4 0 0	AF	Air France	FF	

```

APPEND <wa> TO <itab>.
INSERT <wa> INTO <itab> INDEX <n>.
MODIFY <itab> INDEX <n> FROM <wa>.
DELETE TABLE <itab> FROM <wa>.
    
```

```

READ TABLE <itab> INDEX <n>
      INTO <wa>.
LOOP AT <itab> INTO <wa>.
  WRITE <wa>--<f>.
ENDLOOP.
    
```

### Explicit syntax

```

APPEND <itab>.
INSERT <itab> INDEX <n>.
MODIFY <itab> INDEX <n>.
DELETE TABLE <itab>.
    
```

```

READ TABLE <itab> INDEX <n>.

LOOP AT <itab>.
  WRITE <itab>--<f>.
ENDLOOP.
    
```

### Implicit syntax

© SAP AG 1999

You can define internal tables either with (**WITH HEADER LINE** addition) or without header lines. An **internal table with header line** consists of a work area (header line) and the actual table body. You address both objects **using the same name**. The way in which the system interprets the name depends on the context. For example, the **MOVE** statement applies to the header line, but the **SEARCH** statement applies to the body of the table.

To avoid confusion, you are recommended to use **internal tables without header lines**. This is particularly important when you use nested tables. However, internal tables with header line do offer a shorter syntax in several statements (**APPEND**, **INSERT**, **MODIFY**, **COLLECT**, **DELETE**, **READ**, **LOOP**). Within ABAP Objects, you can only use internal tables **without** a header line.

You can always address the body of an internal table <itab> explicitly by using the following syntax: <itab>[]. This syntax is always valid, whether the internal table has a header line or not.

#### Example

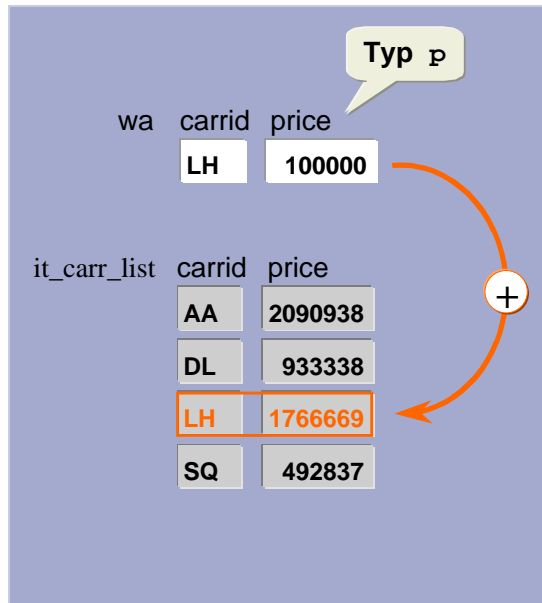
**DATA itab1 TYPE TABLE OF i WITH HEADER LINE.**

**DATA itab2 TYPE TABLE OF i WITH HEADER LINE.**

**itab1 = itab2.** " Only header lines will be copied

**itab1[] = itab2[].** " Copies table body

**COLLECT** [<wa> INTO] <itab>.



**TYPES:**

```
BEGIN OF t_carr_info,
  carrid TYPE sflight-carrid,
  price  TYPE sflight-price,
END OF t_carr_info.
```

**DATA:**

```
it_carr_list TYPE HASHED
  TABLE OF t_carr_info
  WITH UNIQUE KEY carrid,
```

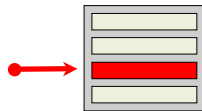
```
wa TYPE t_carr_info.
```

```
SELECT carrid price
  FROM sflight
  INTO wa.
COLLECT wa INTO it_carr_list.
ENDSELECT.
```

You can only use the **COLLECT** statement with internal tables whose **non-key fields** are **all numeric** (type **I**, **P**, or **F**).

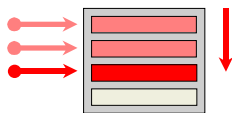
The **COLLECT** statement adds the work area or header line to an internal entry with the same type or, if there is none, adds a new entry to the table. It searches for the entry according to the table type and defined key. If it finds an entry in the table, it adds all numeric fields that are not part of the key to the existing values. If there is not already an entry in the table, it appends the contents of the work area or header line to the end of the table.

## Read



```
READ TABLE <itab> [INDEX <n>
                     FROM <wa>
                     WITH [TABLE] KEY ...] ASSIGNING <<fs_line>>.
```

## Loops



```
LOOP AT <itab> ASSIGNING <<fs_line>> [WHERE ...].
...
ENDLOOP.
```

**You can now access <<fs\_line>>-component directly**

© SAP AG 1999

When you read a table line using **READ** or a series of table lines using **LOOP AT**, you can assign the lines of the internal table to a field symbol using the addition ... **ASSIGNING** <<field symbol>>. The field symbol <<field symbol>> then points to the line that you assigned, allowing you to access it directly. Consequently, the system does not have to copy the entry from the internal table to the work area and back again.

The field symbol <<field\_symbol>> must have the same type as the line type of the internal table.

However, there are also certain restrictions when you use this technique:

- You can only change the contents of key fields if the table is a **standard table**.
- You **cannot** use the **SUM** statement in control level processing. (The **SUM** statement adds all of the numeric fields in the work area.)

This technique is particularly useful for accessing a lot of table lines or nested tables within a loop. Copying values to and from the work area in this kind of operation is particularly runtime-intensive.

## Example I - Declaring Nested Tables



```
TYPES:
  BEGIN OF t_conn,
    cityfrom TYPE spfli-cityfrom,
    cityto   TYPE spfli-cityto,
    carrid   TYPE spfli-carrid,
    connid   TYPE spfli-connid,
  END OF t_conn.

DATA:
  conn_list TYPE STANDARD TABLE OF t_conn,

  BEGIN OF wa_travel,
    dest      TYPE spfli-cityto,
    cofil_list LIKE conn_list,
  END OF wa_travel,

  travel_list LIKE SORTED TABLE OF wa_travel WITH UNIQUE KEY dest.

PARAMETERS
  pa_start TYPE spfli-cityfrom DEFAULT 'FRANKFURT'.
```

© SAP AG 1999

In this example, the user should be able to enter a departure city, for which all possible flights are then listed.

To do this, we declare an inner table (**cofl\_list**) and an outer table (**travel\_list**) with corresponding work areas.

A further internal table (**conn\_list**) buffers and sorts all of the flight connections.

### Note

In order to allow loop access using field symbols, the buffer table and the inner internal table must have the same type. Furthermore, we want to sort the table by various criteria later on. Consequently, we are using standard tables, not sorted tables.

## Example II - Loop Access Using Field Symbols



```
FIELD-SYMBOLS:
  <fs_conn>      TYPE t_conn,
  <fs_conn_int>  TYPE t_conn,
  <fs_travel>    LIKE wa_travel.

SELECT carrid connid cityfrom cityto
      FROM spfli
      INTO CORRESPONDING FIELDS OF TABLE conn_list.
SORT conn_list BY cityfrom cityto ASCENDING AS TEXT.

*** build up nested table:
LOOP AT conn_list ASSIGNING <fs_conn> WHERE cityfrom = pa_start.
  wa_travel-dest = <fs_conn>-cityto.

  LOOP AT conn_list ASSIGNING <fs_conn_int>
    WHERE cityfrom = wa_travel-dest.
    APPEND <fs_conn_int> TO wa_travel-cofl_list.
  ENDLOOP.

  SORT wa_travel-cofl_list BY cityto carrid ASCENDING AS TEXT.
  INSERT wa_travel INTO TABLE travel_list.
ENDLOOP.
```

© SAP AG 1999

We also need to declare three field symbols with the line types of the internal tables.

First, the flight connections starting in the city entered in **pa\_start** are assigned to the field symbol **<fs\_conn>**.

We are only interested in the arrival city of each flight connection. This is the first entry in a line of the outer table **travel\_list**.

Before this can be entered properly sorted, the inner table **wa\_travel-cofl\_list** must be filled with the cities that can be reached from it. To do this, the program looks for the corresponding flight connections and appends them to the inner table.

Next, the field **cityfrom** is initialized, since it is required for control level processing in the display. The table is then sorted by field **cityto** and **carrid**.



```
*** output:
LOOP AT travel_list ASSIGNING <fs_travel>.
  WRITE: / <fs_travel>-dest.

  LOOP AT <fs_travel>-cofl_list ASSIGNING <fs_conn_int>.
    AT NEW cityto.
      WRITE: /8 <fs_conn_int>-cityto.
    ENDAT.

    WRITE: /16 <fs_conn_int>-carrid,
           <fs_conn_int>-connid.
  ENDLOOP.
ENDLOOP.
```

© SAP AG 1999







Because of the **control level processing** used here, only new arrival cities are written in the list.

### Note

It would have been possible to solve this problem using nested **SELECT** statements. However, this is not a realistic proposition because of the excessive load that we would then place on the database server.

## Internal Table Operations: Summary

SAP

	Standard table	Sorted table	Hashed table
<b>Index access</b>			
APPEND	✓	 Sort sequence could be violated	✗
INSERT	✓	 Sort sequence could be violated	✗
READ TABLE	✓	✓	✗ No index access to hashed tables
LOOP AT	✓	✓	✗
MODIFY	✓		✗
DELETE	✓	✓	✗
SORT	✓		✓
<b>Key access</b>			
INSERT	Same as append	✓	Same as append
READ TABLE	✓	✓	✓
LOOP AT	✓	✓	✓
MODIFY	✓	✓	✓
DELETE	✓	✓	✓
COLLECT	✓	✓	✓

© SAP AG 1999

**Index access** (APPEND, INSERT ... INDEX, LOOP ... FROM TO and so on) is possible for standard and sorted tables. A possible consequence of this is that you may cause a runtime error by violating the sort sequence if you use **INSERT** with an index or **APPEND** on a sorted table.

**SORT** can only apply to standard and hashed tables. It has no positive effect in a sorted table, and can lead to a syntax or runtime error if the attempted sort violates the key sequence.

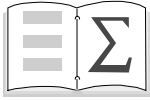
You can use **key access** for any table type, but the runtime required varies according to the table type.

The runtime depends on whether the values are part of the key (so the sequence in which you pass them is also significant). **INSERT** for a standard table or hashed table using the key has the same effect as the **APPEND** statement.

The system supports **control level processing** for all table types. Hashed and standard tables must be sorted beforehand.

### Preview

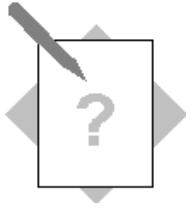
If you want to process data records with different structures, you can use **extracts**. For further information about control level processing and extracts, refer to course **BC405 (Techniques of List Processing and ABAP Query)**.



**You are now able to:**

- **Insert data records**
- **Read data records**
- **Change data records**
- **Delete data records**
- **Recognize the syntax of internal tables with header lines**
- **Evaluate the advantages and disadvantages of using the different types of internal tables in your applications**

## Internal Table Operations: Exercises

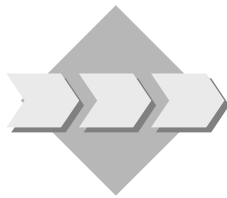


**Unit:** Internal table operations  
**Topic:** Filling internal tables  
and accessing their entries



At the conclusion of these exercises, you will be able to:

- Fill internal tables
- Access their entries



You are a programmer for an airline consortium, and it is your job to write analysis programs for several airlines.

1. Extend your program from task 1 of the previous exercises:  
You should fill the table with the list of airport names, and then the other table with the assignment of airlines to airports and counter numbers. You should then display the second table in a list.  
**## is your two-digit group number.**  
Model solution:  
SAPBC402\_TABS\_COUNTERLIST2
  - 1-1 Copy your solution to exercise 1 from the last unit (or the model solution) to the new program **Z##\_BC402\_COUNTERLIST2**.
  - 1-2 Fill the internal table **it\_airport\_buffer** with the codes of the airports and their names from the transparent table **SAIRPORT**.
  - 1-3 Use a loop to select data from the transparent table **SCOUNTER** for all of the airlines in the user's selection. You can fill the fields **carrid**, **countnum**, and **airport** in the work area **wa\_counter** directly. To fill **airp\_name**, you must use a single-record access to the internal table **it\_airport\_buffer**. Once you have filled the work area, append it to the internal table **it\_carr\_counter**.
  - 1-4 Display the internal table **it\_carr\_counter** sorted by the fields **airport** and **carrid** in ascending order.

2. Extend your program from section 2 of the previous exercises:  
You should now run an authorization check to see if the user is authorized to see the data for the airlines that he or she included in the selection on the selection screen.  
You should then place the airlines that the user chose and for which he or she has authorization in a suitable internal table.

You will then display the output in a list.

**## is your two-digit group number.**

Model solution:

SAPBC402\_TABS\_FLIGHTLIST2

- 2-1 Copy your solution to exercise 2 from the last unit (or the model solution) to the new program **Z##\_BC402\_FLIGHTLIST2**.
- 2-2 Event block **AT SELECTION-SCREEN**:  
Use a loop on the transparent table **SCARR** to fill the **low** field of work area **wa\_allowed\_carr**.  
Now perform an authorization check against the authorization object **S\_CARRID** for this airline and the activity '**Display**' (use the **Pattern** function in the ABAP Editor).  
Only if the check is successful should you fill the **sign** and **option** fields of the work area and append it to the selection table.
- 2-3 Event block **START-OF-SELECTION**:  
Use the view **BC402\_FLIGHTS** to fill the elementary fields of the internal table **it\_flights**.  
However, at this stage, leave the **inner** internal table **it\_planes** initial.  
You **cannot** use the "array fetch" variant of the **SELECT** statement with nested internal tables.  
You must therefore program a loop with the target fields listed in the **INTO** clause, and insert the work area into the table in the correct sort order.
- 2-4 Then, display the contents of the internal table **it\_flights** as a list. To do this, use a field symbol **<fs\_flight>**, appropriately typed.  
Display only those flights for which there is at least one booking. When you display the amount, remember the currency addition.
- 2-5 Maintain appropriate list headers.



You can run the program and then maintain the list headers from the displayed list.

## Internal Table Operations: Solutions



**Unit:** Internal table operations  
**Topic:** Filling internal tables  
and accessing their entries

### 1 Model solution SAPBC402\_TABS\_COUNTERLIST2

```
*&-----*
*& Report SAPBC402_TABS_COUNTERLIST2          *
*&                                           *
*&-----*
*& solution of exercise 1 operations on internal tables *
*&                                           *
*&-----*
```

REPORT sapbc402\_tabs\_counterlist2.

TYPES:

```
BEGIN OF t_airport,
  id TYPE saairport-id,
  name TYPE saairport-name,
END OF t_airport,
```

```
BEGIN OF t_counter,
  airport TYPE scounter-airport,
  airp_name TYPE saairport-name,
  carrid TYPE scounter-carrid,
  countnum TYPE scounter-countnum,
END OF t_counter.
```

DATA:

```
it_carr_counter TYPE STANDARD TABLE OF t_counter,
```

```
wa_counter TYPE t_counter,
```

```
it_airport_buffer TYPE HASHED TABLE OF t_airport
  WITH UNIQUE KEY id,
```

```
wa_airport TYPE t_airport.
```

SELECT-OPTIONS so\_carr FOR wa\_counter-carrid.

**START-OF-SELECTION.**

\* buffer the airport names:

\*\*\*\*\*

```
SELECT id name
  FROM saairport
 INTO CORRESPONDING FIELDS OF TABLE it_airport_buffer.
```

\* fill an internal table with all counters of the selected  
\* carriers:  
\*\*\*\*\*

```
SELECT carrid countnum airport
  FROM scounter
  INTO CORRESPONDING FIELDS OF wa_counter
  WHERE carrid IN so_carr.
```

```
READ TABLE it_airport_buffer
  INTO wa_airport
  WITH TABLE KEY id = wa_counter-airport.
wa_counter-airp_name = wa_airport-name.
APPEND wa_counter TO it_carr_counter.
```

ENDSELECT.

**SORT it\_carr\_counter BY airport carrid ASCENDING AS TEXT.**

\* display list:  
\*\*\*\*\*

```
LOOP AT it_carr_counter INTO wa_counter.
  WRITE: / wa_counter-airport,
         wa_counter-airp_name,
         wa_counter-carrid,
         wa_counter-countnum.
ENDLOOP.
```

## 2      **Model solution SAPBC402\_TABS\_FLIGHTLIST2**

```
*&-----*
*& Report  SAPBC402_TABS_FLIGHTLIST2                *
*&                                     *
*&-----*
*& solution of exercise 2 operations on internal tables *
*&                                     *
*&-----*
```

REPORT sapbc402\_tabs\_flightlist2.

TYPES:

```
BEGIN OF t_flight,
  carrid  TYPE spfli-carrid,
  connid  TYPE spfli-connid,
  fldate  TYPE sflight-fldate,
  cityfrom TYPE spfli-cityfrom,
  cityto  TYPE spfli-cityto,
  seatsocc TYPE sflight-seatsocc,
  paymentsum TYPE sflight-paymentsum,
  currency TYPE sflight-currency,
  it_planes TYPE bc402_typs_planetab,
END OF t_flight,

t_flighttab TYPE SORTED TABLE OF t_flight
             WITH UNIQUE KEY carrid connid fldate.
```

DATA:

```
wa_flight TYPE t_flight,
it_flights TYPE t_flighttab.
```

SELECT-OPTIONS so\_carr FOR wa\_flight-carrid.

**FIELD-SYMBOLS <fs\_flight> TYPE t\_flight.**

```
* for authority-check:
*****
```

DATA:

```
allowed_carriers TYPE RANGE OF t_flight-carrid,
wa_allowed_carr  LIKE LINE OF allowed_carriers.
```



## START-OF-SELECTION.

\* fill a range table with the allowed carriers:

\*\*\*\*\*

```
SELECT carrid
  FROM scarr
  INTO wa_allowed_carr-low
  WHERE carrid IN so_carr.

AUTHORITY-CHECK OBJECT 'S_CARRID'
  ID 'CARRID' FIELD wa_allowed_carr-low
  ID 'ACTVT' FIELD '03'. " display
IF sy-subrc <> 0.
  CLEAR wa_allowed_carr.
ELSE.
  wa_allowed_carr-sign = 'I'.
  wa_allowed_carr-option = 'EQ'.
  APPEND wa_allowed_carr TO allowed_carriers.
ENDIF.
```

ENDSELECT.

\* fill an internal table with connection and flight data

\* for the allowed carriers:

\*\*\*\*\*

```
SELECT carrid connid fldate cityfrom cityto
      seatsocc paymentsum currency
  FROM bc402_flights
  INTO (wa_flight-carrid, wa_flight-connid,
        wa_flight-fldate,
        wa_flight-cityfrom, wa_flight-cityto,
        wa_flight-seatsocc,
        wa_flight-paymentsum, wa_flight-currency)
  WHERE carrid IN allowed_carriers.
```

```
INSERT wa_flight INTO TABLE it_flights.
```

ENDSELECT.

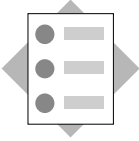
\* display list, using field-symbol:

\*\*\*\*\*

```
LOOP AT it_flights ASSIGNING <fs_flight>
      WHERE seatsocc > 0.
WRITE: / <fs_flight>-carrid,
      <fs_flight>-connid,
      <fs_flight>-fldate,
      <fs_flight>-cityfrom,
      <fs_flight>-cityto,
      <fs_flight>-seatsocc,
      <fs_flight>-paymentsum
      CURRENCY <fs_flight>-currency,
      <fs_flight>-currency.
ENDLOOP.
```

## Contents

- Defining the interface
- Calling subroutines
- Lifetime and visibility
- Use



**At the conclusion of this unit, you will be able to:**

- **Define subroutines**
- **Call subroutines**
- **Use the different techniques that exist for passing and typing interface parameters**

Preface

Course Overview

ABAP Runtime Environment

Data Types and Data Objects

Statements

Internal Table Operations



**Subroutines**

Function Groups and Function Modules

Introduction to ABAP Objects

Calling Programs and Passing Data

© SAP AG 1999

```

FORM <srout> [...]
    [USING    ... ]      "import parameters
    [CHANGING ... ]      "import/export parameters

    [TYPES ...]          "local data types
    [DATA  ...]          "local data objects
    ...                  "statements

ENDFORM.
    
```



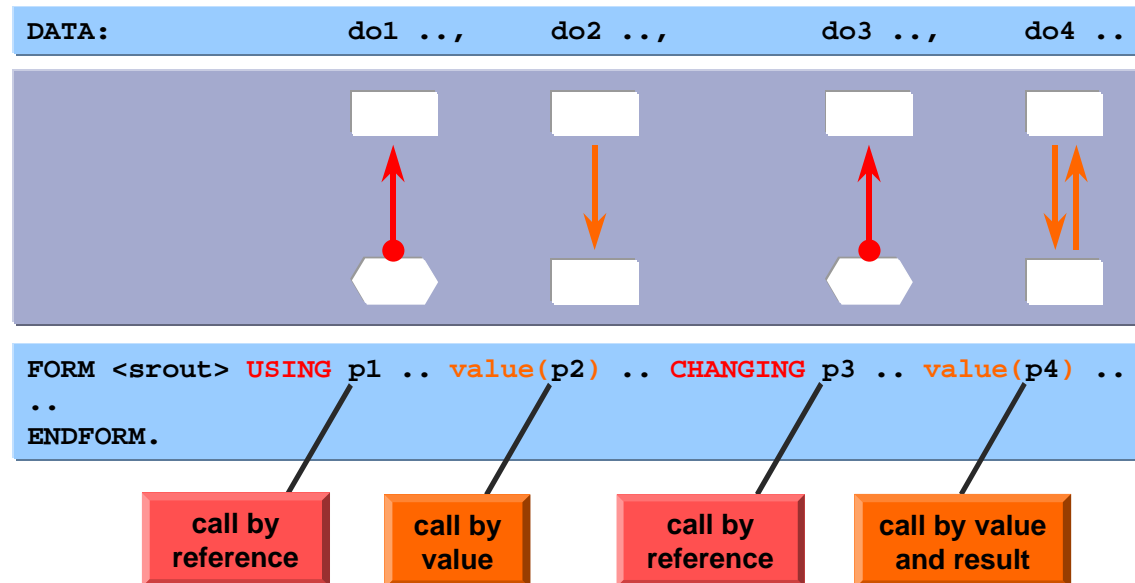
© SAP AG 1999

A subroutine is an **internal** modularization unit within a **program**, to which you can pass data using an interface. You use subroutines to encapsulate parts of your program, either to make the program easier to understand, or because a particular section of coding is used at several points in the program. Your program thus becomes more function-oriented, with its task split into different constituent functions, and a different subroutine responsible for each one.

As a rule, subroutines also make your programs easier to maintain. For example, you can execute them "invisibly" in the Debugger, and only see the result. Consequently, if you know that there are no mistakes in the subroutine itself, you can identify the source of the error faster.

## Structure of a Subroutine

- A subroutine begins with the **FORM** statement and ends with **ENDFORM**.
- After the subroutine name, you program the interface. In the **FORM** statement, you specify the **formal parameters**, and assign them types if required. The parameters **must occur in a fixed sequence** - first the importing parameters, then the importing/exporting parameters. Within the subroutine, you address the data that you passed to it using the formal parameters.
- You can declare **local** data in a subroutine.
- After any local data declarations, you program the statements that are executed as part of the subroutine.



© SAP AG 1999

You define the way in which the data from the main program (**actual parameters do1, do2, do3, and do4**) are passed to the data objects in the subroutine (**formal parameters p1, p2, p3, p4**) in the **interface**.

There are three possibilities:

■ **Call by reference (p1, p3)**

The **dereferenced address** of the actual parameter is passed to the subroutine.

The **USING** and **CHANGING** additions both have the same effect (in technical terms). However, **USING** leads to a warning in the program check.

■ **Call by value (p2)**

A **local "read only"** copy of the actual parameter is passed to the subroutine. Do this using the form **USING value(<formal parameter>)**.

■ **Call by value and result (p4)**

A **local changeable copy** of the actual parameter is passed to the subroutine. Do this using the form **CHANGING value(<formal parameter>)**.

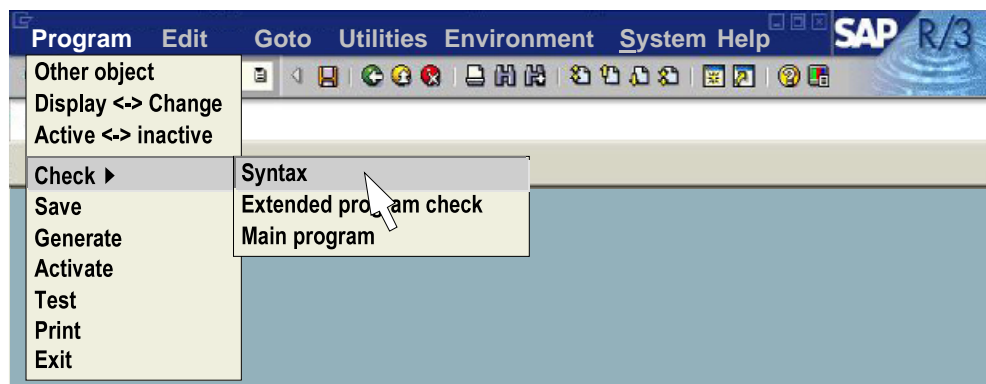
You should use this method when you want to be sure that the value of the actual parameter is not changed if the subroutine terminates early.

- When you use **internal tables** as parameters, you should use **call by reference** to ensure that the system does not have to copy what could be a large internal table.

```
FORM .. <fp> [TYPE ANY]
              [TYPE {p|n|c|string|x|xstring}]
              [TYPE [ANY|INDEX|STANDARD|SORTED|HASHED] TABLE]

              [TYPE {i|f|d|t}]
              [TYPE LINE OF <itab_type>|LIKE LINE OF <itab>]
              [TYPE <data_type>|LIKE <data_object>]
              [TYPE REF TO ..]

..
ENDFORM.
```



© SAP AG 1999

The data objects that you pass to a subroutine can have **any data type**. In terms of specifying data types, there are various rules:

■ You **may** specify the type for **elementary types**.

If you do, the **syntax check** returns an error message if you try to pass an actual parameter with a different type to the formal parameter. Not specifying a type is the equivalent of writing **TYPE ANY**. In this case, the formal parameter "inherits" its type from the actual parameter at runtime. If the statements in the subroutine are not compatible with this inherited data type, a **runtime error** occurs.

Data types **I**, **F**, **D**, and **T** are already fully-specified. If, on the other hand, you use **P**, **N**, **C**, or **X**, the missing attributes are made up from the actual parameter. If you want to specify the type fully, you must define a type yourself (although a user-defined type may itself be generic). When you use **STRING** or **XSTRING**, the full specification is not made until runtime.

■ You **must** specify the type of **structures** and **references**.

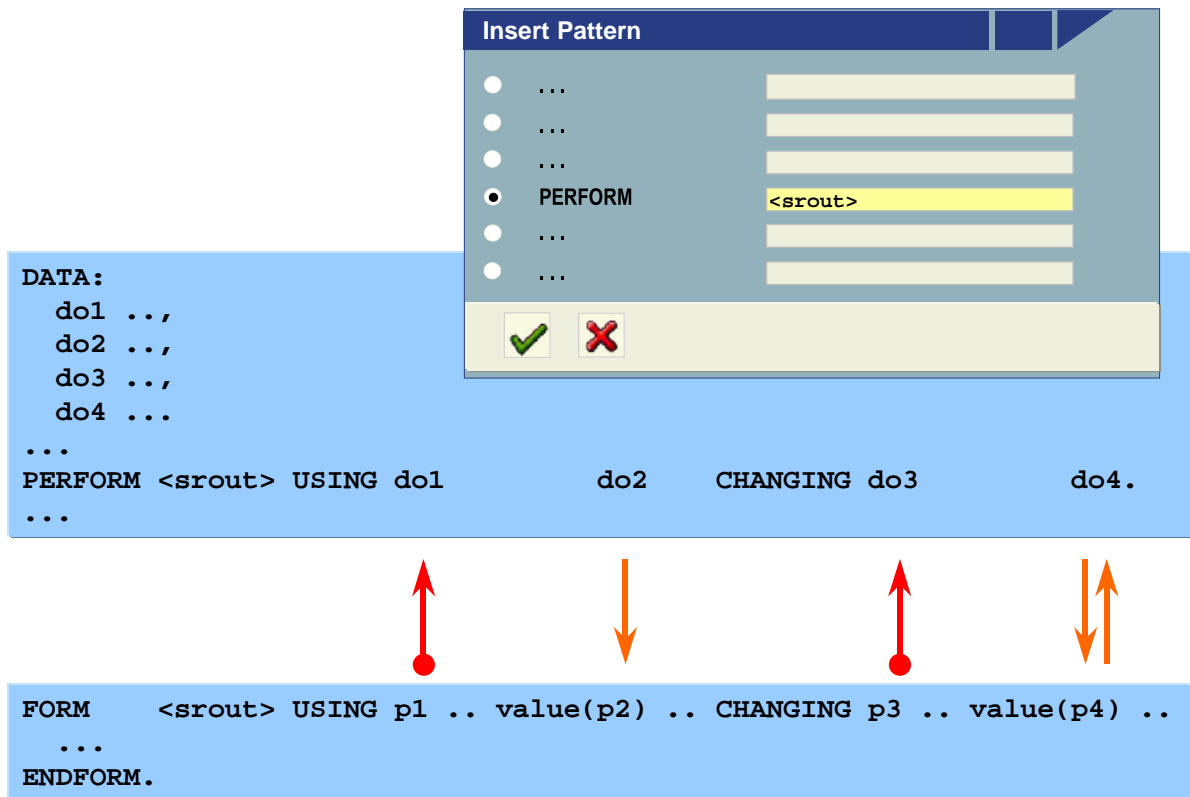
■ You **must** specify the type of an internal table, but you can use a **generic** type, that is, program the subroutine so that the statements are valid for different types of internal table, and then specify the type:

- Using the corresponding **interface specification**:  
**TYPE [ANY|INDEX|STANDARD|SORTED|HASHED] TABLE**,  
 (TYPE TABLE is the short form of TYPE STANDARD TABLE)
- Using a **user-defined** generic table type.



## Calling a Subroutine

SAP



© SAP AG 1999

When you call a subroutine, the parameters are passed **in the sequence in which they are listed**. The type of the parameters and the way in which they are passed is determined in the interface definition. When you call the subroutine, you must list the actual parameters after **USING** and **CHANGING** in the same way. There must be the same number of parameters in the call as in the interface definition. The best thing to do is to define the subroutine and then use the **Pattern** function in the ABAP Editor to generate the call. This ensures that you cannot make mistakes with the interface. The only thing you have to do is replace the formal parameters with the appropriate actual parameters. If you pass an internal table with a header line, the name is interpreted as the **header line**. To pass the body of an internal table with header line, use the form <itab>[]. In the subroutine, the internal table will **not** have a header line.

### Example

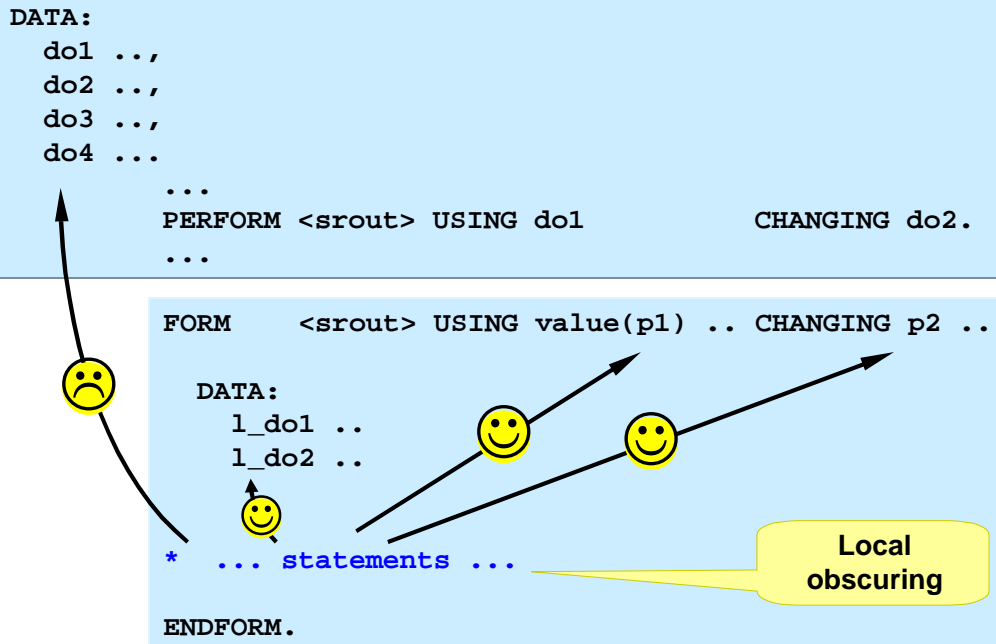
DATA it\_spfli TYPE TABLE OF spfli WITH HEADER LINE.

...  
PERFORM demosub CHANGING it\_spfli[].

...  
FORM demosub CHANGING p\_spfli LIKE it\_spfli[].

DATA wa\_p\_spfli LIKE LINE OF p\_spfli.

...  
ENDFORM.



© SAP AG 1999

Formal parameters and local data objects that you define in a subroutine are only visible while the subroutine is active. This means that the relevant memory space is not allocated until the subroutine is called, and is released at the end of the routine. The data can only be addressed during this time. The general rules are as follows:

- You can address global data objects from within the subroutine. However, you should avoid this wherever possible, since in doing so you bypass the interface, and errors can creep into your coding.
- You can only address formal parameters and local data objects from within the subroutine itself.
- If you have a formal parameter or local data object with the same name as a global data object, we say that the global object is **locally obscured** by the local object. This means that if you address an object with the shared name **in the subroutine**, the system will use the **local object**, if you use the same name **outside the subroutine**, the system will use the **global object**.

## Summary

- Address global data objects in the main program and, if you want to use them in the subroutine, pass them using the interface.
- In the subroutine, address only formal parameters and local data objects.
- Avoid assigning identical names to global and local objects. For example, use a prefix such as **p\_** for a parameter and **l\_** for local data.

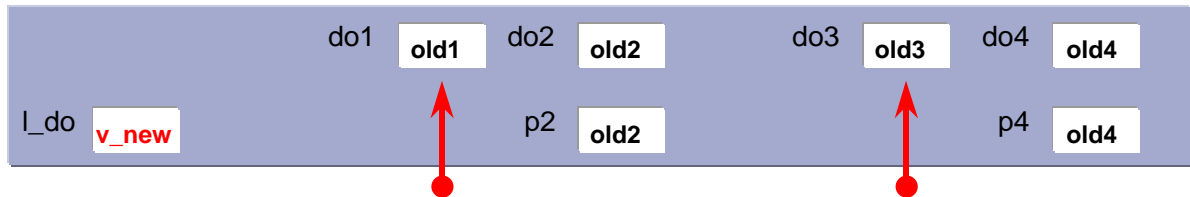
do1 **old1** do2 **old2** do3 **old3** do4 **old4**

...  
**PERFORM** demosub **USING** do1 do2 **CHANGING** do3 do4.  
 ...

**FORM** demosub **USING** p1 .. value(p2) .. **CHANGING** p3 .. value(p4) ..  
  
**DATA:**  
 l\_do .. **VALUE** ..  
  
 p1 = l\_do.  
 p2 = l\_do.  
 p3 = l\_do.  
 p4 = l\_do.  
  
**ENDFORM.**

© SAP AG 1999

This example calls the subroutine **demosub**. It contains a local data object with a starting value, and alters the four formal parameters.



```
...
PERFORM demosub USING do1          do2          CHANGING do3          do4.
...
```

```
FORM      demosub USING p1 .. value(p2) .. CHANGING p3 .. value(p4) ..
```

```
DATA:
```

```
  l_do .. VALUE ..
```

```
p1 = l_do.
```

```
p2 = l_do.
```

```
p3 = l_do.
```

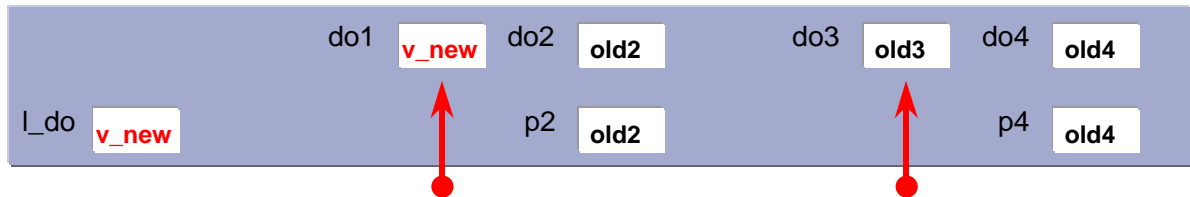
```
p4 = l_do.
```

```
ENDFORM.
```

© SAP AG 1999

The system allocates two memory areas **p2** and **p4** for the two call by value parameters **d2** and **d4**, and fills them with the respective values. It also allocates memory for the local data object **l\_do**, and fills it with the starting value.

There is no **VALUE** addition for **p1** or **p3**, This means that changes at runtime affect the actual parameters **directly**, and you can address **do1** directly via **p1**.



```
...
PERFORM demosub USING do1      do2      CHANGING do3      do4.
...
```

```
FORM      demosub USING p1 .. value(p2) .. CHANGING p3 .. value(p4) ..

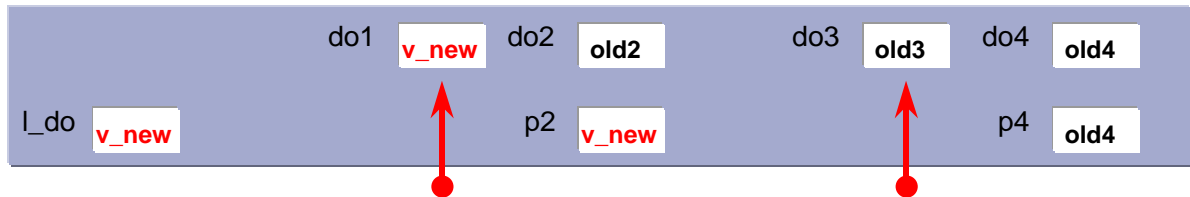
DATA:
  l_do .. VALUE ..

p1 = l_do.
p2 = l_do.
p3 = l_do.
p4 = l_do.

ENDFORM.
```

Here, the change to **p1** directly affects the contents of **do1**.

The formal parameter **p2** is declared as a local copy with read access. This means that any changes will **not** affect the actual parameter **do2** at all.



```
...
PERFORM demosub USING do1      do2      CHANGING do3      do4.
...
```

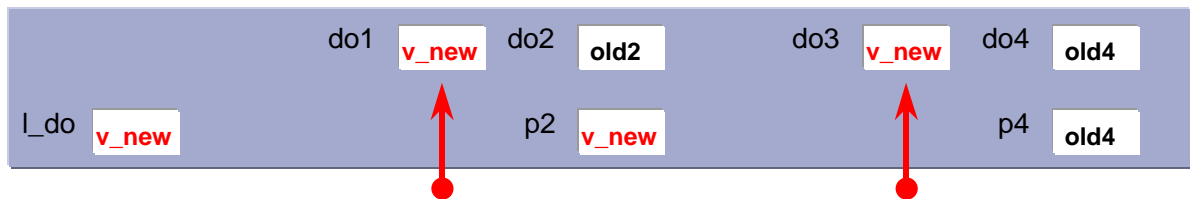
```
FORM      demosub USING p1 .. value(p2) .. CHANGING p3 .. value(p4) ..

DATA:
  l_do .. VALUE ..

p1 = l_do.
p2 = l_do.
p3 = l_do.
p4 = l_do.

ENDFORM.
```

The same applies to **p3** as to **p1**. If you do not use the **VALUE** addition, **USING** and **CHANGING** have the same effect.



```
...
PERFORM demosub USING do1          do2          CHANGING do3          do4.
...
```

```
FORM      demosub USING p1 .. value(p2) .. CHANGING p3 .. value(p4) ..

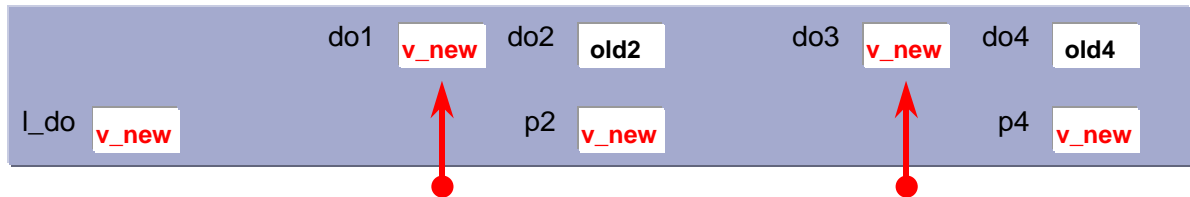
DATA:
  l_do .. VALUE ..

p1 = l_do.
p2 = l_do.
p3 = l_do.
p4 = l_do.

ENDFORM.
```

The contents of **do3** are affected directly by the changes to **p3**.

As for **p2**, we have created a local copy for **p4**. Consequently, the changes to the formal parameter **do not** affect the actual parameter while the subroutine is running.



```
...  
PERFORM demosub USING do1          do2          CHANGING do3          do4.  
...
```

```
FORM      demosub USING p1 .. value(p2) .. CHANGING p3 .. value(p4) ..
```

```
DATA:
```

```
  l_do .. VALUE ..
```

```
p1 = l_do.
```

```
p2 = l_do.
```

```
p3 = l_do.
```

```
p4 = l_do.
```

```
ENDFORM.
```

The changes are not **written back** to the actual parameters until the **ENDFORM** statement.



do1 **v\_new** do2 **old2** do3 **v\_new** do4 **v\_new**

```
...  
PERFORM demosub USING do1          do2          CHANGING do3          do4.  
...
```

```
FORM      demosub USING p1 .. value(p2) .. CHANGING p3 .. value(p4) ..  
  
DATA:  
  l_do .. VALUE ..  
  
p1 = l_do.  
p2 = l_do.  
p3 = l_do.  
p4 = l_do.  
  
ENDFORM.
```

© SAP AG 1999

If **demosub** is interrupted for any reason, **do4** would have the same value afterwards as it had before the call.

Now that **demosub** has finished running, the memory occupied by its local data objects is released. You now cannot address these data objects any more.

Note that **do2** still has its old value, even though **p2** was changed in the subroutine.

## Example: Local Modularization in Programs



```
DATA
  res_seats TYPE saplane-seatsmax.

PARAMETERS:
  pa_type TYPE saplane-planetype,
  pa_occ  TYPE saplane-seatsmax.

PERFORM get_free_seats USING      pa_type
                                pa_occ
                                CHANGING res_seats.

WRITE res_seats.
*****
FORM get_free_seats USING      p_planetype TYPE saplane-planetype
                                p_seatsocc  TYPE saplane-seatsmax
                                CHANGING value(p_seatsfree) TYPE saplane-seatsmax.

  SELECT SINGLE seatsmax FROM saplane
                                INTO p_seatsfree
                                WHERE planetype = p_planetype.

  p_seatsfree = p_seatsfree - p_seatsocc.
ENDFORM.
```

© SAP AG 1999

In the example here, the subroutine should work out the number of free seats on a plane based on the aircraft type and the number of seats already occupied.

The parameters **p\_planetype** and **p\_seatsocc** are passed by reference to the subroutine **get\_free\_seats**. In the interface, **USING** indicates that you can only access them to read them. The result **p\_seatsfree**, on the other hand, is returned by copying its value.

For simplicity, the main program has been restricted to a selection screen on which the user can enter values, the subroutine call itself, and the list display.

It is technically possible to call subroutines from other main programs. However, this technique is obsolete, and you should use function modules instead. Function modules provide considerable advantages, and are important components in the ABAP Workbench. For further information, refer to the unit **Function Groups and Function Modules**.

## Example: Recursive Call I



```
FORM find_conn USING      p_pos      TYPE <citytype>
                        p_dest      TYPE <citytype>
                        CHANGING p_step_list TYPE <step_list_type>.

DATA:
  l_poss_list TYPE <step_list_type>,
  l_wa_poss   TYPE <step_type>.

*** step to p_dest?
READ TABLE conn_list INTO wa_conn
                        WITH TABLE KEY cityfrom = p_pos
                        cityto   = p_dest.

IF sy-subrc = 0.
  ...
ELSE.
  *** all possible next steps where we haven't been yet:
  LOOP AT conn_list INTO wa_conn WHERE cityfrom = p_pos.
    READ TABLE p_step_list WITH KEY cityto = wa_conn-cityto
                        TRANSPORTING NO FIELDS.

    IF sy-subrc <> 0.
      ...
      APPEND l_wa_poss TO l_poss_list.
    ENDIF.
  ENDLOOP.
ENDLOOP.
```

© SAP AG 1999

Another typical use of subroutines is recursive calls. Although all other modularization units can, in principle, be called recursively, the runtime required is often excessive for small easily-programmed recursions.

This example uses a recursive solution to find a connection between two cities. To find a connection between **A** and **Z**, the program looks for a flight from **A** to **B**, and then from **B** to **Z**. The subroutine **find\_conn** calls itself.

- If there is no direct connection, the program uses the current city (**p\_pos**) to compile a list of all cities that can be reached (**l\_poss\_list**), that are not yet in the route list (**p\_step\_list**). The route list is defined as a standard table so that the sequence of the cities on the route is retained.

## Example: Recursive Calls II



```
SORT l_poss_list BY cityto ASCENDING.
DELETE ADJACENT DUPLICATES FROM l_poss_list COMPARING cityto.

*** no good next steps available:
IF l_poss_list IS INITIAL.
  ...
  MODIFY p_step_list FROM wa_step TRANSPORTING no_way_out
    WHERE cityto = p_pos.

ELSE.
*** try to get ahead from the possible next steps:
  LOOP AT l_poss_list INTO l_wa_poss.
    READ TABLE p_step_list WITH KEY cityto      = l_wa_poss-cityto
      no_way_out = 'X'
      TRANSPORTING NO FIELDS.

    IF sy-subrc <> 0.
      ...
      APPEND wa_step TO p_step_list.
      PERFORM find_conn USING      l_wa_poss-cityto
        p_dest
        CHANGING p_step_list.
```

© SAP AG 1999

For simplicity, the system removes duplicate entries from the list of cities. This means that the subroutine ends up with **only one** possible connection.

However, it would also be possible to suppress this, and examine all of the possible connections, for example, for the number of stopovers, total distance, and so on.

- If it is not possible to reach any cities other than those already visited on the same journey, the current city on the route is marked as a "dead end".
- Otherwise, the cities to which it is possible to travel are processed in a loop. Each city is included in the route list, so that the program can continue searching from here. However, before that, the program has to check whether the city has already been marked as a dead end in a previous search.

## Example: Recursive Calls III



```
*** dest reached?
    READ TABLE p_step_list WITH KEY cityto = p_dest
                                TRANSPORTING NO FIELDS.

    IF sy-subrc = 0.
        EXIT.
    ELSE.
        CLEAR wa_step.
        wa_step-no_way_out = 'X'.
        MODIFY p_step_list FROM wa_step TRANSPORTING no_way_out
                                WHERE cityto = l_wa_poss-cityto.
    ENDIF.

    ENDIF.          " no_way_out = ' '

    ENDLOOP.        " get ahead

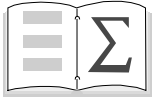
    ENDIF.          " good next steps available

    ENDIF.          " possible next steps where we haven't been yet

    ENDFORM.        " find_conn
```

© SAP AG 1999

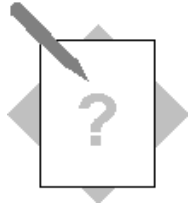
- Once the destination is reached, we can terminate the processing. Any further search from this point would be unsuccessful anyway. The city is marked in the route list, and the search carries on with the next reachable city.
- The processing logic for this subroutine is contained in the function group **LBC402\_SURD\_RECURSION**, include program **LBC402\_SURD\_RECURSIONF01**. The subroutine is called from function module **BC402\_SURD\_TRAVEL\_LIST**, which is itself called from the executable program **SAPBC402\_SURD\_RECURSION**. This program lists all of the possible flights in the flight data mode, and in particular those with stopovers.



**You are now able to:**

- **Define subroutines**
- **Call subroutines**
- **Use the different techniques that exist for passing and typing interface parameters**

## Subroutines: Exercises

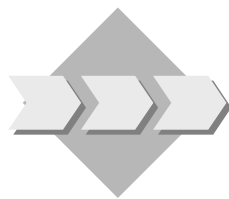


**Unit:** Subroutines  
**Topic:** Interface, functions, and call



At the conclusion of these exercises, you will be able to:

- Implement subroutines
- Call subroutines



You are a programmer for an airline consortium, and it is your job to write analysis programs for several airlines.

1. Extend your program from section 2 of the previous exercises:  
Use a subroutine to encapsulate the code you use to display the flights on the list. Pass the relevant internal table by reference to the subroutine.  
**## is your two-digit group number.**  
Model solution:  
SAPBC402\_SURS\_FLIGHTLIST3
  - 1-1 Copy your solution to exercise 2 from the last unit (or the model solution) to the new program **Z##\_BC402\_FLIGHTLIST3**.
  - 1-2 At the end of the processing logic, define the subroutine **display\_flights**. Declare a parameter for the internal table so that it is passed by reference. Specify an appropriate type for it.
  - 1-3 Remove the field symbol **<fs\_flight>** from the main program and declare it as a local data object in the subroutine.
  - 1-4 Remove the statements used to create the list from the main program and insert them (appropriately modified) in the subroutine.
  - 1-5 From the main program, call the subroutine **display\_flights** (use the *Pattern* function).

## Subroutines: Solutions



Unit: Subroutines  
Topic: Interface, functions, and calling

### 1 Model solution SAPBC402\_SURS\_FLIGHTLIST3

```
*&-----*
*& Report SAPBC402_SURS_FLIGHTLIST3          *
*&                                           *
*&-----*
*& solution of exercise subroutines          *
*&                                           *
*&-----*
```

REPORT sapbc402\_surs\_flightlist3.

TYPES:

```
BEGIN OF t_flight,
  carrid  TYPE spfli-carrid,
  connid  TYPE spfli-connid,
  fldate  TYPE sflight-fldate,
  cityfrom TYPE spfli-cityfrom,
  cityto  TYPE spfli-cityto,
  seatsocc TYPE sflight-seatsocc,
  paymentsum TYPE sflight-paymentsum,
  currency TYPE sflight-currency,
  it_planes TYPE bc402_typs_planetab,
END OF t_flight,

t_flighttab TYPE SORTED TABLE OF t_flight
            WITH UNIQUE KEY carrid connid fldate.
```

DATA:

```
wa_flight TYPE t_flight,
it_flights TYPE t_flighttab.
```

SELECT-OPTIONS so\_carr FOR wa\_flight-carrid.

\* for authority-check:

\*\*\*\*\*

DATA:

```
allowed_carriers TYPE RANGE OF t_flight-carrid,
wa_allowed_carr  LIKE LINE OF allowed_carriers.
```



START-OF-SELECTION.

\* fill a range table with the allowed carriers:

\*\*\*\*\*

```
SELECT carrid
  FROM scarr
  INTO wa_allowed_carr-low
  WHERE carrid IN so_carr.

AUTHORITY-CHECK OBJECT 'S_CARRID'
  ID 'CARRID' FIELD wa_allowed_carr-low
  ID 'ACTVT' FIELD '03'. " display
IF sy-subrc <> 0.
  CLEAR wa_allowed_carr.
ELSE.
  wa_allowed_carr-sign = 'I'.
  wa_allowed_carr-option = 'EQ'.
  APPEND wa_allowed_carr TO allowed_carriers.
ENDIF.
```

ENDSELECT.

\* fill an internal table with connection and flight data

\* for the allowed carriers:

\*\*\*\*\*

```
SELECT carrid connid fldate cityfrom cityto
  seatsocc paymentsum currency
  FROM bc402_flights
  INTO (wa_flight-carrid, wa_flight-connid,
        wa_flight-fldate,
        wa_flight-cityfrom, wa_flight-cityto,
        wa_flight-seatsocc,
        wa_flight-paymentsum, wa_flight-currency)
  WHERE carrid IN allowed_carriers.
```

INSERT wa\_flight INTO TABLE it\_flights.

ENDSELECT.

**PERFORM display\_flights CHANGING it\_flights.**

```

*-----*
*   FORM display_flights
*-----*
* --> p_it_flights
*-----*
FORM display_flights CHANGING p_it_flights TYPE t_flighttab.

FIELD-SYMBOLS <l_fs_flight> TYPE t_flight.

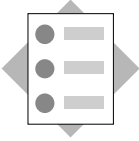
LOOP AT p_it_flights ASSIGNING <l_fs_flight>
    WHERE seatsocc > 0.
    WRITE: / <l_fs_flight>-carrid,
            <l_fs_flight>-connid,
            <l_fs_flight>-fldate,
            <l_fs_flight>-cityfrom,
            <l_fs_flight>-cityto,
            <l_fs_flight>-seatsocc,
            <l_fs_flight>-paymentsum
            CURRENCY <l_fs_flight>-currency,
            <l_fs_flight>-currency.
    SKIP.
ENDLOOP.

ENDFORM.

```

## **Contents:**

- **Defining the interface**
- **Function modules in function groups**
- **Calling a function module**
- **Runtime behavior**



**At the conclusion of this unit, you will be able to:**

- **Create function groups**
- **Create function modules**
- **Call function modules**
- **Handle the exceptions raised in function modules**

Preface

Course Overview

ABAP Runtime Environment

Data Types and Data Objects

Statements

Internal Table Operations

Subroutines

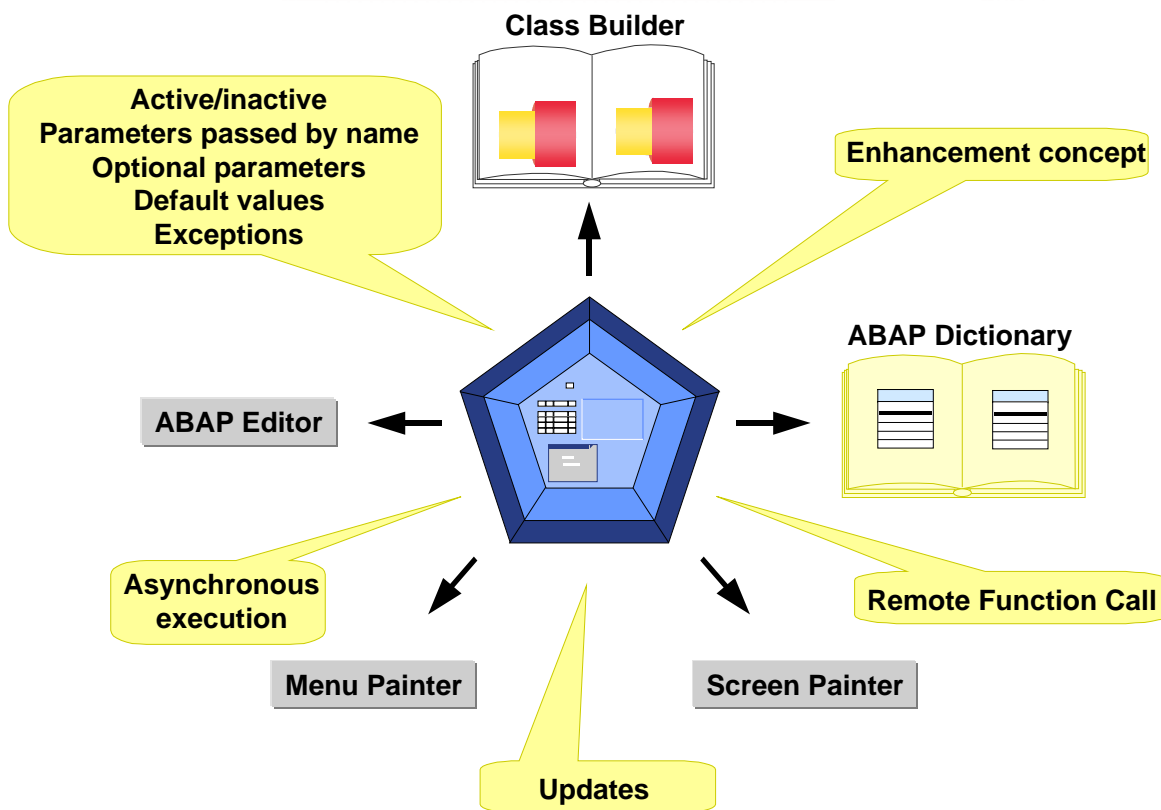


**Function Groups and Function Modules**

Introduction to ABAP Objects

Calling Programs and Passing Data

© SAP AG 1999



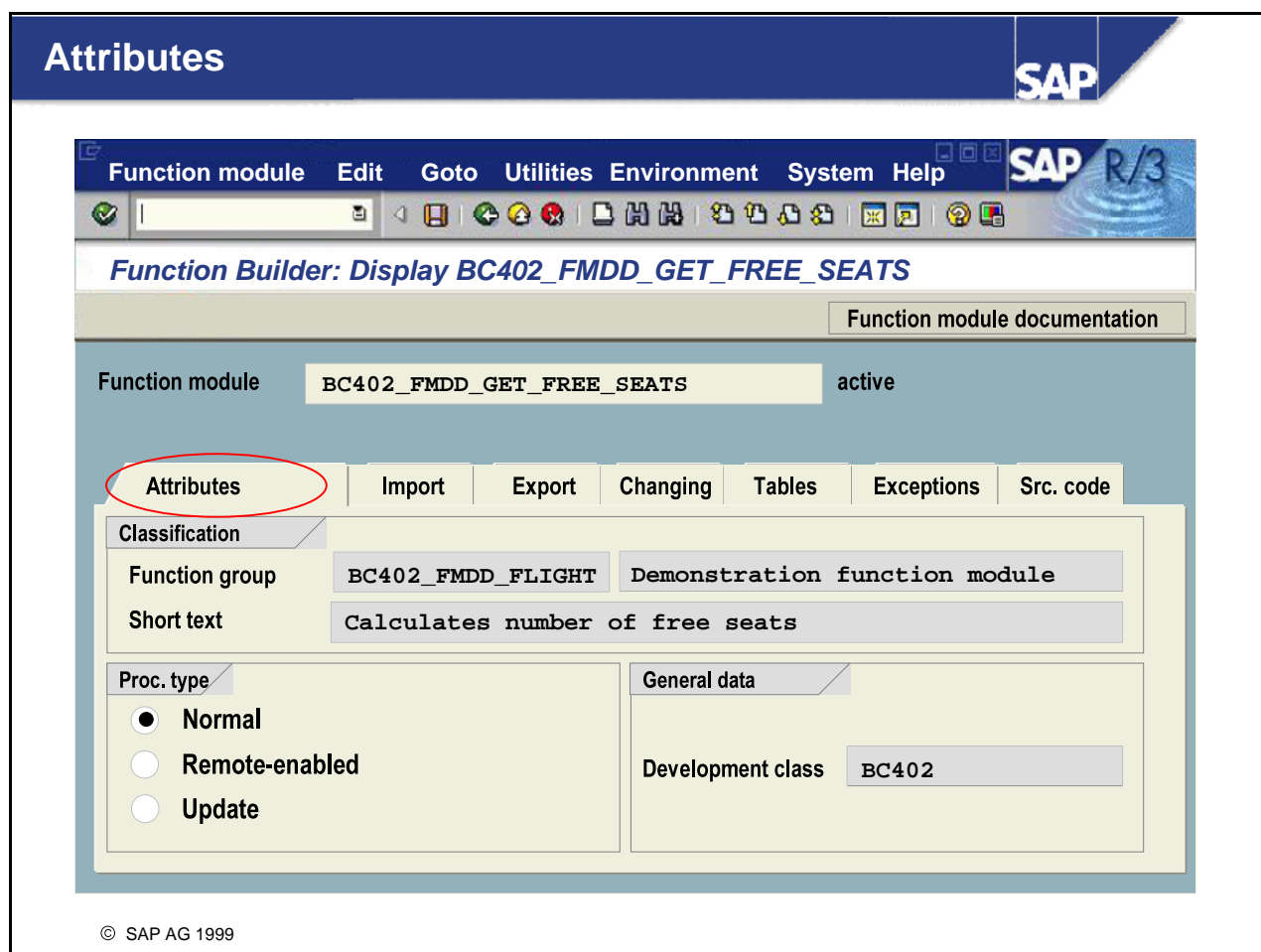
© SAP AG 1999

Function modules are more comfortable to use than subroutines, and have a wider range of uses. The following list, without claiming to be complete, details the essential role that function modules play in the ABAP Workbench:

## Function modules ...

- Are **actively integrated in the ABAP Workbench**. You create and manage them using the *Function Builder*.
- Can have optional importing and changing parameters, to which you can assign **default values**.
- Can **trigger exceptions** through which the return code field **sy-subrc** is filled.
- Can be **remote-enabled**.
- Can be executed **asynchronously**, allowing you to run **parallel** processes.
- Can be enabled for **updates**.
- Play an important role in the **SAP enhancement concept**.

As an example, we are going to calculate the number of free seats on an aircraft. The following slides illustrate the individual steps involved in creating a function module.



In the **Attributes** of a function module, you specify its general administrative data and the **processing type**:

- **Remote-enabled function modules** can be called asynchronously in the same system, and can also be called from other systems (not just R/3 Systems). To call a function module in another system, there must be a valid system connection. For further information, refer to the course **BC415 (Communications Interfaces in ABAP)**.
- **Update function modules** contain additional functions for bundling database changes. For further information, refer to the course **BC414 (Programming Database Updates)** and the online documentation.

The online documentation also details the interface restrictions that apply to remote-enabled and update function modules.

Function module documentation

Function module **BC402\_FMDD\_GET\_FREE\_SEATS** active

Attributes **Import** Export Changing Tables Exceptions Src. code

Parameter name	Typing	Reference type	Default value	Optional	Pass by value	Short text
IP_PLANETYPE	TYPE	S_PLANETYPE				Plane type
IP_SEATSOCC	TYPE	S_SEATSMAX	0	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Curr. occ.

Attributes Import **Export** Changing Tables Exceptions Src. code

Parameter name	Typing	Reference type	Pass by value	Short text
EP_SEATSFREE	TYPE	S_SEATSMAX	<input checked="" type="checkbox"/>	Free seats

© SAP AG 1999

When you exchange data with function modules, you can distinguish clearly between three kinds of parameters:

- **Importing** parameters, which are **received** by the function module
- **Exporting** parameters, which are **returned** by the function module
- **Changing** parameters, which are both **received** and **returned**.

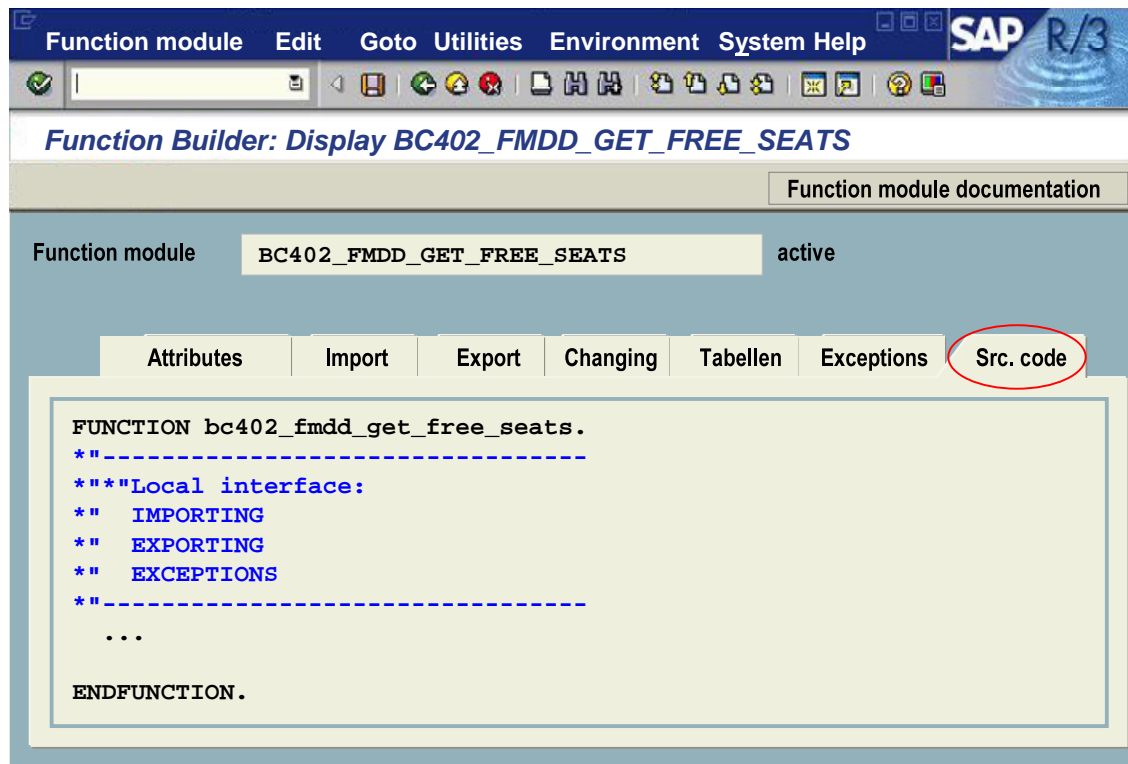
By default, all parameters are passed by **reference**. To avoid unwanted side effects, you can only change exporting and changing parameters in the function module. If you want to pass parameters by **value**, you must select the relevant option when you define the interface.

You can also declare **importing** and **changing** parameters as **optional**. You do not have to pass values to these parameters when you call the function module. Where possible, you should use this option when you **add** new parameters to function modules that are already in use. You can assign a **default value** to an optional parameter. If you do not pass a value of your own when you call the function module, the system then uses the default value instead. **Export** parameters are **always** optional.

You **may** specify the type of an elementary parameter. You **must** specify the type of a structured or table parameter. You can use either ABAP Dictionary types, ABAP Dictionary objects, predefined ABAP types (**I, F, P, N, C, STRING, X, XSTRING, D, T**) or user-defined types. Any type conflicts show up in the **extended program check**.

Tables parameters are obsolete for normal function modules, but have been retained to ensure compatibility for function modules with other execution modes.





© SAP AG 1999

When you save the interface, the system generates the statement framework together with the comment block that lists the interface parameters:

**FUNCTION** <name>.

```

*"-----
*"
*"

```

```

...

```

**ENDFUNCTION.**

The comment block is updated automatically if you make changes to the function module later on. It means that you can always see the interface definition when you are coding the function module.

You program the statements exactly as you would in any other ABAP program in the ABAP Editor.

In the function module, you can create your own local types and data objects, and call subroutines or other function modules.

Function module documentation

Function module: **BC402\_FMDD\_GET\_FREE\_SEATS** active

Attributes	Import	Export	Changing	Tables	Exceptions	Src. code
Exceptions					Short text	
NO_SEATS					Cargo plane	
OVERLOAD					Overbooked	
DB_FAILURE					No data	

```

RAISE <exception>.

*** with default message:
MESSAGE <kind><num>(<id>)
      RAISING <exception>.
    
```

© SAP AG 1999

You can make a function module **trigger exceptions**.

To do this, you must first **declare** the exceptions in the interface definition, that is, assign each one a different **name**.

In the source code of your function module, you program the statements that **trigger** an exception under the required condition. At runtime, the function module is terminated when an exception is triggered. The changes to exporting and changing parameters are the same as in subroutines. There are two statements that you can use to trigger an exception. In the forms given below, <exception> stands for the **name** of an exception that you declared in the interface. The system reacts differently according to whether or not the exception was listed in the function module call:

■ **RAISE <exception>.**

If the exception is listed in the calling program, the system returns control to it directly. If the exception is not listed, a **runtime error** occurs.

■ **MESSAGE <kind><num>(<id>) RAISING <exception>.**

If the exception is listed in the calling program, the statement has the same effect as RAISE <exception>. If it is not listed, the system sends **message** <num> from message class <id> with type <kind>, and **no** runtime error occurs.

- **Function module documentation**

- **Short and long texts**

- ◆ **Parameters**

- ◆ **Exceptions**

- **Functions, notes, and so on**



- **Work list**

- **Revised version**

- **Inactive version**

- **Active version**



- **Testing and debugging**

- **Parameter values**

- **Exceptions**

- **Messages**



© SAP AG 1999

Function modules differ from subroutines in that you must assume that they will be used by other programmers. For this reason, you should ensure that you complete the steps listed here.

- **Documentation (can be translated)**

You should document both your parameters and exceptions with short texts (and long texts if necessary) and your entire function module. The system provides a text editor for you to do this, containing predefined sections for **Functionality**, **Example Call**, **Hints**, and **Further Information**.

- **Work list**

When you change an active function module, it acquires the status **active (revised)**. When you save it, another version is created with the status **inactive**. When you are working on a function module, you can switch between the inactive version and the last version that you activated. When you activate the inactive version, the previous active version is overwritten.

- **Function test**

Once you have activated your function module, you can test it using the built-in test environment in the Function Builder. If an exception is triggered, the test environment displays it, along with any message that you may have specified for it. You can also switch into the **Debugger** and the **Runtime Analysis** tool. You can save test data and compare sets of results.

```

DATA: result TYPE s_seatmax.
PARAMETERS: pa_type TYPE s_planetype, pa_occ TYPE s_seatmax.

CALL FUNCTION 'BC402_FMDD_GET_FREE_SEATS'
  EXPORTING
    ip_planetype = pa_type
    ip_seatsocc  = pa_occ      " default: 0
  IMPORTING
    ep_seatfree  = result
  EXCEPTIONS
    no_seats     = 1
    overload     = 2
    OTHERS       = 3.
CASE sy-subrc.
  WHEN 0.
    WRITE: / result COLOR 5.
  WHEN 1.
    WRITE: / 'You'll have to stand, it's a freighter!'(ftr).
  WHEN 2.
    WRITE: / 'The plane has already been overloaded!'(nos).
  WHEN 3.
    WRITE: / 'Please contact your system administrator!'(adm).
ENDCASE.
    
```

The dialog box titled "Insert pattern" contains a list of radio buttons. The first option, "CALL FUNCTION", is selected and has the text "BC402\_FMDD\_GET\_FREE\_SEATS" entered next to it. Below this are several empty rows with radio buttons. At the bottom of the dialog are two buttons: a green checkmark and a red X.

© SAP AG 1999

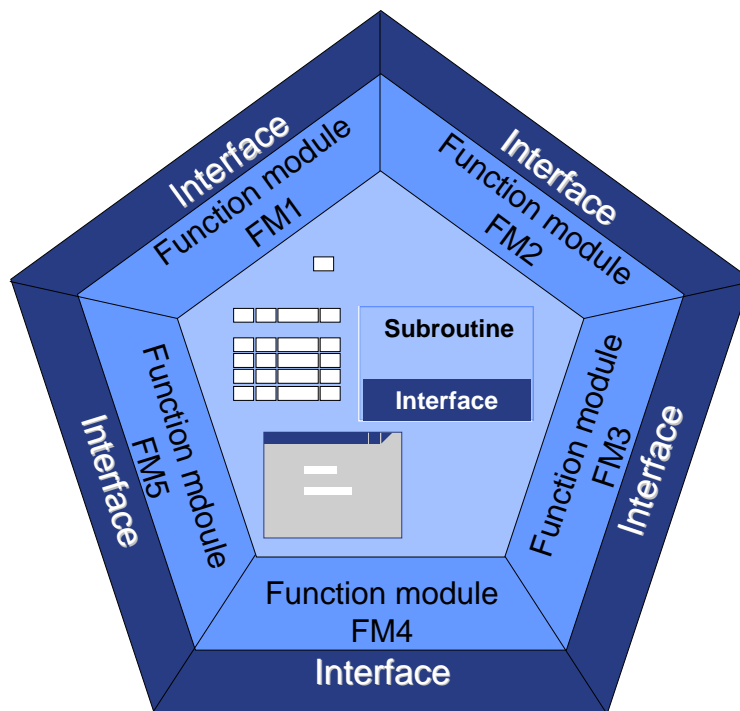
When you insert a function module call in your program, you should use the **Pattern** function. Then, you only need to enter the name of the function module (input help is available). The system then inserts the call and the exception handling (**MESSAGE** statement) into your program.

You assign parameters by name. The **formal** parameters are always on the **left-hand** side of the expressions:

- **Exporting** parameters are **passed** by the program. If a parameter is optional, you do not need to pass it. Default values are displayed if they exist.
- **Importing** parameters are **received** by the program. All importing parameters are optional.
- **Changing** parameters are both **passed** and **received**. You do not have to list optional parameters. Default values are displayed if they exist.

The system assigns a value to each exception, beginning at one, and continuing to number them sequentially in the order they are declared in the function module definition. You can assign a value to all other exceptions that you have not specifically listed using the special exception **OTHERS**.

If you list the exceptions and one is triggered in the function module, the corresponding value is placed in the return code field **sy-subrc**. If you did not list the exception in the function call, a runtime error or a message occurs, depending on the statement you used **in the function module** to trigger the exception.



© SAP AG 1999

When you create a function module, you must assign it to a **function group**. The function group is the **main program** in which a function module is embedded.

A function group is a program with type F, and is **not executable**. The entire function group is loaded in a program the first time that you call a function module that belongs to it.

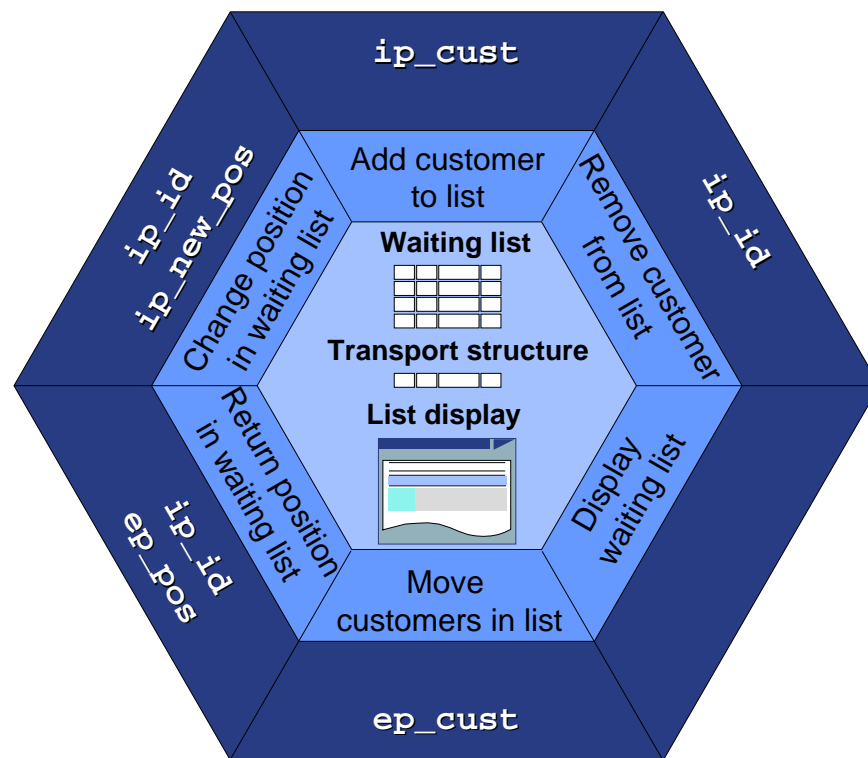
The system also triggers the **LOAD-OF-PROGRAM** event for the function group.

The function group remains active in the background until the end of the calling program. It is therefore a suitable means of retaining data objects **for the entire duration of a program**. All of the function modules in a group can access the group's global data.

The same applies to screens. If you want to call one screen from several different programs, you must create it in a function group. You then create the ABAP data objects with the same names as the screen fields in the function group. The screen and data transport can now be controlled using the function modules in the group.

**Examples:** Function groups **SPO1** to **SPO6**.

For further information about this technique, refer to course **BC410 (Programming User Dialogs)**.



© SAP AG 1999

Let us return to our waiting list example from the **Operations on Internal Tables** unit. Maintaining a waiting list using subroutines would be subject to errors, since the list would be a global object and could be changed within the main program.

Furthermore, waiting lists are a common application, and it is likely that if you write a solution, it can be used by other developers. You should therefore make it available centrally in the ABAP Workbench so that other programmers do not have to do the same work over again. If, for example, they know that a function module **wait\_get\_first** will return the name of the customer at the top of the waiting list, they only need to worry about the required parameters and possible exceptions.

In the example, the waiting list is implemented as an internal table in the global data declarations of the function group. This means that it cannot be changed using any means other than the function modules in that group. You can call these from any program.

```
READ TABLE wait_list FROM ip_cust TRANSPORTING NO FIELDS.  
IF sy-subrc <> 0.  
  APPEND ip_cust TO wait_list.  
ELSE. MESSAGE e202(bc402) RAISING in_list.  
ENDIF.
```

Add waiting customer

```
DELETE wait_list WHERE id = ip_id.  
IF sy-subrc <> 0.  
  MESSAGE e203(bc402) RAISING not_in_list.  
ENDIF.
```

Remove waiting customer

```
READ TABLE wait_list INTO ep_cust INDEX 1.  
IF sy-subrc = 0.  
  DELETE wait_list INDEX 1.  
ELSE. MESSAGE e200(bc402) RAISING list_empty.  
ENDIF.
```

Move customers up list

```
READ TABLE wait_list WITH KEY id = ip_id TRANSPORTING NO FIELDS.  
IF sy-subrc = 0.  
  ep_pos = sy-tabix.  
ELSE. MESSAGE e200(bc402) RAISING not_in_list.  
ENDIF.
```

Return position in list

© SAP AG 1999

The implementation of the individual function modules is similar to the examples in the **Internal Table Operations** unit. To save space, we have only listed the ABAP coding that is relevant for the actual functions.

The types of the parameters and global data objects have been specified by referring to appropriate types in the ABAP Dictionary.

```
IF wait_list IS INITIAL.  
  MESSAGE e200(bc402) RAISING list_empty.  
ELSE.  
  CALL SCREEN 100 STARTING AT 5 5  
    ENDING AT 120 25.  
ENDIF.
```

Display waiting list

```
MODULE display_list_0100 OUTPUT.  
  SUPPRESS DIALOG.  
  LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 0.  
  
  LOOP AT wait_list INTO wa_cust.  
    WRITE: / sy-tabix,  
      wa_cust-id COLOR COL_KEY,  
      wa_cust-name,  
      wa_cust-city,  
      wa_cust-app_date.  
  
  ENDLOOP.  
  
ENDMODULE.
```

List in modal  
dialog box

© SAP AG 1999

Screen 100 belongs to the function group. It is a container screen for list processing that is processed invisibly. It allows the user to display the current contents of the waiting list in a modal dialog box. For simplicity, we have not used a table control in the example. Had we done so, it would have been possible to encapsulate the entire navigation in the function group. For further information about using screen objects such as table controls, refer to course **BC410 (Programming User Dialogs)**.



```
DATA last_pos LIKE sy-tabix.

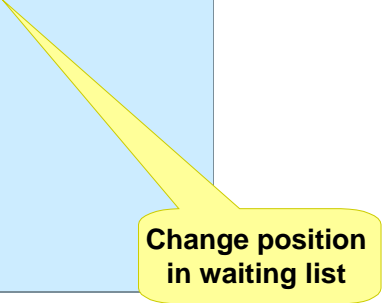
READ TABLE wait_list WITH KEY id = ip_id INTO wa_cust.
IF sy-subrc = 0.

    DELETE wait_list INDEX sy-tabix.
    IF ip_new_pos > 0.

        DESCRIBE TABLE wait_list LINES last_pos.
        IF ip_new_pos > last_pos.
            APPEND wa_cust TO wait_list.
        ELSE.
            INSERT wa_cust INTO wait_list INDEX ip_new_pos.
        ENDIF.

    ELSE.
        INSERT wa_cust INTO wait_list INDEX 1.
    ENDIF.

ELSE.
    MESSAGE e203(bc402) RAISING not_in_list.
ENDIF.
```



Change position  
in waiting list

© SAP AG 1999

To move an entry in the waiting list, we first delete the existing entry. Then, we enter a new one at position **ip\_new\_pos**.

This is only possible if the new index is positive, and not greater than the total number of lines in the list (**last\_pos**). We determine the value of **last\_pos** using the **DESCRIBE TABLE ... LINES** statement. If you specify an index that is too large, the entry is appended to the internal table.

# Organization of a Function Group



Object name	Description
BC402_FMDD_WAITLIST	
Function modules	
Fields	
PBO modules	
Screens	
GUI titles	
Includes	
LBC402_FMDD_WAITLISTU01	Add waiting customer
LBC402_FMDD_WAITLISTU02	Remove waiting customer
LBC402_FMDD_WAITLISTU03	Display waiting list
LBC402_FMDD_WAITLISTU04	Shift waiting list
LBC402_FMDD_WAITLISTU05	Return position in list
LBC402_FMDD_WAITLISTU06	Change position
LBC402_FMDD_WAITLISTUXX	

© SAP AG 1999

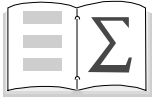
As described in the **ABAP Runtime Environment** unit, the ABAP Workbench helps you to structure your source code when you work with function groups and function modules.

Forward navigation ensures that you always enter the correct object. Include programs are named automatically, and the relevant call statements are inserted automatically in the correct positions. You only have to observe the naming convention for function groups: **{Y|Z}<rem\_name>**.

The system then creates a type F program called **SAPL{Y|Z}<rem\_name>**. This contains automatically-generated **INCLUDE** statements. The include programs are also named automatically:

**L{Y|Z}<rem\_name><abbreviation><number>**. The abbreviation **<abbreviation>** is assigned according to the same principle as described on the **Program Organization** page.

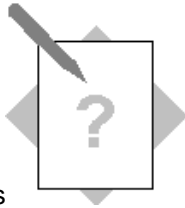
The include program **L{Y|Z}<rem\_name>UXX** is also inserted. This contains an include statement for each function module in the form **L{Y|Z}<rem\_name>U<number>**.



**You are now able to:**

- **Create function groups**
- **Create function modules**
- **Call function modules**
- **Handle the exceptions raised in function modules**

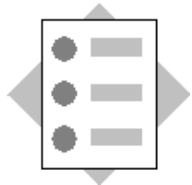
## Function Groups and Function Modules: Exercises



S

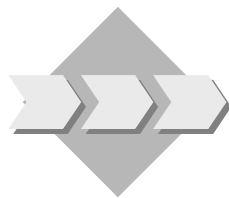
**Unit:** Function Groups and Function Modules

**Topic:** Creating and calling function groups and function modules



At the conclusion of these exercises, you will be able to:

- Create and implement function groups
- Write function modules
- Call function modules



You are a programmer for an airline consortium, and it is your job to write analysis programs for several airlines.

1. You will create an internal table in a function group that buffers all of the aircraft type available to each airline. For simplicity, this will have a flat structure, not a nested one.

**## is your two-digit group number.**

Model solution:

BC402\_FMDS\_FLIGHT

1-1 Create a function group **Z##\_BC402\_FLIGHT**.

1-2 Assign the message class **BC402** to it.



The program ID is in the TOP include (**LBC402\_FLIGHTTOP**).

1-3 Document your function group.

2. You are going to write a function module to fill internal tables for aircraft types. It should only write a replacement aircraft type into the table for a particular airline if the aircraft type has enough seats. It should also calculate the average revenue per seat, based on the total revenue that you will pass to it. Your function module should then sort the list of aircraft by this value in descending order before returning it to the calling program.

**## is your two-digit group number.**

Model solutions:

BC402\_FMDS\_FLIGHT

BC402\_FMDS\_CREATE\_PLANELIST

- 2-1 Create the function module **Z\_##\_BC402\_CREATE\_PLANELIST** in your function group **Z##\_BC402\_FLIGHT**.

- 2-2 Declare the line type **t\_carr\_plane** as a global data type in your function group. It should have the following structure:

Component	Type
<b>carrid</b>	<b>scarplan-carrid</b>
<b>planetype</b>	<b>scarplan-planetype</b>
<b>seatsmax</b>	<b>saplane-seatsmax</b>

You will use this for the airline↔ aircraft type assignment.

- 2-3 Declare the internal table **it\_carr\_planes** as a global data object in your function group. It should have the line type **t\_carr\_plane**. It should be a sorted table with the unique key **carrid** and **planetype**.

- 2-4 Fill the internal table **it\_carr\_planes** using the “array fetch” method with the view **BC402\_CARPLAN**.



Choose a suitable event. Remember that function groups cannot be executed directly. Implement the event block in a suitable include program. Observe the naming convention for include programs in function groups.

- 2-5 Declare the following import parameters for your function module. They should be passed by value.

**ip\_seatsocc** (optional, with default value 0), **ip\_carrid**, **ip\_paymentsum**, **ip\_currency**.

- 2-6 Declare the export parameter **ep\_planelist**. It should be passed by value. Specify its type by referring to your global table type **Z##\_BC402\_PLANETAB**.

- 2-7 Declare and document the exception **no\_planes**.

- 2-8 In the function module, create the local structure **l\_wa\_carr\_plane** with type **t\_carr\_plane**.

- 2-9 From the global internal table, read the aircraft types that are available to the airline that you have passed to the function module, and that have enough seats to accommodate the number of passengers booked on the flight.  
In this loop, calculate the average revenue per seat for each aircraft type. Declare another work area – **I\_wa\_plane** – as a local data object within the function module. Specify its type by referring to your global structure **Z##\_BC402\_PLANE**.  
Once you have **filled the structure fully**, pass it to the internal table that you are going to export.
- 2-10 Before you export the table, sort it by the average revenue per seat.
- 2-11 If there are no suitable aircraft types, trigger the exception. When you raise the exception, use error message **067**. You need to pass the airline to the message.
- 2-12 Document your function module.
- 2-13 Test your function module.

3. Extend your program from task 1 of the previous exercises:  
You should now fill the inner internal table for the aircraft types for each flight, using the function module you created in the last exercise.  
Model solution:  
SAPBC402\_FMDS\_FLIGHTLIST4

- 3-1 Copy your solution from the last exercise, or the model solution.  
New name: **Z##\_BC402\_FLIGHTLIST4**.
- 3-2 Fill the inner internal table in its own step before the list display.  
For each flight on which at least one seat is booked, call your function module **Z##\_BC402\_CREATE\_PLANELIST** (use the *Pattern* function).  
The current line should only be updated if no exception is triggered by the function module.
- 3-3 Extend the subroutine in which you display the list:  
If (and only if) there is at least one replacement aircraft for a flight, display all of the aircraft types, their maximum number of seats, and their average revenues (along with the appropriate currency) on the list.  
To do this, use an appropriately-typed field symbol.  
If there is no replacement aircraft for a particular flight, display an appropriate text.

## Function Groups and Function Modules: Solutions



**Unit:** Function Groups and Function Modules

**Topic:** Creating and calling function groups and function modules

1-1, 2-4

### Model solution SAPLBC402\_FMDS\_FLIGHT

```
*****
* System-defined Include-files.                                *
*****
INCLUDE lbc402_fmds_flighttop.    " Global Data
INCLUDE lbc402_fmds_flightuxx.    " Function Modules

*****
* User-defined Include-files (if necessary).                    *
*****
* INCLUDE LBC402_FMDS_FLIGHTF...    " Subprograms
* INCLUDE LBC402_FMDS_FLIGHTO...    " PBO-Modules
* INCLUDE LBC402_FMDS_FLIGHTI...    " PAI-Modules

INCLUDE lbc402_fmds_flighte01.    " Events
```

1-2, 2-2, 2-3

### Model solution LBC402\_FMDS\_FLIGHTTOP

FUNCTION-POOL bc402\_fmds\_flight MESSAGE-ID bc402.

**TYPES:**

```
BEGIN OF t_carr_plane,
  carrid TYPE scarplan-carrid,
  planetype TYPE scarplan-planetyp,
  seatsmax TYPE saplane-seatsmax,
END OF t_carr_plane.
```

**DATA:**

```
it_carr_planes TYPE SORTED TABLE OF t_carr_plane
  WITH UNIQUE KEY carrid planetype.
```



```

*-----*
* INCLUDE LBC402_FMDS_FLIGHTTE01          *
*-----*

```

**LOAD-OF-PROGRAM.**

```

SELECT carrid planetype seatsmax
  FROM bc402_carplan
  INTO CORRESPONDING FIELDS OF TABLE it_carr_planes.

```

**FUNCTION BC402\_FMDS\_CREATE\_PLANELIST.**

```

*"-----*
*"Local interface:
*" IMPORTING
*"   VALUE(IP_SEATSOCC) TYPE SFLIGHT-SEATSOCC DEFAULT 0
*"   VALUE(IP_CARRID) TYPE SPFLI-CARRID
*"   VALUE(IP_PAYMENTSUM) TYPE SFLIGHT-PAYMENTSUM
*"   VALUE(IP_CURRENCY) TYPE SFLIGHT-CURRENCY
*" EXPORTING
*"   VALUE(EP_PLANELIST) TYPE BC402_TYPS_PLANETAB
*" EXCEPTIONS
*"   NO_PLANES
*"-----*

```

**DATA:**

```

  l_wa_carr_plane TYPE t_carr_plane,
  l_wa_plane      TYPE bc402_typs_plane.

```

```

LOOP AT it_carr_planes INTO l_wa_carr_plane
  WHERE carrid EQ ip_carrid
  AND seatsmax GE ip_seatsocc.
  l_wa_plane-planetype = l_wa_carr_plane-planetype.
  l_wa_plane-seatsmax = l_wa_carr_plane-seatsmax.
  l_wa_plane-avg_price =
    ip_paymentsum / l_wa_carr_plane-seatsmax.
  l_wa_plane-currency = ip_currency.
  APPEND l_wa_plane TO ep_planelist.
ENDLOOP.

```

```

IF sy-subrc NE 0.
  MESSAGE e067 RAISING no_planes WITH ip_carrid.
ELSE.
  SORT ep_planelist BY avg_price DESCENDING.
ENDIF.

```

```

ENDFUNCTION.

```

### 3      **Model solution SAPBC402\_FMDs\_FLIGHTLIST4**

```
*&-----*
*& Report  SAPBC402_FMDs_FLIGHTLIST4                *
*&                                     *
*&-----*
*& solution of exercise 3 function groups            *
*&           and function modules                    *
*&-----*
```

REPORT sapbc402\_fmfs\_flightlist4.

TYPES:

```
BEGIN OF t_flight,
  carrid  TYPE sflight-carrid,
  connid  TYPE sflight-connid,
  fldate  TYPE sflight-fldate,
  cityfrom TYPE spfli-cityfrom,
  cityto  TYPE spfli-cityto,
  seatsocc TYPE sflight-seatsocc,
  paymentsum TYPE sflight-paymentsum,
  currency TYPE sflight-currency,
  it_planes TYPE bc402_typs_planetab,
END OF t_flight,
```

```
t_flighttab TYPE SORTED TABLE OF t_flight
             WITH UNIQUE KEY carrid connid fldate.
```

DATA:

```
wa_flight TYPE t_flight,
it_flights TYPE t_flighttab.
```

SELECT-OPTIONS so\_carr FOR wa\_flight-carrid.

\* for authority-check:

\*\*\*\*\*

DATA:

```
allowed_carriers TYPE RANGE OF t_flight-carrid,
wa_allowed_carr  LIKE LINE OF allowed_carriers.
```

AT SELECTION-SCREEN.

...

\* fill a range table with the allowed carriers:

\*\*\*\*\*

...

...

START-OF-SELECTION.

\* fill an internal table with connection and flight data

\* for the allowed carriers:

\*\*\*\*\*

...

...

\* fill all the inner internal tables with alternate planetypes:

\*\*\*\*\*

**LOOP AT it\_flights INTO wa\_flight WHERE seatsocc > 0.**

**CALL FUNCTION 'BC402\_FMDS\_CREATE\_PLANELIST'**

**EXPORTING**

**ip\_seatsocc = wa\_flight-seatsocc**

**ip\_carrid = wa\_flight-carrid**

**ip\_paymentsum = wa\_flight-paymentsum**

**ip\_currency = wa\_flight-currency**

**IMPORTING**

**ep\_planelist = wa\_flight-it\_planes**

**EXCEPTIONS**

**no\_planes = 1**

**OTHERS = 2.**

**IF sy-subrc = 0.**

**MODIFY TABLE it\_flights FROM wa\_flight**  
**TRANSPORTING it\_planes.**

**ENDIF.**

**ENDLOOP.**

**PERFORM display\_flights CHANGING it\_flights.**

```

*-----*
*   FORM display_flights
*-----*
* --> p_it_flights
*-----*
FORM display_flights CHANGING p_it_flights TYPE t_flighttab.

```

FIELD-SYMBOLS:

```

<l_fs_flight> TYPE t_flight,
<l_fs_plane> TYPE bc402_typs_plane.

```

```

LOOP AT p_it_flights ASSIGNING <l_fs_flight>
    WHERE seatsocc > 0.
WRITE: / <l_fs_flight>-carrid,
    <l_fs_flight>-connid,
    <l_fs_flight>-fldate,
    <l_fs_flight>-cityfrom,
    <l_fs_flight>-cityto,
    <l_fs_flight>-seatsocc,
    <l_fs_flight>-paymentsum
        CURRENCY <l_fs_flight>-currency,
    <l_fs_flight>-currency.

```

\* \* display filled inner internal tables only:

\*\*\*\*\*

```

IF <l_fs_flight>-it_planes IS INITIAL.
    WRITE: /29 'no alternate planes available'(npa).
ELSE.
    LOOP AT <l_fs_flight>-it_planes ASSIGNING <l_fs_plane>.
        WRITE: /29 <l_fs_plane>-planetyp,
            <l_fs_plane>-seatsmax,
            <l_fs_plane>-avg_price
                CURRENCY <l_fs_plane>-currency,
            <l_fs_plane>-currency.
    ENDLOOP.
ENDIF.

```

```

SKIP.
ENDLOOP.

```

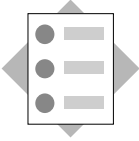
```

ENDFORM.

```

## **Contents:**

- **Declaring and implementing local classes**
- **Creating objects**
- **Accessing components of objects**
- **Defining local interfaces**
- **Triggering and handling events**
- **Position in the ABAP Workbench**



**At the conclusion of this unit, you will be able to:**

- **Declare and implement local classes**
- **Create objects**
- **Access object components**
- **Define local interfaces**
- **Access interface components**
- **Trigger and handle exceptions**

Preface

Course Overview

ABAP Runtime Environment

Data Types and Data Objects

Statements

Internal Table Operations

Subroutines

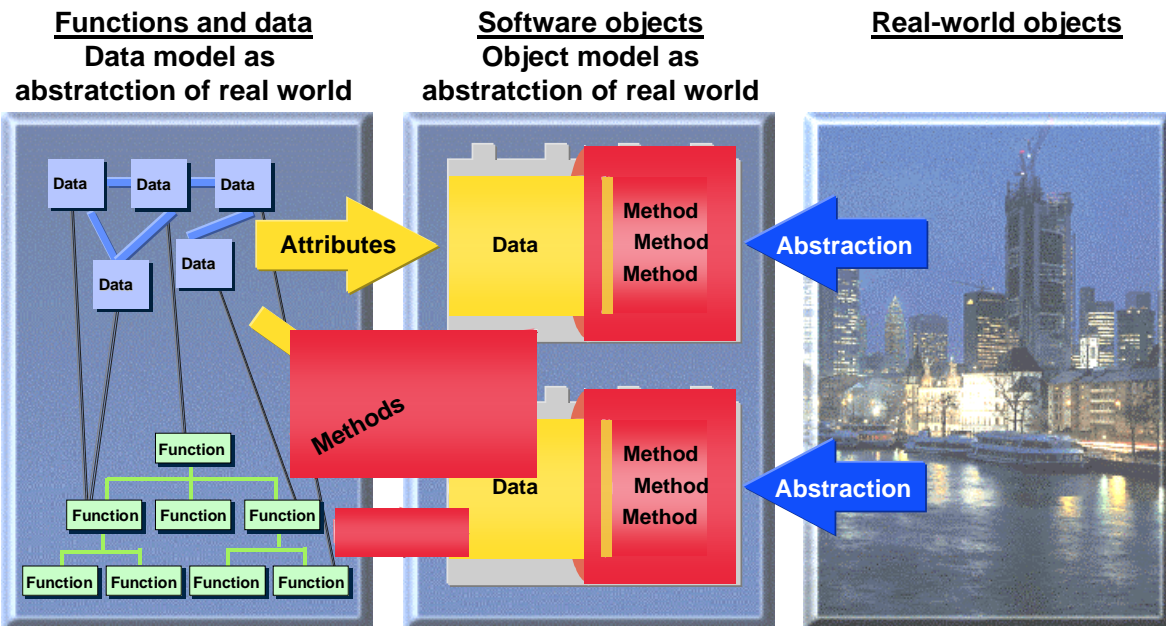
Function Groups and Function Modules



**Introduction to ABAP Objects**

Calling Programs and Passing Data

© SAP AG 1999



© SAP AG 1999

In **functional** programming, functions and data were always kept apart, and linked using input/output relationships.

At the center of **object-oriented** methods are objects. Objects represent abstract or concrete aspects of the real world. You describe them by their characteristics and behavior, and these determine their inner structure and attributes. The behavior of an object is characterized by its methods.

Objects form a capsule, containing attributes and behavior. They should make it possible for the model for a problem and its solution to map onto each other directly. In object-oriented programming, modeling is considerably more important than in function programming. However, once you have **finished** modeling, you can write the program code very quickly. Furthermore, the coding is often so self-explanatory that you do not have to use many comments.

In this unit, we will continue to use our waiting list example. In the function group, the internal table is stored globally, that is, separately from the function modules. The function modules manipulate the data in the table when we call them externally.

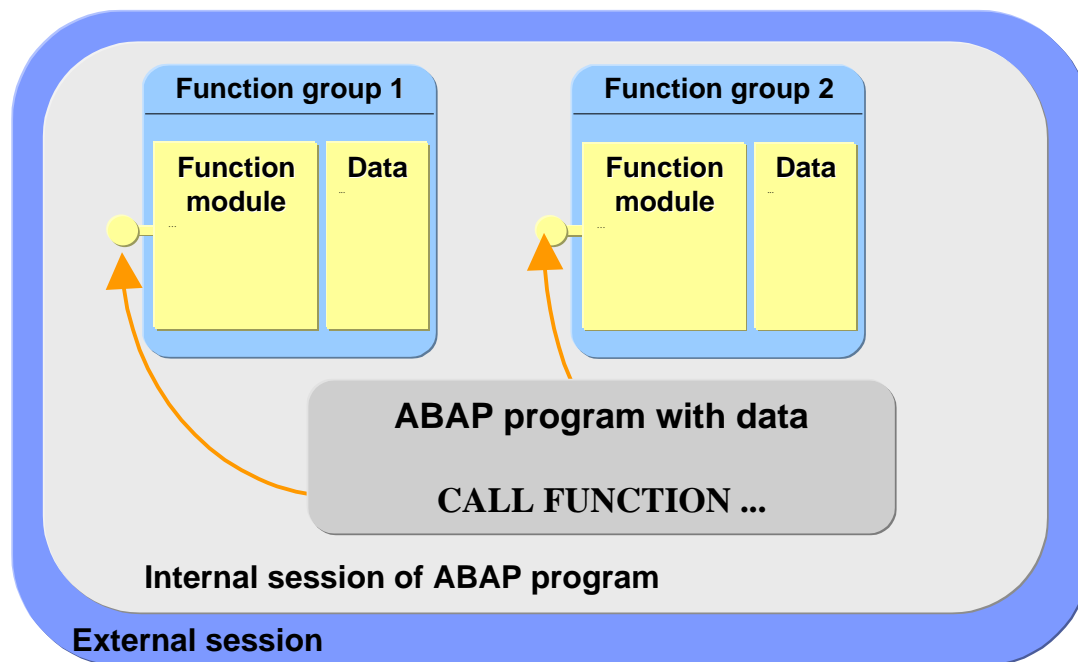
In the object-oriented view, the waiting list has a central role as an **object**, that is, it contains not only the data about waiting customers, but also methods to manipulate that data, which allow it to **manage itself**. Externally, the object is encapsulated - we are not going to call a function module that displays data, but are going to ask the waiting list to **display itself**.



- **ABAP Objects is an upwardly-compatible extension of the existing ABAP language**
- **You can use existing ABAP statements within ABAP Objects**
- **You can use ABAP Objects in existing ABAP programs**
- **ABAP Objects is fully integrated in the ABAP Debugger**

© SAP AG 1999

- Object-orientation in ABAP is an upwards-compatible extension of the existing ABAP language. It provides ABAP programmers with the advantages of object-orientation, such as encapsulation, interfaces, and inheritance, making complex applications easier to control and simplify.
- You can use all conventional ABAP statements and modularization units in ABAP Objects, and, conversely, ABAP Objects can also be used in existing ABAP programs.
- Within ABAP Objects, there are certain syntactic restrictions. For example, you can now only refer to ABAP Dictionary types using the **TYPE** addition, you must specify the types of interface parameters, and the names of components in classes (attributes, methods, and events), may only consist of the characters "A-Z", "0-9" and the underscore character. They may not begin with a digit. To ensure compatibility with earlier releases, the old forms cannot be forbidden outside ABAP Objects. This course has been designed to be fully compatible with ABAP Objects, so **all** forms of all statements used in the course are valid **in the ABAP Objects context** as well.



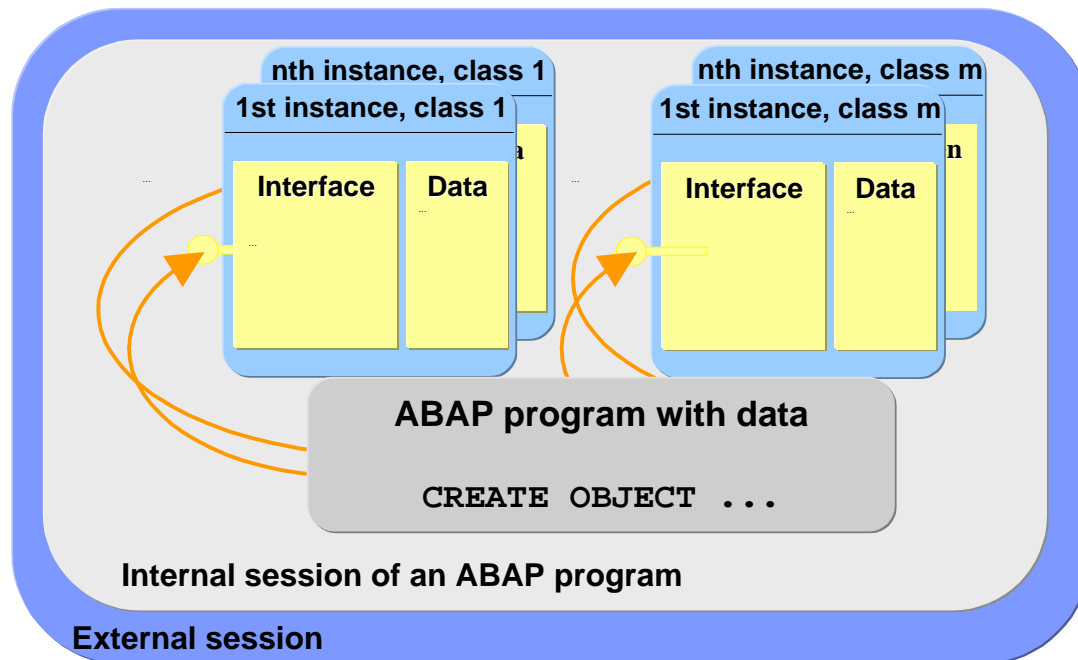
© SAP AG 1999

Before Release 4.0, the ABAP components that most closely resembled objects were function modules and function groups.

When a function module is called, an **instance** of its function group is loaded into the memory area of the internal session. If a single ABAP program calls function modules from more than one function group, it will load more than one instance.

The principal difference between object-orientation and function modules is that **one program** can work with instances of **several** function groups simultaneously, but not with several instances of **a single** function group.

Our waiting list administration in its current form can only create a single instance, or **one** waiting list. It would be very difficult to administer **several** waiting lists at runtime. It would be possible to create new internal tables dynamically, but they would all be part of the same **instance**, namely our function group.

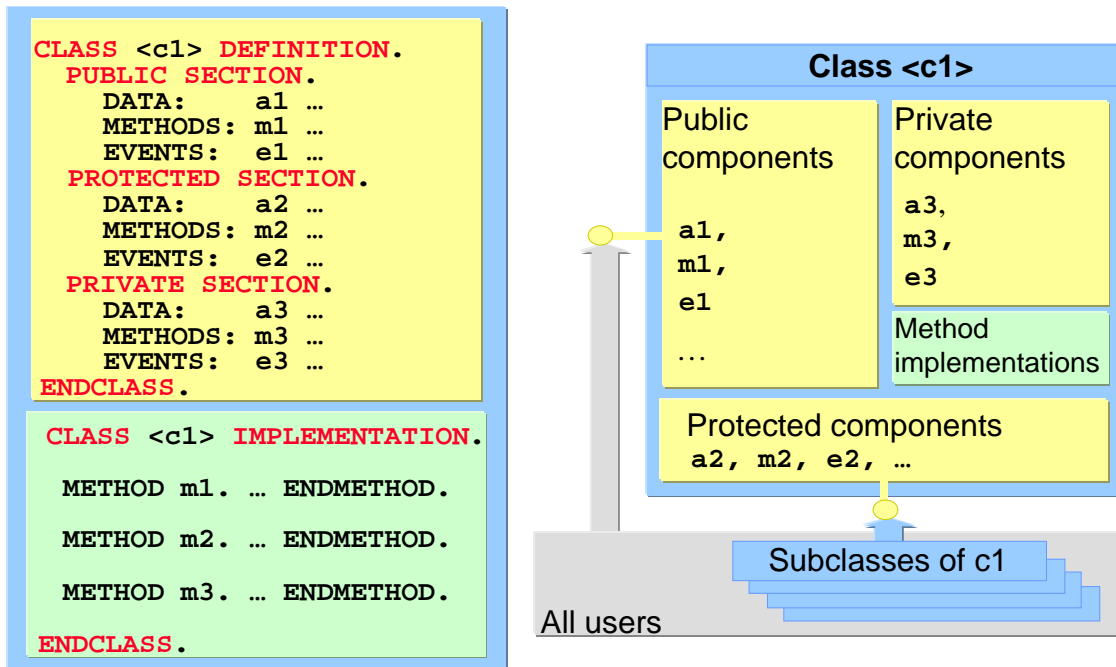


© SAP AG 1999

These problems have been solved with the introduction of **classes**. Now, it is possible to define data and functions in a **class** instead of a function group, and an ABAP program can work with any number of runtime instances of the **same** class. (Of course, you can also create more than one class in a single ABAP program.)

Instead of loading a single runtime instance of a function group implicitly when you call a function module, an ABAP program generates runtime instances of classes explicitly. Each runtime instance is a uniquely-identifiable object that you address using an object reference.

In our example, this means that the waiting list administration can create and delete any number of instances of the waiting list class. All of these instances are separate, but have the same structure. They encapsulate their data. We can specify explicitly in which waiting list we want to include our data by addressing the relevant method of **that** list.



© SAP AG 1999

Classes are structure templates for objects. You can create them **locally** in any ABAP program, or **globally** in the R/3 Repository using the **Class Builder**.

A class **definition** consists of a **declaration** part and an **implementation** part.

The **components** of a class define the attributes and behavior of its instances. Each component must be **declared** in one of the three visibility sections. The visibility sections define the external point of contact between other objects and the class :

- **PUBLIC** - All public components can be addressed both within the class and by all users. They form the external interface of the class.
- **PROTECTED** - All protected components can be addressed both within the class and by all subclasses of the class.
- **PRIVATE** - Private components can only be addressed in the methods of the class itself.
- You must implement all of the methods of the class in its **implementation** part.
- The left-hand side of the graphic shows the declaration and implementation parts of a local class <c1>.
- The right-hand side shows how the class is actually constructed with the components in their visibility sections and the method implementations.

```
CLASS <local_class> DEFINITION.  
  
    PUBLIC SECTION.  
        METHODS constructor  
            [IMPORTING <par> TYPE <type> ... ]  
            [EXCEPTIONS <exc> ... ].  
  
ENDCLASS.
```

```
CLASS <local_class> IMPLEMENTATION.  
  
    METHOD constructor.  
        ...  
    ENDMETHOD.  
  
ENDCLASS.
```

```
PROGRAM ... .  
DATA <cref> TYPE REF TO <local_class>.  
  
CREATE OBJECT <cref> [EXPORTING <par> = <value> ... ].
```

© SAP AG 1999

Constructors (instance or static) are special methods that are called implicitly when you create an object or access a class for the first time. They are executed automatically by the system and are used to set up the initial state of an object (see also **LOAD-OF-PROGRAM** event in function groups).

You use a constructor whenever you want to set the initial state of an object dynamically and using the **VALUE** addition of the **DATA** statement is insufficient.

The method name is always **CONSTRUCTOR**. This is a reserved word.

Instance constructors can have importing parameters and exceptions. You must pass the import parameters in the **CREATE OBJECT** statement.

```

CLASS lcl_waitlist DEFINITION.
  PUBLIC SECTION.
    METHODS constructor IMPORTING im_carrid TYPE sflight-carrid
                                im_connid TYPE sflight-connid
                                im_fldate TYPE sflight-fldate.
    METHODS add IMPORTING im_cust TYPE bc402_typed_cust
                      EXCEPTIONS in_list.

  PRIVATE SECTION.
    DATA:
      carrid TYPE sflight-carrid,
      connid TYPE sflight-connid,
      fldate TYPE sflight-fldate,

      wait_list TYPE bc402_typed_cust_list.
ENDCLASS.
    
```

© SAP AG 1999

As an example, let us look at the local class **lcl\_waitlist** in program **SAPBC402\_AOOD\_WAITLISTS**. It contains:

- An instance constructor, to which you must pass the key data of a flight connection
- A public method **add**, to which you must pass a customer data record.
- The key data for the flight connection, an internal table, and a suitable work area. All of these are defined as private data objects.

```
CLASS lcl_waitlist IMPLEMENTATION.  
  METHOD constructor.  
    carrid = im_carrid.  
    connid = im_connid.  
    fldate = im_fldate.  
  ENDMETHOD.  
  
  METHOD add.  
    READ TABLE wait_list FROM im_cust TRANSPORTING NO FIELDS.  
    IF sy-subrc <> 0.  
      APPEND im_cust TO wait_list.  
    ELSE.  
      MESSAGE e202 RAISING in_list.  
    ENDIF.  
  ENDMETHOD.  
  
ENDCLASS.
```

© SAP AG 1999

- When you create a waiting list object, the constructor writes the key data into the private fields.
- When you call the method **add** for a waiting list object, the system adds the data record to the list in the normal way, as long as it is not already contained in the table.

## Declaring Reference Variables

SAP

DATA

```
o_list1 TYPE REF TO lcl_waitlist.
```

o\_list1 →

© SAP AG 1999

To create an object from a class, you first need a **reference variable**, in our example, **o\_list1**. To declare a reference variable, you use the data type **REF TO <class>**, which was specially introduced for this purpose.

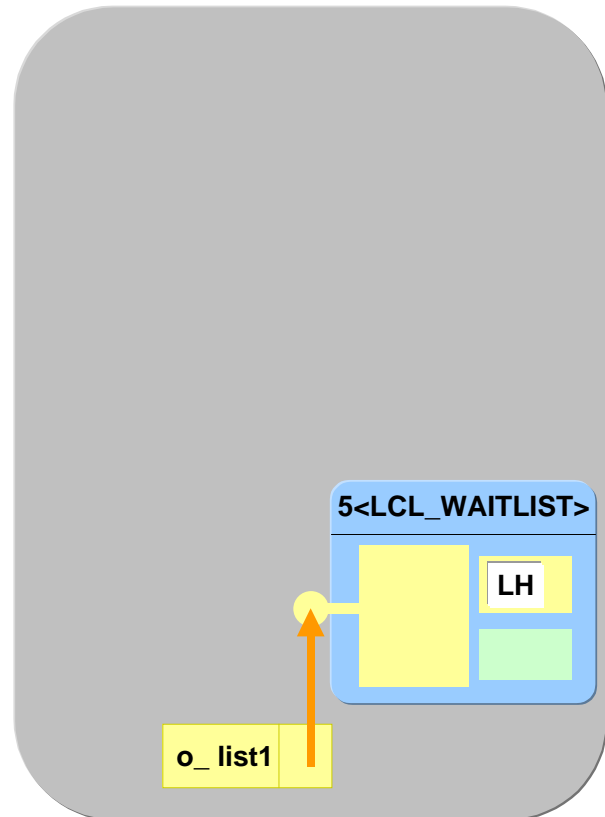
- A reference variable contains an **object reference**, that is, a **pointer** to a data object (see the **Data Types and Data Objects** unit)
- Reference variables use **reference semantics**. When you assign a reference variable to another, only the **address** of the object (the object reference) is passed. After the assignment, the reference points to a different object.
- You can only access objects using object references.
- You can use reference variables as attributes of other objects.



```
DATA
  o_list1 TYPE REF TO lcl_waitlist.

...

CREATE OBJECT o_list1
  EXPORTING im_carrid = 'LH'
           im_connid = '400'
           im_fldate = '19991119'.
```



© SAP AG 1999

An object is an instance of a class. Each instance has a unique identity and its own attributes. All of the instances of a class belong to the context of an internal session (memory area of an ABAP program). You can create any number of instances of the same class.

Once you have defined the reference variable `o_list1` with reference to a class, you can instantiate the class. To do this, use the **CREATE OBJECT** <cref> statement. The reference variable <cref> now contains a reference to the instance.

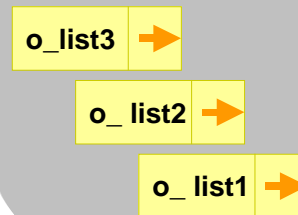
- All classes contain an implicit reference variable called **me**. In each object, it always contains a reference to the object itself, and is consequently known as the self-reference.
- When you work with attributes of a class in methods in the same class, you do not have to specify a reference variable. Instead, the self-reference **me** is set implicitly by the system. In the implementation of our constructor, we could have written **me->carrid = ip\_carrid** instead of just **carrid = ip\_carrid**. Using the self-reference is particularly helpful when you are working in the Debugger.

In the graphic, the instances are displayed as the contents of reference variables are displayed in the Debugger. The prefixed number is assigned randomly.

Instances are displayed with rounded corners to distinguish them graphically from classes.

**DATA:**

```
o_list1 TYPE REF TO lcl_waitlist,  
o_list2 TYPE REF TO lcl_waitlist,  
o_list3 TYPE REF TO lcl_waitlist.
```



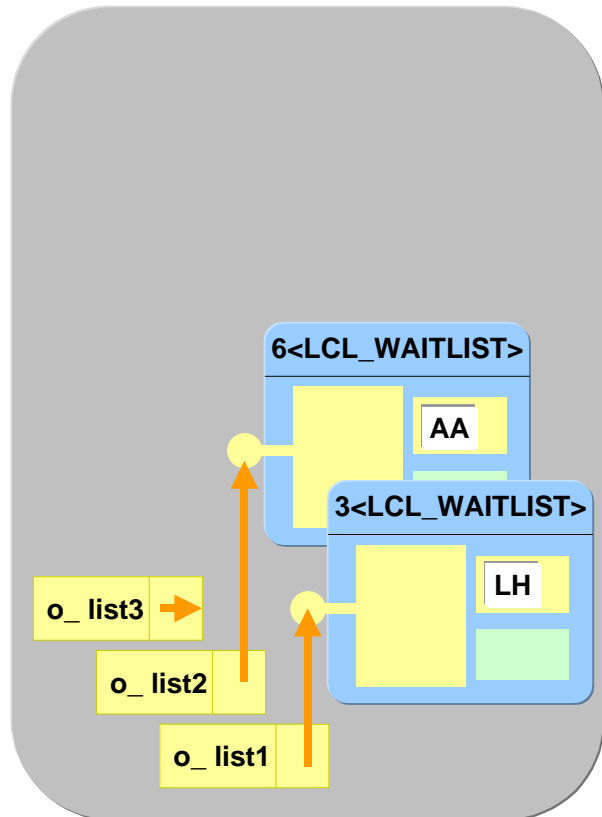
© SAP AG 1999

Here are several reference variables. They are all declared with reference to the same class.

```
DATA:
  o_list1 TYPE REF TO lcl_waitlist,
  o_list2 TYPE REF TO lcl_waitlist,
  o_list3 TYPE REF TO lcl_waitlist.

CREATE OBJECT o_list1
  EXPORTING im_carrid = 'LH'
           im_connid = '400'
           im_fldate = '19991119'.

CREATE OBJECT o_list2
  EXPORTING im_carrid = 'AA'
           im_connid = '17'
           im_fldate = '19991231'.
```



© SAP AG 1999

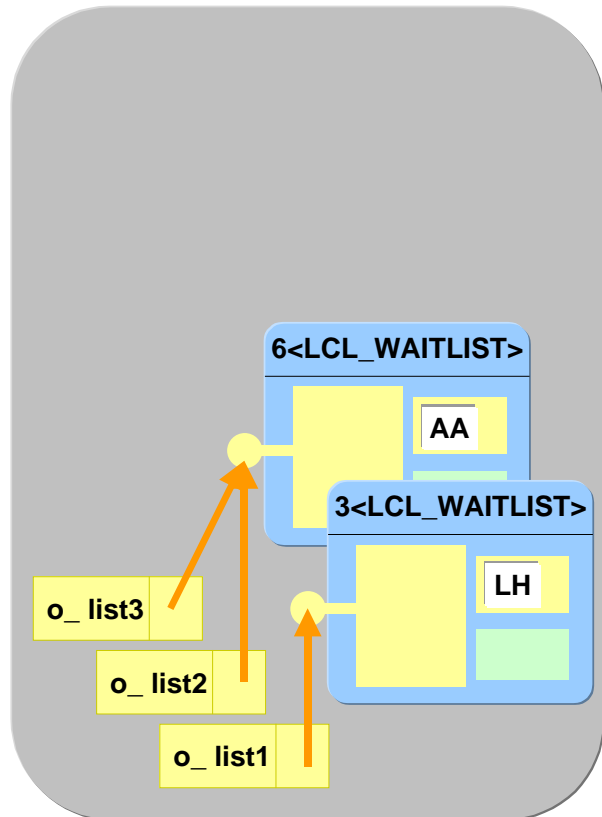
In a given program, you can create any number of instances of the same class. Each instance is fully independent of all the others, with its own identifier and its own attributes. Each **CREATE OBJECT** statement creates a new instance.

```
DATA:
  o_list1 TYPE REF TO lcl_waitlist,
  o_list2 TYPE REF TO lcl_waitlist,
  o_list3 TYPE REF TO lcl_waitlist.

CREATE OBJECT o_list1
  EXPORTING im_carrid = 'LH'
           im_connid = '400'
           im_fldate = '19991119'.

CREATE OBJECT o_list2
  EXPORTING im_carrid = 'AA'
           im_connid = '17'
           im_fldate = '19991231'.

o_list3 = o_list2.
```



© SAP AG 1999

You can assign references to other references using the **MOVE** statement. This means that the reference in several different objects can point to the same object. When you assign between reference variables, the respective types must be either compatible or convertible.

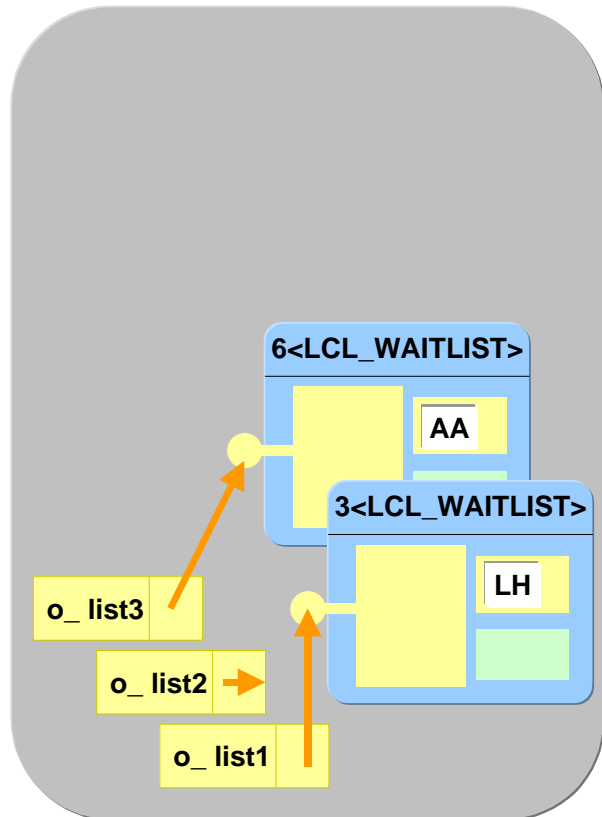
```
DATA:
  o_list1 TYPE REF TO lcl_waitlist,
  o_list2 TYPE REF TO lcl_waitlist,
  o_list3 TYPE REF TO lcl_waitlist.

CREATE OBJECT o_list1
  EXPORTING im_carrid = 'LH'
           im_connid = '400'
           im_fldate = '19991119'.

CREATE OBJECT o_list2
  EXPORTING im_carrid = 'AA'
           im_connid = '17'
           im_fldate = '19991231'.

o_list3 = o_list2.

CLEAR o_list2.
```



© SAP AG 1999

Like other variables, you can initialize a reference variable using the **CLEAR** statement. The initial value of a reference variable is always a reference that **does not** point to an object - an "empty" address.

```
DATA:
  o_list1 TYPE REF TO lcl_waitlist,
  o_list2 TYPE REF TO lcl_waitlist,
  o_list3 TYPE REF TO lcl_waitlist.

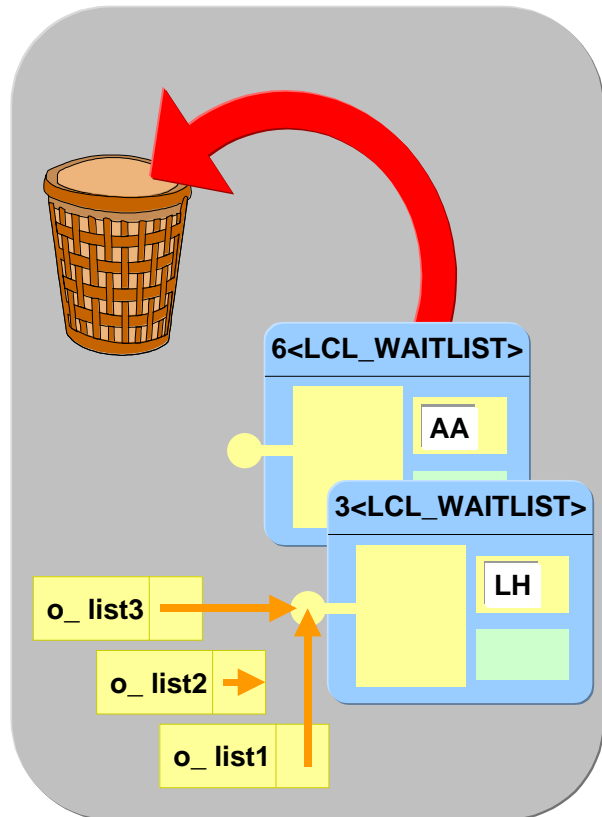
CREATE OBJECT o_list1
  EXPORTING im_carriid = 'LH'
           im_connid = '400'
           im_fldate = '19991119'.

CREATE OBJECT o_list2
  EXPORTING im_carriid = 'AA'
           im_connid = '17'
           im_fldate = '19991231'.

o_list3 = o_list2.

CLEAR o_list2.

o_list3 = o_list1.
```



© SAP AG 1999

If no more references point to an object, it can no longer be addressed in a program, but still exists in memory. This memory space can normally be released under these circumstances.

**Garbage Collection** is a mechanism that ensures that the memory space is released automatically. The garbage collector scans the entire internal session for objects to which no more references are pointing, and deletes them.

### DATA:

```
o_list1 TYPE REF TO lcl_waitlist,  
o_list2 TYPE REF TO lcl_waitlist,  
o_list3 TYPE REF TO lcl_waitlist.
```

### CREATE OBJECT o\_list1

```
EXPORTING im_carrid = 'LH'  
          im_connid = '400'  
          im_fldate = '19991119'.
```

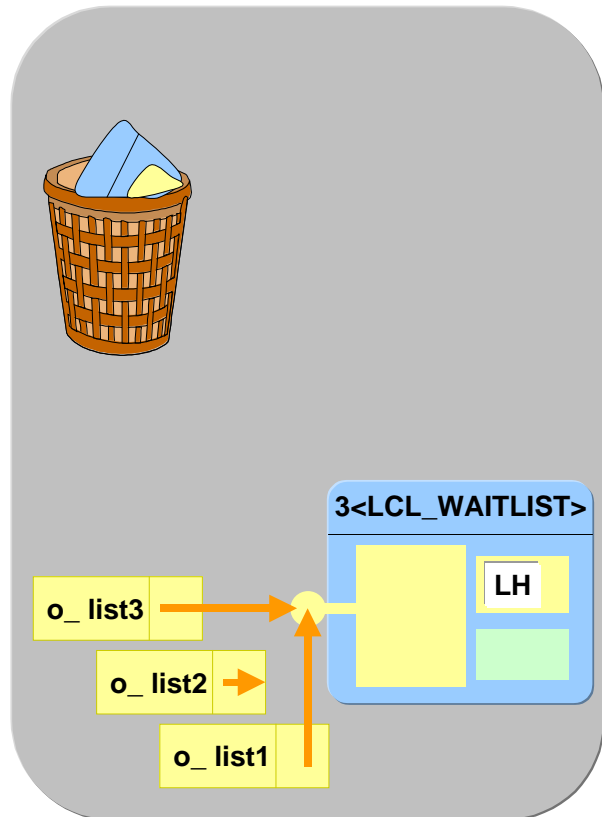
### CREATE OBJECT o\_list2

```
EXPORTING im_carrid = 'AA'  
          im_connid = '17'  
          im_fldate = '19991231'.
```

```
o_list3 = o_list2.
```

```
CLEAR o_list2.
```

```
o_list3 = o_list1.
```



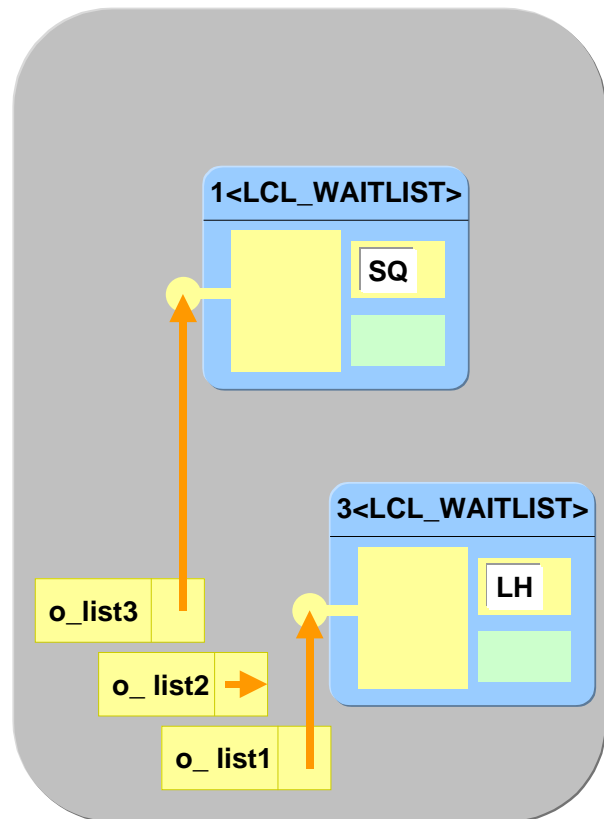
© SAP AG 1999

```
DATA:
  o_list1 TYPE REF TO lcl_waitlist,
  o_list2 TYPE REF TO lcl_waitlist,
  o_list3 TYPE REF TO lcl_waitlist.

...

o_list3 = o_list1.

CREATE OBJECT o_list3
  EXPORTING im_carrid = 'SQ'
           im_connid = '866'
           im_fldate = '20000101'.
```



© SAP AG 1999

We now use **CREATE OBJECT** to create a new object, and the reference in **o\_list3** now points to it. The **CREATE OBJECT** statement has overwritten the previous contents of the reference variable used in the statement with the new reference.



```
DATA
  o_list1 TYPE REF TO lcl_waitlist.

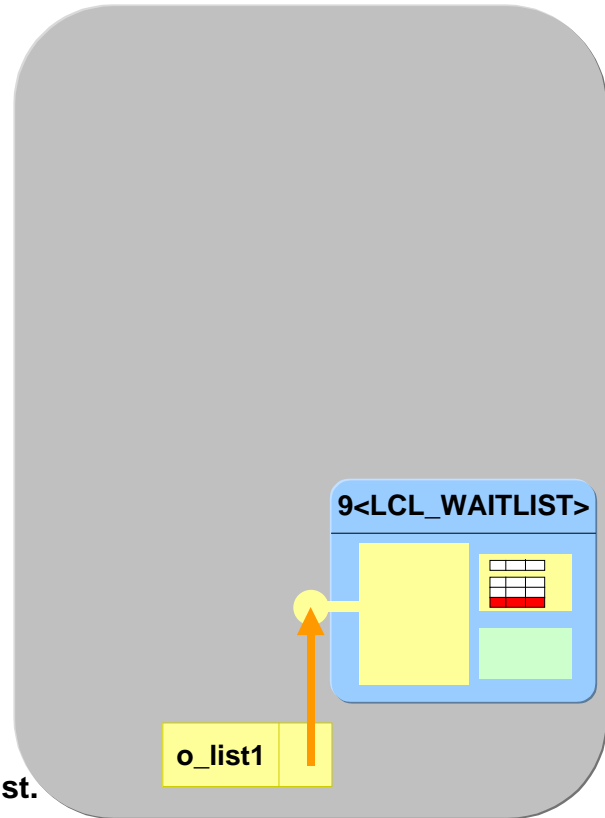
CREATE OBJECT o_list1
  EXPORTING ... .

CALL METHOD o_list1->add
  EXPORTING
    im_cust = wa_cust
  EXCEPTIONS
    in_list = 1
    OTHERS = 2.

CASE sy-subrc.
  WHEN 0.
    WRITE 'ok, inserted'(ins).
  WHEN 1.
    WRITE 'in list already'(it1).
  WHEN 2.
    WRITE 'administrator'(adm).
ENDCASE.
```

**o\_list1->wait\_list now contains the contents of wa\_cust.**

© SAP AG 1999



Once you have created an instance of a class, you can call its methods using the **CALL METHOD** statement. You must specify the name of the method and the object to which you want to apply it. The syntax is **CALL METHOD** <ref>-><meth>.

<ref> is a reference variable that points to an object, and <meth> is a method of the class to which it belongs. The operator -> is called the **object component selector**.

You can call a method **dynamically** using parentheses in the syntax, as is normal in ABAP. Unlike dynamic subroutine and function module calls, you can also pass the parameters and handle the exceptions dynamically. For further information, refer to the online documentation.

```
DATA:
  o_list1 TYPE REF TO lcl_waitlist,
  o_list2 TYPE REF TO lcl_waitlist,
  o_list3 TYPE REF TO lcl_waitlist.
CREATE OBJECT:
  o_list1 EXPORTING ... ,
  o_list2 EXPORTING ... ,
  o_list3 EXPORTING ... .

wa_cust-name = 'SAP AG'.
CALL METHOD o_list1->add
  EXPORTING im_cust = wa_cust ...

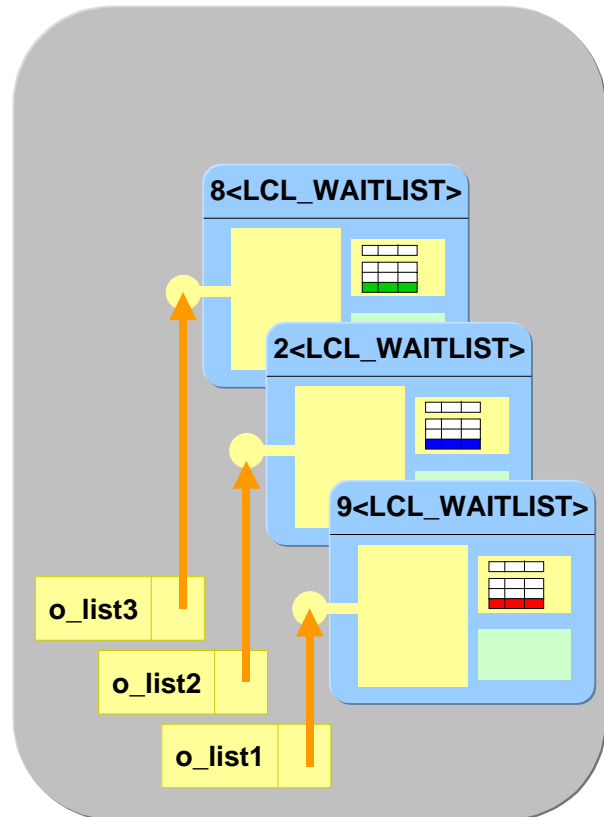
wa_cust-name = 'Dr. Einstein'.
CALL METHOD o_list2->add
  EXPORTING im_cust = wa_cust ...

wa_cust-name = 'IDS Scheer'.
CALL METHOD o_list3->add
  EXPORTING im_cust = wa_cust ...
```



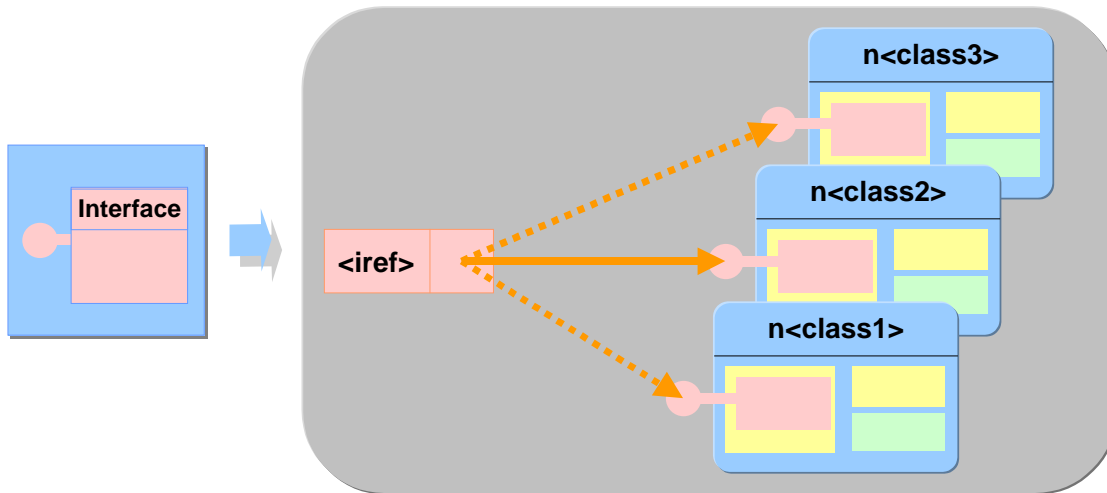
**wait\_list contains different values in each instance**

© SAP AG 1999



In the example above, the same method is called for different objects. Each object has its own set of attributes with which its methods work.

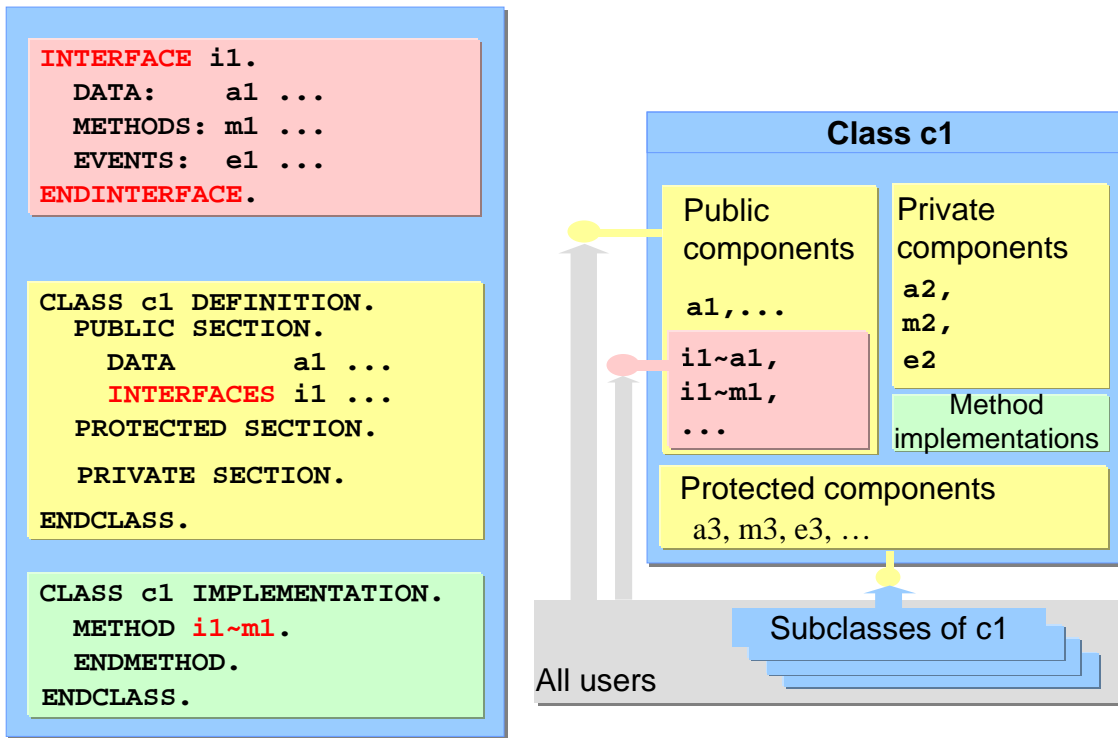
- Definition of a **point of contact to a class without any implementations**
- Classes can **implement several interfaces**
- Uniform access using **interface references**



© SAP AG 1999

An **interface** is a **public** object containing attributes, methods, and events. It only contains a declaration, and is defined **independently** of classes. The components of an interface are implemented **in the classes that use the interface**. The classes then have a uniform external point of contact. Classes that use an interface must provide its **functions** by implementing all of the methods that are defined in it. Although interfaces cannot be instantiated, you can still create reference variables with reference to an interface. An **interface reference** can point to instances of any class that has implemented the components of the relevant interface. You can access these components in a uniform way using the interface reference.

This concept allows related classes to provide similar functions that can be addressed in a uniform way despite being implemented in different classes. In short, interface references allow you to address different classes in the same way. This is often referred to as **polymorphism**.



© SAP AG 1999

The public components of a class define its external point of contact.

An interface is a standalone object that allows you to define or extend this external point of contact.

Interfaces allow you to address the object components of different classes in a uniform way.

- The left-hand side of the graphic shows the definition of a local interface **i1** and the declaration and implementation parts of a local class **c1**, which implements interface **i1** in its **public** section.

The interface method **m1** must be implemented in the implementation part of class **c1** as **i1~m1**.

- The right-hand side of the graphic shows the structure of the class, with the components in their visibility sections and the method implementations. The interface components extend the public section of the class, which means that all users can access both the public components that belong to the class alone and the components that have been added by implementing the interface.

A class can implement several interfaces in **parallel**, as long as it implements **all** of the methods of all of those interfaces.

Interfaces can be nested. (**INTERFACE ... INTERFACES ... ENDINTERFACE.**)

```

INTERFACE lif_status.
    METHODS display.
ENDINTERFACE.

CLASS lcl_waitlist DEFINITION.
    PUBLIC SECTION.
        ...
        INTERFACES lif_status.
        ...
ENDCLASS.

CLASS lcl_clerk DEFINITION.
    PUBLIC SECTION.
        ...
        INTERFACES lif_status.
        ...
    PRIVATE SECTION.
        DATA cnt_look_for TYPE i.
ENDCLASS.
    
```

© SAP AG 1999

As an example, let us look at the local interface **lif\_status** of program **SAPBC402\_AOOD\_WAITLISTS**. It contains a method declaration **display**. This method is to allow various objects of different types to indicate their status.

The second class, **lcl\_clerk**, describes the statuses of clerks at the airport counter. More detailed information about this will follow. The counter **cnt\_look\_for** stands for a particular status of a clerk. This interface method must now be implemented in the two classes. First, however, the interface must be declared in the public section of the classes.

```

CLASS lcl_waitlist IMPLEMENTATION.
...
METHOD lif_status~display.
    CALL FUNCTION 'BC402_AOOD_WAIT_DISPLAY'
        EXPORTING
            ip_waitlist = wait_list
    EXCEPTIONS
        list_empty    = 1
        OTHERS        = 2.
...
ENDMETHOD.

ENDCLASS.

CLASS lcl_clerk IMPLEMENTATION.
...
METHOD lif_status~display.
    MESSAGE i194 WITH cnt_look_for ... .
ENDMETHOD.

ENDCLASS.
    
```

© SAP AG 1999

- In the waiting list, we are going to display the status by calling a function module that displays the list in a modal dialog box.
- In the clerk class, the interface method will be implemented differently, and a method will be displayed containing information about the status of a clerk.

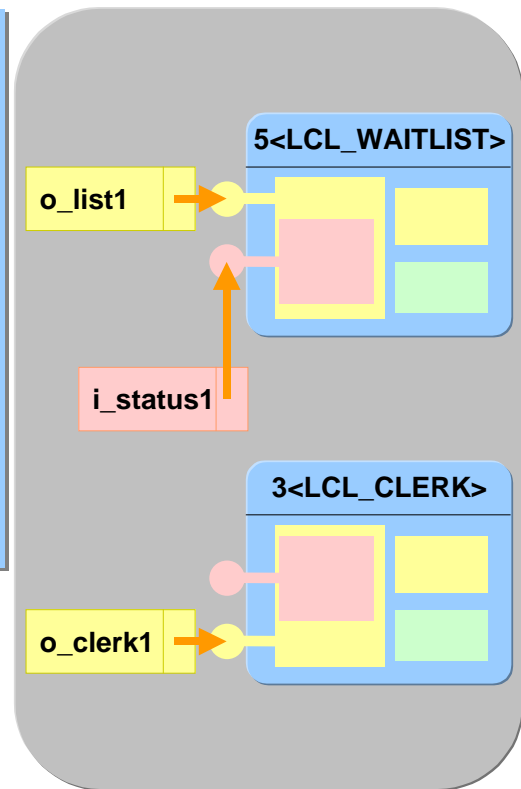
```
DATA:
  o_list1  TYPE REF TO lcl_waitlist,
  o_clerk1 TYPE REF TO lcl_clerk,

  i_status1 TYPE REF TO lif_status.

CREATE OBJECT o_list1
  EXPORTING im_carrid = 'LH'
           im_connid = '400'
           im_fldate = '19991119'.

i_status1 = o_list1.

CREATE OBJECT o_clerk1.
```



© SAP AG 1999

In order to access an object, we always need a reference variable.

Instead of creating reference variables with reference to a class, we can also create them by referring to an interface (in this case, **i\_status1**). This kind of reference variable can contain references to objects of any class that implements the corresponding interface.

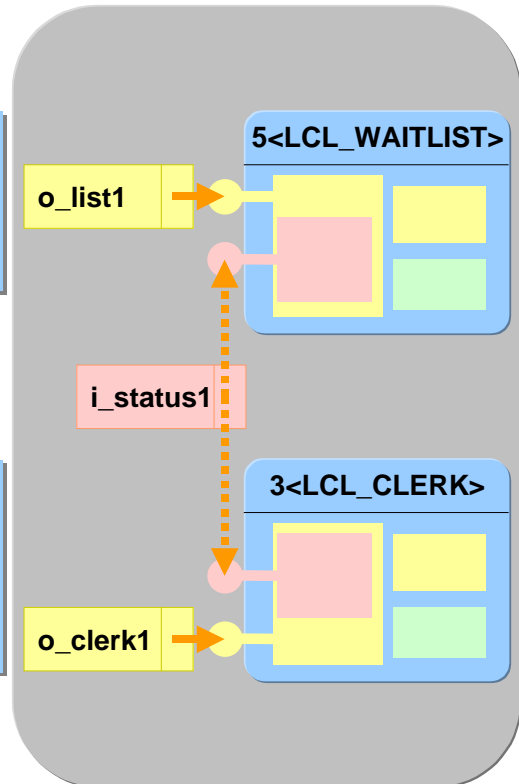
- The example then creates a class instance **o\_list1**. We can access **all** components of **o\_list1** in the normal way. After we have assigned **o\_list1** to **i\_status1**, we can also use **i\_status1** to address the interface components. If **i\_status1** were an object of class **lcl\_clerk**, the interface components would behave differently when we accessed them.

```
* display state of waitlist:
*****
CALL METHOD o_list1->lif_status~display.
* or:
CALL METHOD i_status1->display.
```

**i\_status1->display would call the function module**

```
* display state of clerk:
*****
CALL METHOD o_clerk1->lif_status~display.
* or:
i_status1 = o_clerk1.
CALL METHOD i_status1->display.
```

**i\_status1->display would display the message**



© SAP AG 1999

- If a **class reference** <cref> points to a class instance, you can call a method <meth> of an interface <intf> that has been implemented by the class using the form  
**CALL METHOD** <cref>-><intf>~<meth> ... .
- If an **interface reference** <iref>, created with reference to the interface <intf>, points to a class instance, you can call the method <meth> of the interface <intf> using the form  
**CALL METHOD** <iref>-><meth> ... .

The same principle applies to interface attributes.

By using interface references, you can access those components of a class that have been added by the implementation of an interface.





- Declaring reference variables

```
DATA: <cref> TYPE REF TO <class>,  
      <iref> TYPE REF TO <interface>.
```

- Creating objects

```
CREATE OBJECT <cref> ... .
```

- Accessing attributes and calling methods

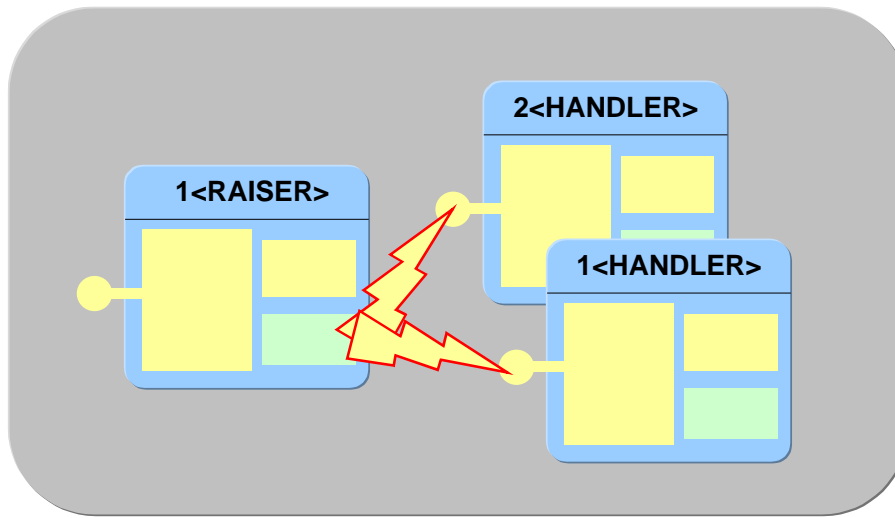
```
x =    <cref>-><attr>  
      + <cref>-><interface>~<attr>  
      - <iref>-><attr>.  
  
CALL METHOD: <cref>-><method> ... ,  
            <cref>-><interface>~<method> ... ,  
            <iref>-><method> ... .
```

- **Functional methods**
- **Static attributes and static methods (including static constructor)**
- **Global classes**
- **Global interfaces**
- **Inheritance**

© SAP AG 1999

The topics listed above would be a logical continuation from the stage that we have just reached. However, it would not be possible to do them justice in the time available to us here. Instead, let's continue with another important concept of object-oriented programming - **events**.

- Events are components of classes
- Methods can trigger the events of their class
- Handler methods can react to events



© SAP AG 1999

A particular feature of ABAP Objects is that it allows you to trigger events and react to them. ABAP Objects events are not to be confused with the events that are part of the runtime environment and occur during user dialog processing (see the **ABAP Runtime Environment** unit).

An event in object orientation can be triggered by a single object, for example, to indicate that its state has changed. Other objects can then react to selected events. This is implemented using special events, which you do not have to call explicitly. Instead, they are called implicitly when the event is triggered **as long as** you have registered them to react to that particular event.

During the definition and at runtime, neither the sender nor receiver of an event "knows" anything about the other. Objects trigger events without knowing if it will be received or how any receiver will react to it.

Conversely, receivers are registered for events without knowing whether the event will be triggered at all, or by which object.

At runtime, however, it is possible to identify the trigger of the event using the parameter **sender**.

```
CLASS <raiser_class> DEFINITION.  
  
  {PUBLIC|PROTECTED|PRIVATE} SECTION.  
  
    METHODS <event_raiser> ...  
  
    EVENTS <evt> [EXPORTING value(<par>) TYPE <type> [OPTIONAL]].  
  
ENDCLASS.
```

```
CLASS <raiser_class> IMPLEMENTATION.  
  
  METHOD <event_raiser>.  
    ...  
    RAISE EVENT <evt> [EXPORTING <par> = <value> ... ].  
    ...  
  ENDMETHOD.  
  
ENDCLASS.
```

© SAP AG 1999

- You declare events as components of a class in the declaration part. They can have export parameters, but these must be passed by value.
- To trigger an event in a method, use the **RAISE EVENT** statement. You must pass all of its parameters (except any you defined as optional) using the **EXPORTING** addition.

```
CLASS <handler_class> DEFINITION.  
  
  {PUBLIC|PROTECTED|PRIVATE} SECTION.  
  
    METHODS <event_handler> FOR EVENT <evt> OF <raiser_class>  
      [IMPORTING ... [sender] ].  
  
ENDCLASS.
```

```
CLASS <handler_class> IMPLEMENTATION.  
  
  METHOD <event_handler>.  
    ...  
  ENDMETHOD.  
  
ENDCLASS.
```

© SAP AG 1999

Any class can contain event handler methods for selected events of selected classes. These methods react to the events when they are triggered.

- The interface of an event handler method **can** have importing parameters, but may only contain the formal parameters that are defined as exporting parameters for the **event** in question. The attributes of the parameter are inherited from the event (their types are **not** specified explicitly). The handler method does not have to receive all of the parameters passed in the **RAISE EVENT** statement.
- Events have an implicit parameter called **sender**, which you can also receive by listing it in the **IMPORTING** addition. This allows the handler method to access the trigger, and makes it possible for it to make its own behavior dependent on the name of the trigger.

If you declare event handler methods in a class, you **make the instances of the class able** to handle the event.

## Event trigger

```

CLASS c1 DEFINITION.
  PUBLIC SECTION.
    EVENTS e1 EXPORTING VALUE(p1)
              TYPE i.

    METHODS m1.

  PRIVATE SECTION.
    DATA a1 TYPE i.

ENDCLASS.

```

```

CLASS c1 IMPLEMENTATION.
  METHOD m1.
    a1 = ...
    RAISE EVENT e1
      EXPORTING p1 = a1.
  ENDMETHOD.
ENDCLASS.

```

## Event handler

```

CLASS c2 DEFINITION.
  PUBLIC SECTION.
    METHODS m2 FOR EVENT e1 OF c1
              IMPORTING p1.

  PRIVATE SECTION.
    DATA a2 TYPE i.

ENDCLASS.

```

```

CLASS c2 IMPLEMENTATION.
  METHOD m2.
    ...
    a2 = p1.
    ...
  ENDMETHOD.
ENDCLASS.

```

© SAP AG 1999

- Here, class **c1** contains an event **e1**, which exports an integer when it is triggered by method **m1**.
- Class **c2** contains a method **m2**, which **can** handle event **e1** of class **c1** and import a value. At runtime, this parameter of **m2** is assigned type **i**.

```
PROGRAM ...

DATA:
  o_raiser      TYPE REF TO lcl_raiser,
  o_handler_1 TYPE REF TO lcl_handler,
  o_handler_2 TYPE REF TO lcl_handler.

CREATE OBJECT:
  o_raiser, o_handler_1, o_handler_2.

SET HANDLER o_handler_1->event_handler FOR o_raiser.
SET HANDLER o_handler_2->event_handler FOR o_raiser.

...

CALL METHOD o_raiser->event_raiser.
```

© SAP AG 1999

In order for an event handler method to react to an event, you must specify **at runtime** the **trigger(s)** (object or objects) to which it should react). When you declare the class, you can only specify the **class** of the triggering object.

To link a handler method to an event trigger, use the **SET HANDLER** statement. For instance events, you must use the **FOR** addition to specify the triggering instance for which you want to register the handler:

- You can either register for a **single** triggering instance, using a reference variable (here **o\_raiser**).
- Or you can register the handler for **all** instances using the **FOR ALL INSTANCES** addition. This registration also includes potential triggering instances that have not yet been created.

The optional **ACTIVATION** addition, available for both variants, allows you to recall individual registrations or set new ones dynamically.

The argument in this addition must be a field with type **C** and length 1. You can set the expression

**ACTIVATION** <flag> as follows:

- <flag> = **space**: Deregisters the event handler method
- <flag> = **'X'**: Registers the event handler method.

```

CLASS lcl_waitlist DEFINITION.
...
EVENTS list_full EXPORTING value(ex_carrid) TYPE sflight-carrid
                        value(ex_connid) TYPE sflight-connid
                        value(ex_fldate) TYPE sflight-fldate.
...
ENDCLASS.
    
```

```

CLASS lcl_waitlist IMPLEMENTATION.
CONSTANTS max_entries LIKE sy-tabix VALUE '10'.
DATA last_pos LIKE sy-tabix.
METHOD add.
...
DESCRIBE TABLE wait_list LINES last_pos.
IF last_pos < max_entries.
    APPEND ip_cust TO wait_list.
ELSE.
    RAISE EVENT list_full EXPORTING ex_carrid = carrid
                                ex_connid = connid
                                ex_fldate = fldate.
...
ENDMETHOD.
ENDCLASS.
    
```

© SAP AG 1999

Let us now extend our waiting list example: If the number of entries in the list reaches a certain maximum value, we want the **waiting list** to trigger an event. There are various things that the clerk needs to do when the event is triggered.

- In the declaration part of the class **lcl\_waitlist**, we declare the event **list\_full**. This has three parameters that can be exported when the event is triggered.
- In the implementation part, we change the add method accordingly.



```
CLASS lcl_clerk DEFINITION.  
  PUBLIC SECTION.  
    ...  
    METHODS say_something    FOR EVENT list_full OF lcl_waitlist.  
    METHODS change_planetype FOR EVENT list_full OF lcl_waitlist  
                                IMPORTING ex_carrid ex_connid ex_fldate.  
ENDCLASS.
```

```
CLASS lcl_clerk IMPLEMENTATION.  
  
  ...  
  
  METHOD say_something.  
    MESSAGE i197 WITH ... .  
  ENDMETHOD.  
  
  METHOD change_planetype.  
    MESSAGE i195 WITH ... ex_carrid ex_connid ex_fldate.  
  ENDMETHOD.  
  
ENDCLASS.
```

© SAP AG 1999

We have now created another class **lcl\_clerk** for our clerk:

- In the declaration part, we specify that the two methods can be executed when instances of the class **lcl\_waitlist** trigger the event **list\_full**. The method **change\_planetype** should be able to process all three export parameters.
- In the implementation part, we program the functions of the methods as normal. In this case (for simplicity) the methods just send various messages. In the method **change\_planetype**, the parameters received at runtime can also be displayed with the message.

```
PROGRAM ...

DATA:
  o_list1  TYPE REF TO lcl_waitlist,
  o_clerk1 TYPE REF TO lcl_clerk.

CREATE OBJECT o_list1 EXPORTING im_carrid = 'LH'
                        im_connid = '400'
                        im_fldate = '19991119'.

CREATE OBJECT o_clerk1.

SET HANDLER o_clerk1->say_something    FOR o_list1.
SET HANDLER o_clerk1->change_planetype FOR o_list1.
```

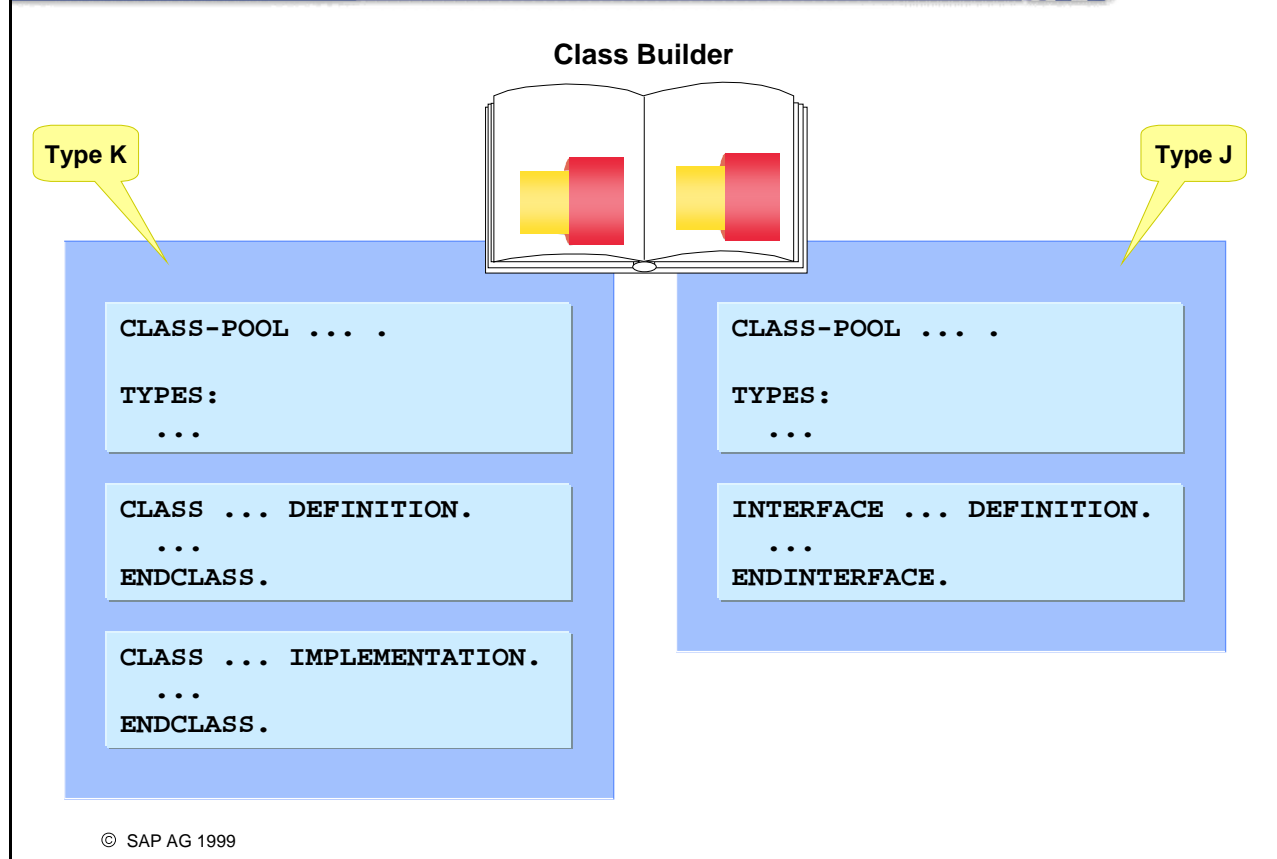
© SAP AG 1999

Once we have created an instance **o\_clerk1** of class **lcl\_clerk**, we can register the methods of the object to react to events triggered by the object **o\_list1**. If we do not register the methods, they will not be executed, even if the event is triggered.

This allows us to specify exactly when the method should react to the event, as well as to define the groups of triggers and handlers.

If, for example, we were to create more waiting lists from the class, which could also trigger events, our clerk would only be able to react to events from waiting list **o\_list1** unless we registered its methods for other instances.

Conversely, if we create more objects from the clerk class, but do not register their methods to react to an event, only our first clerk **o\_clerk1** will react. We could also make it react "partially" by only registering some of its handler methods. So you can see that the event registration technique is very flexible!



The **Class Builder** is a tool in the ABAP Workbench that you use to create **global** classes and interfaces. You define the components using a graphical interface, then the Class Builder generates the corresponding ABAP source code automatically. You have to write the implementation part of the class yourself. For this, the Class Builder calls the ABAP Editor.

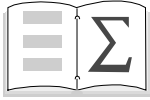
Global classes are stored in special ABAP container programs called class pools (type K program). Global interfaces are stored in interface pools (type J program). Each class or interface pool contains the definition of a single global class or interface. The class or interface pool is also generated automatically by the **Class Builder**.

Class pools are similar to function groups. They contain both declarative statements and executable statements, and they cannot be started directly. You use the **CREATE OBJECT** statement to create an instance of a global class (an object) in a program.

## Further Information:

Various other courses explain further syntax variants used in ABAP Objects and how to use global classes, in particular, **BC400 (ABAP Workbench: Concepts and Tools)**, **BC410 (Programming User Dialogs)**, and **BC412 (ABAP Dialog Programming With EnjoySAP Controls)**.

There is also a dedicated **ABAP Objects** course, **BC404**.

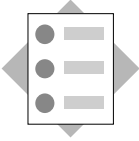


**You are now able to:**

- **Declare and implement local classes**
- **Create objects**
- **Access object components**
- **Define local interfaces**
- **Access interface components**
- **Trigger and handle exceptions**

### **Contents:**

- **Techniques for calling programs**
- **Memory model**
- **Techniques for passing data**
- **Uses**



**At the conclusion of this unit, you will be able to:**

- **Describe the R/3 memory model**
- **Call executable programs**
- **Call transactions**
- **Use the various memory areas to pass data**

Preface

Course Overview

ABAP Runtime Environment

Data Types and Data Objects

Statements

Internal Table Operations

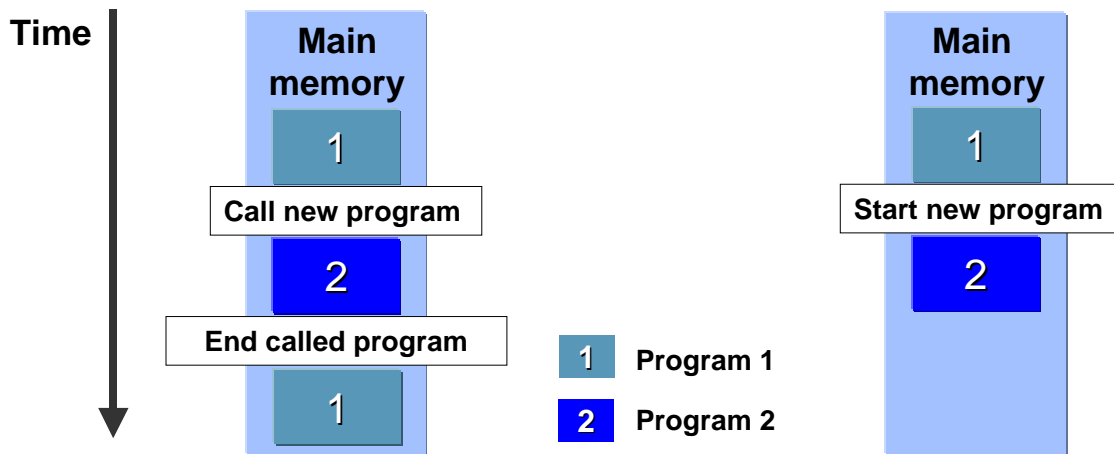
Subroutines

Function Groups and Function Modules

Introduction to ABAP Objects

**Calling Programs and Passing Data**





© SAP AG 1999

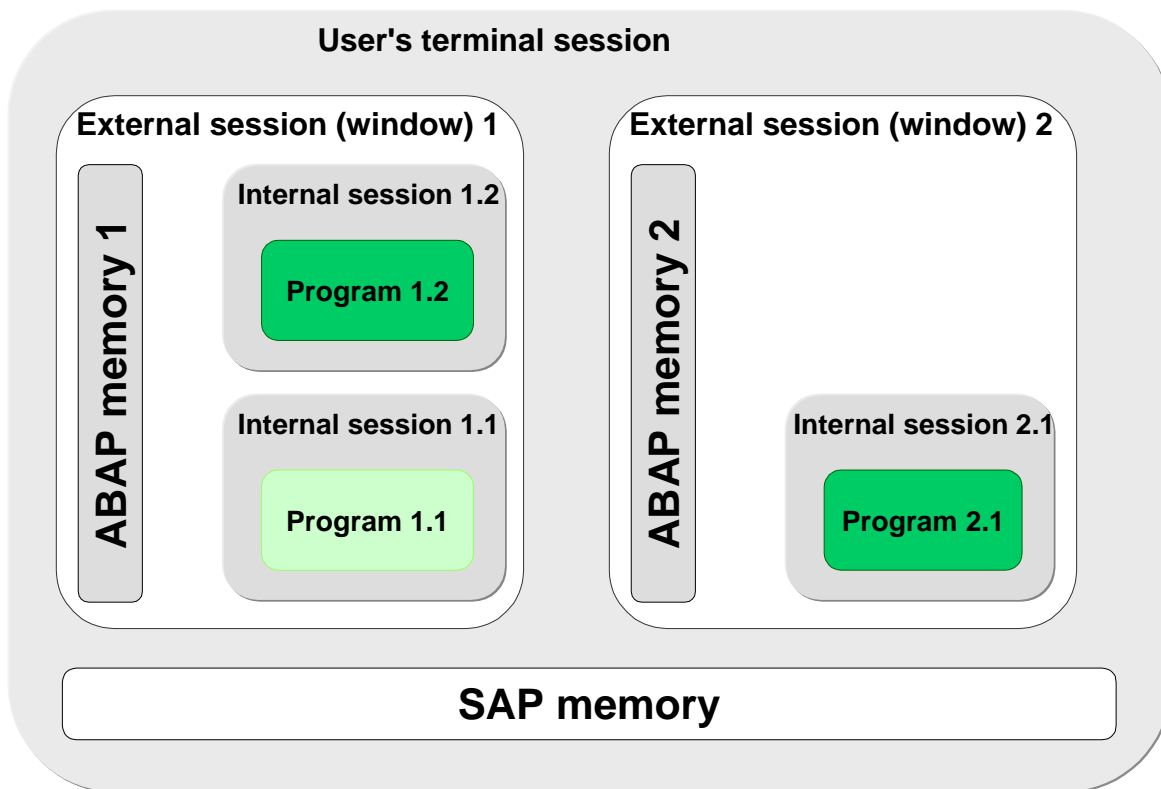
There are two ways of starting an ABAP program from another ABAP program that is already running:

- By interrupting the current program to run the new one - the called program is executed, and afterwards, processing returns to the program that called it.
- By terminating the current program and then running the new one.

Complete ABAP programs within a single user session can only run sequentially. We refer to this technique as using **synchronous calls**.

If you want to run functions in parallel, you must use function modules. For further information about this technique, refer to course **BC415 (Communication Interfaces in ABAP)**, and the documentation for the **CALL FUNCTION ... STARTING NEW TASK...** statement.





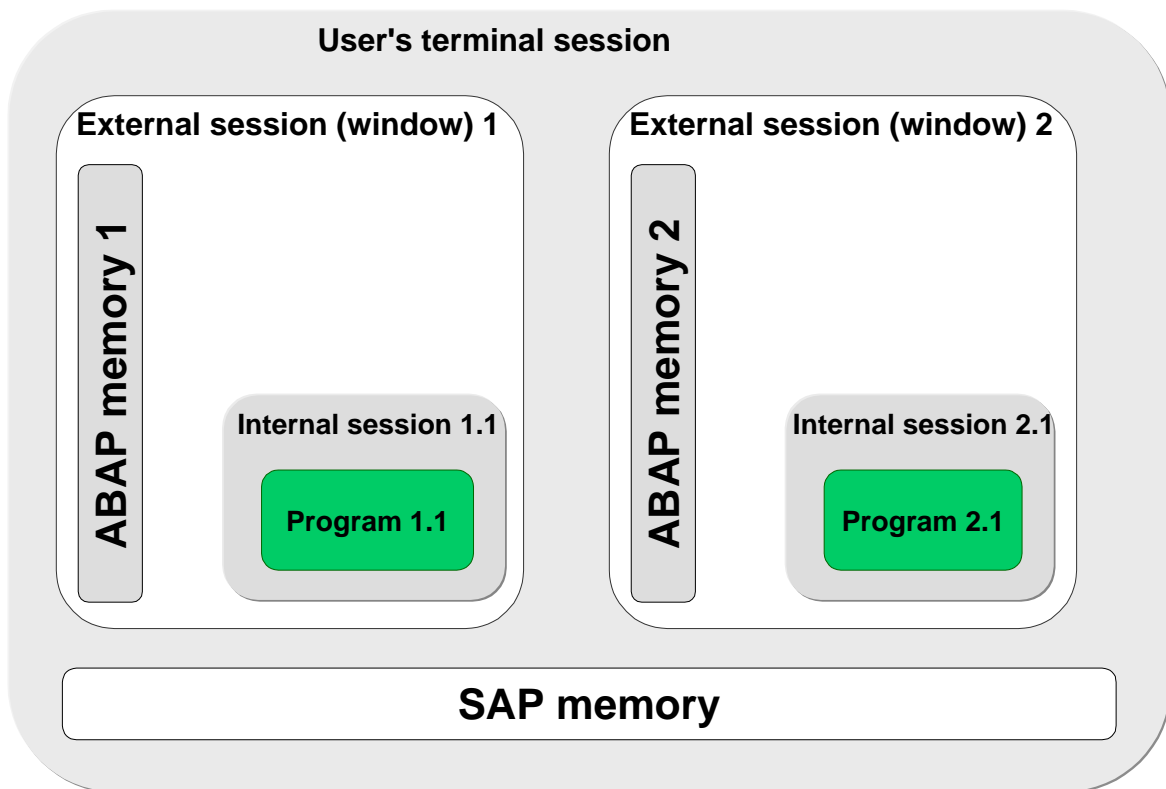
© SAP AG 1999

The way in which main memory is organized from the program's point of view can be represented easily in the above model. There is a distinction between internal and external sessions:

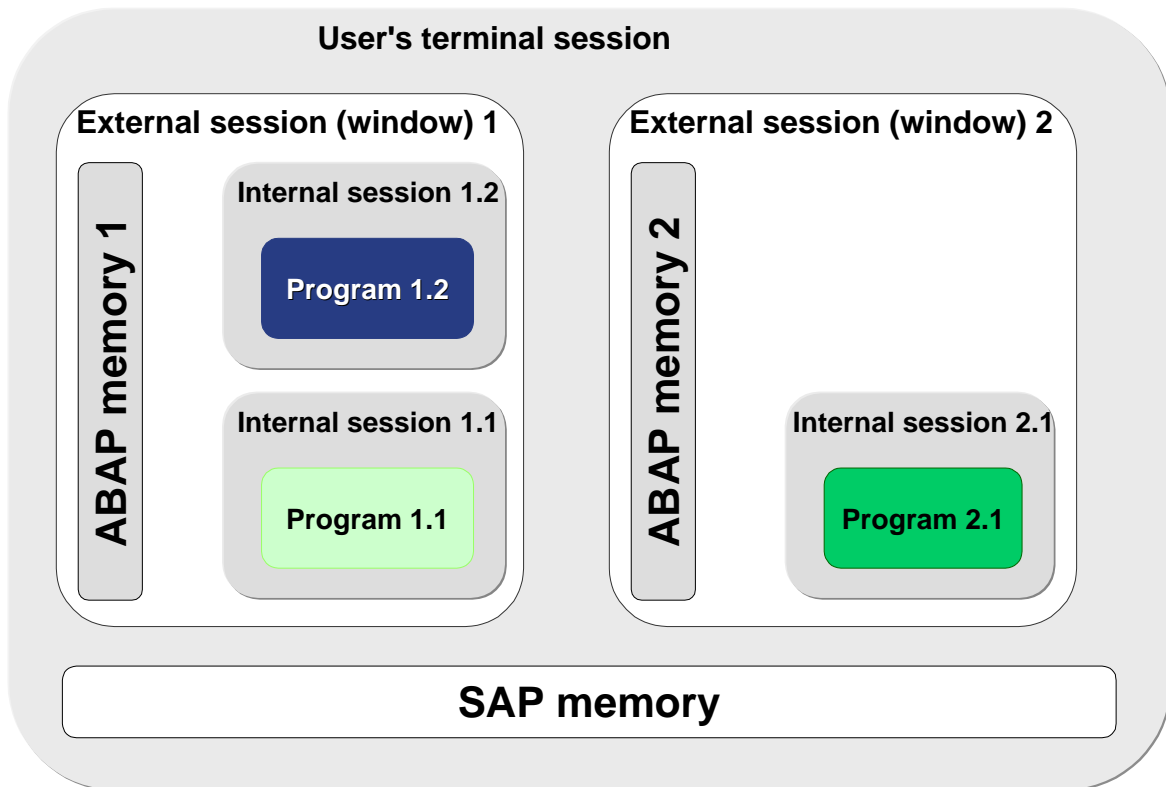
- Generally, an **external session** corresponds to an R/3 window. You create new external sessions by choosing *System → Create session* or entering `/o<code>` in the command field. You can have up to six external sessions open simultaneously in one terminal session.
- External sessions are subdivided into **internal sessions**. Each program that you run occupies its own internal session. Each external session can contain up to nine internal sessions.

The data in a program is only visible **within** that internal session, so it is only visible to the program.

The following pages illustrate how the stack inside an external session changes with various program calls.



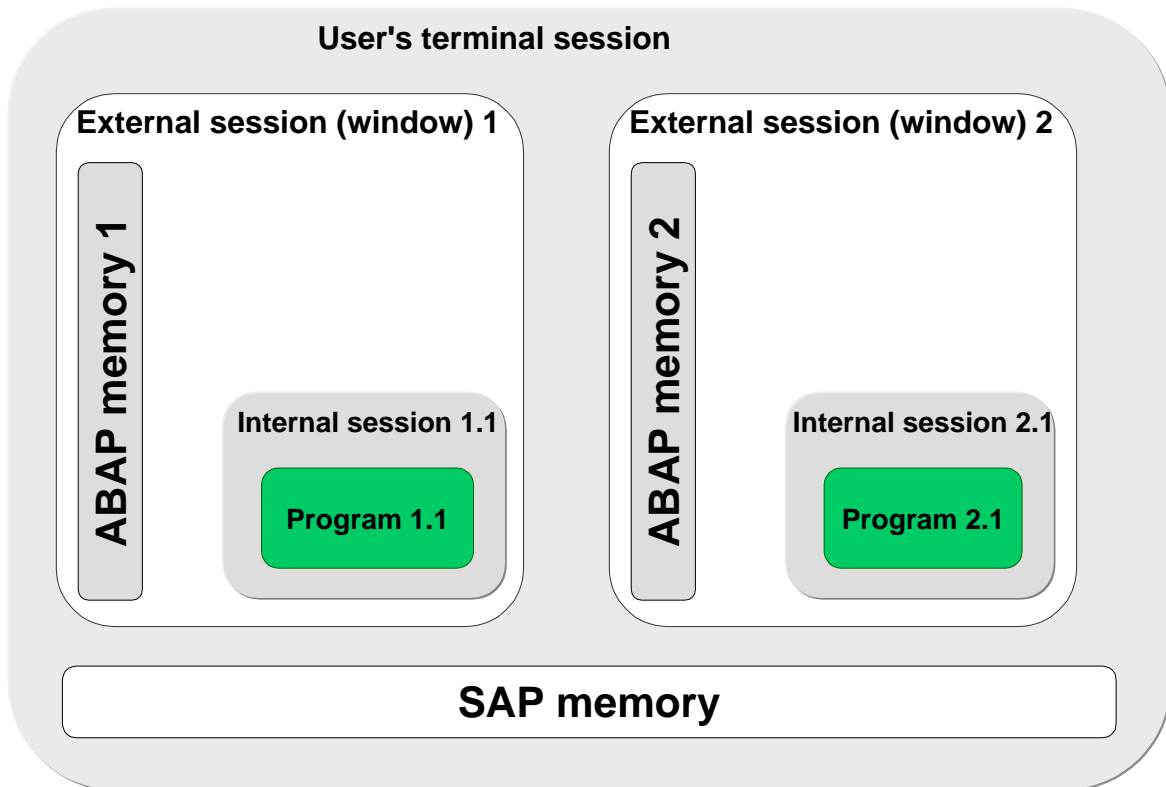
© SAP AG 1999



© SAP AG 1999

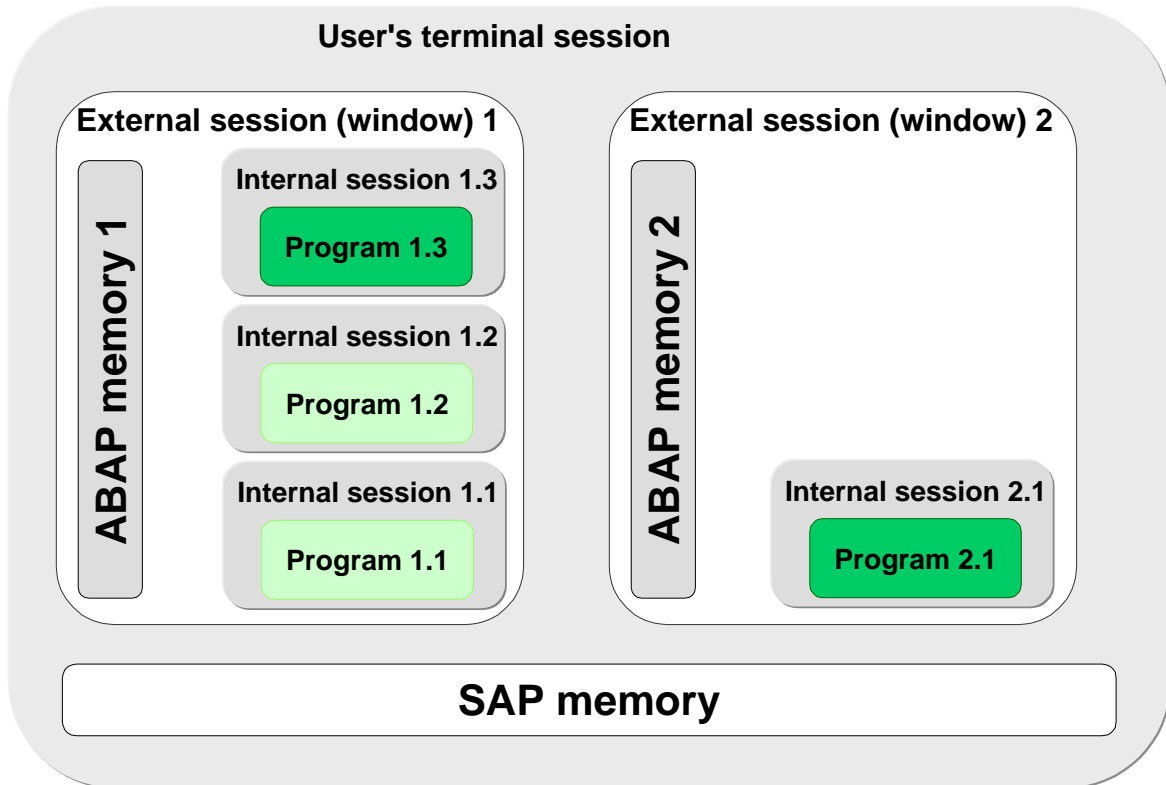
When you insert a program, the system creates a new internal session, which contains the new program context.

The new session is placed on the stack. The program context of the calling program also remains on the stack.



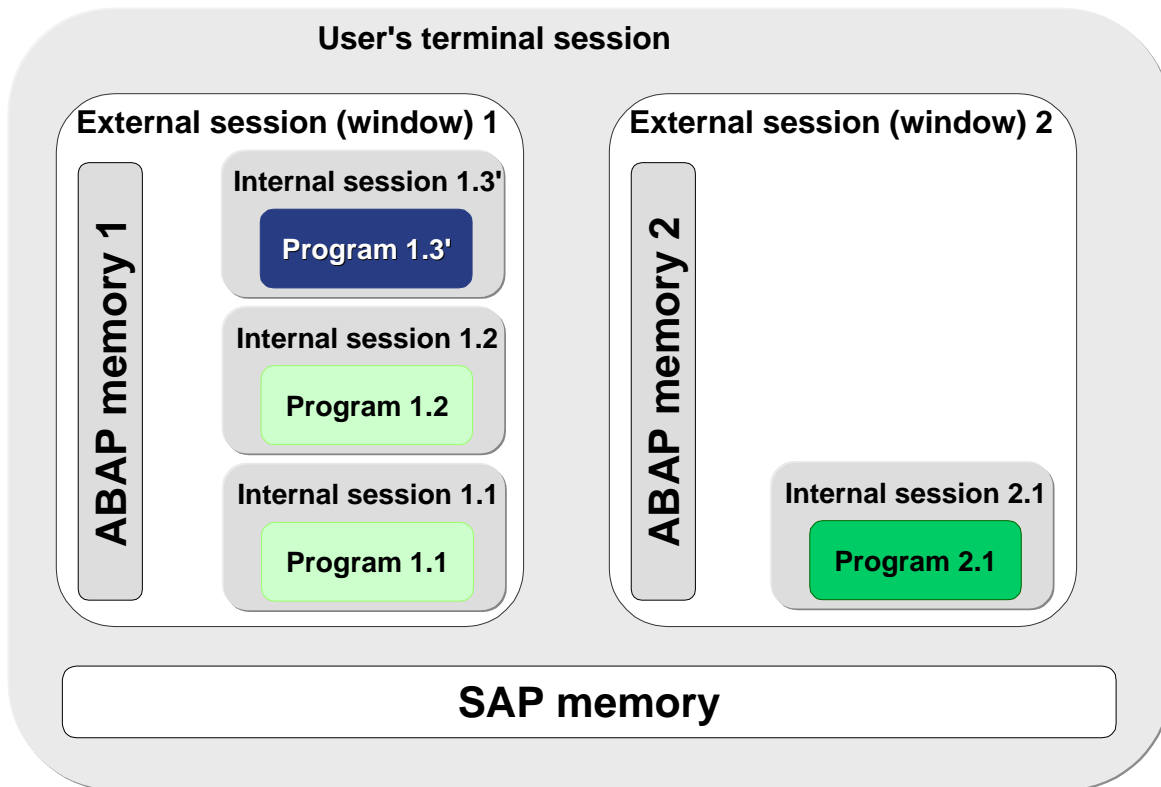
© SAP AG 1999

When the called program finishes, its internal session (the top one in the stack) is deleted. Processing is resumed in the next-highest internal session in the stack.



© SAP AG 1999

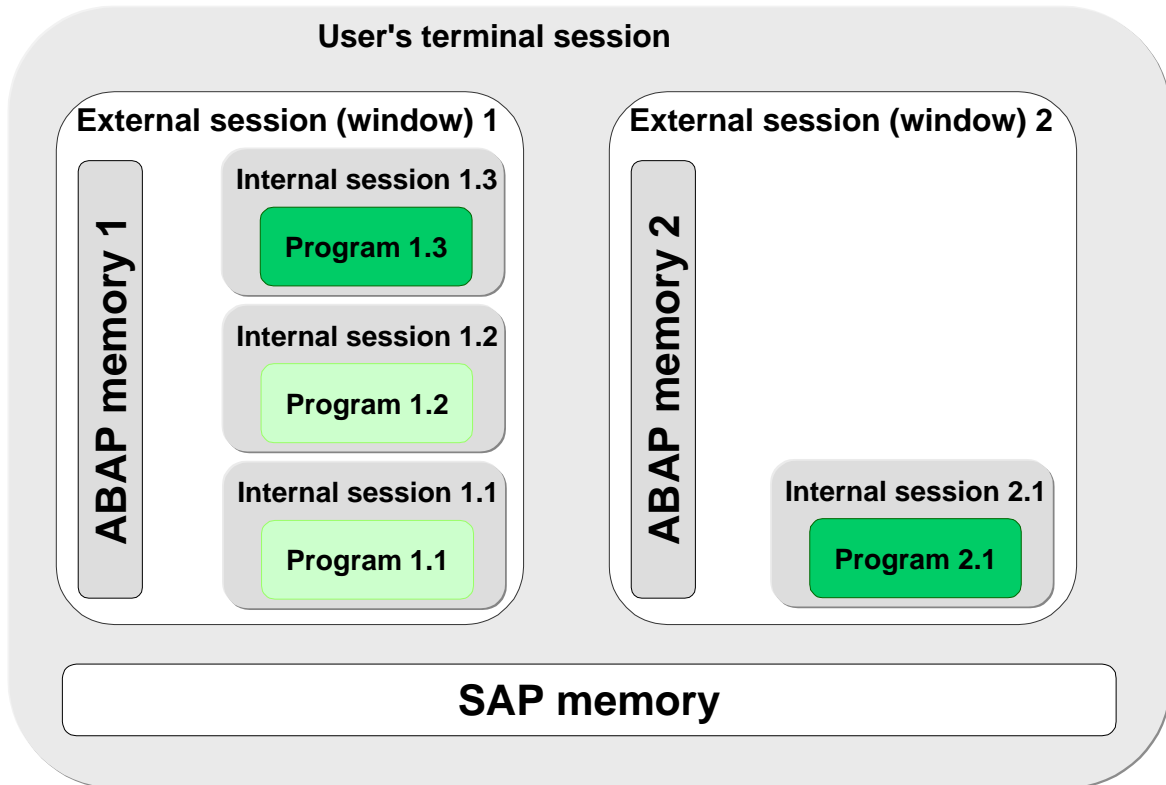
When you end a program and start a new one, there is a difference between calling an executable program and calling a transaction.



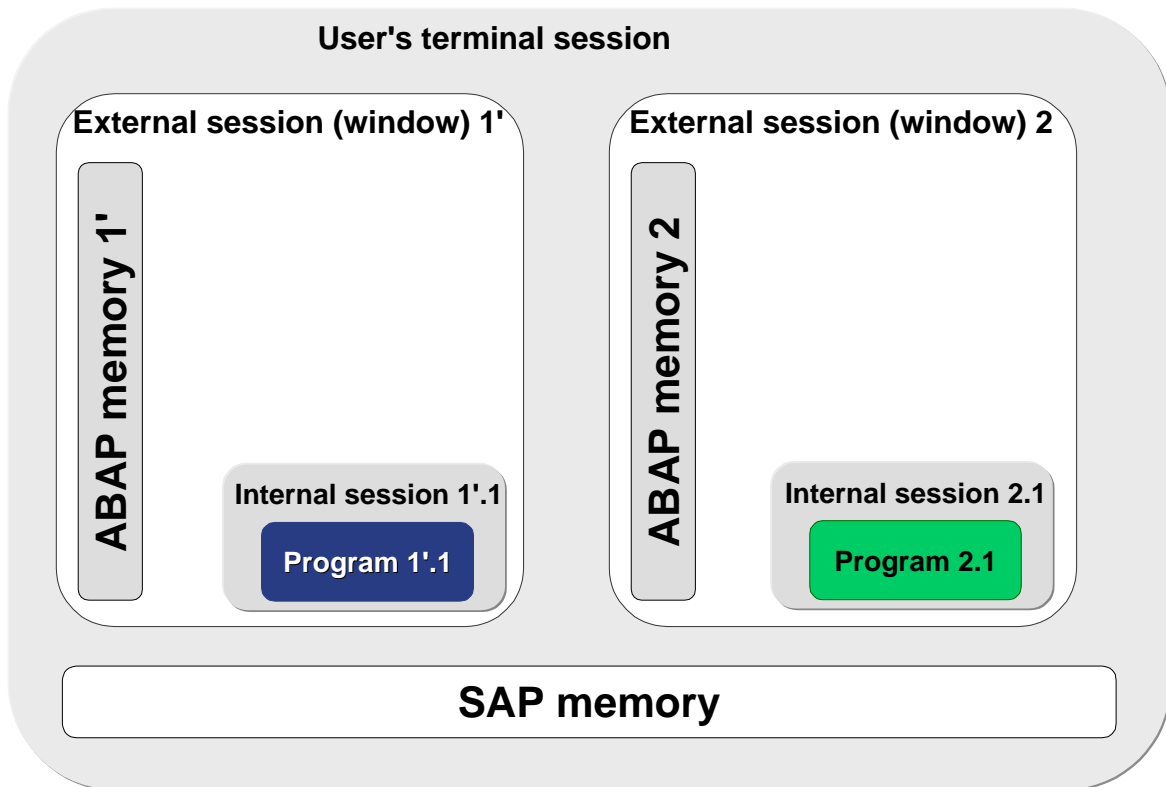
© SAP AG 1999

If you start an **executable program** using its **program name**, the internal session of the program you are ending (the top one) is removed.

The system creates a new internal session, which contains the program context of the called program. The new session is placed on the stack. Any program contexts that already existed are retained. The topmost internal session on the stack is replaced.



© SAP AG 1999

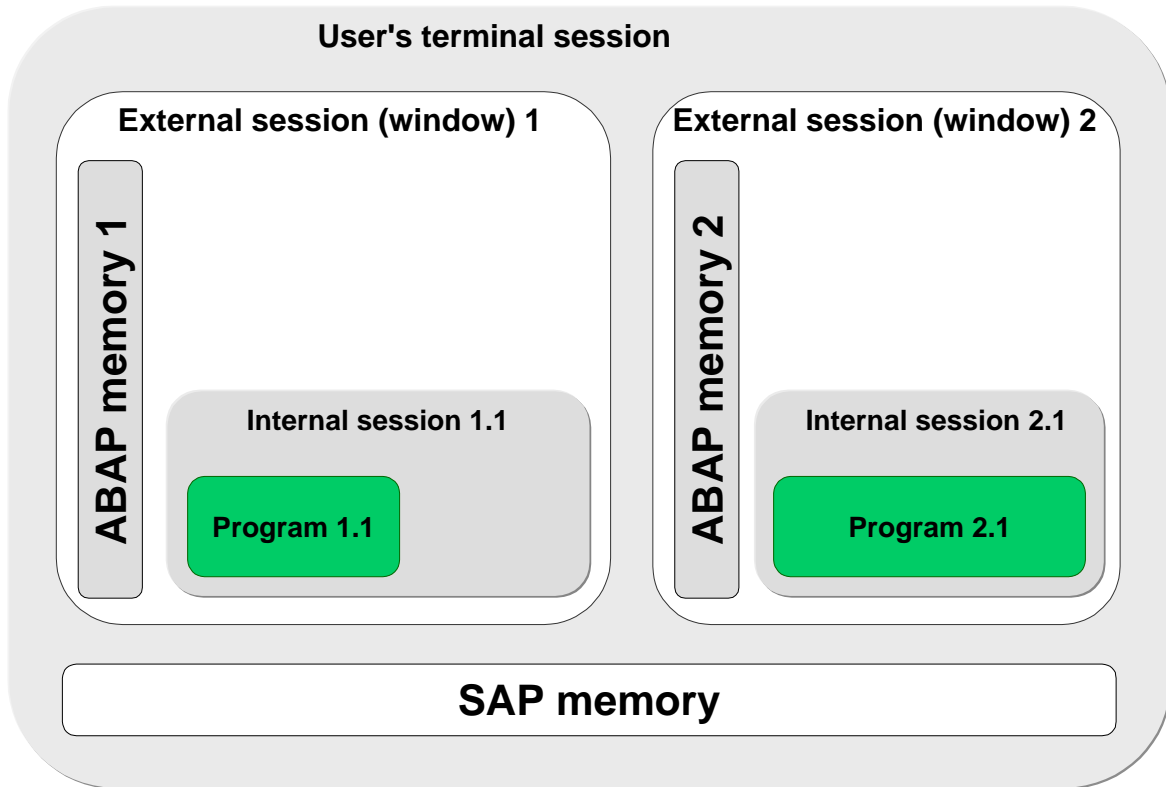


© SAP AG 1999

If you start a program using its transaction code (if one is assigned), all of the existing internal sessions are removed from the stack.

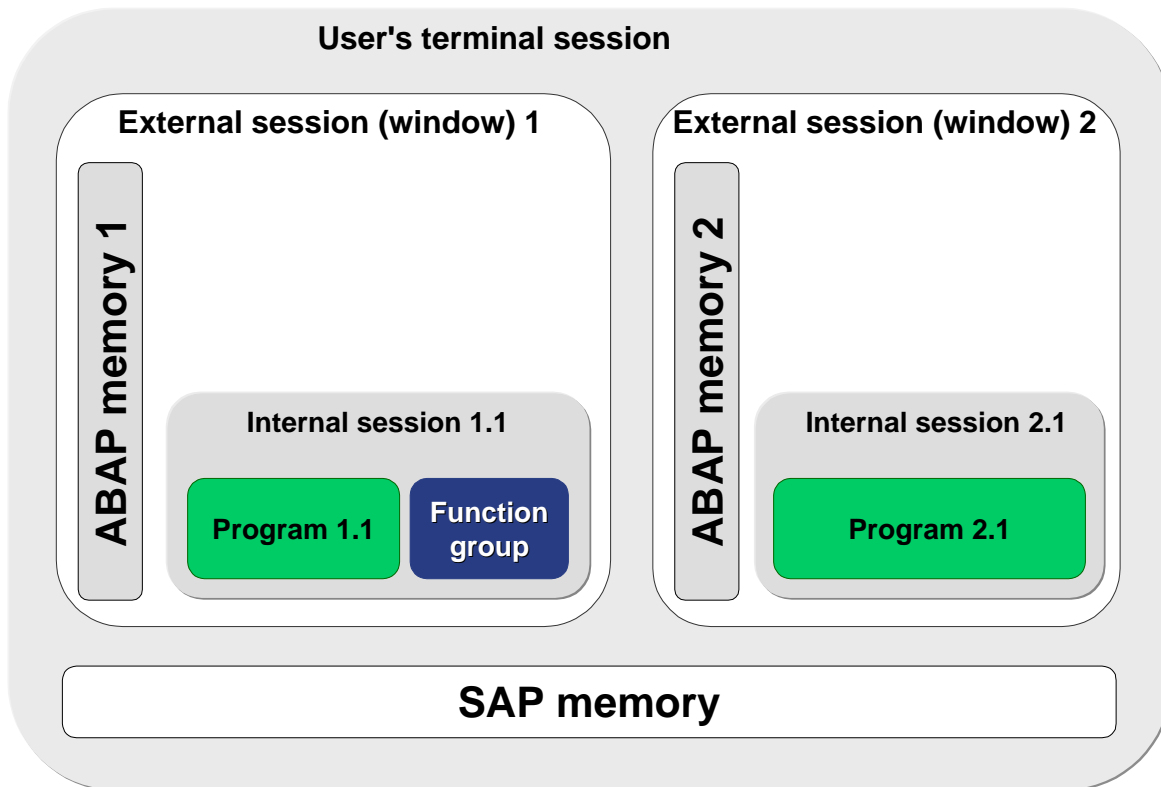
The system creates a new internal session, which contains the program context of the called program. After the call, the ABAP memory is **reset**.





© SAP AG 1999

When you call a function module, the ABAP runtime system checks whether you have already called a function module from the same function group in the current program.



© SAP AG 1999

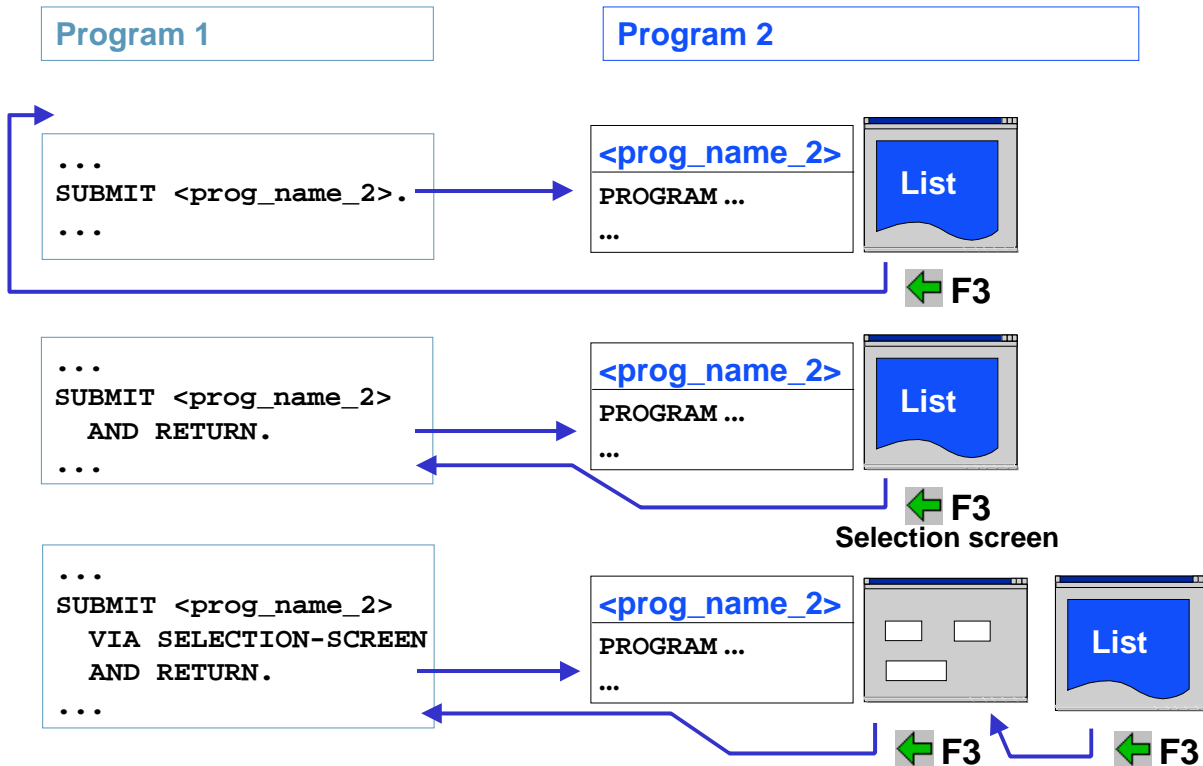
- If this is not the case, the system loads the relevant function group **into the internal session of the calling program**. Its global data is initialized and the **LOAD-OF-PROGRAM** event is triggered.
- If your program had already used a function module from the same function group before the call, the function group is already resident in the internal session, and the new call can access the same global data. We say that the function group **remains active until the end of the program that called it**.

The data is only visible in the corresponding program - each program can only address its own data, even if there are identically-named objects in both programs. The same applies when the stack is extended. If a program is added to the stack that calls a function module from a function group already called by another program, the function group is **loaded again** into the **new internal session**. The system creates **new** copies of its data objects, initializes them, and, as before, they are only visible within the function group, and only in the internal session in which the function group was loaded.

The graphic shows the first call to a function module in a particular function group.

## Starting an Executable (Type 1) Program

SAP



© SAP AG 1999

To start an executable (type 1) program, use the **SUBMIT** statement.

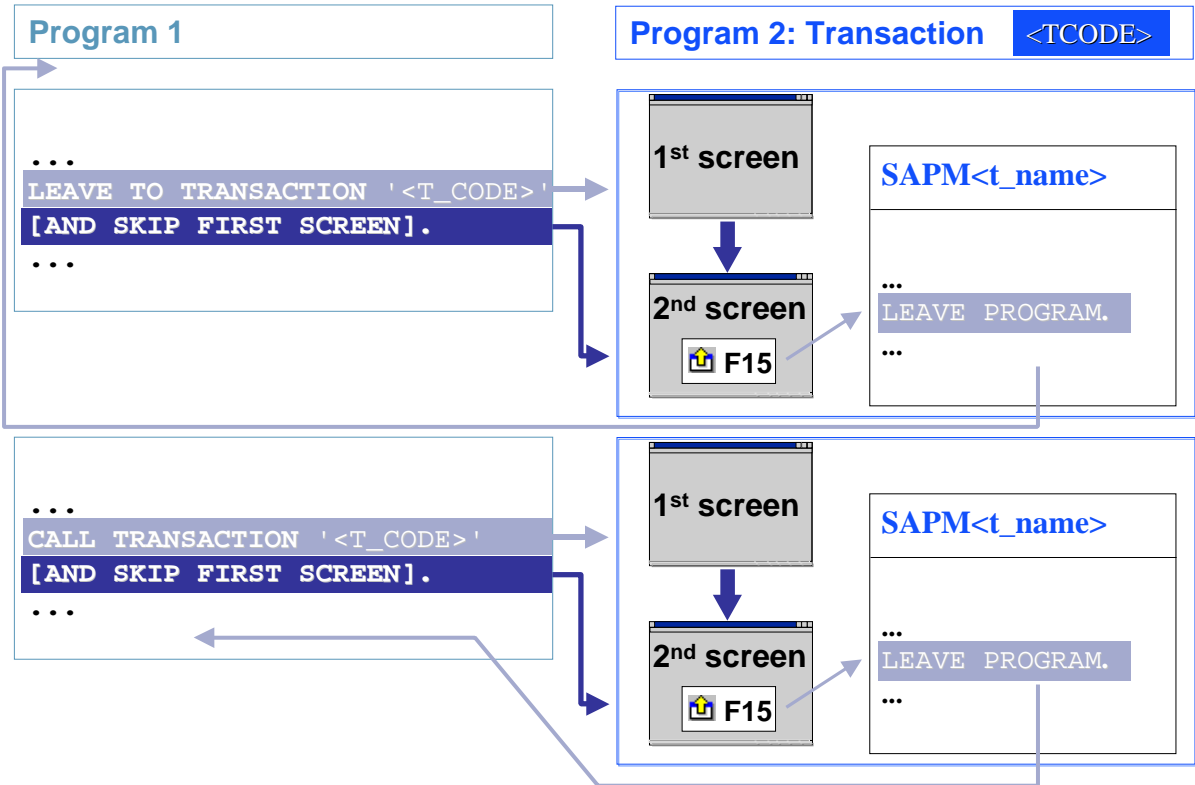
If you use the **VIA SELECTION-SCREEN** addition, the system displays the standard selection screen of the program (if one has been defined).

If you use the **AND RETURN** addition, the system resumes processing with the first statement after the **SUBMIT** statement once the called program has finished.

For further information, refer to the documentation for the **SUBMIT** statement.

# Calling a Transaction

SAP



© SAP AG 1999

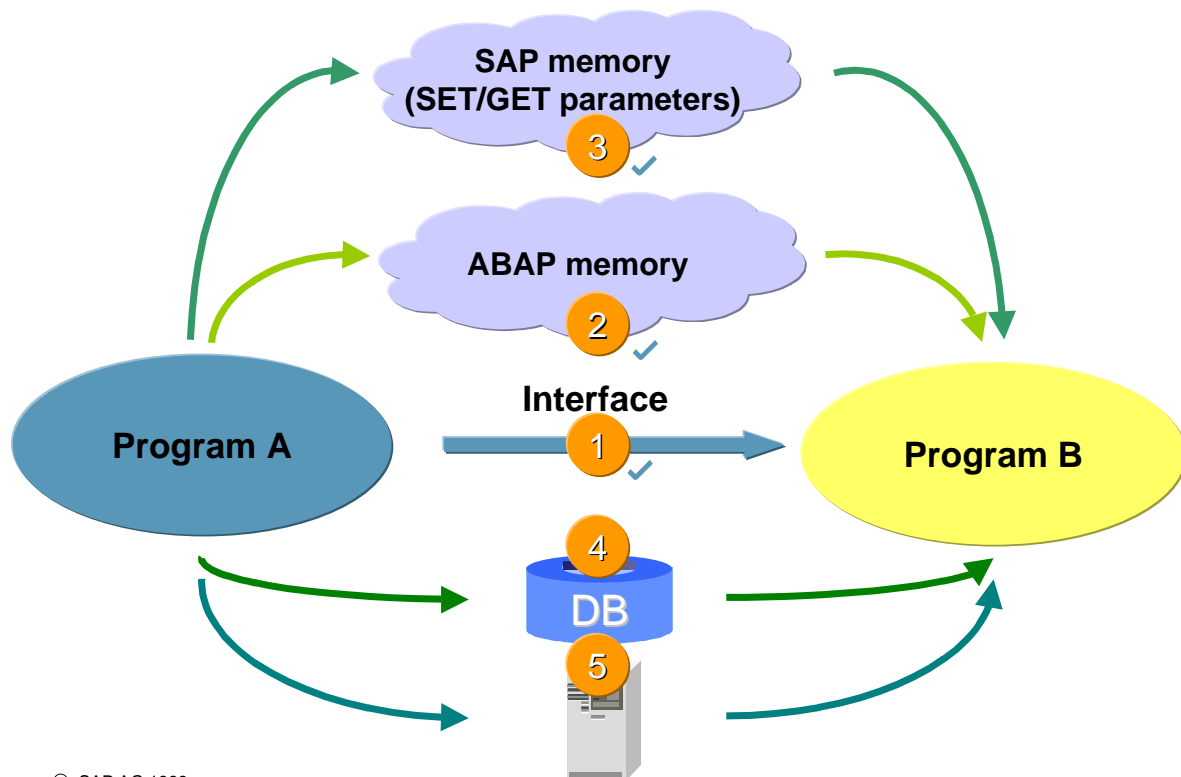
When you use the **LEAVE TO TRANSACTION '<T\_CODE>'** statement, the system terminates the current program and starts the transaction with transaction code <T\_CODE>. The statement is the equivalent of entering /n<T\_CODE> in the command field.

**CALL TRANSACTION '<T\_CODE>'** allows you to insert an ABAP program with a transaction code into the call chain.

To terminate an ABAP program, use the **LEAVE PROGRAM** statement. If the statement occurs in a program that you called using **CALL TRANSACTION '<T\_CODE>'** or **SUBMIT <prog\_name> AND RETURN**, the system resumes processing at the next statement after the call in the calling program. In all other cases, the user returns to the application menu from which he or she started the program.

If you use the **...AND SKIP FIRST SCREEN** addition, the system **does not** display the **screen contents** of the first screen in the transaction. However, it **does** process the flow logic.

If you started a transaction using **CALL TRANSACTION** that uses update techniques, you can use the **UPDATE...** addition to specify the update technique (asynchronous (default), synchronous, or local) that the program should use. For further information, refer to course **BC414 (Programming Database Updates)** and the online documentation.



© SAP AG 1999

There are various ways of passing data to programs running in separate internal sessions:

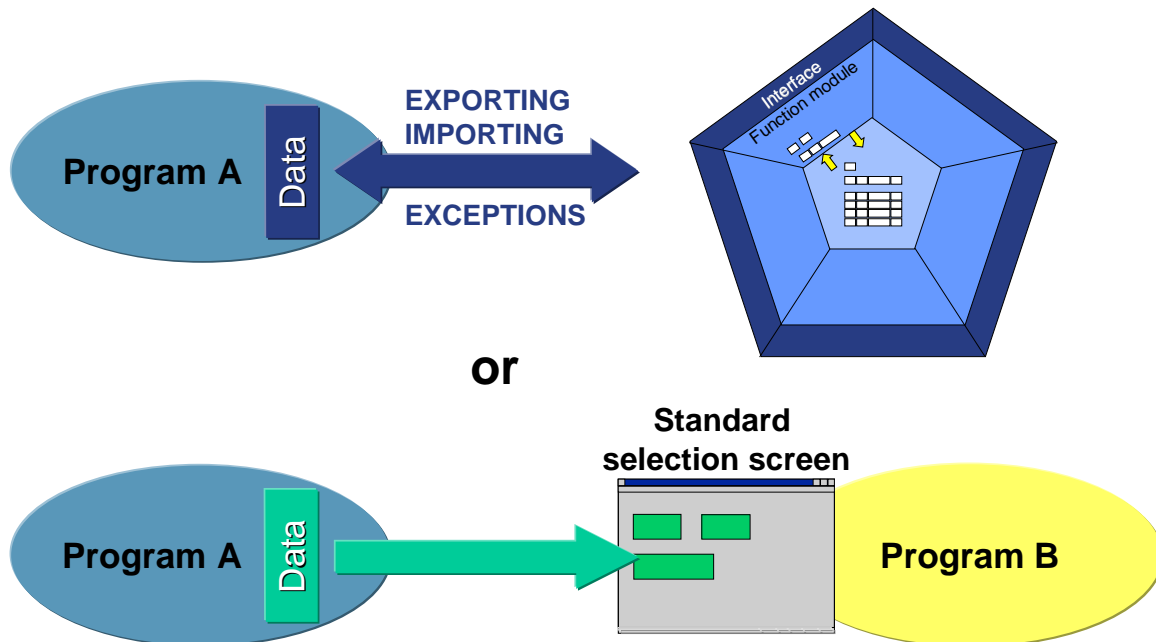
You can use

- ① The interface of the called program (usually a standard selection screen)
- ② ABAP memory
- ③ SAP memory
- ④ Database tables
- ⑤ Local files on your presentation server

The following pages illustrate methods ①, ② and ③.

For further information about passing data using database tables or the *shared buffer*, refer to the documentation for the **EXPORT** and **IMPORT** statements.

For further information about transferring data between an ABAP program and your presentation server, refer to the documentation of function modules **WS\_UPLOAD** and **WS\_DOWNLOAD**.



© SAP AG 1999

Function modules have an interface that the calling program and the function module use to exchange data. Subroutines also use a similar technique. Certain restrictions apply to the interfaces of remote-enabled function modules.

When you call ABAP programs that have a standard selection screen, you can pass data for the input fields in the call. There are two ways to do this:

- By specifying a variant for the selection screen when you call the program.
- By specifying values for the input fields when you call the program.

## Passing Values for Input Fields

SAP

```
DATA <set> {TYPE|LIKE} RANGE OF {<type>|<dataobject>}.
```

```
SUBMIT <prog_name> AND RETURN [VIA SELECTION-SCREEN]
```

```
WITH <parameter> {EQ|NE|...} <val>
```

```
WITH <sel_opt> {EQ|NE|...} <val> SIGN {'I'|'E'}
```

```
WITH <sel_opt> BETWEEN <val1> AND <val2> SIGN {'I'|'E'}
```

```
WITH <sel_opt> NOT BETWEEN <val1> AND <val2> SIGN {'I'|'E'}
```

```
WITH <sel_opt> IN <set>
```

```
... .
```

```
MODULE user_command_0200 INPUT.
```

```
...
```

```
CASE save_ok.
```

```
WHEN 'COORDFR'.
```

```
SUBMIT sapbc402_tabd hashed
```

```
WITH pa_city = sdyn_conn-cityfrom
```

```
WITH pa_ctry = sdyn_conn-countryfr
```

```
AND RETURN.
```

```
...
```

```
ENDCASE.
```

```
ENDMODULE.
```

```
" USER_COMMAND_0200 INPUT
```

### Insert pattern

© SAP AG 1999

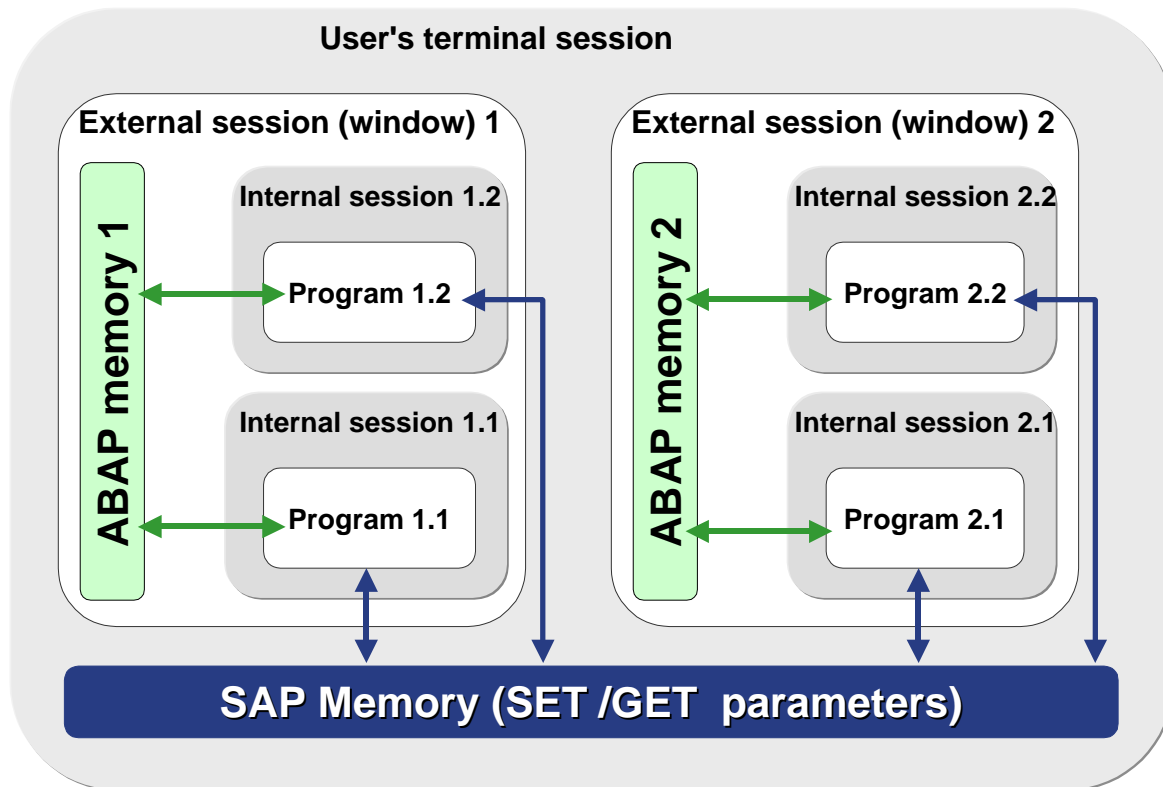
The **WITH** addition in the **SUBMIT** statement allows you to assign values to the fields on a standard selection screen. The abbreviations "EQ, NE, ... , I, E" have the same meanings as with select-options. If you want to pass several selections to a selection option, you can use the **RANGES** statement instead of individual **WITH** additions. The **RANGES** statement creates a selection table, which you can fill as though it were a selection option. You then pass the whole table to the executable program.

If you want to display the standard selection screen when you call the program, use the **VIA SELECTION-SCREEN** addition.

When you use the **SUBMIT** statement, use the **Pattern** function in the ABAP Editor to insert an appropriate statement pattern for the program you want to call. It automatically supplies the names of the parameters and selection options that are available on the standard selection screen.

The example shown above is an extract from transaction **BC402\_CALD\_CONN**. When the user requests the coordinates of a city, the **executable program SAPBC402\_TABD\_HASHED** is called. The parameters are filled with the city and country code from the transaction. The standard selection screen **does not** appear.

For further information about working with variants and about other syntax variants of the **WITH** addition, refer to the documentation for the **SUBMIT** statement.



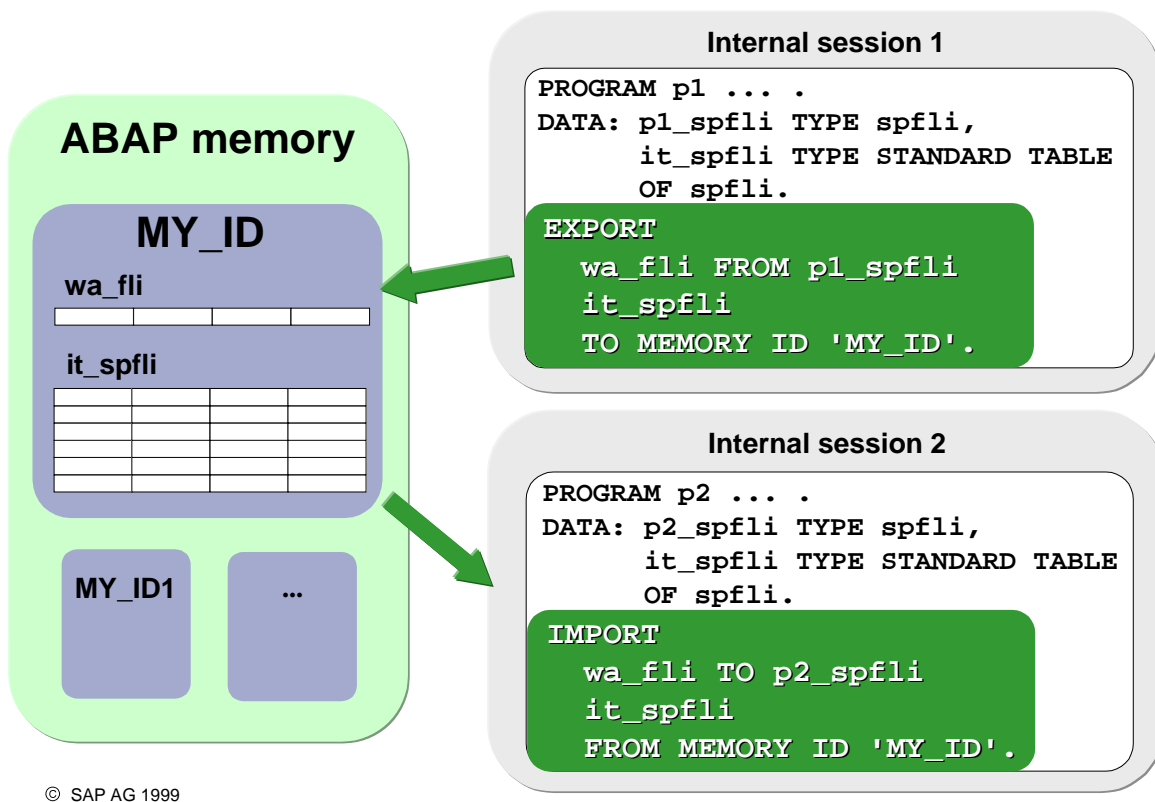
© SAP AG 1999

To pass data between programs, you can use either the SAP memory or the ABAP memory.

- **SAP memory** is a user-specific memory area that you can use to store **field values**. It is only of limited value for passing data between internal sessions. Values in SAP memory are retained for the duration of the user's terminal session. The memory can be used **between sessions** in the same terminal session. You can use the contents of SAP memory as default values for screen fields. All **external sessions** can use the SAP memory.
- ABAP memory is also user-specific. There is a local ABAP memory for each external session. You can use it to exchange any ABAP variables (fields, structures, internal tables, complex objects) between the **internal sessions** in any one **external session**.

When the user exits an external session (**/i** in the command field), the corresponding ABAP memory is automatically initialized or released.





Use the **EXPORT ... TO MEMORY** statement to copy any number of ABAP variables with their current values (data cluster) to ABAP memory. The **ID...** addition (maximum 32 characters long) enables you to identify different clusters.

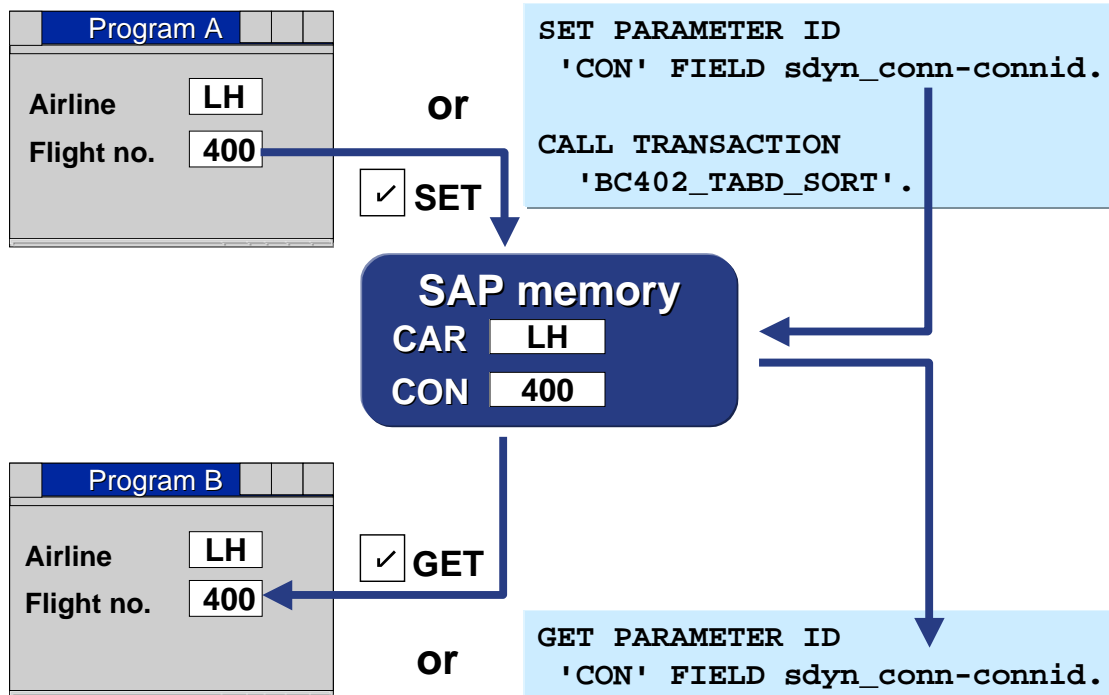
If you use a new **EXPORT TO MEMORY** statement for an existing data cluster, the new one will overwrite the old.

The **IMPORT... FROM MEMORY ID...** statement allows you to copy data from ABAP memory into the corresponding fields of your ABAP program. In the **IMPORT** statement, you can also restrict the selection to a part of the data cluster.

The variables into which you want to read data from the cluster in ABAP memory must have the same types in both the exporting and the importing programs.

To release a data cluster, use the **FREE MEMORY ID...** statement.

Remember when you call programs using transaction codes that you can only use the ABAP memory to pass data **to** the transaction.



© SAP AG 1999

You can define memory areas (parameters) in the SAP memory in various ways:

- By creating input/output fields with reference to the ABAP Dictionary. These take the parameter name of the data element to which they refer.

Alternatively, you can enter a name in the attributes of the input/output fields. Then, you can also choose whether the entries from the field should be transferred to the parameter (SET), or whether the input field should be filled with the value from the parameter (GET).

To find out about the names of the parameters assigned to input fields, display the field help for the field (F1), then choose *Technical info*.

- You can also fill a memory area directly using the statement

**SET PARAMETER ID '<PAR\_ID>' FIELD <var>.**

and read it using the statement

**GET PARAMETER ID '<PAR\_ID>' FIELD <var>.**

- You can also define parameters using the **Object Navigator** and fill them with values.

The example shown here is an extract from transaction **BC402\_CALD\_CONN**. When the user maintains the flight times, the program calls transaction **BC402\_TABD\_SORT**. The name of the airline is passed using parameter **CAR** (using a statement). The flight number is passed using parameter **CON** (SET option selected for the field in the Screen Painter).

## Preview: Passing Data Using an Internal Table

SAP

### Program 1

```
...
DATA:
  <bi_itab> TYPE TABLE OF bdcdata,
  <bi_wa>   TYPE bdcdata.

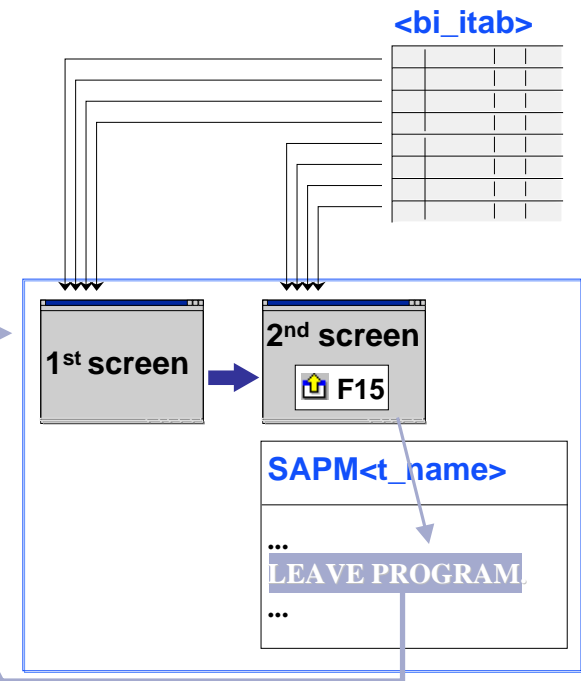
* fill <bi_itab>
...

* call other program
CALL TRANSACTION '<T_CODE>'
  USING <bi_itab>.
IF sy-subrc = 0.
...
ELSE.
...

```

### Program 2: Transaction

<T\_CODE>



© SAP AG 1999

When you call a transaction using the statement **CALL TRANSACTION '<T\_CODE>' USING <bi\_itab>...** you can run the transaction <T\_CODE> using the values from <bi\_itab> in the screen fields. **The internal table must have the structure bdcdata.**

The **MODE** addition allows you to specify whether the screen contents should all be displayed ('A' - the default setting), only when an error occurs ('E'), or not at all ('N'). You can use the **MESSAGE INTO <mess\_itab>** to specify an internal table into which any system messages should be written. **The internal table must have the structure bdcmsgcoll.**

You can find out if the transaction was executed successfully from the system field **sy-subrc**.

You might use this technique if

- You are processing in the foreground, but the input fields have not been filled using GET parameters,
- You want to process the transaction invisibly. In this case, you normally have to pass the function codes in the table as well.

This technique is also one of the possible ways of transferring data from non-SAP systems. When you do this, the internal table with the structure **bdcdata** must be filled **completely**.

## Fields in the Global Type BDCDATA



Field name:	→ program	dynpro	dynbegin	fnam	fval
Length:	→ 40	4	1	132	132
Meaning:	→ Program name	Screen number	First record	Field name	Field value
Filled:	→ Only for first record of new screen	Only for first record of new screen	'X' for first record on screen, otherwise ''		Upper-/lowercase!

© SAP AG 1999

Filling the internal table in batch input format:

- Each screen that you want to process automatically in the transaction must be identified by a line in which only the fields **program**, **dynpro**, and **dynbegin** are filled.
- After the record that identifies the screen, use a new **bdcdata** record for each field you want to fill. These records use the fields **fnam** and **fval**. You can fill the following fields:
  - Input/output fields (with data)
  - The command field **bdc\_okcode**, (with a function code)
  - The cursor positioning field, **bdc\_cursor** (with a field name)

For information about how to use this technique for **data transfer**, refer to course **BC420 (Data Transfer)** or the online documentation.

## Example: Passing Data Using an Internal Table

SAP

program	dynpro	dynbegin	fnam	fval
SAPBC402_CALD_CREATE_CUSTOMER	0100	X		
			SCUSTOM-NAME	<current_name>
			SCUSTOM-CITY	<current_city>
			BDC_OKCODE	SAVE

```
DATA:
  wa_bdcdata TYPE bdcdata,
  it_bdcdata LIKE TABLE OF wa_bdcdata.

*** fill the bdcdata-table ...

CALL TRANSACTION 'BC402_CALD_CRE_CUST'
  USING it_bdcdata
  MODE 'N'.

IF sy-subrc <> 0.
  MESSAGE ... WITH sy-subrc.
ENDIF.
```

© SAP AG 1999

Save

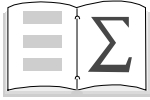
Indicates entry in  
command field

The above example refers to transaction **BC402\_FMDD\_FG**. When the user creates a new customer entry, the program calls transaction **BC402\_CALD\_CRE\_CUST**. This transaction has not implemented import from ABAP memory, and its input fields are not set as GET parameters. The customer data is therefore passed using an internal table and processed invisibly.

If the operation is successful, the new customer record can be entered in the waiting list.

The filled internal table in **bdcdata** format is illustrated above. **At runtime**, <current\_name> stands for the customer name from the input field, <current\_city> stands for the city.

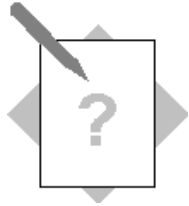
You use the field **BDC\_OKCODE** to address the command field, into which you enter the function code that would have been triggered by the user choosing a function key, pushbutton, or menu entry in dialog mode (or by entering a code directly in the command field).



**You are now able to:**

- **Describe the R/3 memory model,**
- **Call executable programs**
- **Call transactions**
- **Use the various memory areas to pass data**

## Calling Programs and Passing Data: Exercises

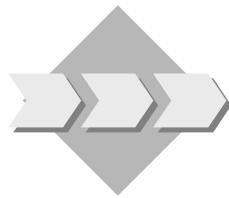


**Unit:** Calling Programs and Passing Data:  
**Topic:** Calling an executable program



At the conclusion of these exercises, you will be able to:

- call an executable program from another program
- Preassign values to its selection options



You are a programmer for an airline consortium, and it is your job to write analysis programs for several airlines.

1. Extend your program from task 3 of the previous exercises:  
The user should be able to display a list of all sales counters of the airlines entered on the selection screen, as long as he or she is authorized to see the data.  
**## is your two-digit group number.**  
Model solution:  
SAPBC402\_CALS\_FLIGHTLIST5
  - 1-1 Copy your solution from the last exercise, or the model solution.  
New name: **Z##\_BC402\_FLIGHTLIST5**.
  - 1-2 For simplicity, the function should be triggered when the user double-clicks a line or single-clicks it and presses <F2>. Add an appropriate event block to your program.
  - 1-3 Program a call to your executable program **Z##\_BC402\_COUNTLIST2** or the model solution **SAPBC402\_TABS\_COUNTLIST2** (use the *Pattern* function).  
Ensure that the standard selection screen of the called program is not displayed. Once the program has finished running, the user should be able to return to the original program.

## Calling Programs and Passing Data: Solutions



Unit: Calling Programs and Passing Data:  
Topic: Calling an executable program

### 1 Model solution SAPBC402\_CALS\_FLIGHTLIST5

```
*&-----*
*& Report SAPBC402_CALS_FLIGHTLIST5 *
*& *
*&-----*
*& solution of exercise 1 *
*& calling programs and transmitting data *
*&-----*
```

REPORT sapbc402\_cals\_flightlist5.

...  
...

\* for authority-check:  
\*\*\*\*\*

#### DATA:

allowed\_carriers TYPE RANGE OF t\_flight-carrid,  
wa\_allowed\_carr LIKE LINE OF allowed\_carriers.

START-OF-SELECTION.

\* fill a range table with the allowed carriers:  
\*\*\*\*\*

...  
...

\* fill an internal table with connection and flight data  
\* for the allowed carriers:  
\*\*\*\*\*

...  
...

\* fill all the inner internal tables with alternate planetypes:  
\*\*\*\*\*

...  
...

PERFORM display\_flights CHANGING it\_flights.



## AT LINE-SELECTION.

\* display list of counters for all allowed carriers:

\*\*\*\*\*

**SUBMIT sapbc402\_tabs\_counterlist2**

**WITH so\_carr IN allowed\_carriers AND RETURN.**

\*-----\*

\*     FORM display\_flights

\*-----\*

\* --> p\_it\_flights

\*-----\*

FORM display\_flights CHANGING p\_it\_flights TYPE t\_flighttab.

...

ENDFORM.

This document was created with Win2PDF available at <http://www.daneprairie.com>.  
The unregistered version of Win2PDF is for evaluation or non-commercial use only.