

BC400

Introduction to the ABAP Workbench

mySAP Technology

Date _____
Training Center _____
Instructors _____

Education Website _____

Participant Handbook

Course Version: 2003 Q2
Course Duration: 5 Day(s)
Material Number: 50061840



An SAP course - use it to learn, reference it for work

Copyright

Copyright © 2003 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Trademarks

- Microsoft®, WINDOWS®, NT®, EXCEL®, Word®, PowerPoint® and SQL Server® are registered trademarks of Microsoft Corporation.
- IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, MVS/ESA®, RS/6000®, AIX®, S/390®, AS/400®, OS/390®, and OS/400® are registered trademarks of IBM Corporation.
- ORACLE® is a registered trademark of ORACLE Corporation.
- INFORMIX®-OnLine for SAP and INFORMIX® Dynamic ServerTM are registered trademarks of Informix Software Incorporated.
- UNIX®, X/Open®, OSF/1®, and Motif® are registered trademarks of the Open Group.
- Citrix®, the Citrix logo, ICA®, Program Neighborhood®, MetaFrame®, WinFrame®, VideoFrame®, MultiWin® and other Citrix product names referenced herein are trademarks of Citrix Systems, Inc.
- HTML, DHTML, XML, XHTML are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.
- JAVA® is a registered trademark of Sun Microsystems, Inc.
- JAVASCRIPT® is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.
- SAP, SAP Logo, R/2, RIVA, R/3, SAP ArchiveLink, SAP Business Workflow, WebFlow, SAP EarlyWatch, BAPI, SAPPHIRE, Management Cockpit, mySAP.com Logo and mySAP.com are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other products mentioned are trademarks or registered trademarks of their respective companies.

Disclaimer

THESE MATERIALS ARE PROVIDED BY SAP ON AN "AS IS" BASIS, AND SAP EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR APPLIED, INCLUDING WITHOUT LIMITATION WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THESE MATERIALS AND THE SERVICE, INFORMATION, TEXT, GRAPHICS, LINKS, OR ANY OTHER MATERIALS AND PRODUCTS CONTAINED HEREIN. IN NO EVENT SHALL SAP BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR PUNITIVE DAMAGES OF ANY KIND WHATSOEVER, INCLUDING WITHOUT LIMITATION LOST REVENUES OR LOST PROFITS, WHICH MAY RESULT FROM THE USE OF THESE MATERIALS OR INCLUDED SOFTWARE COMPONENTS.

About this Handbook

This handbook is intended to complement the instructor-led presentation of this course, and serve as a source of reference. It is not suitable for self-study.

Typographic Conventions

The following typographic conventions are used in this guide.

Type Style	Description
<i>Example text</i>	Words or characters that appear on the screen. These include field names, screen titles, pushbuttons as well as menu names, paths, and options. Also used for cross-references to other documentation both internal (in this documentation) and external (in other locations, such as SAPNet).
Example text	Emphasized words or phrases in body text, titles of graphics, and tables
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example SELECT and INCLUDE.
Example text	Screen output. This includes file and directory names and their paths, messages, names of variables and parameters, and passages of the source text of a program.

Type Style	Description
Example text	Exact user entry. These are words and characters that you enter in the system exactly as they appear in the documentation.
<Example text>	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.

Icons in Body Text

The following icons are used in this handbook.

Icon	Meaning
	For more information, tips, or background
	Note or further explanation of previous point
	Exception or caution
	Procedures
	Indicates that the item is displayed in the instructor's presentation.

Contents

Course Overview.....	vii
Course Goals.....	vii
Course Objectives	viii
Unit 1: Flow of an ABAP Program	1
System Architecture and ABAP Program Using Example of Selection Screen and List	2
Unit 2: Introduction to the ABAP Workbench.....	17
<i>Repository and Object Navigator</i>	18
Developing Programs and Organizing Developments	28
Unit 3: Introduction to the ABAP Dictionary	55
Transparent Tables and Views	56
Unit 4: Basic ABAP Language Elements.....	75
Working with Elementary Data Objects.....	76
Working with Structures.....	99
Working with Internal Tables	111
Unit 5: Data Retrieval	133
Reading Database Tables	134
Authorization Check	161
Unit 6: Internal Program Modularization	175
Subroutines	176
Unit 7: Introduction to Event Blocks.....	191
Event Blocks.....	192
Unit 8: User dialogs.....	199
List	201
Selection Screen.....	223
Screen.....	245

User interface	295
Unit 9: Outlook: Developing Internet Applications	317
Developing Internet Applications	318
Unit 10: Cross-Program Modularization.....	331
Working with Function Modules.....	332
Working with Methods.....	349
Working with BAPIs	369
Unit 11: Software Logistics and Software Adjustment	377
Outlook: Organization of Development and Transport.....	378
Outlook: Enhancing the Standard Software.....	386
Index.....	395

Course Overview

There are two aspects to this course: The SAP programming language, ABAP Objects; and its associated development environment, the *ABAP Workbench*. In both cases, we will be focusing on concepts and fundamental principles. We also introduce the appropriate terminology, so that you will find it easier to understand the in-depth documentation.

We always handle these topics using practical application examples, so that you can immediately implement what you have learnt here. Above all, it should also be clear how little effort is needed, using the *ABAP Workbench*, to create high-performance business applications rapidly.

We are convinced that this course enables you to start developing immediately and provides the knowledge you will need to focus on the essentials in subsequent courses.

That is why this course is a prerequisite for all more in-depth *ABAP Workbench* programming courses.

Target Audience

This course is intended for the following audiences:

- Project team members
- ABAP Objects: Programmers

Course Prerequisites

Required Knowledge

- Programming knowledge
- SAPTEC (Basics of the mySAP Technology solution)



Course Goals

This course will prepare you to:

- Understand and use basic ABAP Objects syntax elements
- Implement different types of user dialog
- Program read accesses to the database
- Use the ABAP Workbench development tools
- Understand how developments are organized and transported



Course Objectives

After completing this course, you will be able to:

- Create an ABAP Objects program containing user dialogs and database accesses
- Describe the different type of development objects and their typical intended purposes
- Create basic examples for all those Repository objects introduced in the course using the appropriate tools

SAP Software Component Information

The information in this course pertains to the following SAP Software Components and releases:

Unit 1

Flow of an ABAP Program

Unit Overview

Refer to the lesson summaries for an overview of this unit.



Unit Objectives

After completing this unit, you will be able to:

- Basically describe the architecture of the *SAP Web Application Server*
- Basically describe how a simple dialog program is executed by the ABAP runtime system

Unit Contents

Lesson: System Architecture and ABAP Program Using Example of Selection Screen and List.....	2
--	---

Lesson: System Architecture and ABAP Program Using Example of Selection Screen and List

Lesson Overview

In this lesson you will learn how a simple dialog program is executed by the *SAP Web AS 6.20*.



Lesson Objectives

After completing this lesson, you will be able to:

- Basically describe the architecture of the *SAP Web Application Server*
- Basically describe how a simple dialog program is executed by the ABAP runtime system

Business Example

You are to explain the basics of the *SAP Web AS 6.20* architecture to a colleague and also how ABAP Objects programs run.

System Architecture and ABAP Program

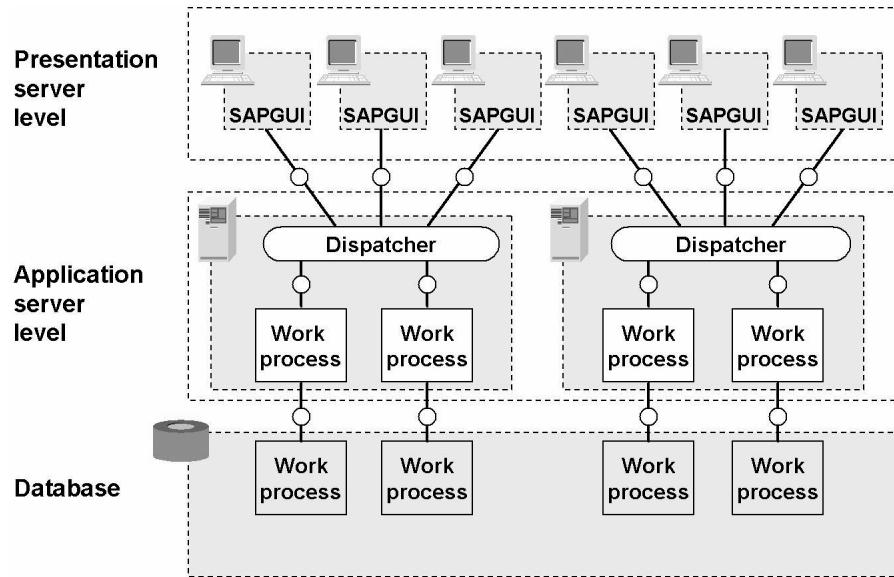


Figure 1: Client-Server Architecture of the *SAP Web AS 6.20*

The *SAP Web Application Server* has a modular architecture that follows the **software-oriented** client/server principle.

In the *SAP Web Application Server*, presentations, application logic, and data storage can be assigned to different systems. This serves as the basis for the **scalability** of the system.

The lowest level is the **database level**. Here data is managed with the help of a relational database management system (RDBMS). This data includes, apart from application data, the programs and the metadata that the system requires for self-management.

The ABAP programs run at the **application server level**, that is, both the applications provided by SAP and the ones you develop yourself. The ABAP programs read data from the database level, process the data, and possibly also store data there.

The third level is the **presentation server level**. This level contains the user interface through which each user can access the program, enter new data, and receive the results of a work process.

The technical distribution of software is independent of its physical location on the hardware. Vertically, all levels can be installed on top of each other on one computer or each level on a separate computer. Horizontally, the presentation and application servers can be divided among any number of computers. The horizontal distribution of database components, however, depends on the type of database installed.

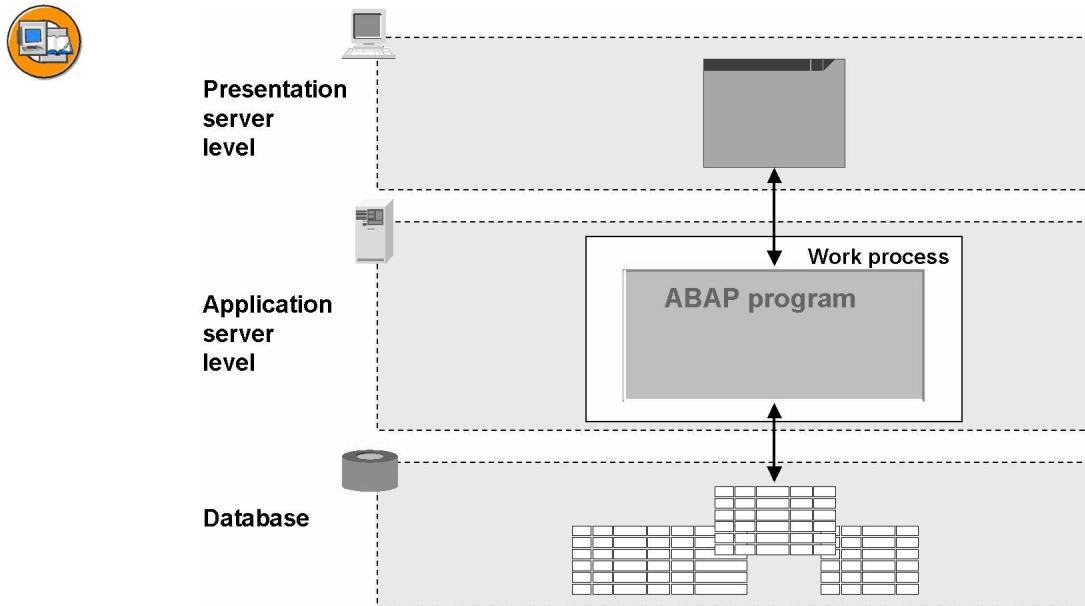


Figure 2: Excerpt for an ABAP Program

This graphic can be simplified for most topics discussed during this course. The interaction between **one** user and **one** ABAP program will be of primary interest to us during this course.

The exact processes involved in user dispatching on an application server are secondary to understanding how to write an ABAP program. Therefore, we will be working with a simplified graphic that does not explicitly show the dispatcher and the work process. Certain graphics will, however, be enhanced to include these details whenever they are relevant to ABAP programming.

ABAP programs are processed on the application server. The design of **user dialogs** and **database accesses** is therefore of particular importance when writing application programs.

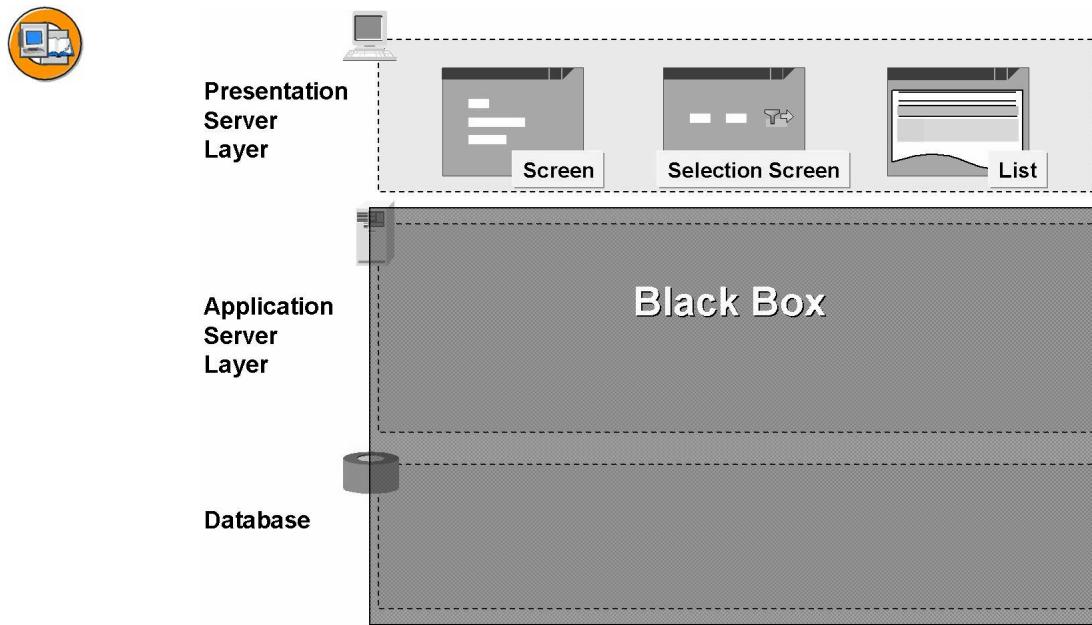


Figure 3: View for the User

The user is only interested in how his or her business transaction flows and how data can be entered and output within the transaction. The technical aspects of programming are less interesting for the user. The user does not need to know the precise flow of the ABAP program on the application server. To the average user, the SAP System is like a "Black Box".

Technically speaking, however, there are three different types of screens: standard screens, selection screens, and lists. Each type of screen provides different services to the user. Here the task of the developer is to select for each user dialog exactly those types that are most suitable for the accomplishment of the tasks concerned.

For this purpose, exact knowledge of the technical aspects is very helpful.

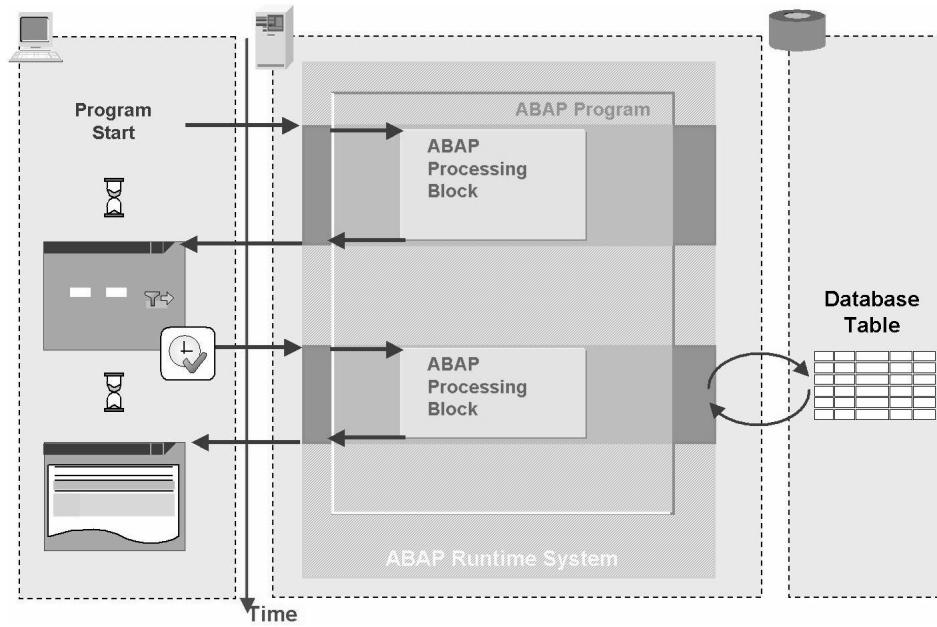


Figure 4: Interaction Between Server Layers

Once the user performs a user action (choosing Enter, a function key, a menu function, or a pushbutton, for example), control is passed from the presentation server to the application server.

Only certain parts of the ABAP program are processed.

If a further user dialog is triggered from within the ABAP program, the system transmits the screen, and control is once again passed to the presentation server.

Sample Program with Selection Screen and List

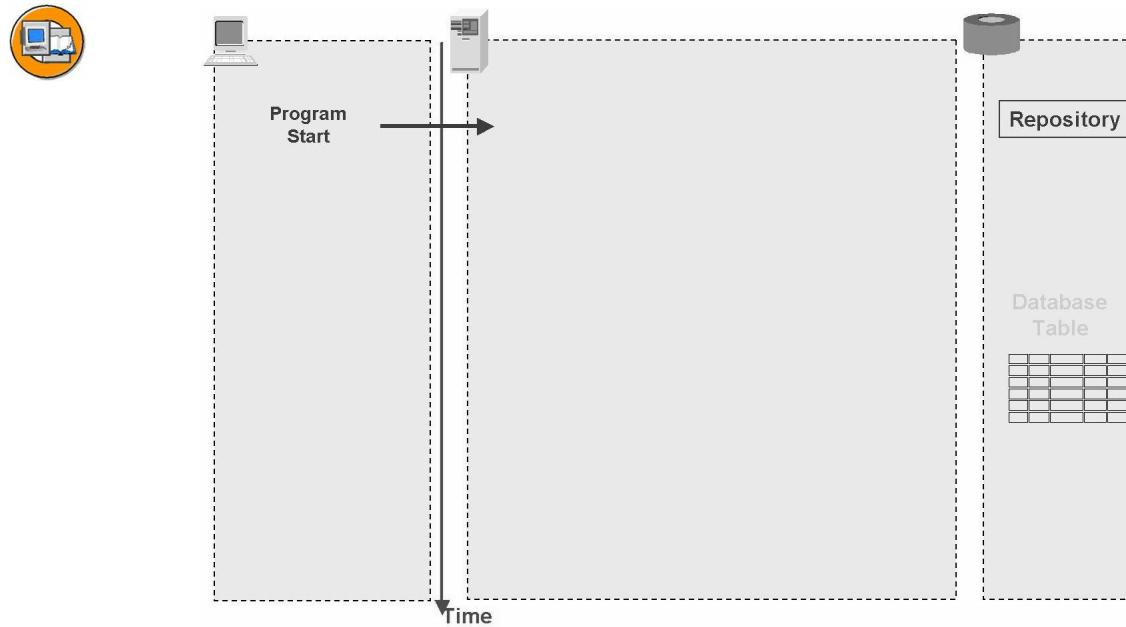


Figure 5: Program Start

Whenever a user logs on to the system, a screen is displayed. From this screen, the user can start an ABAP program through the menu path.

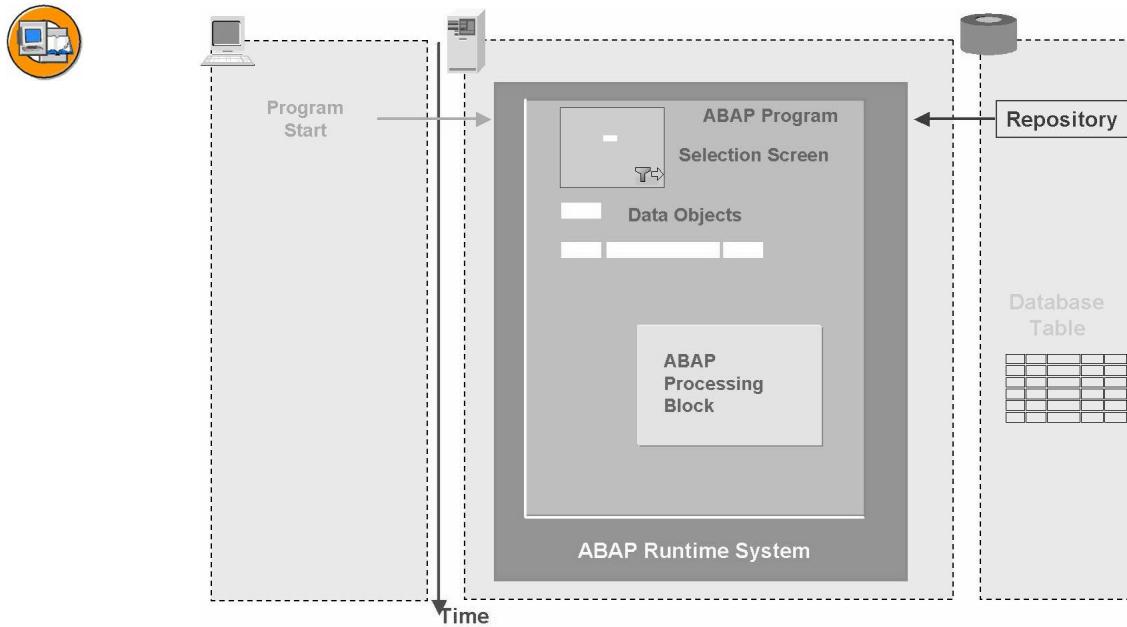


Figure 6: System Loads Program Context

In this case, the system first loads the program context onto the application server. The program context contains memory areas for variables and complex data objects, information on the screens for user dialogs, and ABAP processing blocks. The runtime system gets all this program information from the *Repository*, which is a special part of the database.

The sample program has a selection screen as the user dialog, a variable and a structure as data objects, and one ABAP processing block. The list that is used to display the data is created dynamically at runtime.

The ABAP runtime system controls the subsequent program flow.

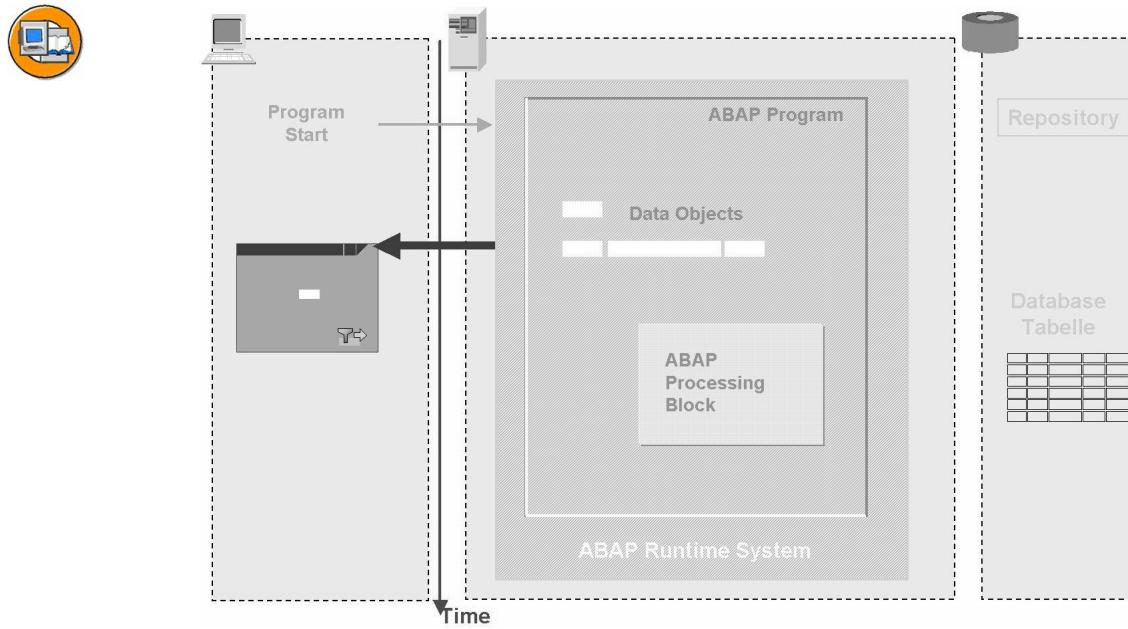


Figure 7: Runtime System Sends Selection Screen

Since the program contains a selection screen, the ABAP runtime system then sends it to the presentation server. The presentation server controls the program flow for as long as the user enters data in the input fields.

Selection screens allow users to enter selection criteria required by the program for it to continue.

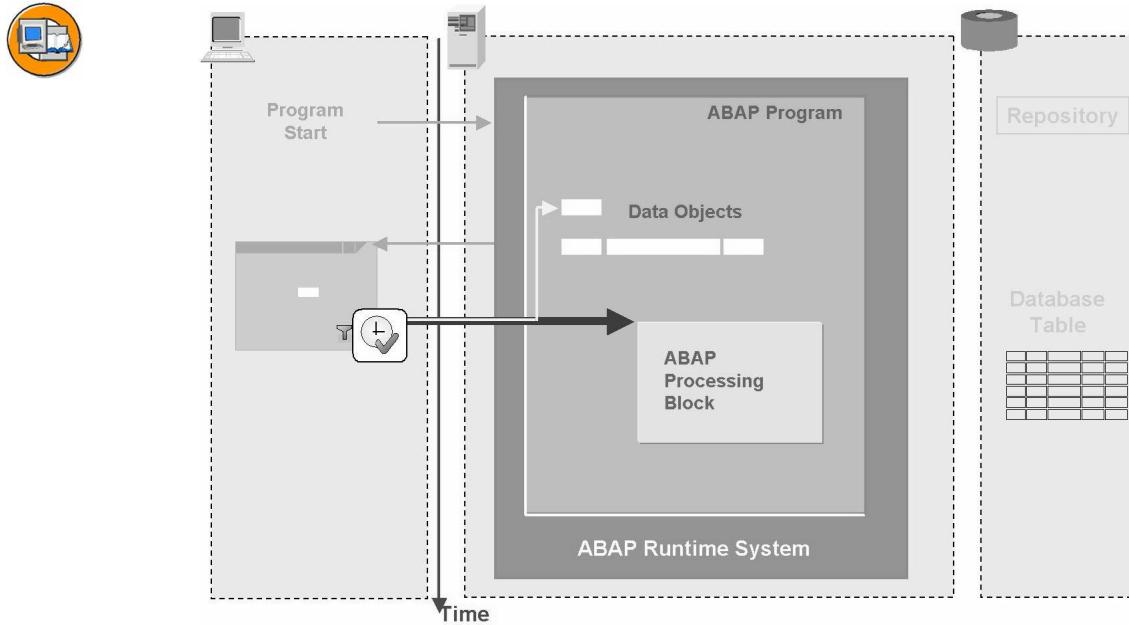


Figure 8: Input Values Are Inserted into Data Objects

As soon as the user has finished entering data on the selection screen, he or she can trigger further processing of the program by choosing *Execute*.

The entered data is then automatically placed in its corresponding data objects in the program and the ABAP runtime system resumes control of processing.

In our simple program example there is only one ABAP processing block. The ABAP runtime system triggers sequential processing of this ABAP processing block.

If the entries made by the user do not have the correct type, then an error message is **automatically** triggered. The user must correct his/her entries.

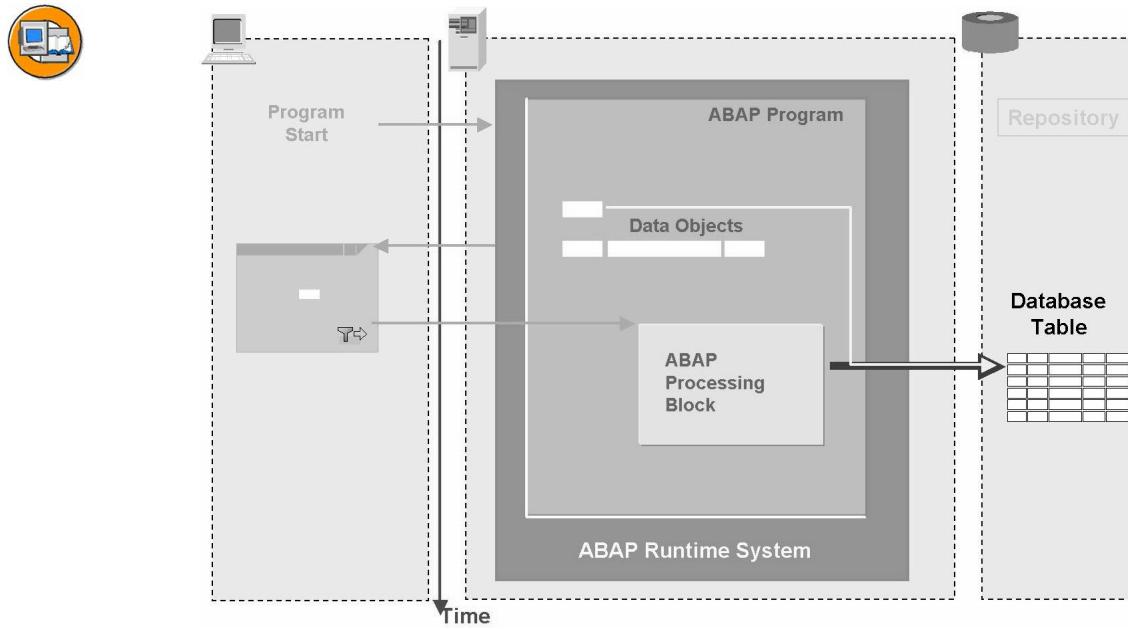


Figure 9: Program Requests Data Record from Database

In the example program, read access to the database is programmed in the ABAP processing block. Therefore, information about which database table is to be accessed and which row in the table is to be read is passed to the database.

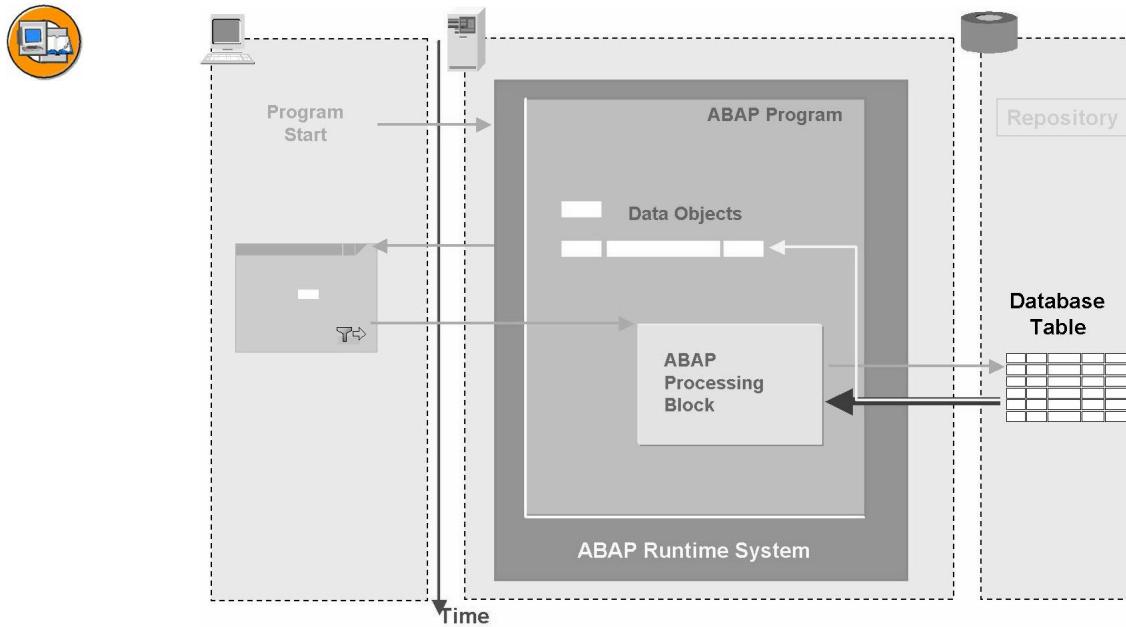


Figure 10: Database Returns Data Record to Program

The database returns the requested data record to the program and the runtime system ensures that this data is placed in the appropriate data objects.

If a single record is accessed, this data object is usually a structure that contains components for all the required database fields.

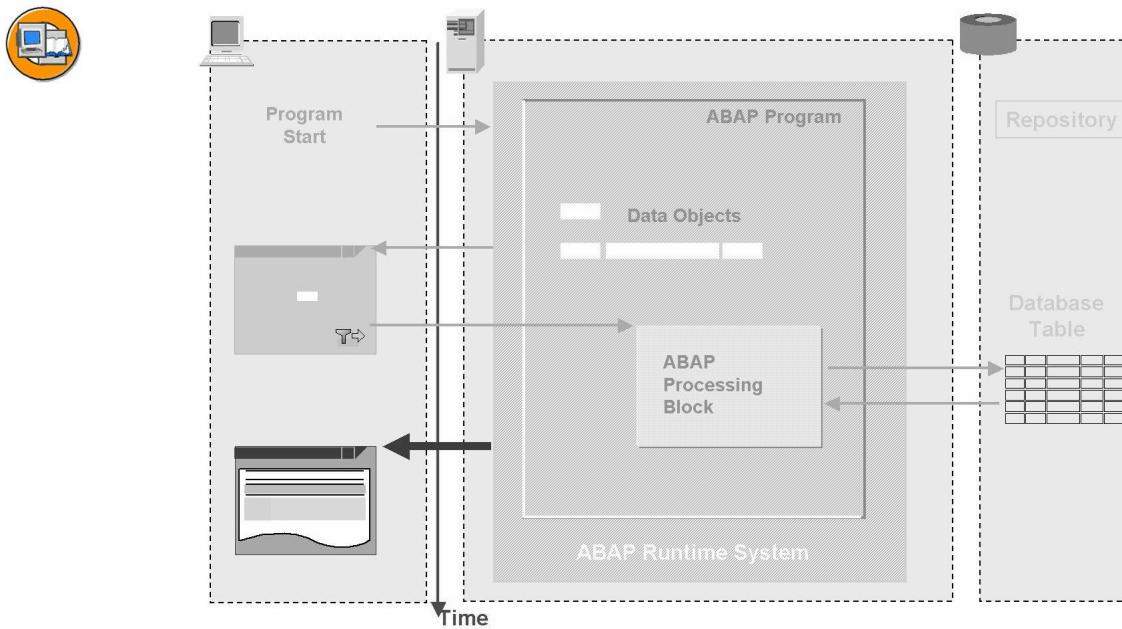


Figure 11: Runtime System Sends List

The layout of the subsequent list display has also been programmed into the processing block. After its completion, the runtime system sends this list as a screen to the presentation server.

How the Topics are Organized in this Course

Using this example program, we can cover numerous topics that are part of this course. We want to take the opportunity to again supply an orientation aid for the course flow.

Which Components of an ABAP Program Are Discussed and in Which Unit?



Unit 1

Flow of an ABAP Program

Unit 2

Introduction to the *ABAP Workbench*

Unit 3

Introduction to the *ABAP Dictionary*

Unit 4

Basic ABAP Language Elements

Unit 5

Data Retrieval

Unit 6

Internal Program Modularization

Unit 7

Introduction to Event Blocks

Unit 8

User Dialogs, Lists, Selection Screens, Screens, User Interfaces

Unit 9

Outlook: Developing Internet Applications

Unit 10

Cross-Program Modularization

Unit 11

Software Logistics and Software Adjustment



Lesson Summary

You should now be able to:

- Basically describe the architecture of the *SAP Web Application Server*
- Basically describe how a simple dialog program is executed by the ABAP runtime system



Unit Summary

You should now be able to:

- Basically describe the architecture of the *SAP Web Application Server*
- Basically describe how a simple dialog program is executed by the ABAP runtime system

Related Information

... Refer to the article “Overview of Application Program Components” in the online documentation.

Unit 2

Introduction to the ABAP Workbench

Unit Overview

Refer to the lesson summaries for an overview of this unit.



Unit Objectives

After completing this unit, you will be able to:

- Describe the structure of the *Repository*
- Name and use the search tools of the *Repository*
- Use the *Object Navigator* for your development work
- Name and use the utilities for orderly software development
- Create packages
- Create programs
- Create a transaction code

Unit Contents

Lesson: Repository and Object Navigator	18
Lesson: Developing Programs and Organizing Developments	28
Procedure: Creating Packages	32
Procedure: Creating an ABAP Program	38
Procedure: Creating a Transaction Code	43
Procedure: Creating Transaction Codes for Favorites in Your Role Menu	44
Exercise 1: Organizing Developments	45
Exercise 2: Developing ABAP Programs	47
Exercise 3: Transaction Codes	51

Lesson: Repository and Object Navigator

Lesson Overview

This lesson gives a short description of the *Repository* and a small overview of the most important components of the *ABAP Workbench*. Also, it presents the *Object Navigator* as a central development tool.



Lesson Objectives

After completing this lesson, you will be able to:

- Describe the structure of the *Repository*
- Name and use the search tools of the *Repository*
- Use the *Object Navigator* for your development work

Business Example

You need to search for repository objects and analyze their structure.

Introduction to the Repository

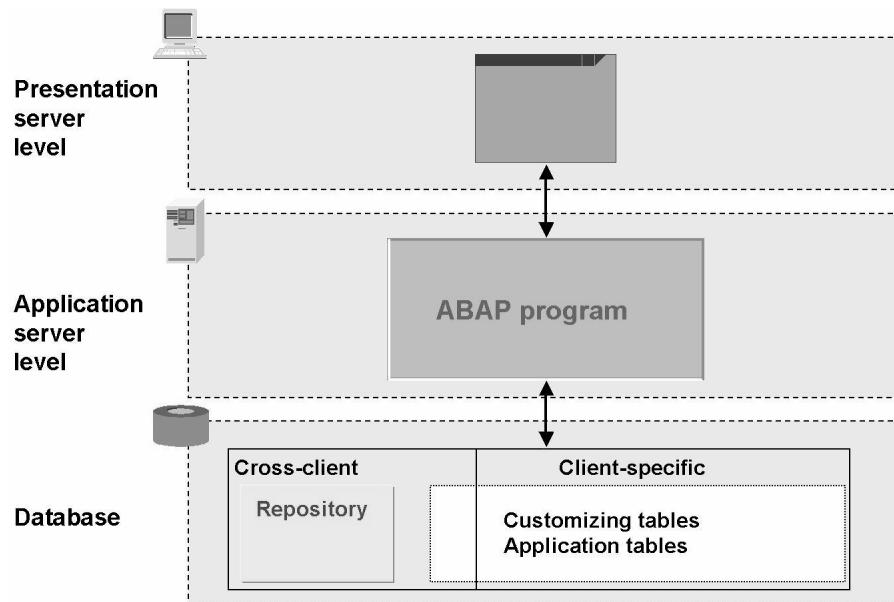


Figure 12: The Repository

The database contains application and Customizing tables that are usually **client-specific**.

In addition, the database also contains the **Repository**. Part of the **Repository** makes up the *ABAP Dictionary*.

The **Repository** thus contains all the development objects, for example, programs, definitions of database tables, or central data type definitions. Development objects are therefore also known as repository objects.

Repository objects are always **cross-client**. They can therefore be used and changed in all clients.

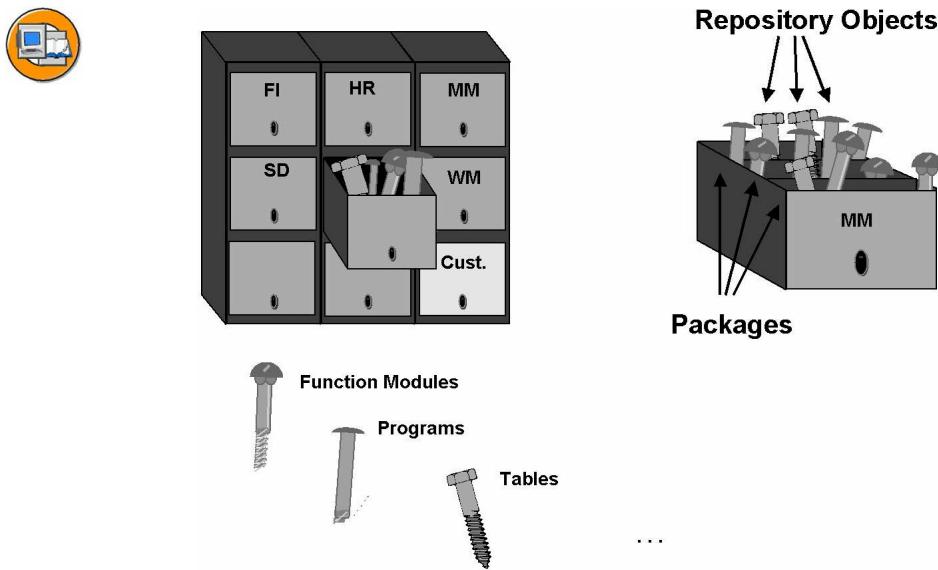


Figure 13: Structure of the Repository

The **Repository** is subdivided according to application components.

Within each application component, there are several **packages**, which are an even finer logical subdivision.

Repository objects are often made up of subobjects that are themselves repository objects.

Each repository object must be assigned to a package when it is created.

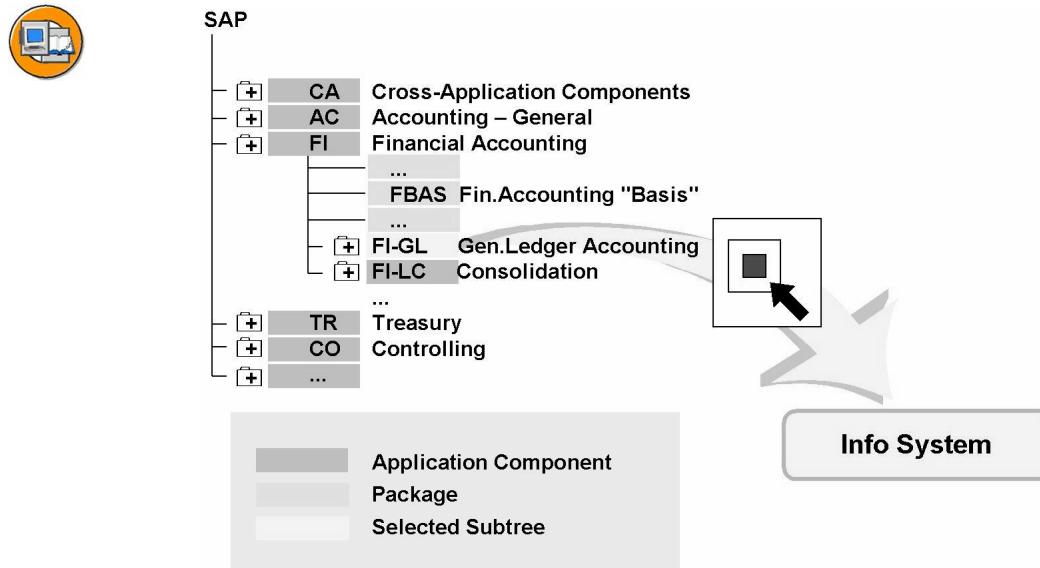


Figure 14: SAP Application Hierarchy

The structure of the application components is depicted in the **application hierarchy**. From the SAP Easy Access Menu, you can easily branch to the application hierarchy by choosing *Tools→ABAP Workbench→Overview→Application Hierarchy*.

The application components are displayed in a tree structure in the application hierarchy. Expanding a component displays all the packages that are assigned to that component.

You can select subtrees and navigate from the application hierarchy into the *Repository Information System*. The system then collects all packages for the subtrees selected and passes them to the Information System.

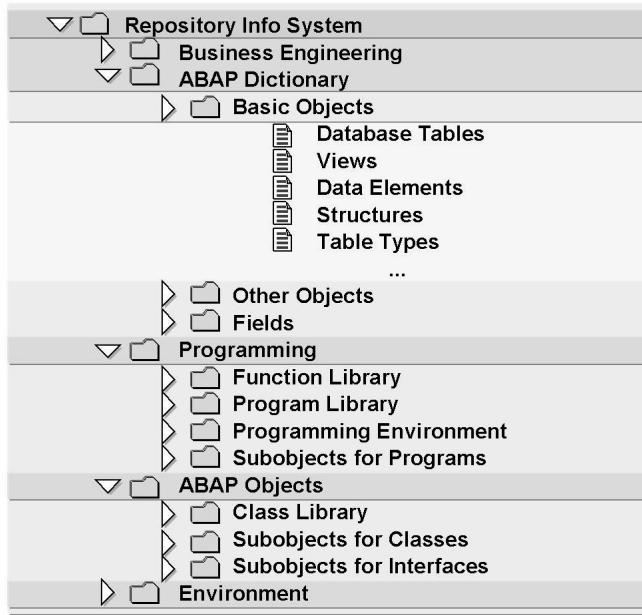


Figure 15: Repository Information System

You can use the Repository Information System to search for specific repository objects. Suitable search criteria are available for the various repository objects.

Which objects are to be found using the Repository Information System depends on how you navigate to get there:

- From within the application hierarchy, select the pushbutton *Information System* (double-click the selected application components or packages), and then the respective repository objects will be “filtered”.
- From the SAP Easy Access Menu, choose the path *Tools* → *ABAP Workbench* → *Overview* → *Information System*; then **all** the repository objects in the system are available for searching.

Working with the *Object Navigator*

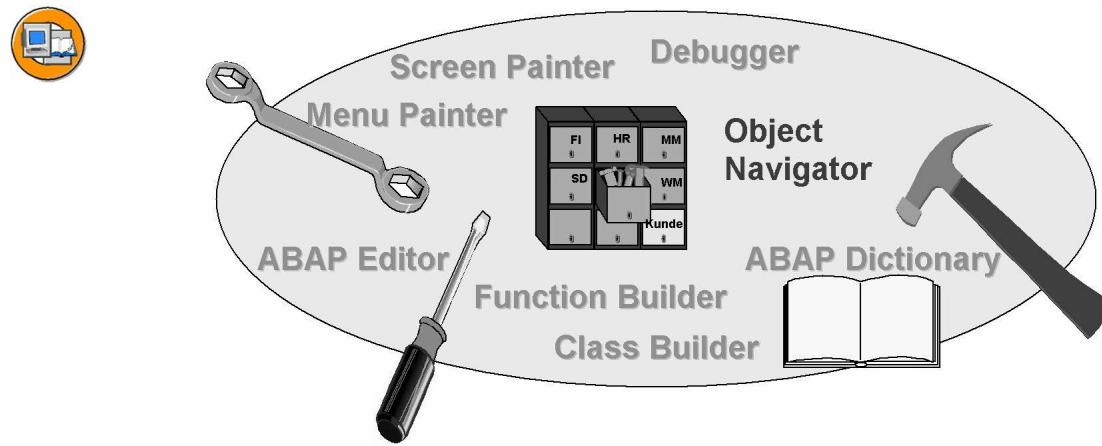


Figure 16: ABAP Workbench Tools

The *ABAP Workbench* contains various tools for processing repository objects. These tools cover the entire software development cycle. The most important tools for creating and processing repository objects are:

- The *ABAP Editor* for editing source code
- The *ABAP Dictionary* for editing database table definitions, central data types, and so on
- The *Screen Painter* for configuring screens (screens together with functions for user dialogs)
- The *Menu Painter* for designing user interfaces (menu bar, standard toolbar, application toolbar, function key settings)
- The *Function Builder* for processing function modules
- The *Class Builder* for processing global classes

There are three ways to call the respective tool:

- You call each tool directly from the SAP Easy Access menu.
- You are working with the *Object Navigator*. This transaction provides you with a tree-like overview of all objects within a package or program. Double-click on an object to start the respective tool.
- You navigate by double-clicking from one tool to the next, and back again.

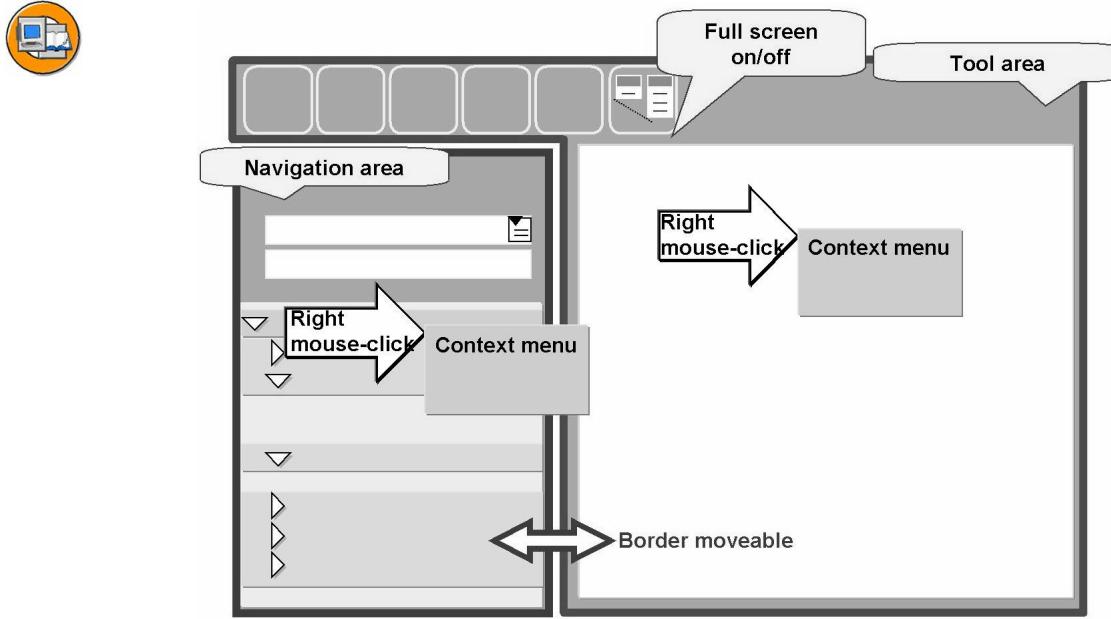


Figure 17: Screen Structure in the *Object Navigator*

The *Object Navigator* screen is split into two areas:

- A navigation area for displaying an object list as a hierarchy tree
- A tool area for displaying and editing a development object using the correct tool

You can display or hide the hierarchy area.

You can select functions from a context menu in both screen areas. You are only given a choice of those functions that are provided for displaying or editing the object on which the cursor is positioned. You open the context menu for an object you have selected previously by right-clicking the object (if you have a mouse for left-handed operation, then the left mouse button).

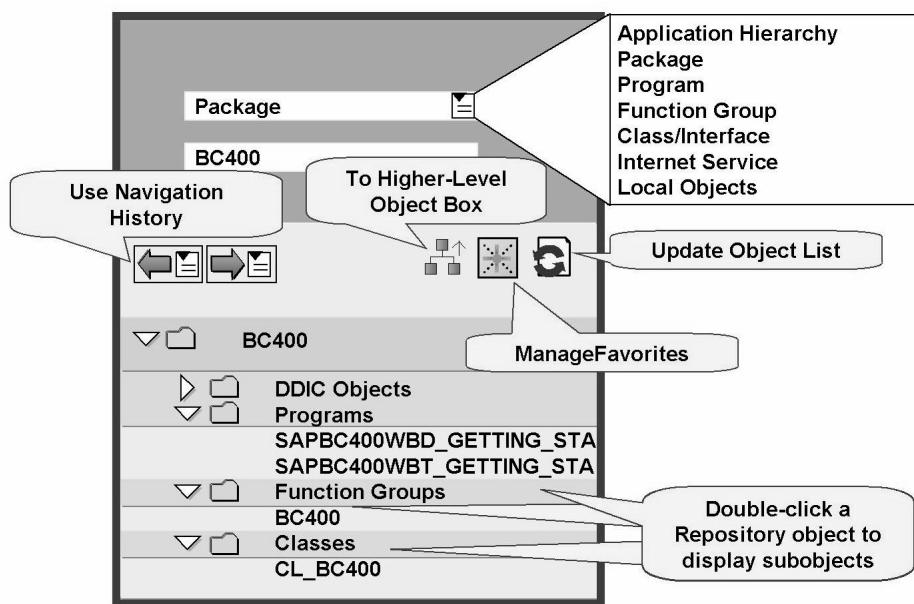


Figure 18: Working with the Navigation Area

The repository objects are organized in a hierarchy:

- An application component can have different packages.
- A package can contain different repository objects: Programs, function groups, ABAP Dictionary objects, and so on.
- Each repository object can have different subobjects assigned to it:

Programs can contain:

Global data, types, events, subroutines, and so on

Function groups can contain:

Global data, function modules, and so on

You can enter the type of object list and the object name in the upper part of the hierarchy area. The object list is then displayed in the navigation area.

Note: As of SAP Web AS 6.20, you can switch to navigation tools for other system information and objects using the appropriate pushbuttons. However, these are not required during this training course.

To display the subobjects, double-click an object within the object list.

Double-clicking on an object that does not have an object list means the object is displayed using the respective tool in the tool area.

You can navigate between object lists either **by history** or **by hierarchy**.

You can add object lists that you edit frequently to your favorites.

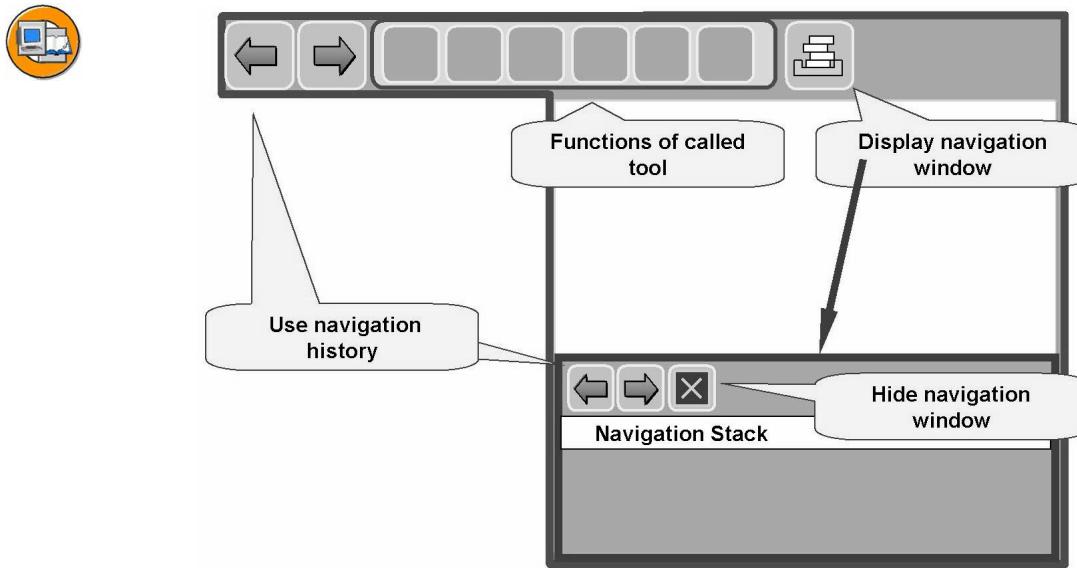


Figure 19: Navigation in the Tool Area

In the tool area, you can display a navigation window showing your navigation **history**. There you see a list of the objects that you have displayed since starting the *Object Navigator* **in the tool area**.

The object currently displayed in the tool area is highlighted in color.

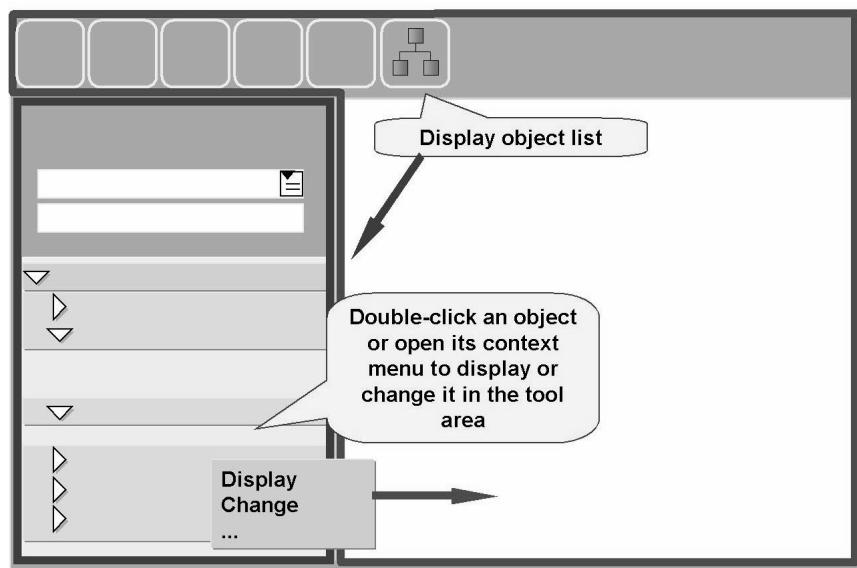


Figure 20: Synchronization of Navigation and Tool Areas

Navigation in the navigation area is **independent** from navigation in the tools area. This allows the navigation area to be used in a very flexible manner.

If you wish, you can synchronize the two areas:

- You can **display an object** in the tool area by double-clicking it, or by selecting the context menu from within the navigation area. The system then automatically selects the appropriate tool for processing the object selected.
- To **display the object list** that belongs to an object you are currently processing in the tool area, choose *Display Object List*.

To **create** objects from an object list, you can use the context menu for that object type. If there is no entry in the object list for the required object type, you can create any number of objects using *Edit Object* or *Other Object*.



Lesson Summary

You should now be able to:

- Describe the structure of the *Repository*
- Name and use the search tools of the *Repository*
- Use the *Object Navigator* for your development work

Lesson: Developing Programs and Organizing Developments

Lesson Overview

In this lesson you will learn how to create packages and how to assign repository objects to a package. You will write a program with a WRITE statement within a SELECT loop. In addition, you will create a transaction code for this program and include this in your favorites for the SAP Easy Access menu.



Lesson Objectives

After completing this lesson, you will be able to:

- Name and use the utilities for orderly software development
- Create packages
- Create programs
- Create a transaction code

Business Example

You are to create a new package and ABAP programs within the framework of a development project.

Organizing Developments

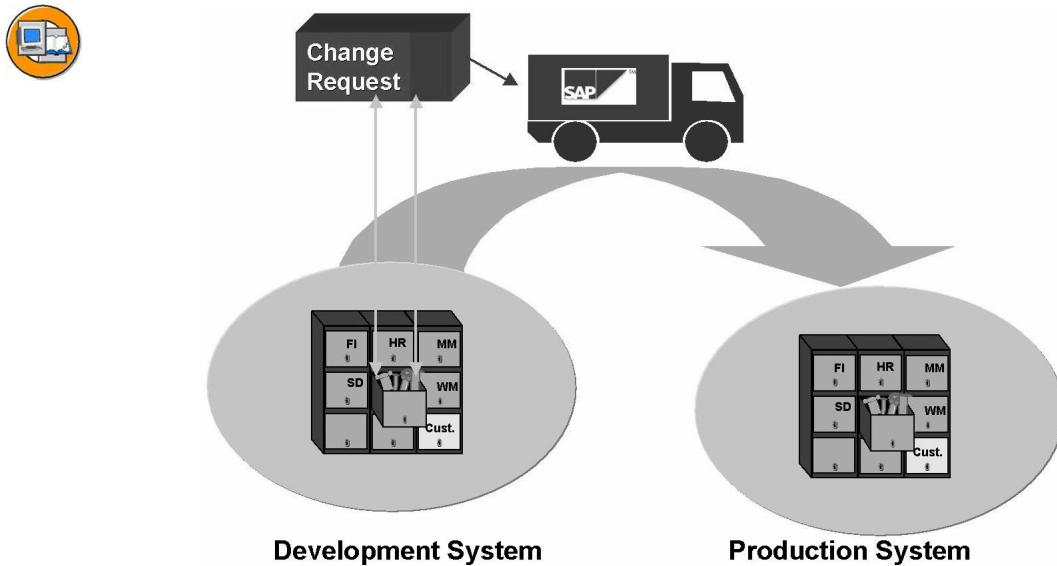


Figure 21: Transporting Repository Objects

Development projects are always implemented in a development system and then transported to the next system. Repository objects are automatically assigned to a **transport route** on the basis of their belonging to a particular development class and their actual function. A decisive criterion for the combination of development projects is, therefore, which repository objects need to be transported together because of their dependencies.

You perform the **scheduling** of the **common transport** for the repository objects using a **change request**.

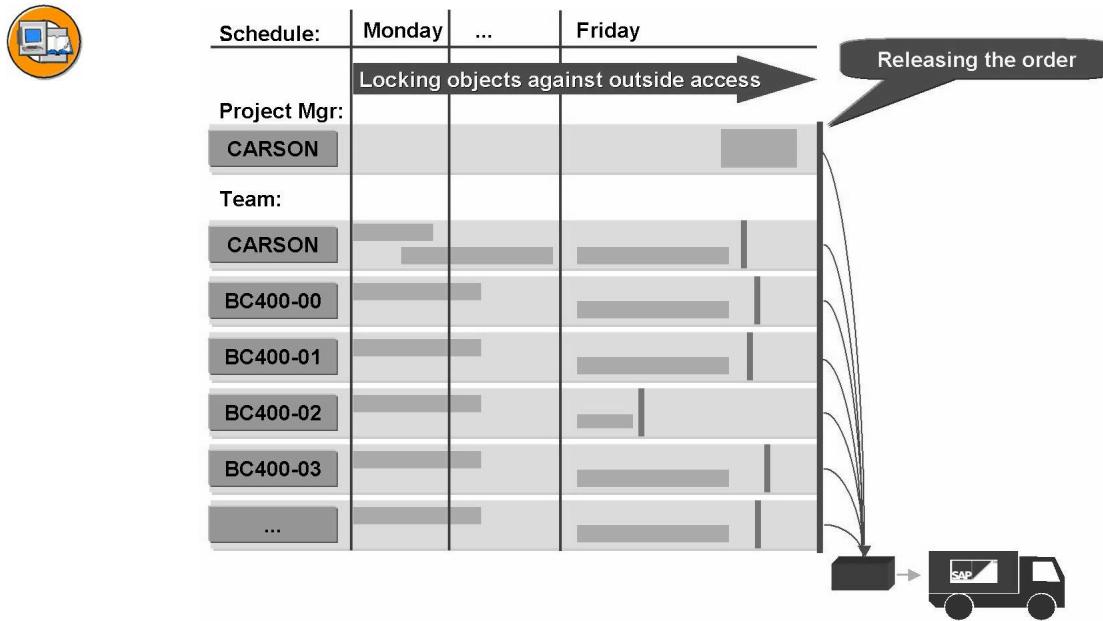


Figure 22: Scheduling a Development Project

Scheduling a development project using a change request covers the following aspects:

- All employees in the project start their work at a defined time/date.
- Employees have the chance to follow up and check their activities separately.
- The respective repository objects can be processed by all employees involved in the project. For those developers who do not belong to the team, the repository objects remain locked until the project is completed.
- The group triggers the transport of all the respective repository objects.

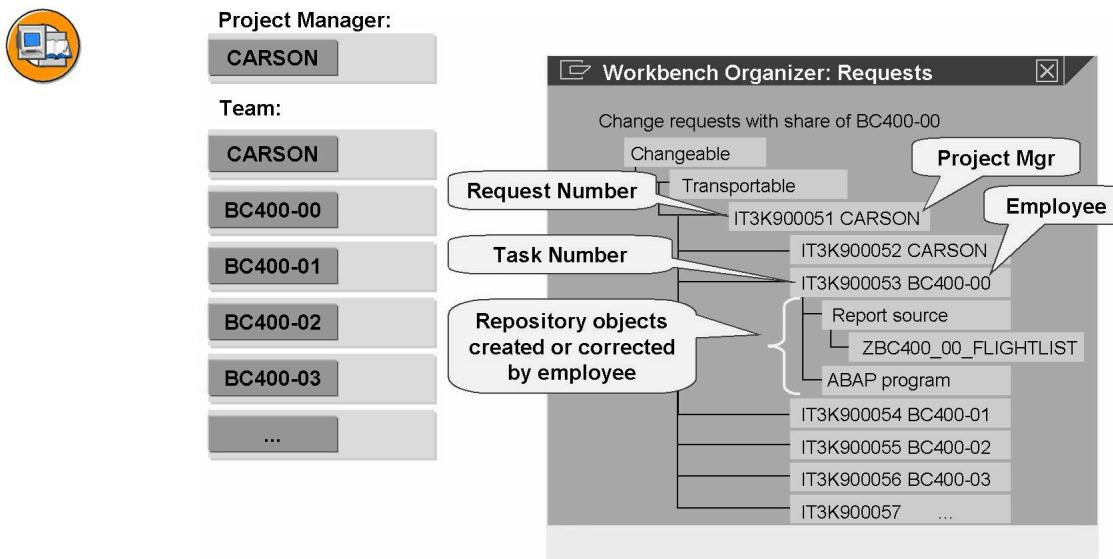


Figure 23: Organization of a Development Project in the Change Request

At the beginning of a development project, the **project manager** creates a change request in the *Transport Organizer*. The project manager assigns all project team members to the change request. The *Transport Organizer* then creates a **task** for each project developer.

Whenever an employee assigns a repository object to this change request, the repository object is automatically entered as that employee's **task**. Thus, all repository objects that an employee works on during a development project are collected within his or her task. The repository objects are therefore locked for all developers who do not belong to this particular project.

Unlike the logical functional divisions that separate Repository objects using packages, change requests are project-related and therefore have a limitation with regard to the **time schedule** involved. Thus, although a program always belongs to only one package, it can - at different times - belong to different projects.



Creating Packages

1. Navigate to the *Object Navigator*. You now have two options:
 - Choose the object type *Package* and enter the name of the package in the input field. Make sure you comply with the customer namespace conventions. If the package class does not already exist, the system branches to a dialog for creating a package.
 - For Release 4.6A or 4.6B, you must choose *Edit Object* or *Other Object* in the initial screen. Under *More*, enter the name of the package under *Package*.
2. Create the attributes of the new package.

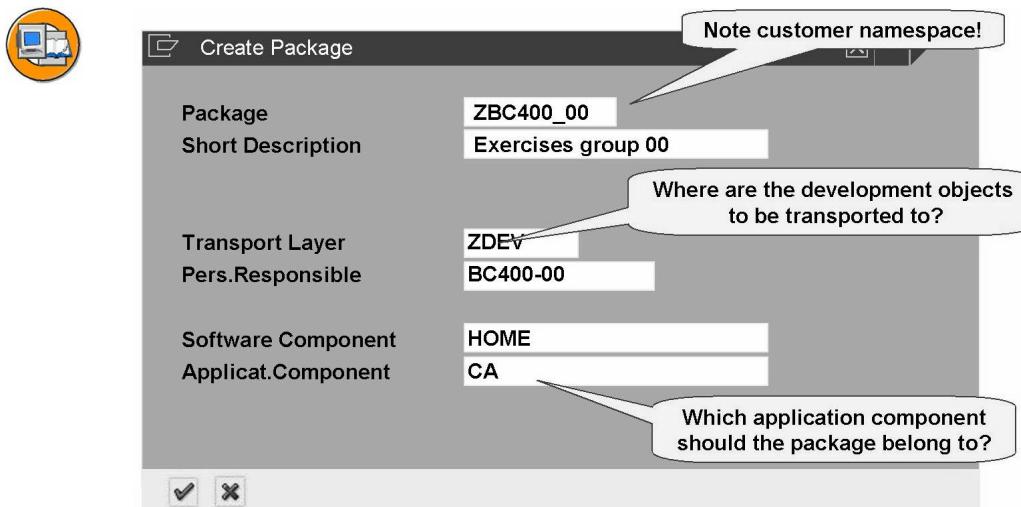


Figure 24: Setting Package Attributes (Example)

You should check the settings for the attributes listed below. For more detailed information, select the field help (*F2*).

Transport Layer

If you are executing your own developments, you must set up a transport layer for customer developments.

Person Responsible

This person has full responsibility for the objects in this package. The system automatically uses your user name as a default value.

Continued on next page

Software Component

For customer developments you should enter **HOME** as the software component.

Application Component

Define the location of the package within the application hierarchy.

3. Assign the package to a **change request**.

You can display all change requests in which you have a task using the **My Tasks** pushbutton.

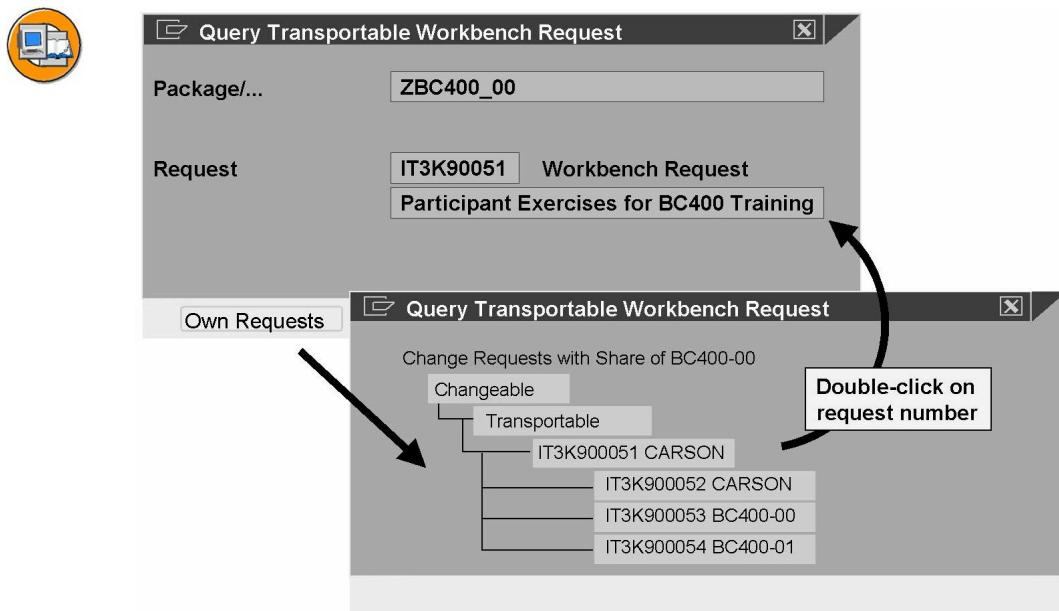


Figure 25: Assignment to a Change Request

→ **Note:** The system takes care of the exact assignment for your task.



Hint: All repository objects that are created or changed must be assigned to the change request of the respective project. A change request has already been created for the project "Exercises for Training Participants BC400" in this training course. Each group has a task within this change request.

Introduction to the Programming Language ABAP Objects

Features of the ABAP Objects Programming Language

The ABAP Objects programming language ...



- Is typed
- Enables multi-language applications
- Fully integrates an SQL standard
- Has been enhanced as an object-oriented language
- Is platform-independent
- Is upward-compatible

Therefore, it is designed for dialog-based, business applications.

To support data processing by types, ABAP Objects contains a systematic type hierarchy. Type conversions and type casting are supported.

Using translatable text elements, you can develop multi-language applications.

The OPEN SQL standard embedded in ABAP Objects allows direct database accesses.

ABAP Objects is the object-oriented enhancement of the ABAP programming language.

The ABAP runtime system is a component of the *SAP Web Application Server*. Thus, the ABAP syntax is platform-independent.

ABAP has been developed further from release to release. If an old syntax statement has been replaced by a better one, you can still use the old statement. The **ABAP runtime system** is therefore **downward-compatible**.

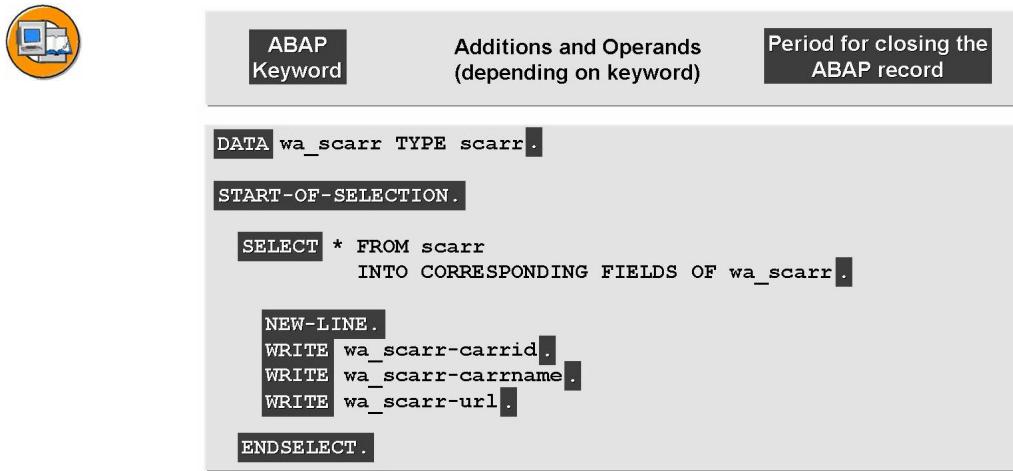


Figure 26: General ABAP Syntax I

ABAP programs are made up of individual statements.

Each statement must end with a period.

The first word in a statement is called a keyword.

Words must always be separated by at least one space.

Statements can be indented.

Statements can take up more than one line.

You may have multiple statements in a single line.

The ABAP runtime system does **not** distinguish between uppercase and lowercase letters for keywords, additions, and operands. In order to keep a better overview in this course, we will display all keywords and their additions in uppercase letters, and all operands in lowercase letters.

For indents and upper-case/lower-case lettering, you can use the *Pretty Printer*: Utilities → Settings → ABAP Editor → Pretty Printer.

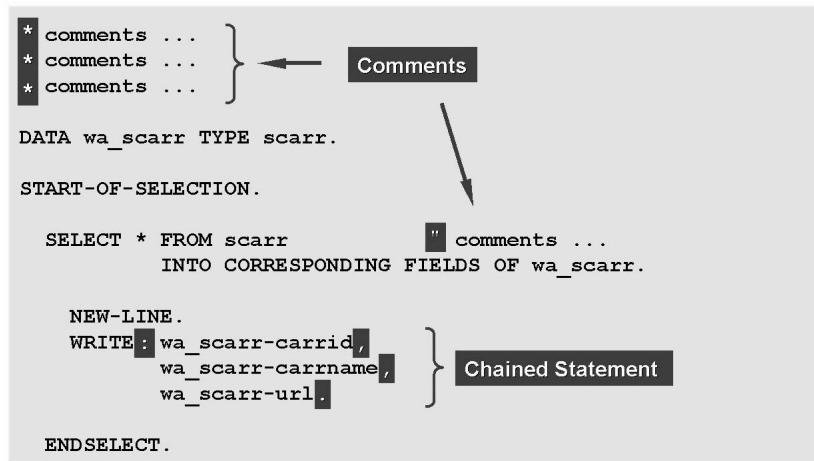


Figure 27: General ABAP Syntax II

You introduce **comment lines** with an asterisk *. The respective comment line is then ignored by the ABAP runtime system.

If you wish to have the rest of a line set as a comment, you must use double quotation marks ("").

You can combine consecutive statements with an **identical beginning** into a **chained statement**:

- In chained statements, close the identical beginning part with a colon.
- Separate the individual chained elements that come after the colon by commas.
- Blank spaces and line breaks are allowed before and after the separators (colons, commas, periods).
- Be aware that the ABAP runtime system still considers the individual parts of a chained statement to be single statements.



```
PARAMETERS pa_carr TYPE scarr-carrid.  
  
DATA wa_scarr TYPE scarr.  
  
START-OF-SELECTION.  
  
    SELECT SINGLE * FROM scarr  
        INTO CORRESPONDING FIELDS OF wa_scarr  
        WHERE carrid = pa_carr.  
    IF sy-subrc EQ 0.  
        NEW-LINE.  
        WRITE: wa_scarr-carrid,  
               wa_scarr-carrname,  
               wa_scarr-url.  
  
    ELSE.  
        WRITE 'Sorry, no data found!'.  
    ENDIF.
```

Figure 28: Return Value After ABAP Statements

If there are several ABAP statements, the ABAP runtime system writes a numeric value into the system field SY-SUBRC in order to provide information as to how successfully the particular statement was executed. The value zero means that the statement was successful. Read the keyword documentation for the respective statements to find out whether and how this return value is set in individual cases.



Creating an ABAP Program

1. Navigate to the *Object Navigator*. You now have two options:
 - In the navigation area, choose the *Program* object type and enter a program name. Comply with the customer namespace conventions. If the program does not exist, the system goes to the dialog sequence that lets you create a program.
 - Display the package for which you want to create the program. Trigger the dialog sequence for creating a program using the package context menu or the *Program* node.
 - Choose *Other Object* or *Edit Object*. Choose the tab *Program Objects* and enter the name. Then choose *Create*.
2. In this training course, remove the indicator *With TOP Include*. (Otherwise, your source text would be distributed to several programs.)
3. Change the title into a self-explaining short text.

Choose the appropriate *Status*: **Test Program** for all ABAP training courses.

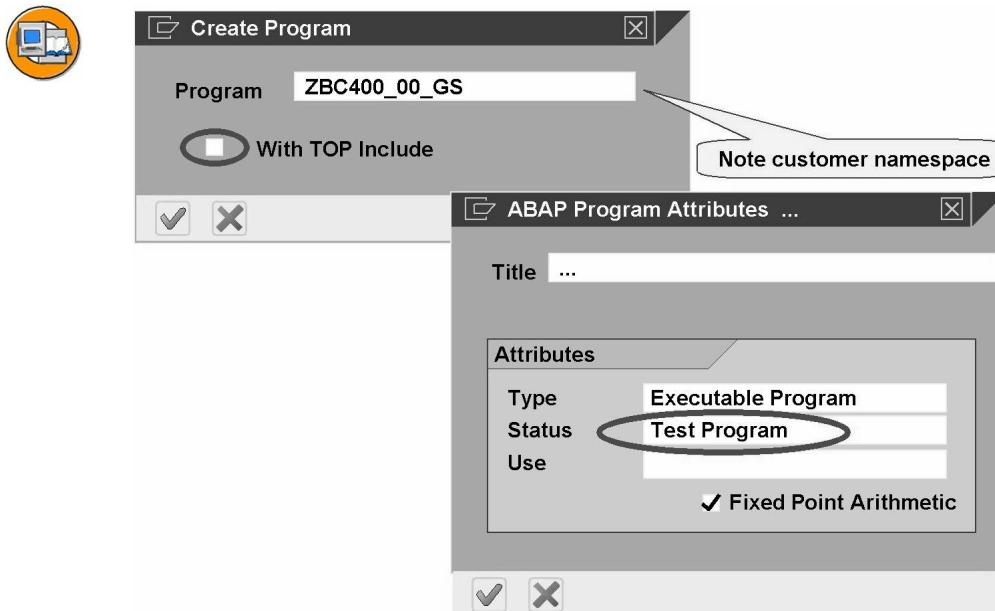


Figure 29: Creating an ABAP Program

Developing an ABAP Program

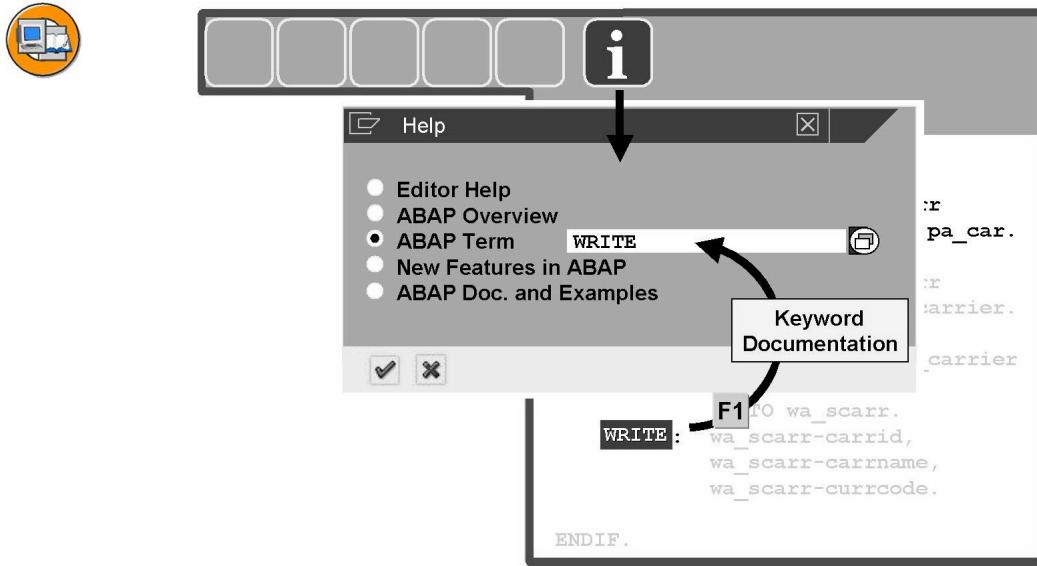


Figure 30: Keyword Documentation in the *ABAP Editor*

There are various ways of navigating to keyword documentation for an ABAP statement:

- The *F1* key takes you directly to the documentation for the statement on which the cursor is positioned.
- Through the *Help for ...* pushbutton, you can navigate to a display frame for the ABAP documentation. These provide you with different views of the keyword documentation.

In addition, under the path *Help → SAP Library → Basis*, you will find detailed online documentation for all tools and topics connected with the *ABAP Workbench*.

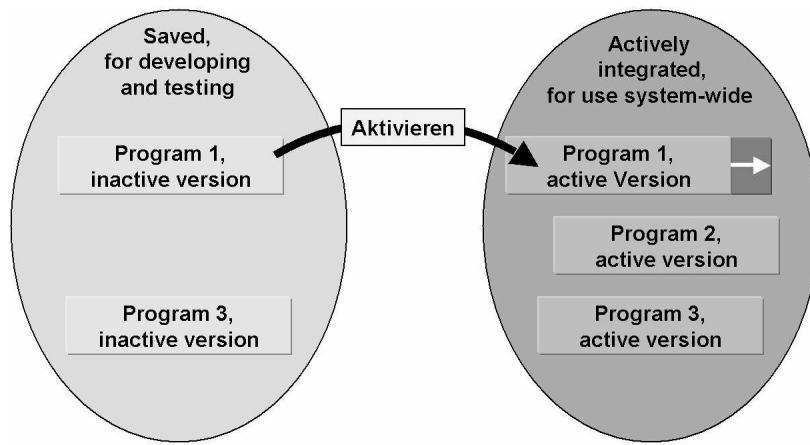


Figure 31: Inactive and Active Development Objects

Whenever you create a development object, or change and then save it, the system first stores only one **inactive version** in the *Repository*.

So that a repository object is available on a system-wide basis, it must first be available as an **active version**. This active version of your program will be used, for example, whenever a user wishes to execute your program.

If your program is available in both versions, you can switch from one version to the other in display mode.

Whenever you activate a program, the system first displays a list of all inactive objects that you have processed. This is called the **worklist**. If you trigger activation, the following happens:

- The system saves the program, that is, it stores or overwrites the inactive version.
- This is subject to a syntax check.
- The active version is stored or overwritten.

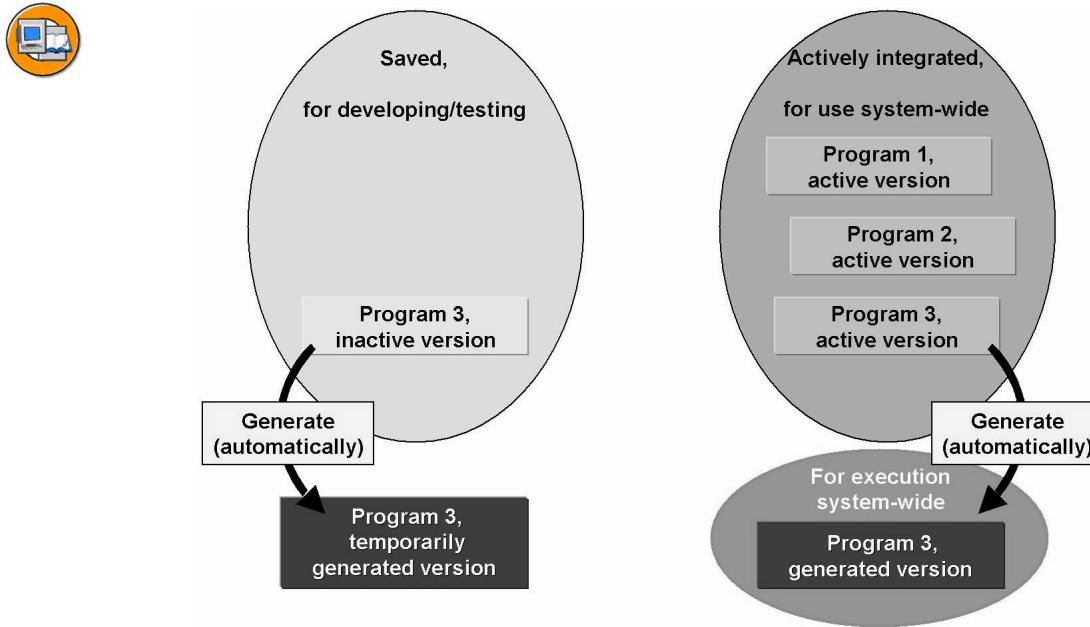


Figure 32: Generating Runtime Objects

When you generate a development object, the system creates (compiles) a separate runtime object (LOAD) and also stores it in the *Repository*. This generated version is then the version that is executed (interpreted). Generation takes place automatically, but it can also be triggered explicitly.

If, for example, a user wishes to execute your program and there is still no generated version, the system will use the active version for generation.

However, you can decide yourself which version you wish to execute: Start your program from the navigation area, and the system will use the active version. *Testing* On the other hand, if you *test* an executable program from the *ABAP Editor*, the system will access the **inactive** version. The system temporarily creates a generated version.

In this way it is possible to continue the development of a repository object **without changing the current system status**. All changes to the development object only become “visible” system-wide when the program is re-activated. With inactive versions, you get a local, separate view of the *repository*, and this provides the basis for a “**local runtime system**”.

Creating Transaction Codes

If you want to include a program in role menus or in the Favorites in your SAP Easy Access menu, you must assign a transaction code to it. Also, you can then start the program alternatively by entering the transaction code in the command field. In addition, transaction codes provide several additional link options.

Any number of transaction codes can be assigned to a program.



Creating a Transaction Code

1. In the *Object Navigator*, display the object list for your program.
2. In the context menu for the program name, choose *Create → Transaction*. Make sure you comply with the customer namespace conventions.
Enter a short text.
3. If the program is an *executable program*, choose *Program and Selection Screen (Report Transaction)*.
4. On the next screen, enter the name of the program.
Choose *Professional User Transaction*.
Under *GUI enabled*, enter *SAP GUI for HTML*, *SAP GUI for Java*, and *SAP GUI for Windows*.

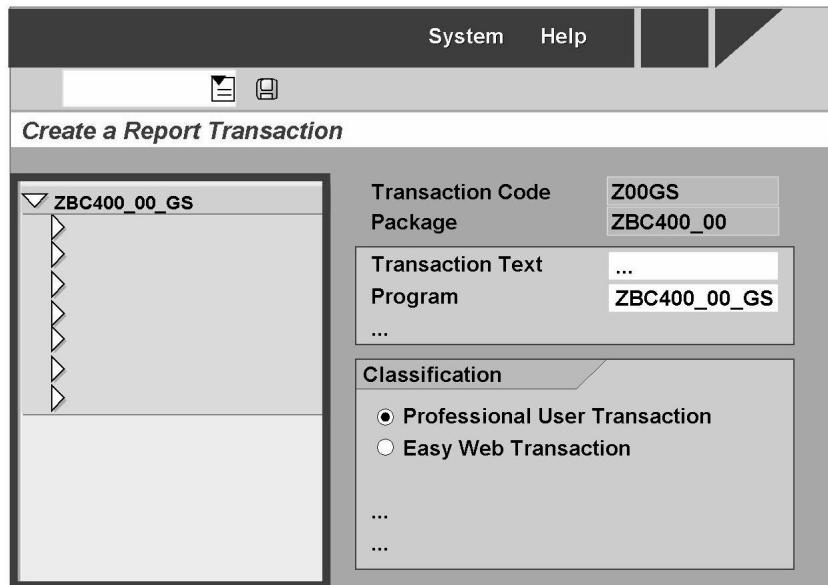


Figure 33: Creating Transaction Codes

5. Save the transaction code.
6. Since each transaction code is a Repository object, you must assign it to a package and to a change request on the following screens.



Creating Transaction Codes for Favorites in Your Role Menu

1. Navigate to the initial screen (*SAP Easy Access Menu*).
2. In the Favorites context menu, choose *Insert Transaction*.
3. In the dialog box that appears, enter the required transaction code.

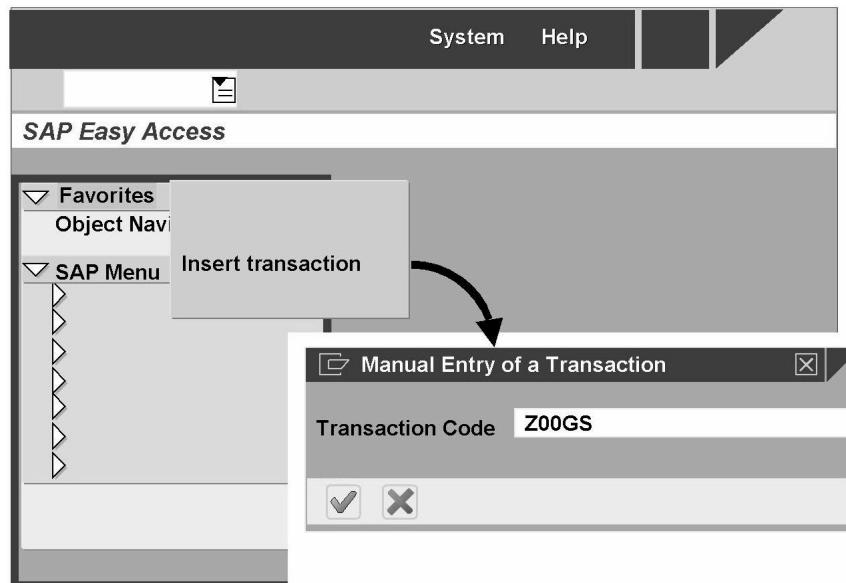


Figure 34: Adding a Transaction Code in the SAP Easy Access Menu

Result

The transaction code short text appears under the *Favorites* node. You can start the appropriate program from the context menu for this new node.

Exercise 1: Organizing Developments

Exercise Objectives

After completing this exercise, you will be able to:

- Create packages
- Assign changes in repository objects to a change request

Business Example

You are a developer for an airline group and you are given the task of developing evaluation programs for several airline companies.

To group the repository objects together logically and also transport them, you must create a package.

Task 1

Log on to the training system

1. Log on to the operating system and then to the specified training system using the user name the instructor has given you.
2. Enter a new personal password.

Task 2

Create Package

1. Create the package **ZBC400_##**.
Assign it to the change request the instructor has given you.
2. Assign all your repository objects in the following exercises to this package and this change request.



Hint: It is best if you always work with the *Object Navigator* tool. It provides you with an overview of all repository objects in your package. From here, you can edit each object directly by selecting it from this list.

Solution 1: Organizing Developments

Task 1

Log on to the training system

1. Log on to the operating system and then to the specified training system using the user name the instructor has given you.
 - a) Carry out this step as usual.
2. Enter a new personal password.
 - a) Carry out this step as usual.

Task 2

Create Package

1. Create the package **ZBC400_##**.
Assign it to the change request the instructor has given you.
 - a) Carry out this step as described in the training material.
2. Assign all your repository objects in the following exercises to this package and this change request.



Hint: It is best if you always work with the *Object Navigator* tool. It provides you with an overview of all repository objects in your package. From here, you can edit each object directly by selecting it from this list.

- a) Carry out this step as described in the training material.

Exercise 2: Developing ABAP Programs

Exercise Objectives

After completing this exercise, you will be able to:

- Create, edit, save, and process ABAP programs
- Check ABAP programs for correct syntax
- Test and activate ABAP programs

Business Example

You need to create an ABAP program that displays additional information on all existing flight departure times in a list.

The data is to be read from the database table SFLIGHT.

Task 1

Create program

1. Create the executable program **ZBC400_##_GS_LIST** without a “TOP include”.

Task 2

Edit program

1. Define a work area for data retrieval and output (name suggestion: `wa_flight):DATA wa_flight TYPE sflight.`
2. Program a loop access to all data records in the database table `SFLIGHT:SELECT * FROM sflight INTO CORRESPONDING FIELDS OF wa_flight....ENDSELECT.`
3. For each data record, display the contents of the fields `carrid, connid, fldate, seatsocc, and seatsmax` on the list: `NEW-LINE. WRITE wa_flight-carrid. WRITE ...`
4. Check your program for syntax errors, test it, and activate it

Solution 2: Developing ABAP Programs

Task 1

Create program

1. Create the executable program **ZBC400_##_GS_LIST** without a “TOP include”.
 - a) Carry out this step as described in the training material.

Task 2

Edit program

1. Define a work area for data retrieval and output (name suggestion: `wa_flight):DATA wa_flight TYPE sflight.`
 - a) See source code excerpt in the model solution.
2. Program a loop access to all data records in the database table `SFLIGHT:SELECT * FROM sflight INTO CORRESPONDING FIELDS OF wa_flight....ENDSELECT.`
 - a) See source code excerpt in the model solution.
3. For each data record, display the contents of the fields `carrid`, `connid`, `fldate`, `seatsocc`, and `seatsmax` on the list: `NEW-LINE. WRITE wa_flight-carrid. WRITE ...`
 - a) See source code excerpt in the model solution.

Continued on next page

4. Check your program for syntax errors, test it, and activate it
 - a) Carry out this step as described in the training material.

Result

Source code excerpt: **SAPBC400WBS_GS_LIST**

```
REPORT sapbc400wbs_gs_list.

DATA wa_flight TYPE sflight.

SELECT * FROM sflight
      INTO CORRESPONDING FIELDS OF wa_flight.

NEW-LINE.
WRITE:
      wa_flight-carrid,
      wa_flight-connid,
      wa_flight-fldate,
      wa_flight-seatsocc,
      wa_flight-seatsmax.

ENDSELECT.
```


Exercise 3: Transaction Codes

Exercise Objectives

After completing this exercise, you will be able to:

- Assign transaction codes to ABAP programs
- Add transaction codes to your Favorites list in the SAP Easy Access Menu

Business Example

Give your first ABAP program a transaction code, in addition to its technical name, which you use to execute it.

Task 1

Create a transaction code

1. Create transaction code **ZBC400_##_GS** for the executable program **ZBC400_##_GS_LIST**. *Start object: Program and selection screen (report transaction).*
2. Execute your program by entering the new transaction code.

Task 2

Add to favorites.

1. Include the transaction code **ZBC400_##_GS** in your favorites in the SAP Easy Access menu.
2. Execute your program by double-clicking the new favorite entry.

Solution 3: Transaction Codes

Task 1

Create a transaction code

1. Create transaction code **ZBC400_##_GS** for the executable program **ZBC400_##_GS_LIST**. *Start object: Program and selection screen (report transaction).*
 - a) Carry out this step as described in the training material.
2. Execute your program by entering the new transaction code.
 - a) Carry out this step as usual.

Task 2

Add to favorites.

1. Include the transaction code **ZBC400_##_GS** in your favorites in the SAP Easy Access menu.
 - a) Carry out this step as described in the training material.
2. Execute your program by double-clicking the new favorite entry.
 - a) Carry out this step as usual.



Lesson Summary

You should now be able to:

- Name and use the utilities for orderly software development
- Create packages
- Create programs
- Create a transaction code



Unit Summary

You should now be able to:

- Describe the structure of the *Repository*
- Name and use the search tools of the *Repository*
- Use the *Object Navigator* for your development work
- Name and use the utilities for orderly software development
- Create packages
- Create programs
- Create a transaction code

Related Information

... Refer to the online documentation for each tool.

Unit 3

Introduction to the *ABAP Dictionary*

Unit Overview

Refer to the lesson summaries for an overview of this unit.



Unit Objectives

After completing this unit, you will be able to:

- Describe the SAP flight data model
- Describe how a structure is structured
- Describe the structure of a structure
- List the basic possible uses of transparent tables and structures in ABAP programs.

Unit Contents

Lesson: Transparent Tables and Views	56
Exercise 4: Using Global Structures for Data Objects and Dialogs.....	67

Lesson: Transparent Tables and Views

Lesson Overview

In this lesson, you will become familiar with the description and the structure of transparent tables. In addition, the distinction between structures and transparent tables will be explained. You will also learn why it is appropriate to use data types from the *ABAP Dictionary*. This is particularly the case in connection with the PARAMETERS statement.



Lesson Objectives

After completing this lesson, you will be able to:

- Describe the SAP flight data model
- Describe how a structure is structured
- Describe the structure of a structure
- List the basic possible uses of transparent tables and structures in ABAP programs.

Business Example

When you define data objects in your ABAP program, you should use the advantages of the *ABAP Dictionary*.

The ABAP Dictionary as a Central Tool of the ABAP Workbench

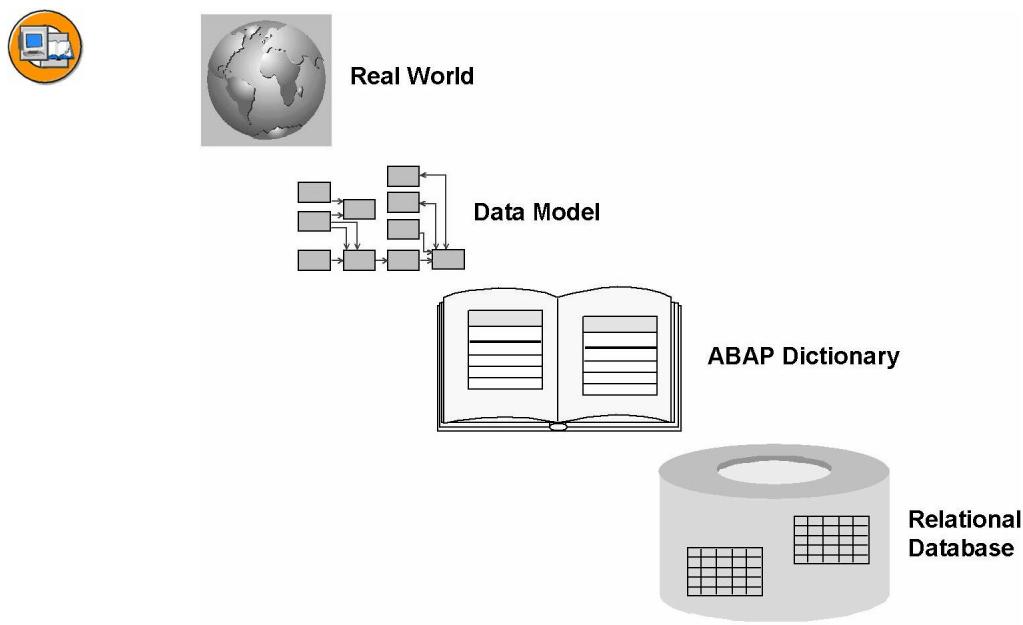


Figure 35: Data Modeling

In the development of business application software, parts of the real world must be represented in data form. A business unit then represents an entity. These entities relate to each other. The entities and relationships are visible from the data model. We also refer to an **Entity Relationship Model (ERM)**.

Based on this data model, you can find the conversion of the table definitions and their relationships in the *ABAP Dictionary*.

The actual application data is then on the database itself (physically).



Figure 36: Flight Data Model for ABAP Training Courses

ABAP training courses, online documentation, and ABAP keyword documentation all use the same flight data model. The Repository objects for the flight data model are in the package **BC_DATAMODEL**.

In the context of this training course, it is sufficient to show one simple excerpt from this data model; this can be modified to suit your own needs, if necessary: If a **customer of a travel agency** wants to travel from one place to another, then he or she requests the following information from the travel agency:

- What connection offers me the best and most direct flight?
- At what times are flights offered on the date that I want to travel?
- How can I optimize the conditions under which I am traveling to find the best solution; that is, what is the cheapest flight, the fastest connection, the connection that gets me there closest to the time I want to arrive?

This view is different to that of a **travel agency**: In the data model, which is tailored for **managing** the necessary data, the data is stored in tables in a central database, organized according to **technical** criteria. The amount of data stored far exceeds the demands of a customer.

You must be able to compile the data to suit the demands of the customer using application programs.

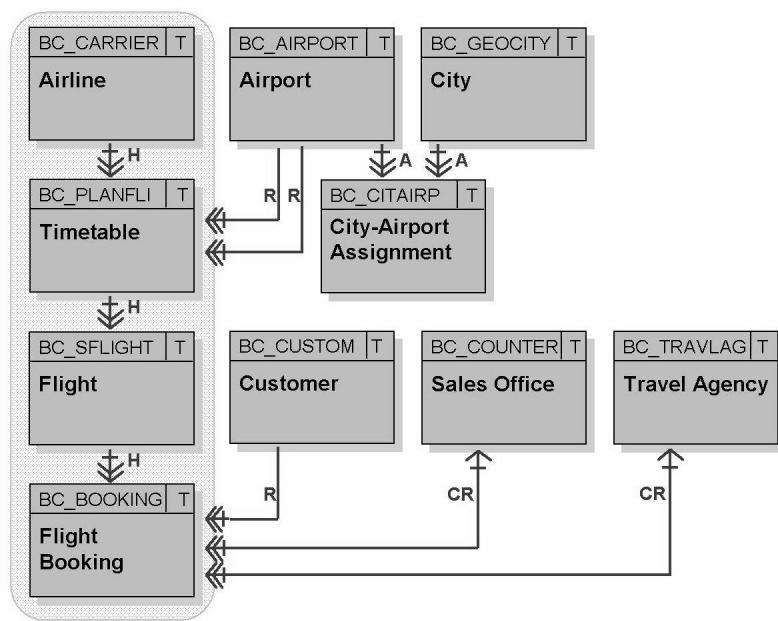


Figure 37: Relational Data Model

The flight data model contains entities for all business information that is logically connected, such as:

- Cities
- Airports
- Airlines
- Flight routes
- Flights
- ...

These entities all relate to each other in certain ways:

- Each flight schedule contains exactly one airline, one departure airport, and one destination airport.
- Each bookable flight always belongs to exactly one existing flight schedule.
- Assignments can be set between cities and nearby airports.

You can manage all necessary data, without redundancies, using these relationships. At the same time, the travel agency is able to obtain all data requested by the customer.

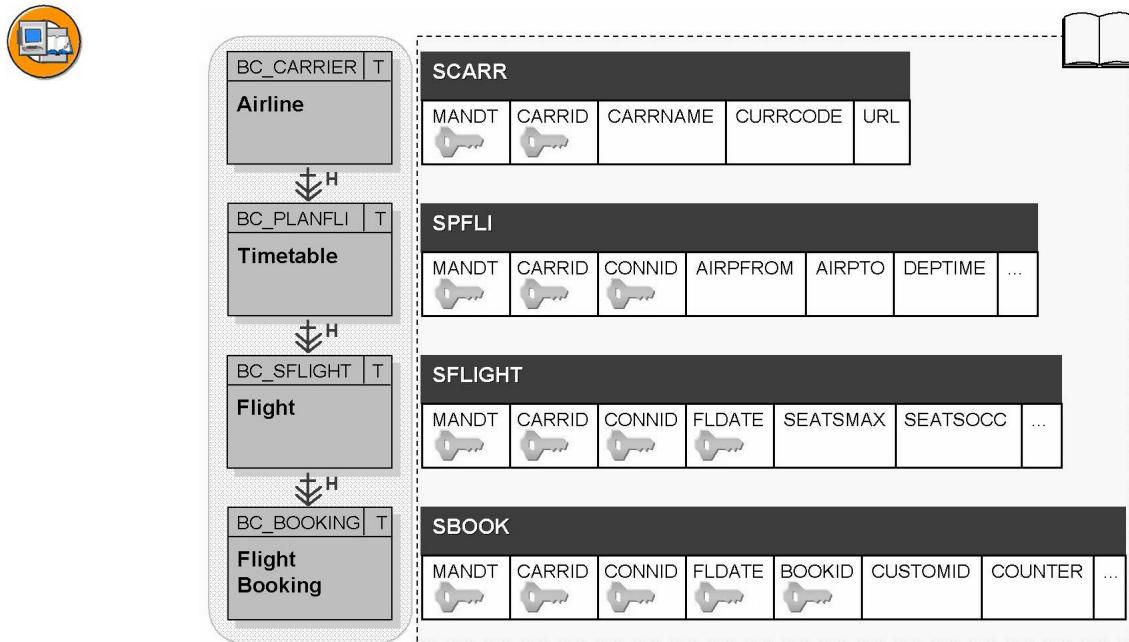


Figure 38: Converting Using Transparent Tables

You define transparent tables in the *ABAP Dictionary*.

A transparent table contains different columns to allow structured management of the data records. In this context, the term **table key** is of particular importance. The table key is the **combination of key fields**. The values in the key fields allow the system to **uniquely identify** all data records.

Key fields that link to other tables are called **foreign key fields**. Using these foreign key relationships, you can manage the data in a hierarchy and in a consistent fashion.

In any other case we refer to **primary key fields**.

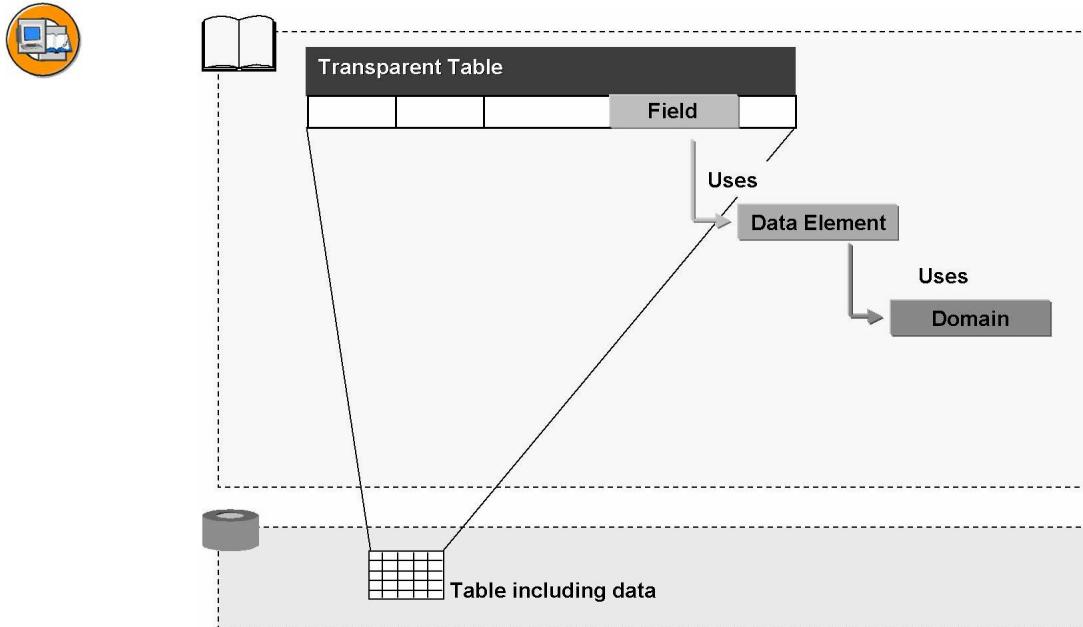


Figure 39: Technical Structure of a Transparent Table

You normally use a **data element** to type a table field. **Semantic** properties, such as field labels, are stored in the data element.

You normally use a **domain** for the **technical** properties. This is where the data type, amongst other things, is stored.

So the *ABAP Dictionary* contains the **platform-independent** description of the relevant database table. The database table has the same name as the transparent table in the *ABAP Dictionary*. The fields, too, have the same names in both the database and in the *ABAP Dictionary*.



The screenshot shows the SAP R/3 ABAP Dictionary interface. The title bar reads "Dictionary: Display Table". The main area shows a table for the transparent table "SPFLI". The table has columns: Fields, Key, Field Type, Data Type, Length, Dec. Places, and Short Text. The first three rows are highlighted with ovals: MANDT (Key: checked, Field Type: S_MANDT, Data Type: CLNT, Length: 3, Dec. Places: 0, Short Text: Client for training), CARRID (Key: checked, Field Type: S_CARR_ID, Data Type: CHAR, Length: 3, Dec. Places: 0, Short Text: Airline), and CONNID (Key: checked, Field Type: S_CONN_ID, Data Type: NUMC, Length: 4, Dec. Places: 0, Short Text: Flight connection). The table also includes columns for Attributes, Fields, and Currency/Quantity Fields.

Fields	Key	Field Type	Data Type	Length	Dec. Places	Short Text
MANDT	<input checked="" type="checkbox"/>	S_MANDT	CLNT	3	0	Client for training
CARRID	<input checked="" type="checkbox"/>	S_CARR_ID	CHAR	3	0	Airline
CONNID	<input checked="" type="checkbox"/>	S_CONN_ID	NUMC	4	0	Flight connection
COUNTRYFR	<input type="checkbox"/>	LAND_1	CHAR	3	0	Country key
CITYFROM	<input type="checkbox"/>	S_FROM_CIT	CHAR	20	0	City of departure
AIRPFROM	<input type="checkbox"/>	S_FROMAIRP	CHAR	3	0	Airport of departure
COUNTRYTO	<input type="checkbox"/>	LAND_1	CHAR	3	0	Country key
CITYTO	<input type="checkbox"/>	S_TO_CITY	CHAR	20	0	Arrival city
...	<input type="checkbox"/>

Figure 40: Transparent Tables in the ABAP Dictionary

The **fields** make up the columns of the transparent table. The **Key** indicator marks a field as a key field.

You normally **type** a field using a data element (*Field Type*). The **short description** stored for the data element is displayed here once again. The same applies for the technical attributes of the field (*Data Type*, *Length*, and *Dec.Places*). A data element normally uses a domain!

This provides us with a **logical view** of the table. It is a purely technical description. As soon as it is physically converted, the table is in the **database**. The **table content**, that is, the actual application data is also there.

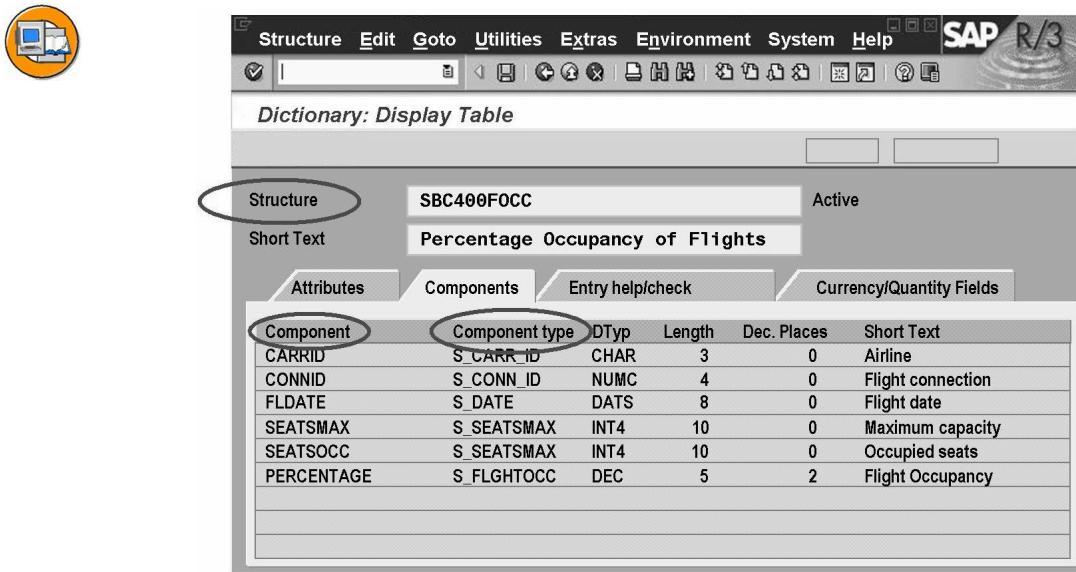


Figure 41: Structures in the ABAP Dictionary

The definition of a flat structure is constructed in almost the same way as that of a transparent table. Only the property of being a key field is not appropriate for structures in accordance with a data model. They do not serve to represent a database table in the *ABAP Dictionary*, but are a **pure type definition** instead. (The *DB Properties* in the menu *Extras* have nothing to do with this topic, but are provided for other implementation options.)

However, we refer to **component** and **component type** as opposed to field and field type, because, in contrast to transparent tables, it is possible to define complex structures. This means that, for example, it is possible to store a (sub)structure component under a component name.



```
DATA wa_focc TYPE sbc400focc.
TABLES spfli.
PARAMETERS pa_carr TYPE spfli-carrid.
```

	carrid	connid	fldate	seatsmax	seatsocc	percentage
wa_focc	L H	400	20021231	4 5 0	1 8 0	4 0

	mandt	carrid	connid	countryfr	cityfrom	airpfrom	countryto	...
spfli	4 0 0	A A	64	US	New York	JFK	US	...

pa_carr	L H
---------	-----

Figure 42: Using Global Structures for Data Objects

A structure normally serves the purpose of centrally providing a data **type** in which **fields from different tables are compiled**. In the ABAP program, you can then define data **objects** using the (Dictionary) structure (structured data type).

A transparent table merely represents a technical description in the *ABAP Dictionary*. Therefore you can use it as a central data type definition, as is the case with a structure. This means that the following syntax variants are also possible, similar to the data object definitions listed above,:

- **DATA wa_spfli TYPE spfli** supplies a data object, with the same type as the line in the transparent table SPFLI.
- **TABLES sbc400focc** supplies a data object, with the same type as the structure SBC400FOCC.
- **PARAMETERS pa_carr TYPE sbc400focc-carrid** supplies an input field, with the same type as the component CARRID of the structure SBC400FOCC.

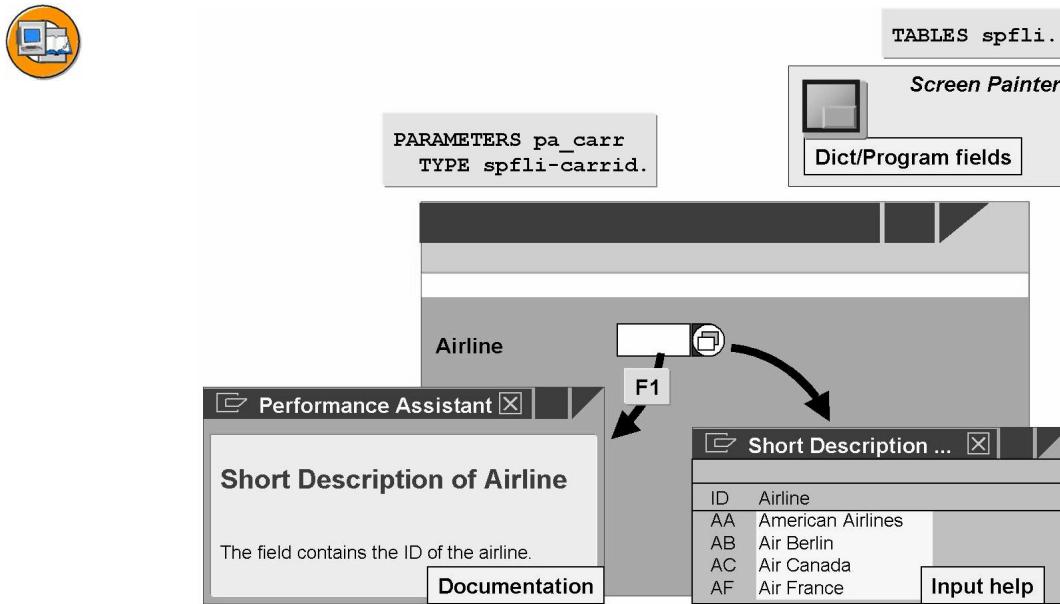


Figure 43: Using Global Structures for Dialogs

If you use an elementary structure component to define an **input field** for a **selection screen**, you can use the semantic attributes as well as the technical attributes: On request, the system automatically generates a **documentation dialog** and an **input help dialog** for the user. It takes the necessary information from the *ABAP Dictionary*. The same applies to fields of transparent tables.

If you use an elementary structure component to set the type for an **input/output field** on a **screen**, you can use the same attributes as with selection screens.

If you have defined a foreign key relationship, the resulting value combinations will appear in the input help on the screen. If there is an error, the system also processes an automatic **error dialog**. The same applies to fields of transparent tables.

For this you must define the data object for the data transport between the screen and the program **using the TABLES statement**.

Exercise 4: Using Global Structures for Data Objects and Dialogs

Exercise Objectives

After completing this exercise, you will be able to:

- Establish the technical names of data types and data objects that are used in user dialogs.
- Run where-used lists within an ABAP program.

Business Example

You have to examine a specified ABAP program to see whether global data types are used in it, and if so, which ones.

Task 1

Executing a Program

1. Display the object list for package **BC400**. Find the program **SAPBC400WBT_GETTING_STARTED**, and navigate to the object list. Throughout the exercise, make sure that you remain in display mode.
2. Run the program to find out how it works.

Task 2

There is an input field on the selection screen.

Continued on next page

1. What information must you pass to the program? (Use the field help.)

2. What values can you enter? (Use input help.)

3. What information does the program provide?

4. What is the technical name of the input field? (Use the Technical Information for the field help.)

Task 3

Two output fields appear on the screen. Find out the screen number. (Choose *Status* from the *System* menu.)

Continued on next page

1. What information is displayed? (Use the field help.)

2. What are the technical names of the display fields? Do the fields belong to a structure with a globally defined type? (Use the Technical Information for the field help.)

Task 4

Use the object list to analyze the source code of the program.

Continued on next page

1. What data objects are there? Where in the program are they defined?

2. Where in the program are they used? (Use the *Where-Used List* in the *ABAP-Editor*.)

3. What data object corresponds to the input field on the selection screen? (Search the object list for a data object with the same name as the field that you found out in step 2.4.)

Solution 4: Using Global Structures for Data Objects and Dialogs

Task 1

Executing a Program

1. Display the object list for package BC400. Find the program **SAPBC400WBT_GETTING_STARTED**, and navigate to the object list. Throughout the exercise, make sure that you remain in display mode.
 - a) Carry out this step as usual.
2. Run the program to find out how it works.
 - a) Carry out this step as usual.

Task 2

There is an input field on the selection screen.

1. What information must you pass to the program? (Use the field help.)
Answer: Short description of an airline
2. What values can you enter? (Use input help.)
Answer: You see a list of airline descriptions: **AA**, **AB**, and so on.
3. What information does the program provide?
Answer: Description and local currency of the selected airline
4. What is the technical name of the input field? (Use the Technical Information for the field help.)
Answer: PA_CAR

Task 3

Two output fields appear on the screen. Find out the screen number. (Choose *Status* from the *System* menu.)

1. What information is displayed? (Use the field help.)
Answer: The short description and the local currency of the selected airline

Continued on next page

2. What are the technical names of the display fields? Do the fields belong to a structure with a globally defined type? (Use the Technical Information for the field help.)

Answer: SBC400_CARRIER-CARRID and SBC400_CARRIER-CURRCODE.

These are the components of the structure SBC400_CARRIER, as visible under *Field Data*.

Task 4

Use the object list to analyze the source code of the program.

1. What data objects are there? Where in the program are they defined?

Answer: Open the node *Fields* and display the respective statements in edit mode, as described in the training material.

2. Where in the program are they used? (Use the *Where-Used List* in the *ABAP-Editor*.)

Answer: After you have selected a data object and executed the where-used list *in programs*, the system displays a list of the locations found.

3. What data object corresponds to the input field on the selection screen? (Search the object list for a data object with the same name as the field that you found out in step 2.4.)

Answer: Open the node *Fields*. There is an entry PA_CAR.



Lesson Summary

You should now be able to:

- Describe the SAP flight data model
- Describe how a structure is structured
- Describe the structure of a structure
- List the basic possible uses of transparent tables and structures in ABAP programs.



Unit Summary

You should now be able to:

- Describe the SAP flight data model
- Describe how a structure is structured
- Describe the structure of a structure
- List the basic possible uses of transparent tables and structures in ABAP programs.

Related Information

... Refer to the online documentation for the *ABAP Dictionary*.

Unit 4

Basic ABAP Language Elements

Unit Overview

Refer to the lesson summaries for an overview of this unit.



Unit Objectives

After completing this unit, you will be able to:

- Define elementary data objects
- Use basic ABAP statements with elementary data objects
- Execute and analyze programs in debugging mode
- Define structured data objects
- Use basic ABAP statements in structured data objects
- Analyze structured data objects in debugging mode
- Define internal tables
- Use basic ABAP statements with internal tables
- Analyze internal tables in debugging mode

Unit Contents

Lesson: Working with Elementary Data Objects.....	76
Exercise 5: Basic ABAP Statements.....	93
Lesson: Working with Structures	99
Exercise 6: Working with Structures	105
Lesson: Working with Internal Tables	111
Exercise 7: Working with Internal Tables	127

Lesson: Working with Elementary Data Objects

Lesson Overview

In this lesson you will become familiar with the difference between data types and data objects (first only the elementary ones) and you will learn how to define these in a program. Also, you will learn basic ABAP statements. You will get to know the *ABAP Debugger* for analyzing program flow.



Lesson Objectives

After completing this lesson, you will be able to:

- Define elementary data objects
- Use basic ABAP statements with elementary data objects
- Execute and analyze programs in debugging mode

Business Example

You will develop your own first data processing tasks and search your programs for semantic errors using the *ABAP Debugger*.

Data Types and Data Objects



Data types describe the attributes of:

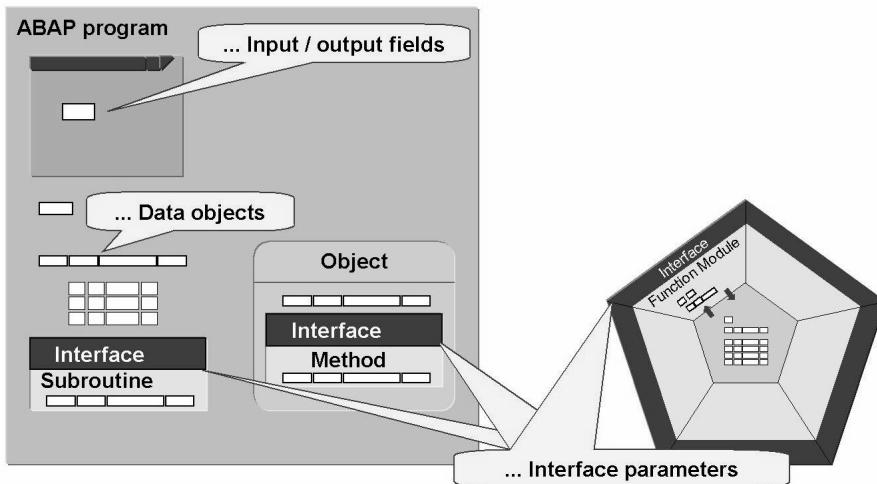


Figure 44: Using Data Types

The type of a data object defines its technical attributes.

The type of an interface parameter defines the type of the values (actual parameters) that are passed when the modularization unit is called.

As well as the technical attributes, the type of an input or output field can provide further information; for example, you can use it to control the field or input help.

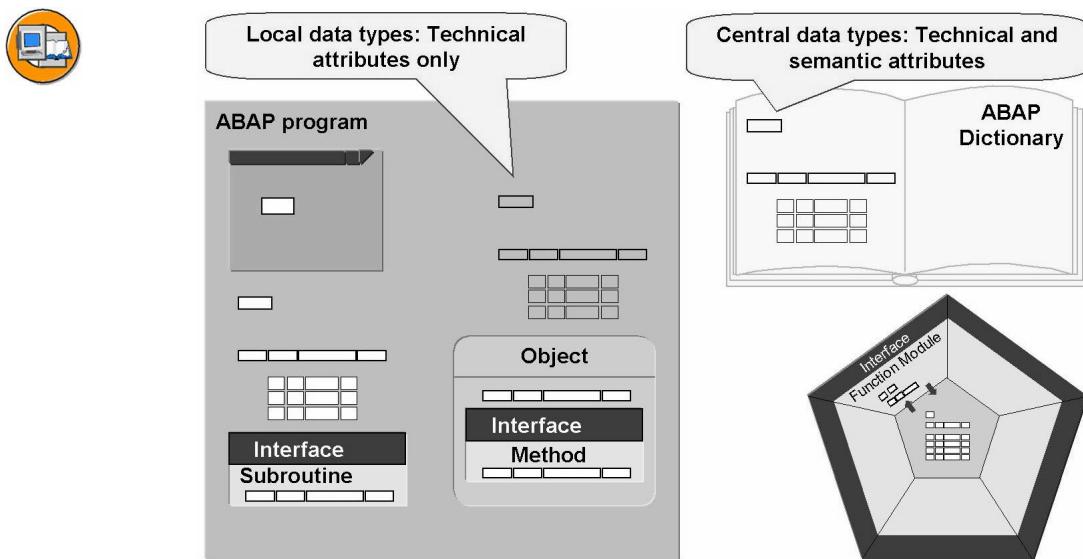


Figure 45: Local and Global Data Types

You can define data types in an ABAP program (local, key word TYPES) or in the *ABAP Dictionary* (global). Using global data types has several advantages:

- Global types are **managed centrally** and you can use them in all repository objects. This increases the consistency of the system and significantly reduces the maintenance effort.
- In the *ABAP Dictionary* you can use the environment analysis (*Where-Used List*).
- Global data types can be linked, through their contents, to business elements, that is, they also contain technical and semantic information. You can then also use them to design screens.

Therefore, when searching for global types, make sure that you take both the technical and semantic attributes into account. Only then you will be able to make full use of the user-friendly maintenance functions and robust nature of your program.

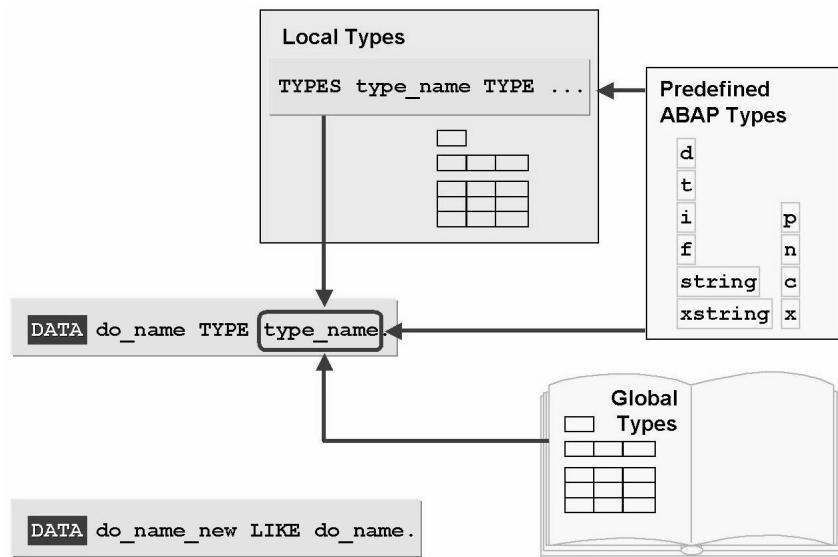


Figure 46: Defining Data Objects

You can use a predefined ABAP type, a local type, or a global type to type a data object.

The following predefined ABAP types are completely specified:

D

(Date), Format: **YYYYMMDD**

T

(Time), Format: **HHMMSS**

I

(Integer)

F

(Floating point number)

STRING

Character sequence, variable length

XSTRING

Byte sequence (Hexadecimal string), variable length

You must define the length for these predefined types when defining the data objects:

C

(Character)

N

(Numeric character)

X

Byte (HeXadecimal)

P

(Packed number), that is, binary-coded decimal number

(Here you must also specify the number of decimal places.)

For more information on predefined ABAP types, refer to the keyword documentation on the TYPES or DATA statement.

You can use the LIKE addition to **refer to the type of a data object already defined.**

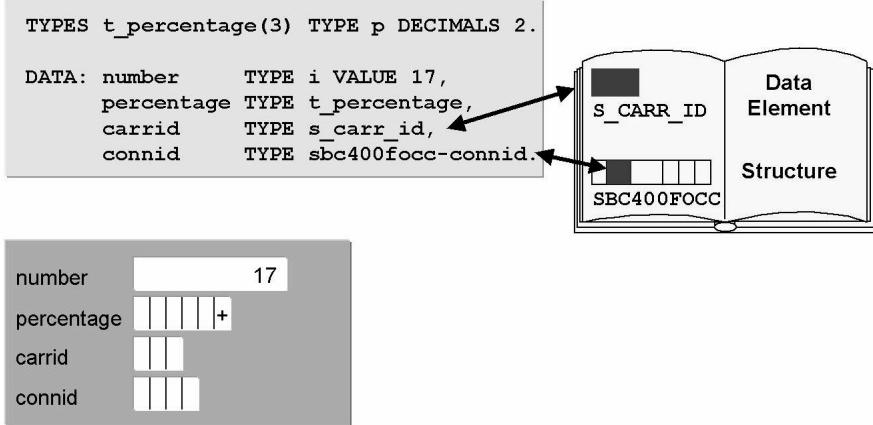


Figure 47: Defining Elementary Data Objects

The predefined ABAP types **D**, **T**, **I**, **F**, **STRING**, and **XSTRING** are complete. This means you can define an elementary data object of these types in the following way:

```
DATA do_name TYPE predefined_ABAP_type.
```

In the case of the predefined ABAP types **C**, **N**, **P**, and **X**, the length has yet to be defined. This means that you should first define an elementary type:

```
TYPES type_name(length) TYPE predefined_ABAP_type.
```

As of SAP Web AS 6.20, this would be:

TYPES type_name **TYPE** predefined_ABAP_type **LENGTH** length.

Using this, you then define the data object:

DATA do_name **TYPE** type_name.

For type P, you can also define the number of decimal places using the DECIMALS addition.

You can use the VALUE addition to pre-assign the value of an elementary data object.

For compatibility reasons, you can define data objects using the DATA statement, **without** having defined the type beforehand (locally in the program using a TYPES statement).

In the case of incompletely predefined ABAP types, the system adds standard values to replace the missing type information.

If there is no type specification at all, the system defines an elementary data object of type C with length one.

You can get the standard lengths as well as other information from the keyword documentation on the TYPES or DATA statement.

The same syntax applies to global types as does to user-defined types: **DATA** do_name **TYPE** global_type_name.



Literal: Fixed data object without ID code

Numeric literal: Type i or p
Positive integers: 123
Negative integers: -123

Text literal: Type c
Character string: 'aBcdE'
Decimal numbers: '123.45'
Floating point numbers: '123.45E01'

Constant: Fixed data object with ID code

CONSTANTS const_name **TYPE** type_name **VALUE** [literal | IS INITIAL].

Figure 48: Fixed Data Objects

You assign a value to each fixed data object in the source code. Therefore, it cannot be changed at runtime.

You can use **literals** to pass fixed values to ABAP statements. There are **numeric literals** and **text literals**. The latter are always enclosed in inverted commas:

- You **can** display integers as numeric literals (in the case of negative numbers, with a preceding sign). The system then maps them onto the data type **I**, if they are within the value range of 4-byte integers. Higher numeric literals are mapped onto data type **p**.
- You **must** display all other literals (character strings, numbers with decimal places, floating point numbers) as text literals. They are mapped onto data type **C** and converted later, if necessary. If you want to include an inverted comma in a text literal, you must enter it twice.

You define constants using the CONSTANTS statement. Their type is defined similarly to the definition of elementary data objects. The VALUE addition is **required** for constants. This is how you define the value of the constants.



Caution: If possible, avoid literals completely when using statements. Instead, define constants. This significantly improves program maintenance.

Basic ABAP Statements

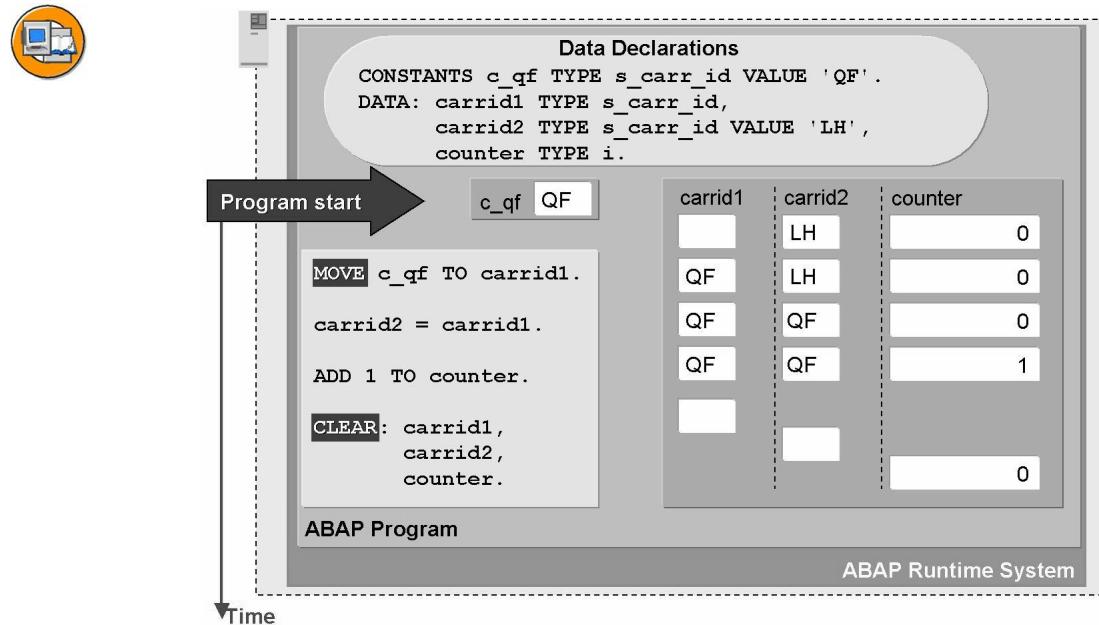


Figure 49: Value Assignments

When a program is started, the program context is loaded into a storage area of the application server and made available for all the data objects. Every elementary data object is pre-assigned the type-related initial value, except if a different value was pre-assigned using the VALUE addition.

You can use the MOVE statement to copy the contents of data objects. Two syntax variants are available for this purpose:

- **MOVE source TO target.**
- **target = source.**

If the two data objects **source** and **target** have different types, the type is automatically converted if there is a conversion rule. For detailed information about copying and the conversion rules, refer to the keyword documentation for the MOVE statement.

The CLEAR statement resets the contents of a data object to the **type-related** initial value. For detailed information about the initial values for a particular type, refer to the keyword documentation on the CLEAR statement.



Data Declarations

```
DATA: max      TYPE sbc400focc-seatsmax,
      occ      TYPE sbc400focc-seatsocc,
      percentage TYPE sbc400focc-percentage.
```

COMPUTE percentage = occ * 100 / max.

The keyword is optional:

```
percentage = occ * 100 / max.
```

ABAP Program

Figure 50: Calculations

In ABAP Objects, you can program **arithmetic expressions** nested to any "level" you require. Valid operations include:

- + Addition
- - Subtraction
- * Multiplication
- / Division
- ** Power
- DIV integral division without remainder
- MOD Remainder after integral division

 **Caution:** Parentheses and operators are keywords and must therefore be enclosed between at least one blank.

The COMPUTE keyword is optional.

Several functions for different data types are predefined in the ABAP runtime environment. For example, the following statement supplies the current length of the content of a character string.

length = STRLEN(cityfrom).

In the case of functions, the opening parenthesis is part of the ID. The rest must again be separated by at least one space.

In general, the standard algebraic rules apply to the **processing sequence**: Expressions in parentheses come first, then powers, then multiplication/division, and finally addition/subtraction.

For detailed information about the available operations and functions, refer to the keyword documentation on the COMPUTE statement.

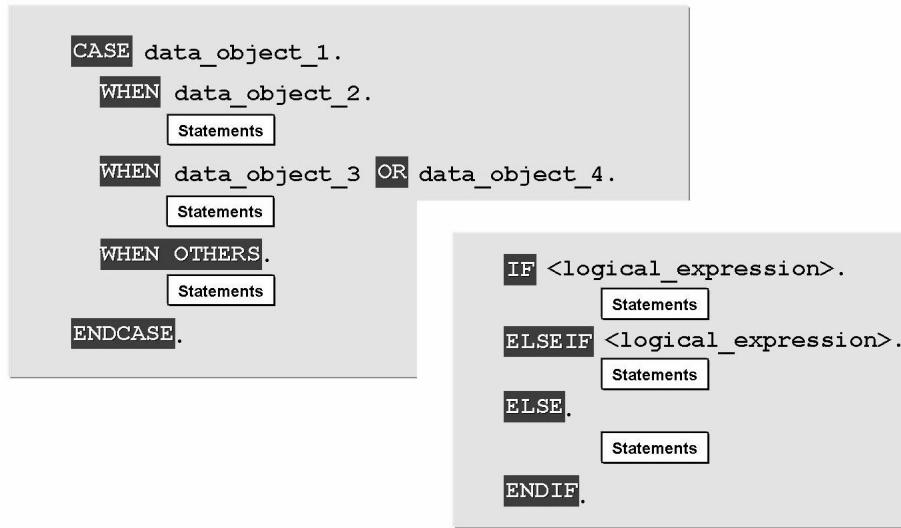


Figure 51: Conditional Branches

In ABAP Objects, you have two ways to execute different sequences of statements, depending on certain conditions.

- With the CASE-ENDCASE construction, the criterion for the execution of a statement block is the **similarity of the data objects**. If no comparison is successful, the system executes the OTHERS branch if it is available. Except for the first WHEN branch, all further additions are optional.
- With the IF-ENDIF construction, you can use **any logical expressions**. If the condition is met, the system executes the relevant statement sequence. If no comparison is successful, then the system executes the ELSE branch, if it is available. Except for the first query, all further branches are optional.

For both constructions the system only ever executes one statement sequence, and this always for the first valid case.



Hint: If, in every condition, you are checking that a variable is equal to a given value, you should use the CASE-ENDCASE construct. It is **more clearly arranged and less runtime-intensive**.

Outside of loops, you can also use CHECK instead of IF. This query places the execution of all statements up to **the end of the current processing block** under one condition. If this is unsuccessful, the system continues with the first statement in the next processing block.

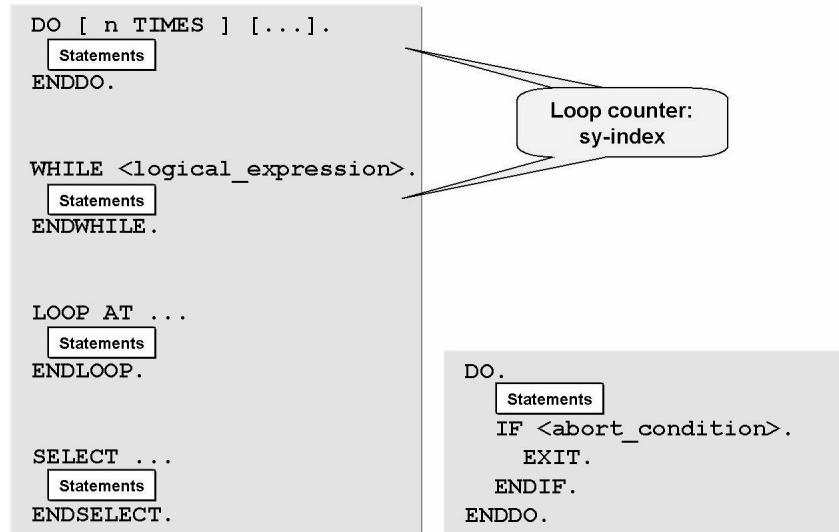


Figure 52: Loops

In ABAP Objects, there are four loop constructions, whereby LOOP-ENDLOOP and SELECT-ENDSELECT represent special cases. In the first two types of loop, the system stores the number of the current loop pass in the **SY-INDEX** field; in the case of nested loops, this always refers to the current loop.

Unconditional/Index-Controlled Loops

The statements between DO and ENDDO are executed until the loop is exited through other statements. You also have the option of specifying the maximum number of loop passes; otherwise you may get an **endless loop**.

Header-Controlled Loops

The statements between WHILE and ENDWHILE are executed only if the condition <logical_condition> has been met.

You can use the statements CHECK and EXIT for different effects on the way the loop is processed. For example, you can construct a footer-controlled loop.



MESSAGE tnnn(message_class) [WITH v1 [v2] [v3] [v4]].			
Type t	Meaning	Behavior	Message appears in
s	Status message	Program continues without interruption	Status line in next screen
i	Information	Program continues after interruption	Modal dialog box
w	Warning	Context dependent	Status bar
e	Error	Context dependent	Status bar
a	Termination	Program aborted	Modal dialog box
x	Short dump	Runtime error MESSAGE_TYPE_X Is triggered	Short dump

Figure 53: Dialog Messages

You can use the MESSAGE statement to send dialog messages. You must specify the three-digit message number and the message class. Furthermore, you can use the WITH addition to include up to four parameters. The system, depending on the message text, inserts these values into the message text.

You control the dialog behavior using the type abbreviation t. There are six different dialog types for messages:

A

(Abort)

The program is ended after a dialog box.

X

(Exit)

The program is exited and a short dump analysis is issued.

E

(Error)

The runtime behavior is context-dependent.

W

(Warning)

The runtime behavior is context-dependent.

I

(Information)

The program is continued after a dialog box.

S

(Success)

The message appears in the status line on the subsequent screen.

For further information about the syntactical alternatives to the MESSAGE statement, refer to the keyword documentation.

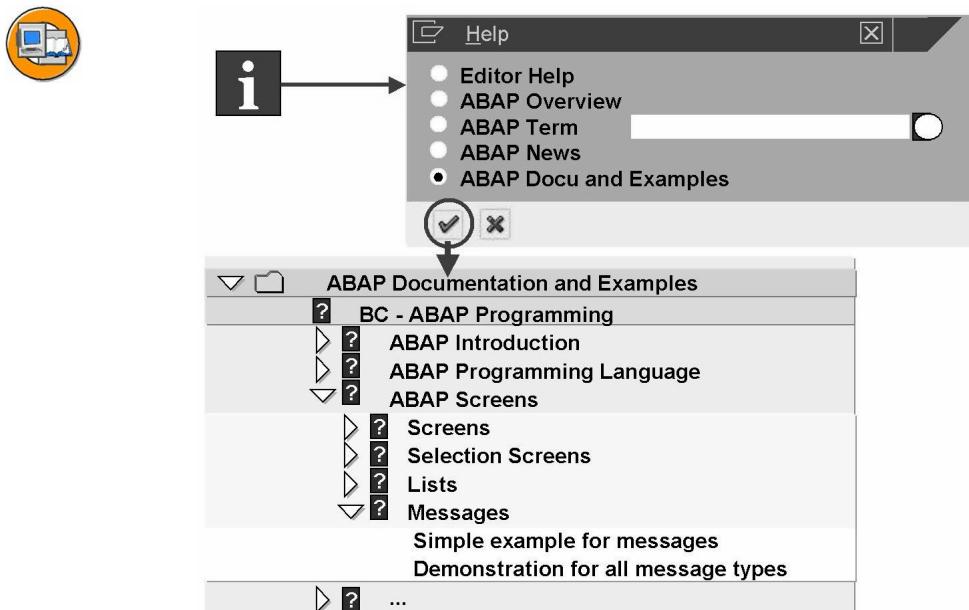


Figure 54: Runtime Behavior for Dialog Messages

Depending on the context in which they appear, some message types show different dialog behavior. You can find a program for testing the resulting possibilities in the ABAP documentation under the section *ABAP Documentation and Examples*



Hint: There you will find executable examples, even for many other keywords and programming techniques.



Note: You can call up the ABAP documentation from the ABAP Editor using the pushbutton *Help for ...*.

Working with the ABAP Debugger

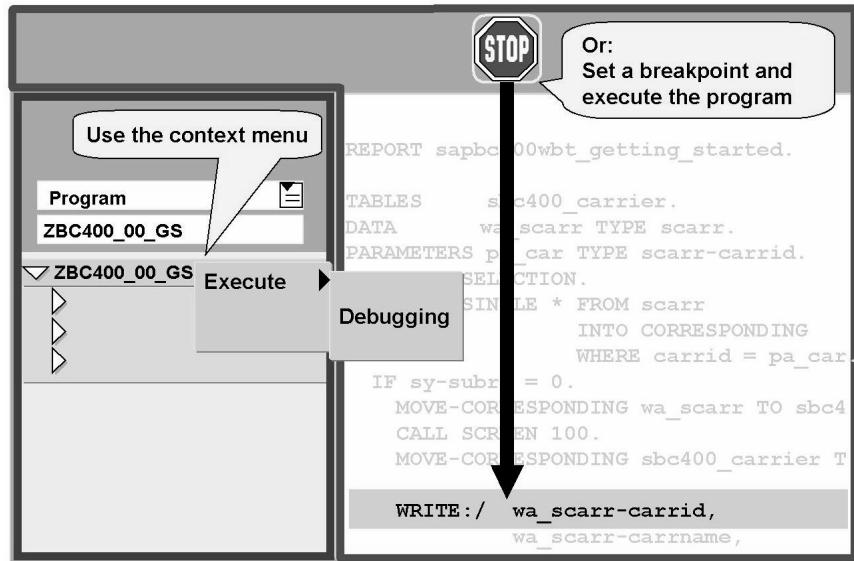


Figure 55: Executing a program in debugging mode

There are several ways to start a program in debugging mode from the *Object Navigator* without having to change the program:

- In the navigation area, choose the path *Execute* → *Debugging* for the selected program.
- In the editing area, select the desired program line and choose *Set/Delete Breakpoint*. Afterwards, start the program by choosing *(Execute) Direct Processing* or *F8*.

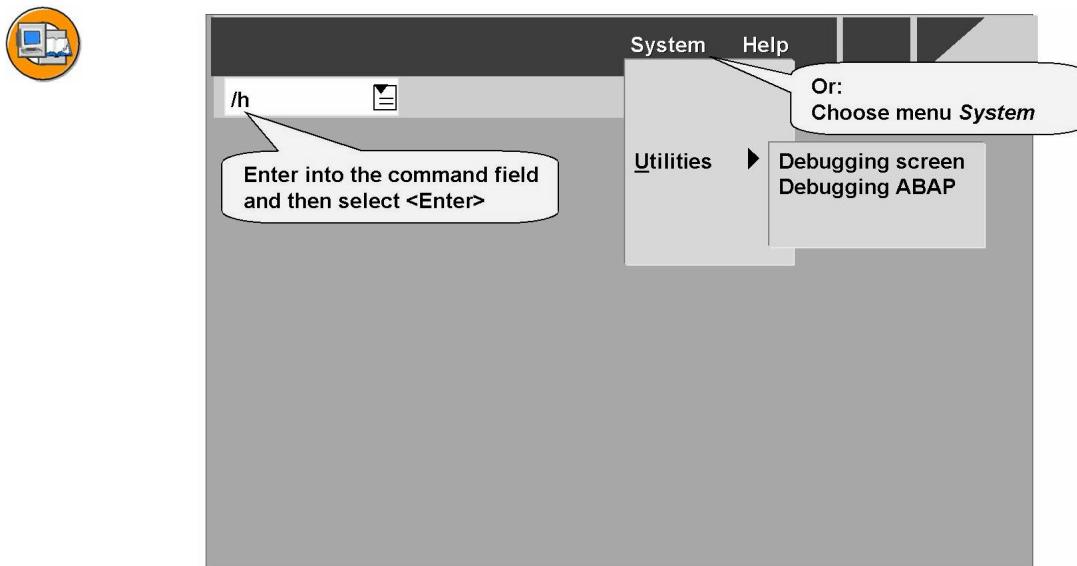


Figure 56: Switching to Debugging Mode at Runtime

If you only want to “debug” certain parts of a program, you should start it first and then switch to debugging mode **before a screen change**. You have two ways of doing this:

- Choose *System* → *Utilities* → *Debugging ABAP or Screen*.
- Enter */H* in the command field in the standard toolbar and press *Enter*.

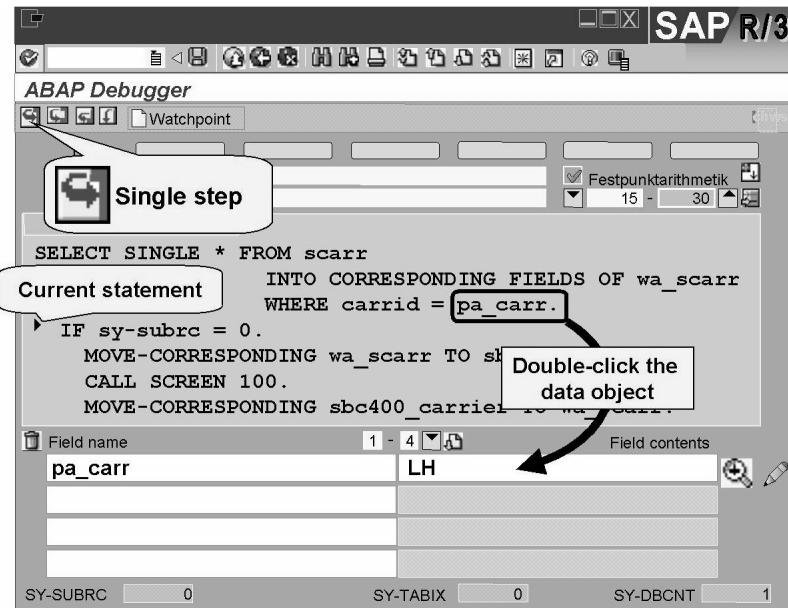


Figure 57: Debugging Mode: Single Step and Field Contents

In debugging mode, you can choose *Single step* to execute the program statement by statement.

You can display the contents of up to eight data objects. Enter the names in the left input field or “double-click” the name in the displayed source code.

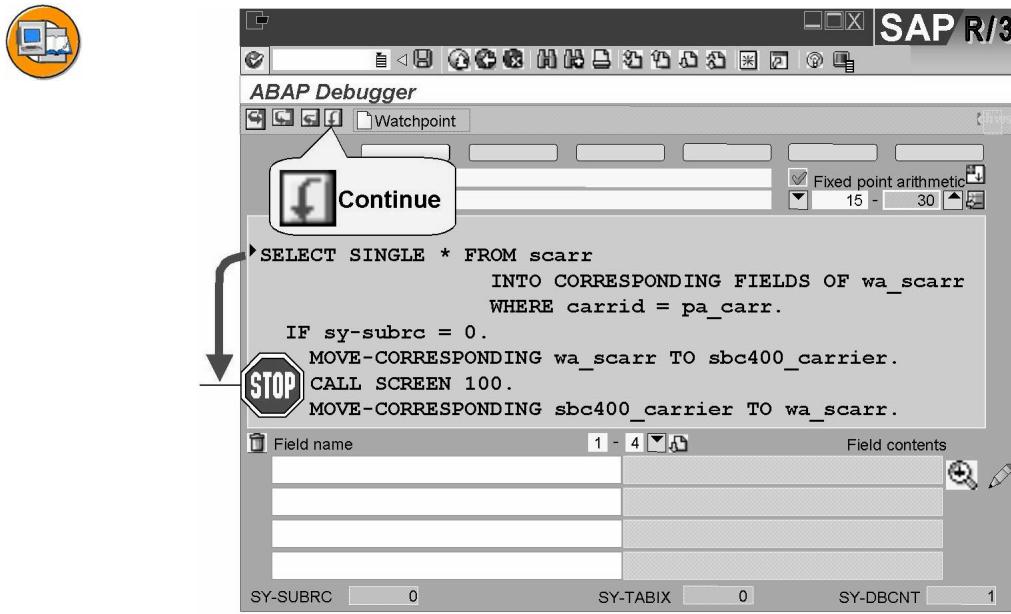


Figure 58: Debugging Mode: Breakpoints

You can set a breakpoint by double-clicking in front of a line of source code. You can also set breakpoints for a specific ABAP statement: *Breakpoint* → *Breakpoint at* → *Statement*.

If you choose *Continue*, the program is executed up to the next breakpoint.

If you *save* the breakpoints, they remain intact while you are logged on to the system. With breakpoints, the *ABAP Workbench* only switches to debugging mode for the developer who set the breakpoints. All other users can execute the program directly or set their own breakpoints.

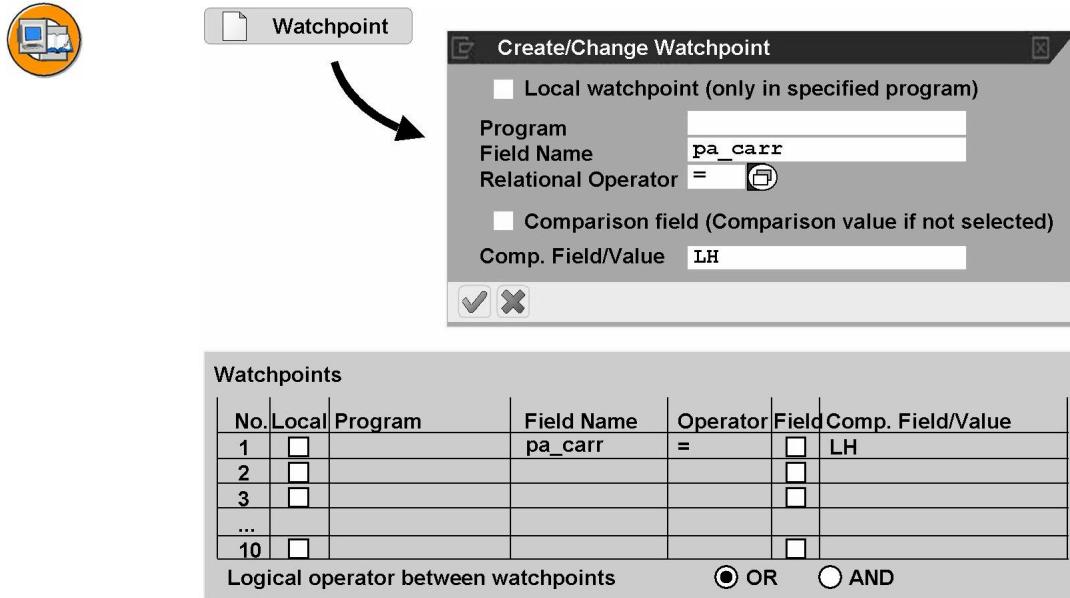


Figure 59: Debugging Mode: Tracing Data Changes

Watchpoints are breakpoints that depend on the field content. You can create the following types of watchpoints:

Field Name

The program is executed until the content of the field has changed.

Field Name, Relational Operator, Comparison Value

The program is executed until the content of the field has changed so that the defined logical condition, relative to a value, is met.

You can set up to ten watchpoints and link these using the logical operators AND or OR.

Exercise 5: Basic ABAP Statements

Exercise Objectives

After completing this exercise, you will be able to:

- Define elementary data objects
- Assign values
- Implement conditional branching
- Perform Calculations

Business Example

You are to create a simple ABAP program for the four basic calculation types. You must be able to enter the values and the arithmetic operator on a selection screen. Display the result in a list.

Task 1

Create program

1. Create the executable program **ZBC400_##_COMPUTE** without a “TOP include”.

Task 2

Design the selection screen.

1. Define the input parameters for two integers (name suggestion: pa_int1, pa_int2) and an arithmetic operator (name suggestion: pa_op).

Task 3

Execute calculations and set up the list output.

1. Additionally, define an elementary data object for the result; type: Packed number with two decimal places (name suggestion: result).
2. Execute the calculation in connection with the specified arithmetic operator.



Hint: Use the CASE statement to do this.

Continued on next page

3. Display the result in a list.

Task 4

Catch errors.

1. Display an error message on the list if the user has specified an invalid arithmetic operator.



Hint: Use the IF statement to do this.

2. Display an error message on the list if the user tries to divide by zero.



Hint: In a later unit you will learn how you can implement the catching of these errors with a more "elegant" user dialog.

Solution 5: Basic ABAP Statements

Task 1

Create program

1. Create the executable program ZBC400_##_COMPUTE without a "TOP include".
 - a) Carry out this step as usual.

Task 2

Design the selection screen.

1. Define the input parameters for two integers (name suggestion: pa_int1, pa_int2) and an arithmetic operator (name suggestion: pa_op).
 - a) See source code excerpt in the model solution.

Task 3

Execute calculations and set up the list output.

1. Additionally, define an elementary data object for the result; type: Packed number with two decimal places (name suggestion: result).
 - a) See source code excerpt in the model solution.
2. Execute the calculation in connection with the specified arithmetic operator.



Hint: Use the CASE statement to do this.

- a) See source code excerpt in the model solution.
3. Display the result in a list.
 - a) See source code excerpt in the model solution.

Task 4

Catch errors.

Continued on next page

1. Display an error message on the list if the user has specified an invalid arithmetic operator.



Hint: Use the IF statement to do this.

- a) See source code excerpt in the model solution.
2. Display an error message on the list if the user tries to divide by zero.



Hint: In a later unit you will learn how you can implement the catching of these errors with a more “elegant” user dialog.

- a) See source code excerpt in the model solution.

Result

Source code extract: SAPBC400TSS_COMPUTE

```
REPORT sapbc400tss_compute.
```

```
PARAMETERS:
```

```
pa_int1  TYPE i,
pa_op(1) TYPE c,
pa_int2  TYPE i.
```

```
DATA result TYPE p DECIMALS 2.
```

```
IF NOT ( pa_op = '+' OR
         pa_op = '-' OR
         pa_op = '*' OR
         pa_op = '/' ).

        WRITE: 'Invalid operator!' (iop).
ELSEIF pa_op = '/' AND pa_int2 = 0.
        WRITE: 'Division by zero!' (dbz).
ELSE.
```

```
CASE pa_op.
  WHEN '+'.
    result = pa_int1 + pa_int2.
  WHEN '-'.
```

Continued on next page

```
        result = pa_int1 - pa_int2.  
WHEN '**'.  
        result = pa_int1 * pa_int2.  
WHEN '/'.  
        result = pa_int1 / pa_int2.  
ENDCASE.  
WRITE: 'Result:'(res), result.  
  
ENDIF.
```



Note: You will learn about the additions in parentheses for the WRITE statement in a later lesson. You can omit them here.



Lesson Summary

You should now be able to:

- Define elementary data objects
- Use basic ABAP statements with elementary data objects
- Execute and analyze programs in debugging mode

Lesson: Working with Structures

Lesson Overview

In this lesson, we will continue with the definition of structured data objects and their analysis using the *ABAP Debugger*. Also, this chapter will teach you how to use basic ABAP statements in structured data objects.



Lesson Objectives

After completing this lesson, you will be able to:

- Define structured data objects
- Use basic ABAP statements in structured data objects
- Analyze structured data objects in debugging mode

Business Example

You are to process your own first table-type data structures and search your programs for semantic errors using the *ABAP Debugger*.

Working with Structures

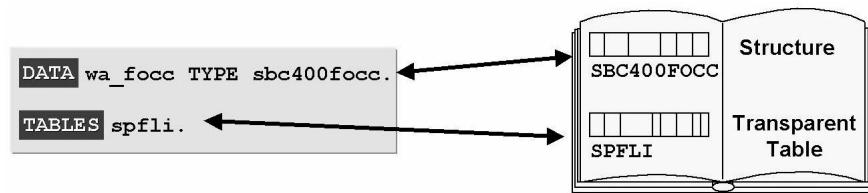


Figure 60: Definition of Structures with Global Types

In ABAP Objects, you can define structured data objects (simply called structures). This allows you to combine values that belong together logically into one object. Structures can be nested. This means that components can be made up of more structures or even tables.

There are two ways to define structures:

- For **DATA struc_name TYPE struc_type** you can use local or global types **struc_type**.
- **TABLES global_struct_type** defines a structure that is technically managed in a special memory area. It has the same name as the transparent table **GLOBAL_STRUC_TYPE** or the global structure **GLOBAL_STRUC_TYPE**. For this second variant, the global structure must be “flat”, that is, all components must have an elementary type. Since SAP R/3 4.0, TABLES structures only serve as interfaces for screen fields.



```
TYPES: BEGIN OF st_flightinfo,
         carrid      TYPE s_carr_id,
         connid      TYPE s_conn_id,
         fldate      TYPE s_date,
         seatsmax    TYPE sflight-seatsmax,
         seatsocc    TYPE sflight-seatsocc,
         percentage(3) TYPE p DECIMALS 2,
      END OF st_flightinfo.

DATA wa_flightinfo TYPE st_flightinfo.
```

	carrid	connid	fldate	seatsmax	seatsocc	percentage
wa_flightinfo	<input type="text"/>					

Figure 61: Defining Structures with Local Types

You can also define structure types locally using the TYPES statement. Here the components are enclosed by

```
BEGIN OF structure_type_name,
...
END OF structure_type_name
```

Each individual component can, in turn, be assigned any type you wish. For more details, refer to the keyword documentation for the TYPES statement.

You define the data object itself in the usual way.

For reasons of compatibility, you can also define a structured data object directly.

```
DATA: BEGIN OF structure_name,  
      ...  
    END OF structure_name.
```



```
DATA: wa_scarr TYPE scarr.  
  
wa_scarr->carrid = 'LH'.  
  
SELECT SINGLE * FROM scarr  
      INTO wa_scarr  
      WHERE carrid = wa_scarr->carrid.  
  
WRITE:/ wa_scarr->carrid,  
       wa_scarr->carrname,  
       wa_scarr->url.
```

	mandt	carrid	carrname	currcode	url
wa_scarr					

Figure 62: Access to Structure Components

Components of a structure are always addressed using a hyphen -:
structure_name-component_name.

For this reason, you should not use hyphens in names.



```
DATA: wa_sflicht TYPE sflicht,  
      wa_focc TYPE sbc400focc.  
  
MOVE-CORRESPONDING wa_sflicht TO wa_focc.
```

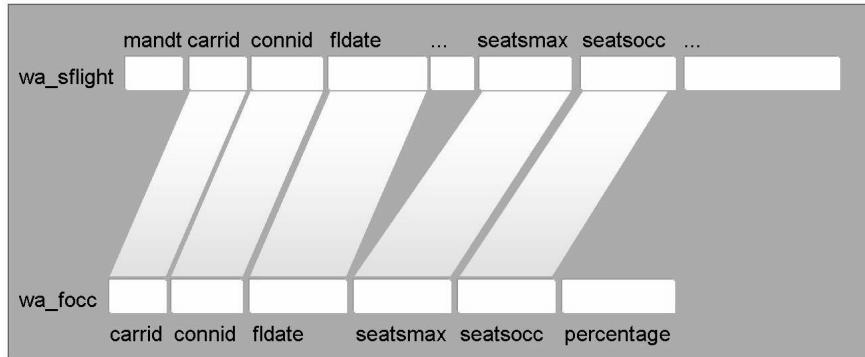


Figure 63: Copying Structure Components with the Same Name

The statement **MOVE-CORRESPONDING source_struct TO target_struct** copies the contents of the structure **source_struct** into the structure **target_struct**, component by component. The value assignment only works if the components have identical names. All other components remain unchanged.

The individual value assignments can be executed using MOVE.

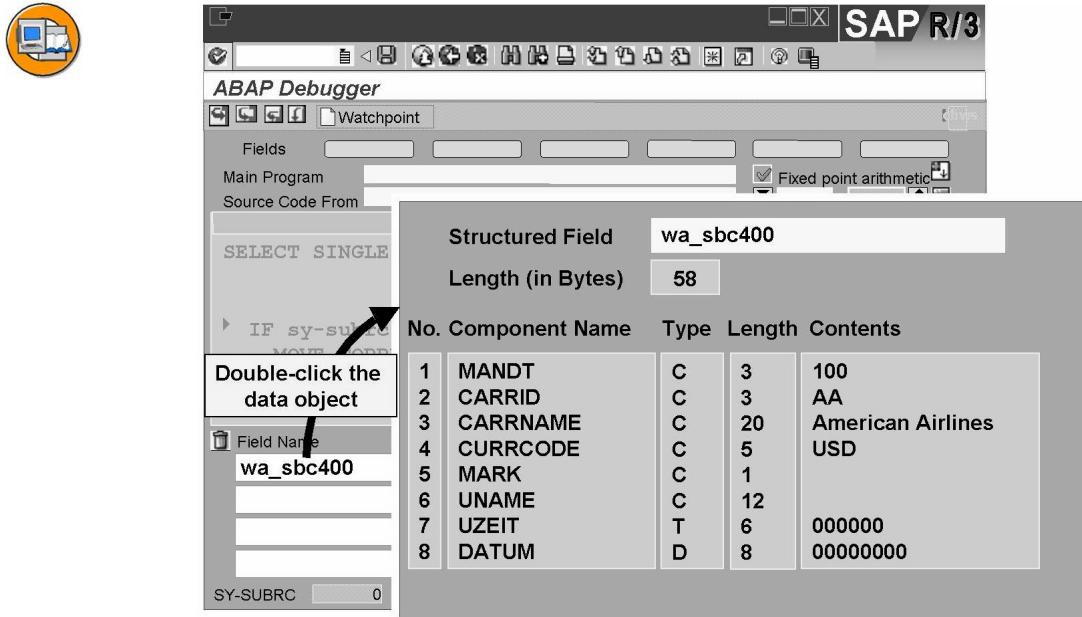


Figure 64: Structures in Debugging Mode

You can trace the field contents of a structure by entering the name of the structure in the left column (using double-click). Double-click this entry again and you will see the component display of the structure.

Exercise 6: Working with Structures

Exercise Objectives

After completing this exercise, you will be able to:

- Use the ABAP Debugger to trace the data flow and understand the connections between processing blocks.
- Use the ABAP statement MOVE-CORRESPONDING to assign values between structures.

Business Example

You have to examine a specified ABAP program to ascertain when and how specific data is transported between certain structured data objects.

Task 1

Executing a program in debugging mode

1. Start the program **SAPBC400WBS_GETTING_STARTED**. On the selection screen, give the field *Airline* the value **LH**. In the command field, enter **/H** and then execute the program. You are now in debugging mode.
2. Check that all of the data objects are initial. Transfer all of the data objects defined in the program into the field view. Find out about the structure and type of the individual components.
3. Step through the program using the single-step function.

Task 2

Analyzing field contents

Continued on next page

1. Which fields of the structure WA_SCARR does the SELECT statement fill? What is the value of system field SY-SUBRC after the statement?

2. Now observe how fields are copied from the structure WA_SCARR to the structure SBC400_CARRIER. Which field values are copied?

3. The ABAP statement CALL SCREEN 100 processes screen 100. On the screen, enter appropriate values for the user name, date, and time, and continue with the program. Now observe how fields are copied from the structure SBC400_CARRIER into the structure WA_SCARR.

Task 3

Following up the list structure

1. Finally, observe how the WRITE statement constructs the list.



Hint: After the first WRITE statement, an extra button appears in the application toolbar, which allows you to display the current contents of the list buffer at any time.

Task 4

Setting breakpoints and changing field contents

Continued on next page

1. Restart the program in Debugging mode. Set a breakpoint at all MOVE-CORRESPONDING statements. Before the screen is processed, assign a (different) name to the field UNAME of the structure SBC400_CARRIER.



Hint: The *Change Field Content* button is beside the input/output field.

2. Repeat the exercise 1.1. Now set a breakpoint at the statement CALL SCREEN. Additionally, set a field content-dependent watchpoint for the condition that a field value of the structure WA_SCARR changes. Continue the program by selecting *Continue*, and check why the debugging of the program is stopped in each case.

Solution 6: Working with Structures

Task 1

Executing a program in debugging mode

1. Start the program **SAPBC400WBS_GETTING_STARTED**. On the selection screen, give the field *Airline* the value **LH**. In the command field, enter **/H** and then execute the program. You are now in debugging mode.
 - a) Carry out this step as described in the training material.
2. Check that all of the data objects are initial. Transfer all of the data objects defined in the program into the field view. Find out about the structure and type of the individual components.
 - a) Carry out this step as described in the training material.
3. Step through the program using the single-step function.
 - a) Carry out this step as described in the training material.

Task 2

Analyzing field contents

1. Which fields of the structure **WA_SCARR** does the **SELECT** statement fill? What is the value of system field **SY-SUBRC** after the statement?
Answer: all, zero
2. Now observe how fields are copied from the structure **WA_SCARR** to the structure **SBC400_CARRIER**. Which field values are copied?
Answer: Fields with the same name - therefore, **MANDT**, **CARRID**, **CARRNAME**, and **CURRCODE**.
3. The ABAP statement **CALL SCREEN 100** processes screen 100. On the screen, enter appropriate values for the user name, date, and time, and continue with the program. Now observe how fields are copied from the structure **SBC400_CARRIER** into the structure **WA_SCARR**.
 - a) Carry out this step as described in the training material.

Task 3

Following up the list structure

Continued on next page

1. Finally, observe how the WRITE statement constructs the list.



Hint: After the first WRITE statement, an extra button appears in the application toolbar, which allows you to display the current contents of the list buffer at any time.

- a) Carry out this step as described in the training material.

Task 4

Setting breakpoints and changing field contents

1. Restart the program in Debugging mode. Set a breakpoint at all MOVE-CORRESPONDING statements. Before the screen is processed, assign a (different) name to the field UNAME of the structure SBC400_CARRIER.



Hint: The *Change Field Content* button is beside the input/output field.

- a) Carry out this step as described in the training material.
2. Repeat the exercise 1.1. Now set a breakpoint at the statement CALL SCREEN. Additionally, set a field content-dependent watchpoint for the condition that a field value of the structure WA_SCARR changes. Continue the program by selecting *Continue*, and check why the debugging of the program is stopped in each case.
 - a) Carry out this step as described in the training material.



Lesson Summary

You should now be able to:

- Define structured data objects
- Use basic ABAP statements in structured data objects
- Analyze structured data objects in debugging mode

Lesson: Working with Internal Tables

Lesson Overview

In this lesson, you will learn how to define internal tables and use them in ABAP programs. You will also see how they appear in the *ABAP Debugger*.



Lesson Objectives

After completing this lesson, you will be able to:

- Define internal tables
- Use basic ABAP statements with internal tables
- Analyze internal tables in debugging mode

Business Example

You need to develop your own first processing of table-type data structures and search your programs for semantic errors using the *ABAP Debugger*.

Working with Internal Tables

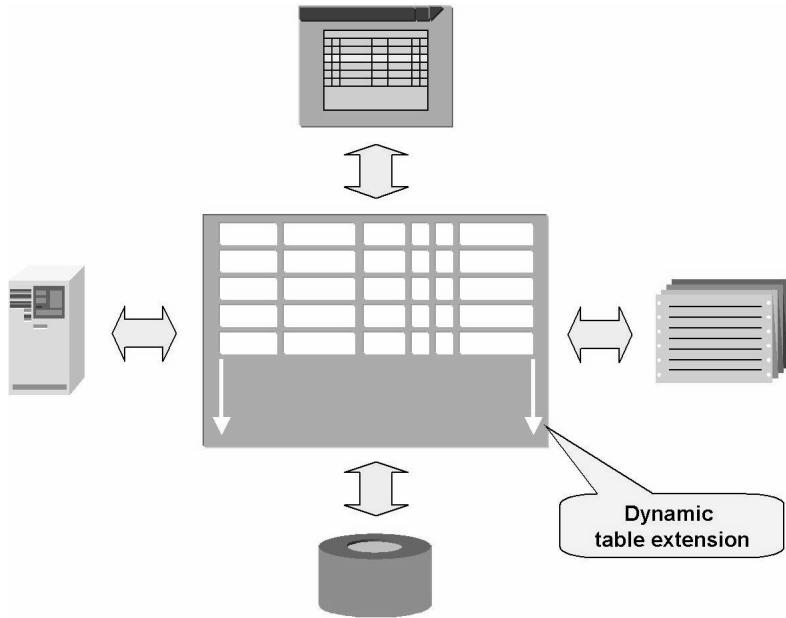


Figure 65: Internal Tables

Internal tables are data objects that allow you to retain several data records with the same structure in the working memory. In principle, the number of data records is unlimited. It is only restricted by the capacity limits of specific system installations.

The **ABAP runtime system** dynamically manages the length of internal tables. This removes the necessity for you to carry out any work concerning working memory management.

The individual data records in an internal table are known as table **rows** or table entries. For this reason, the individual components in a row are referred to as **columns** of the internal table. The line type of an internal table can be any ABAP data type - elementary, structured, or another internal table.

Internal tables are therefore a simple way of processing large data sets in a structured manner. Typical uses include:

- Temporarily storing data from database tables for future processing
- Structuring and formatting data for output
- Formatting data for the use of other services



CARRID CONNID DISTANCE				Row type
Row index	CARRID	CONNID	DISTANCE	
①	AA	0017	2,572	• Components • Sequence • Unique / non-unique key
②	LH	0400	6,162	
③	LH	0400	7,273	
④	QF	0005	10,000	
⑤	SQ	0866	1,625	
⑥	UA	0007	2,572	Index access Key access

Figure 66: Attributes of Internal Tables

The data type of an internal table is completely specified by the following attributes:

Line type

This is the source of the attributes of the individual columns. You normally specify a structure type but any data types are possible.

Key definition

The key columns and their sequence define the criteria by which the table rows are identified. Depending on the access type, the key can be defined as **unique** or **non-unique**. In the case of unique keys, there are no multiple entries in internal tables with identical keys.

Access type

- With a **key access** - as with database tables - you access using the field contents.

Example: A read access with the search term `carrid = 'UA'` and `connid = '0007'` to an internal table using the unique key CARRID and CONNID and the data shown above returns exactly one data record.

- Index access:** Unlike database tables, the system may number the rows for an internal table. It is therefore possible to access table rows using this **index**.

Example: A read access to a data record with index 5 returns the fifth data record of the internal table.



	Index tables		Hashed table
Table kind	STANDARD TABLE	SORTED TABLE	HASHED TABLE
Index access 	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Key access 	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Unique / non-unique key	NON- UNIQUE	UNIQUE NON-UNIQUE	UNIQUE
Access Using			Keys only

Figure 67: Connection Between Table Kind and Access Type

Another internal table attribute is the **table kind**. Internal tables can be divided into three table kinds according to the access types that are possible in each case:

- With **standard tables** the row numbering is maintained internally. Both index and key accesses are possible.
 - With **sorted tables** the data records are always sorted according to key and saved. Here too, the index is maintained internally. Both index and key accesses are possible.
 - With **hashed tables** the data records are managed **optimized to runtime**. A unique key is required. With hashed tables only key accesses are possible.

Which table type you should use in each case depends on how that table's entries are generally going to be accessed:

- For index accesses you should normally use standard tables.
 - Sorted tables are best for unique keys and fixed sorting.
 - With hashed tables the runtime optimization is only noticeable if the accesses are mainly of the read type with a unique key.

In this course, we only deal with standard tables; however - with the exception of a few special cases - the syntax is **identical for all three table kinds**.

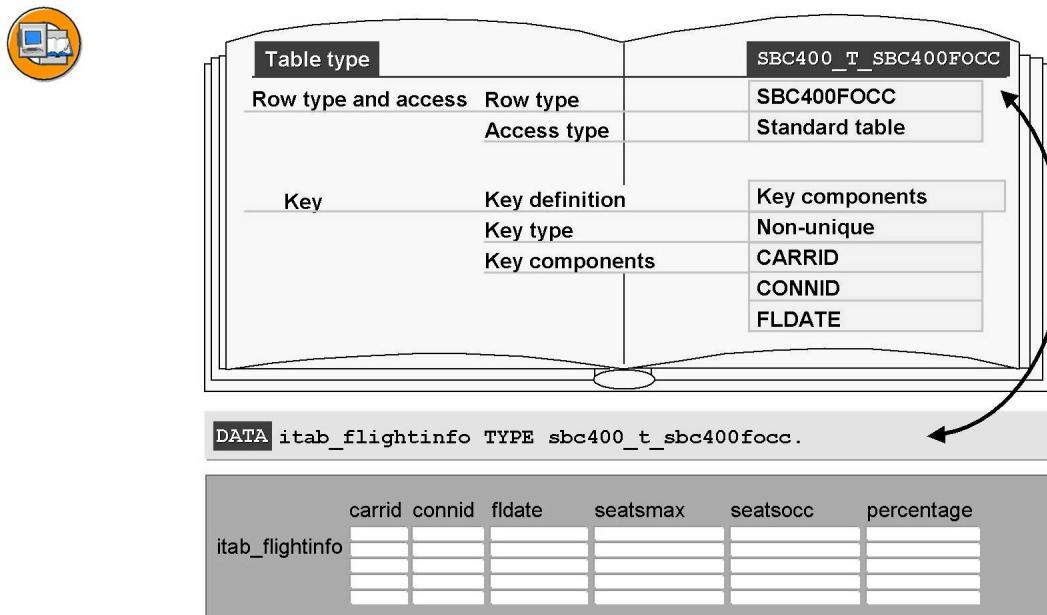


Figure 68: Defining Internal Tables with Global Types

Table types can be defined locally in a program or globally in the ABAP Dictionary.

DATA itab_name TYPE itab_type can have a **local or global** type **itab_type**. Detailed information about the definition of global table types in the ABAP Dictionary is available in the online documentation under *SAP Web Application Server → ABAP Workbench → BC - ABAP Dictionary → Types → Table Types*.



```
TYPES: tt_flightinfo
      TYPE STANDARD TABLE OF sbc400focc
      WITH NON-UNIQUE KEY carrid connid fldate.

DATA itab_flightinfo TYPE tt_flightinfo.
```

	carrid	connid	fldate	seatsmax	seatsocc	percentage
itab_flightinfo						

Figure 69: Defining Internal Tables with Local Types

You can also define internal table types locally using the TYPES statement. The table type is specified between **TYPE** and **TABLE OF**. The line type comes after **TABLE OF**. The key fields are listed after the **WITH** addition. Note that the order of the key fields plays a part here. For detailed information about defining local table types, refer to the keyword documentation for the TYPES statement.

You define the data object itself in the usual way.

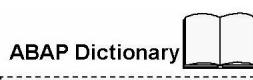
For compatibility reasons you can also define an internal table directly:

```
DATA itab_name TYPE table_kind TABLE OF struc_type WITH
key_def.
```



ABAP program

```
DATA wa TYPE spfli.
SELECT ... FROM spfli
INTO wa ...
TABLES sbc400focc.
DATA wa TYPE
sbc400focc.
DATA it TYPE
sbc400_t_sbc400focc.
AA 17 ...
AA 64 ...
...
LH400 ...
LH402 ...
```



Transparent table
SPFLI (Description)

Database

SPFLI
AA 17 ...
AA 64 ...
... ...
LH400 ...
LH402 ...

Structure
SBC400FOCC

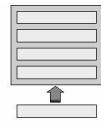
Structure
SBC400FOCC

Table type
SBC400_T_SBC400FOCC

Figure 70: Overview: Typing Structured Data Objects

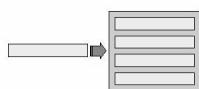


Append



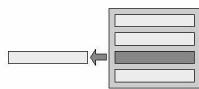
APPEND wa to itab.

Insert



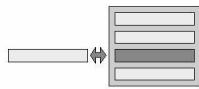
INSERT wa INTO TABLE itab <condition>.

Read



READ TABLE itab INTO wa <condition>.

Change



MODIFY TABLE itab FROM wa [<condition>].

Delete



DELETE itab <condition>.

Figure 71: Overview: Single Record Accesses

The following single record operations are available for internal tables: In each case, **wa** represents a structure that **must** have the **same type** as the **line type** of the internal table **itab**.

APPEND

Appends the contents of a structure to an internal table. This operation can only be used with standard tables.

INSERT

Inserts the contents of a structure into an internal table. In a standard table these contents are appended, in a sorted table they are inserted in the right place, and in a hashed table they are inserted according to the hash algorithm.

READ

Copies the contents of an internal table row into a structure.

MODIFY

Overwrites an internal table row with the contents of a structure.

DELETE

Deletes a row of an internal table.

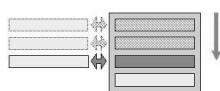
COLLECT

Accumulates the contents of a structure in an internal table. This statement can only be used for tables whose non-key fields are all numeric. If the keys are identical, the numeric values are added.

For detailed information about the ABAP statements described here, refer to the relevant keyword documentation.



Loop: For all single record operations



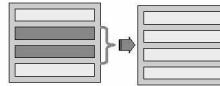
```
LOOP AT itab INTO wa <condition>.  
ENDLOOP.
```

Delete



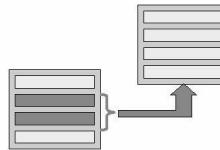
```
DELETE itab <condition>.
```

Inserting several rows from another internal table



```
INSERT LINES OF itab2  
<condition2>  
INTO itab1 <condition1>.
```

Appending several rows from another internal table



```
APPEND LINES OF itab2  
<condition2>  
TO itab1.
```

Figure 72: Overview: Processing Sets of Records

The following set operations are available for internal tables: In each case, **wa** represents a structure that **must** have the **same type** as the **line type** of the internal table **itab**.

LOOP ... ENDLOOP

The LOOP places the rows of an internal table one by one into the structure specified in the INTO clause. All single record operations can be executed within the LOOP. In this case, for the single record operations, the system identifies the row to be processed.

DELETE

Deletes the rows of the internal table that satisfy the condition <condition>.

INSERT

Copies the contents of several rows of an internal table to another internal table.

APPEND

Appends the contents of several rows of an internal table to another standard table.

For detailed information about the ABAP statements described here, refer to the relevant keyword documentation.

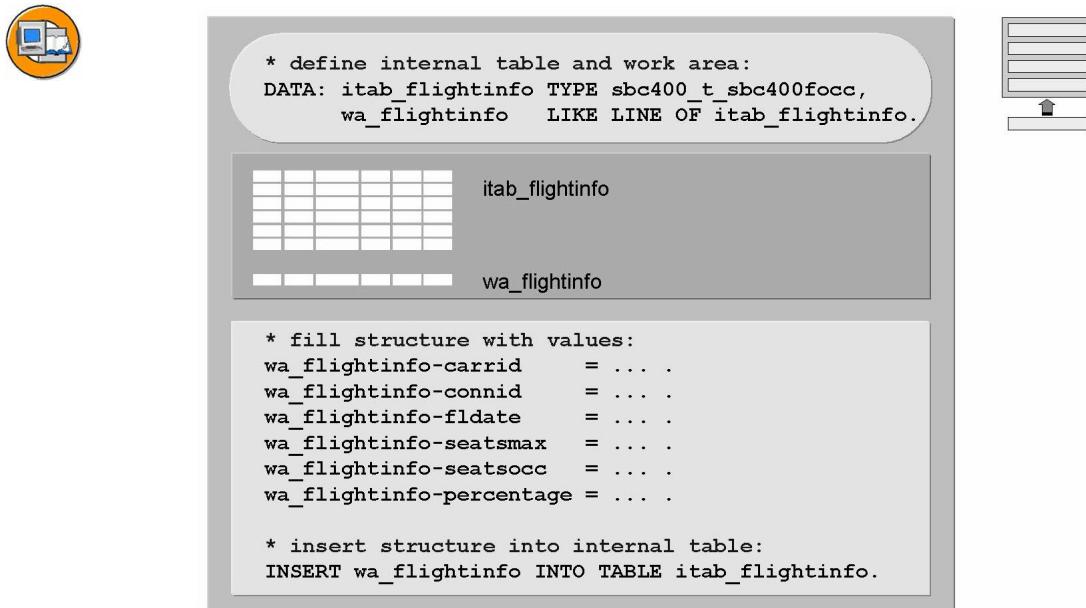


Figure 73: Syntax Example: Filling Line by Line

You can insert rows into an internal table by first filling a structure with the required values and then inserting it into the internal table using the INSERT statement. In the case of standard tables, the row is appended to the table. For sorted tables and hashed tables, the system inserts the row by referring to the key.

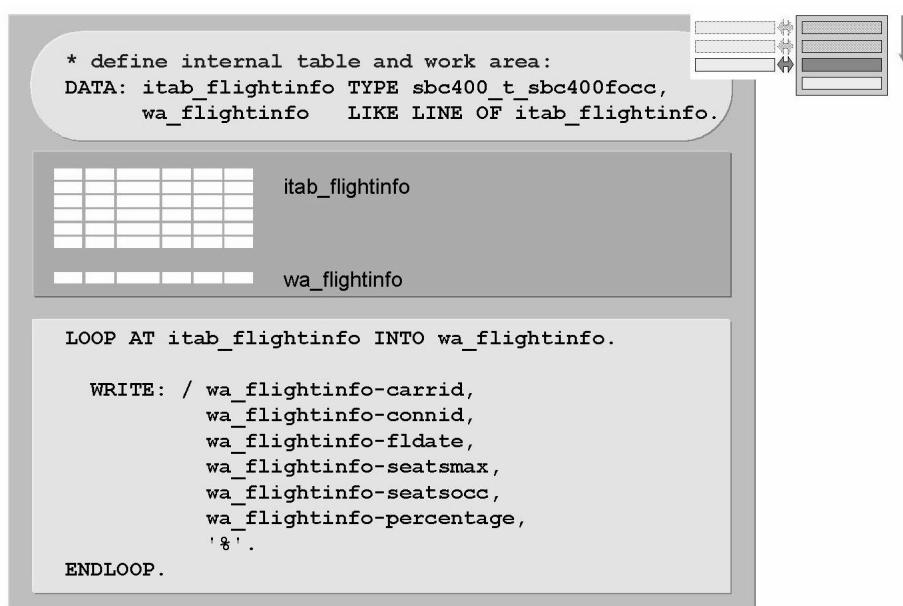


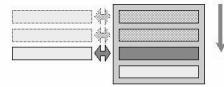
Figure 74: Syntax Example: Reading Contents Using a Loop

You can use a LOOP to read and process the contents of an internal table by accessing the structure components within the loop. In this example, the WRITE statement is used to create a list from the field contents.

If you want to change the contents of the internal table, you must first change the value of the components within the loop and then overwrite the internal table row using the MODIFY statement.

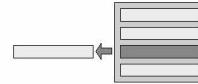


```
LOOP AT itab_flightinfo INTO wa_flightinfo  
      FROM 1 TO 5.
```



```
      WRITE: / wa_flightinfo-carrid,  
              wa_flightinfo-connid,  
              wa_flightinfo-fldate,  
              wa_flightinfo-seatsmax,  
              wa_flightinfo-seatsocc,  
              wa_flightinfo-percentage,  
              '%'.  
ENDLOOP.
```

```
READ TABLE itab_flightinfo INTO wa_flightinfo  
      INDEX 3.
```



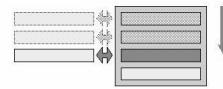
```
      WRITE: / wa_flightinfo-carrid,  
              wa_flightinfo-connid,  
              wa_flightinfo-fldate,  
              wa_flightinfo-seatsmax,  
              wa_flightinfo-seatsocc,  
              wa_flightinfo-percentage,  
              '%'.  
.
```

Figure 75: Syntax Example: Reading Using the Index

You can use the INDEX addition to restrict access to specific row numbers. Index operations can only be performed on index tables. Therefore, these operations are only supported for standard and sorted tables. In the above example, the system only runs through the first five rows in the internal table. In the example below, the system reads the third row of the internal table.



```
LOOP AT itab_flightinfo INTO wa_flightinfo
  WHERE carrid = 'LH'.
  WRITE: / wa_flightinfo-carrid,
          wa_flightinfo-connid,
          wa_flightinfo-fldate,
          wa_flightinfo-seatsmax,
          wa_flightinfo-seatsocc,
          wa_flightinfo-percentage,
          '%'.
ENDLOOP.
```



```
READ TABLE itab_flightinfo INTO wa_flightinfo
  WITH TABLE KEY carrid = 'LH'
        connid = '0400'
        fldate = sy-datum.
IF sy-subrc = 0.
  WRITE: / wa_flightinfo-seatsmax,
          wa_flightinfo-seatsocc,
          wa_flightinfo-percentage,
          '%'.
ENDIF.
```

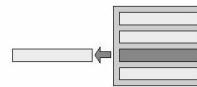


Figure 76: Syntax Example: Reading Using the Key

With the WHERE addition you can restrict access to rows with certain values. Key operations are supported for **all table kinds**. With a fully specified key, key accesses to sorted and hashed tables are generally less runtime-intensive than key accesses to standard tables.

In the above example, the system only runs through those internal table rows in which the CARRID field has the value **LH**.



Hint: With regards to runtime requirements, the sorted table is most suitable for this type of processing.

The second example shows the syntax for reading a line in the internal table with a fully specified key. The return code **SY-SUBRC** is set to zero if a corresponding row is in the internal table.



Hint: With regards to runtime requirements, the hashed table is most suited to this type of single record access if the number of contained data records is **sufficiently high**.

Note that with single record accesses using a key, you must fill **all** key fields using the **WITH TABLE KEY** addition. If you want to limit the contents of any field, you must use the **WITH KEY** addition.

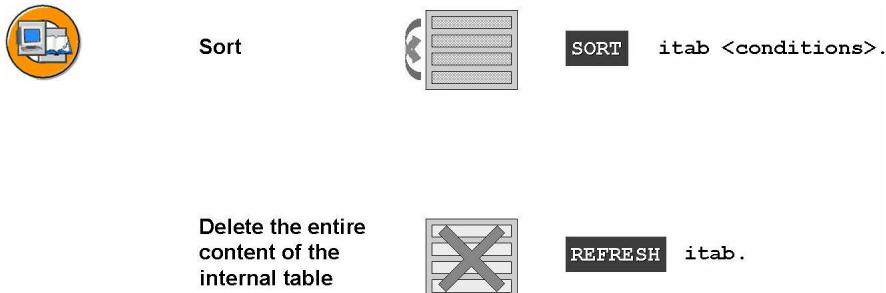


Figure 77: Operations on the Whole Internal Table

The following operations affect the **whole** internal table:

SORT

You can use this to sort any number of columns in a standard or hashed table in ascending or descending order. Culture-specific sorting rules can be taken into account, if necessary.

REFRESH

This deletes the entire contents of the internal table. A part of the previously used working memory remains available for future insertions.

FREE

This deletes the entire contents of the internal table and releases the previously used working memory.

CLEAR

Unlike all other data objects, on internal tables with **no header** this statement has the same effect as the REFRESH statement.

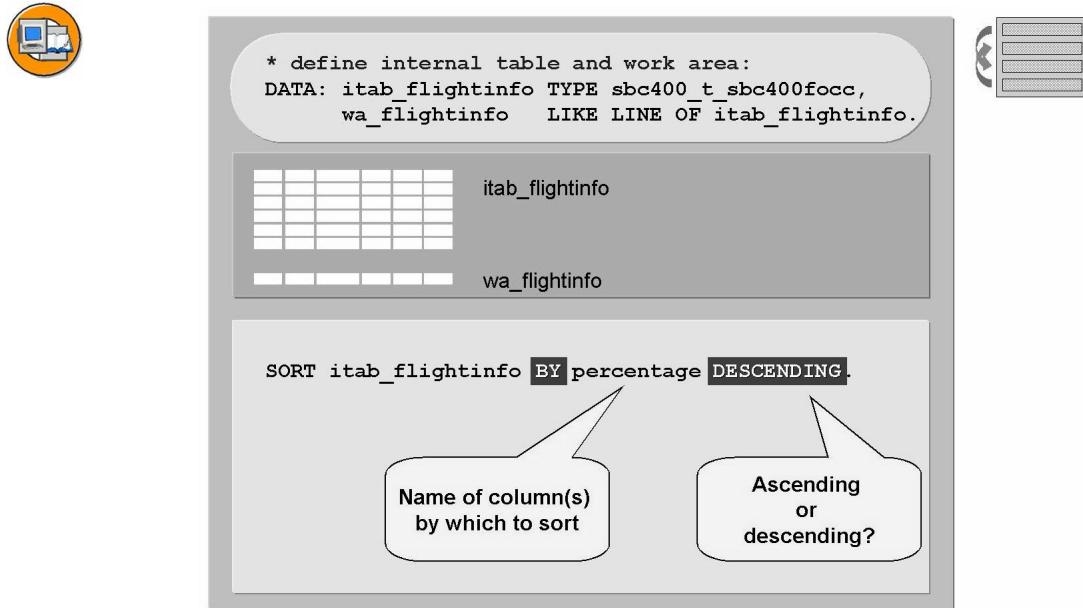


Figure 78: Syntax Example: Sorting a Standard Table

You can sort standard tables by the table key or by any column, simply by entering the column name after the BY addition to the SORT statement.

```

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

Ascending sorting is the default. If you use the AS TEXT addition, the system will take the culture-specific sorting rule into account when sorting by character-type fields. **Only** if you use the STABLE addition will the relative order of data records, which have identical sort keys, remain intact during sorting.

In this example, the data records of the internal table are sorted descendingly by the percentage occupancy of flights.

For more details, refer to the keyword documentation for the SORT statement.

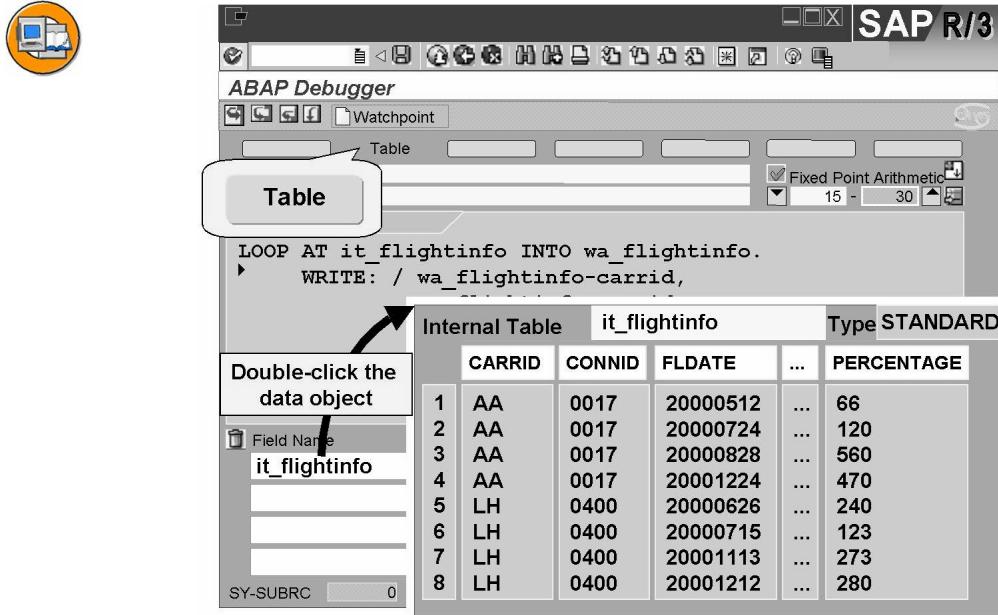


Figure 79: Internal Tables in Debugging Mode

You can trace the row contents of an internal table by entering the name in the left column. To see the table display, double-click this entry.

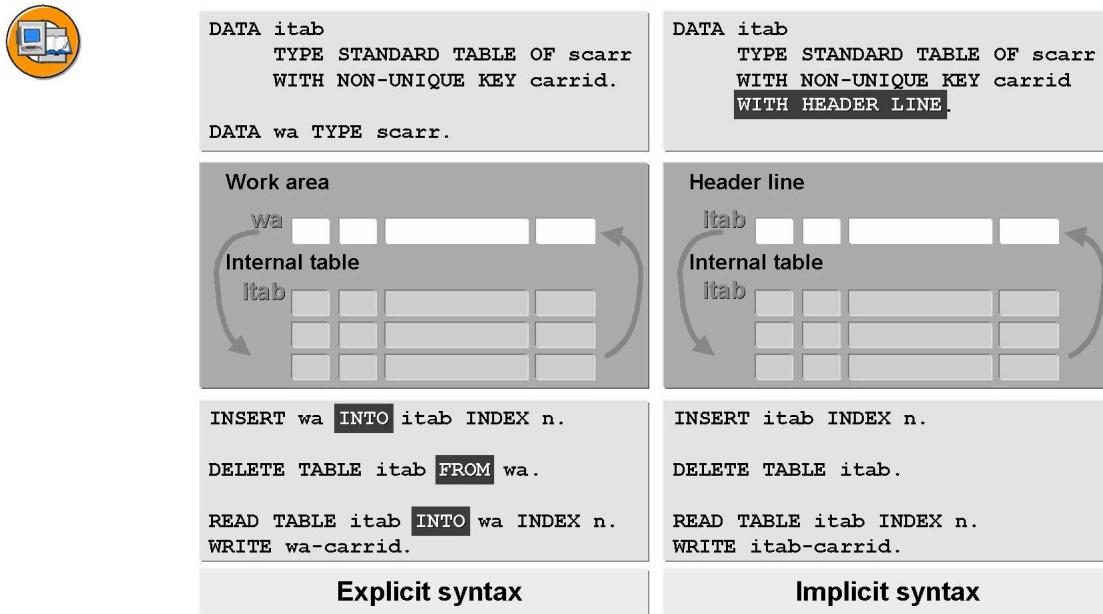


Figure 80: Comparison: Internal Tables with and Without Header Lines

You can define internal tables with or without a header line (WITH HEADER LINE addition). An **internal table with a header line** consists of a work area (the header line) and the actual table body. Both objects are addressed **with the same name**.

How this name is interpreted depends on the context. For example, with MOVE the name refers to the header line, while with SEARCH it is interpreted as the table body. To avoid confusion, it is recommended that you use **internal tables without header lines**. This is especially apparent in the case of nested tables. Furthermore, within object-oriented contexts of ABAP Objects, you can now **syntactically** only use internal tables without a header.

On the other hand, you can often use a shortened syntax (APPEND, INSERT, MODIFY, COLLECT, DELETE, READ, LOOP AT) for operations on internal tables with headers.

You can always uniquely address the body of an internal table `itab` by using the following syntax: `itab[]`. This is irrespective of whether the internal table has a header or not.

Example:

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

```
DATA itab2 LIKE itab1.
```

`itab1 = itab2`. Only the header is copied.

`itab1[] = itab2[]`. The actual table content is copied.

Exercise 7: Working with Internal Tables

Exercise Objectives

After completing this exercise, you will be able to:

- Search for suitable table types in the *ABAP Dictionary*
- Define internal tables based on a global table type
- Fill internal tables using array fetch
- Process the content of internal tables using a loop

Business Example

You need to create an ABAP program that displays additional information on all existing flight departure times in a list.

The data is to be read from the database table SPFLI.

Task 1

Define internal table

1. Create the executable program **ZBC400_##_ITAB_LOOP** without a "TOP include".
2. Buffer the data from the database table SPFLI in an internal table. You should therefore define an internal table with a line type that is compatible with the line structure of SPFLI.

In the *ABAP Dictionary*, search for all (internal) table types that match this condition.



Hint: Use the *Where-Used List* function in the *ABAP Dictionary*.

3. Define an internal table (name suggestion: **it_spfli**) based on one of the found global table types.
4. Define a suitable work area for the internal table (name suggestion: **wa_spfli**).

Task 2

Fill the internal table and display contents

Continued on next page

1. Program an array fetch access to all data records in the database table SPFLI:

```
SELECT * FROM spfli INTO TABLE it_spfli.
```

2. Display the buffered data in a list.



Hint: Use the LOOP statement to do this.

Solution 7: Working with Internal Tables

Task 1

Define internal table

1. Create the executable program **ZBC400_##_ITAB_LOOP** without a "TOP include".
 - a) Carry out this step as usual.
2. Buffer the data from the database table SPFLI in an internal table. You should therefore define an internal table with a line type that is compatible with the line structure of SPFLI.

In the *ABAP Dictionary*, search for all (internal) table types that match this condition.



Hint: Use the *Where-Used List* function in the *ABAP Dictionary*.

- a) Carry out this step as usual.
3. Define an internal table (name suggestion: **it_spfli**) based on one of the found global table types.
 - a) See source code excerpt in the model solution.
4. Define a suitable work area for the internal table (name suggestion: **wa_spfli**).
 - a) See source code excerpt in the model solution.

Task 2

Fill the internal table and display contents

1. Program an array fetch access to all data records in the database table SPFLI:

```
SELECT * FROM spfli INTO TABLE it_spfli.
```

- a) See source code excerpt in the model solution.

Continued on next page

2. Display the buffered data in a list.



Hint: Use the LOOP statement to do this.

- a) See source code excerpt in the model solution.

Result

Source code excerpt: SAPBC400TSS_ITAB_LOOP

```
REPORT sapbc400tss_itab_loop.
```

```
DATA: it_spfli TYPE sbc400_t_spfli. DATA: wa_spfli TYPE spfli.
```

```
SELECT * FROM spfli INTO TABLE it_spfli.
```

```
* at least one dataset selected
```

```
IF sy-subrc = 0.
```

```
* move each single dataset from internal table to
```

```
* structure WA_SPFLI in order to write data in list
```

```
LOOP AT it_spfli INTO wa_spfli.
```

```
WRITE: / wa_spfli-carrid,
```

```
      wa_spfli-connid,
```

```
      wa_spfli-cityfrom,
```

```
      wa_spfli-cityto,
```

```
      wa_spfli-deptime,
```

```
      wa_spfli-arrrtime.
```

```
ENDLOOP.
```

```
ENDIF.
```



Lesson Summary

You should now be able to:

- Define internal tables
- Use basic ABAP statements with internal tables
- Analyze internal tables in debugging mode



Unit Summary

You should now be able to:

- Define elementary data objects
- Use basic ABAP statements with elementary data objects
- Execute and analyze programs in debugging mode
- Define structured data objects
- Use basic ABAP statements in structured data objects
- Analyze structured data objects in debugging mode
- Define internal tables
- Use basic ABAP statements with internal tables
- Analyze internal tables in debugging mode

Related Information

... Refer to the online documentation for the relevant ABAP statement.

Unit 5

Data Retrieval

Unit Overview

Refer to the lesson summaries for an overview of this unit.



Unit Objectives

After completing this unit, you will be able to:

- List various ways of finding database tables
- Program read access to specific columns and rows within a particular database table
- List the different kinds of read access possibilities for multiple database tables
- Explain the SAP authorization concept
- Implement authorization checks

Unit Contents

Lesson: Reading Database Tables	134
Exercise 8: Data Retrieval Using a SELECT Loop.....	149
Exercise 9: Data Retrieval and Buffering in an Internal Table	155
Lesson: Authorization Check	161
Exercise 10: Authorization Check	167

Lesson: Reading Database Tables

Lesson Overview

In this lesson, you will learn how to get information about database tables. You will also become familiar with the statements SELECT SINGLE, SELECT ... ENDSELECT, and SELECT ... INTO TABLE. An overview of techniques that allow you to access multiple database tables will also be provided.

This lesson concludes with a note about database table accesses that initiate changes.



Lesson Objectives

After completing this lesson, you will be able to:

- List various ways of finding database tables
- Program read access to specific columns and rows within a particular database table
- List the different kinds of read access possibilities for multiple database tables

Business Example

You need to evaluate data from database tables.

Data Retrieval

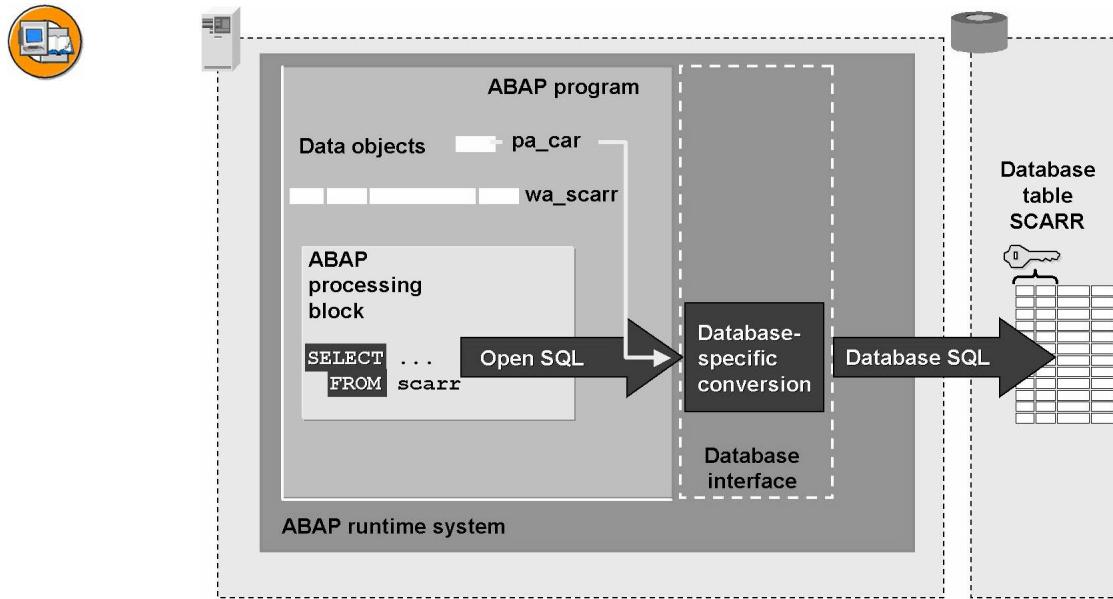


Figure 81: Querying the Database

Open SQL statements are a subset of **Standard SQL** that is fully integrated in the ABAP language.

They allow the ABAP programmer uniform access to data, regardless of the database system installed. Open SQL statements are converted into database-specific SQL statements by the **database interface**.



Searching within a particular application component: *Application hierarchy*

Searching with the help of a program that accesses the database tables searched for:

- Search for the **SELECT** statement in the **ABAP Editor**
- Change to debugging mode during program execution and set breakpoint at the **SELECT** statement
- If the program transmits a screen, display the structure field using **<F1>** and technical information, or navigate directly to the **data element** (double-click), and then call the **where-used list** in the tables

Figure 82: Searching for Database Tables

You have different options for searching for database tables: Using the application hierarchy or the Repository Information System.

If you have the name of a program that accesses the database table you are searching for:

Debugger

Start this program in debugging mode and set a *breakpoint* at the **SELECT** statement.

ABAP Editor

Search for the **SELECT** statement in the source code.

Input field on a screen

If you know of a program that contains a screen with input fields connected to the table you are looking for, you can choose *F1→Technical Info* and double-click the technical name of the screen field to navigate to the *ABAP Dictionary*. This is often a structure field. Double-click the **data element** and then use the *where-used list* function to search for transparent tables with the field type.

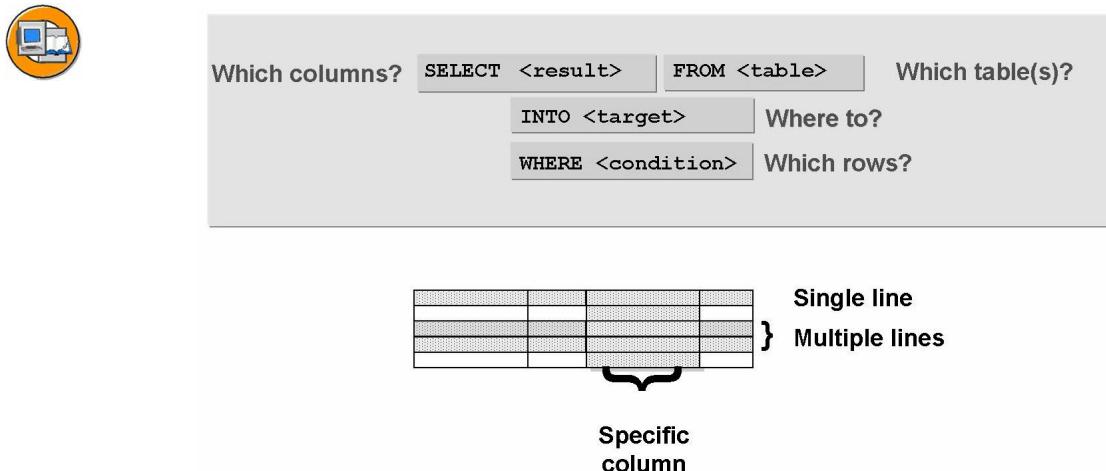


Figure 83: Overview: Database Read Access

You use the Open SQL statement SELECT to program database read access. The SELECT statement contains a series of clauses, each of which has a different task:

- The SELECT clause describes, among other things, whether the result of the selection will comprise several lines or a single data record, and which fields of the table are to be read.
- The FROM clause names the source (database table or view) from which the data is to be selected.
- The INTO clause determines the internal data objects into which the selected data is to be placed.
- The WHERE clause defines the conditions that the selection results must fulfill. It therefore specifies the rows to be selected from the table.
- For information about other clauses, refer to the keyword documentation for the SELECT statement.

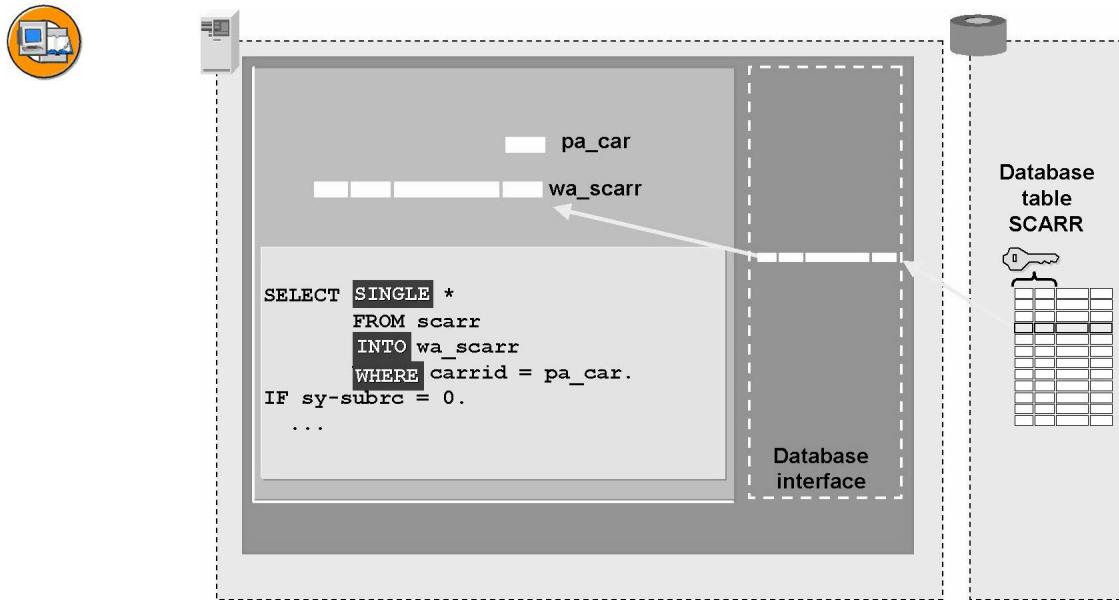


Figure 84: Reading by Single Record Access

The **SELECT SINGLE *** statement allows you to read a **single record** from the database table. To ensure a unique access, all key fields must be filled in the WHERE clause. The * informs the *database interface* that all columns in that row of the database table should be read. If you only wish a specific selection of columns, you can list the required fields instead.

In the INTO clause, enter the destination to where the *database interface* is to copy the data. The target area should have a structure identical to the columns of the database table being read and be left-justified.

If you use the CORRESPONDING FIELDS OF addition to the INTO clause, you can fill the target area component by component. The system only fills those components that have identical names to columns in the database table. If you do not use this addition, the system fills the target area left-justified, irrespective of its structure.

If the system finds a suitable record, the return value SY-SUBRC has the value 0.

The SINGLE addition informs the database that only one record is to be read. The database can then terminate the search as soon as it has found the appropriate record. Therefore, SELECT SINGLE produces better performance for single-record accesses than a SELECT loop if you supply values for all key fields.

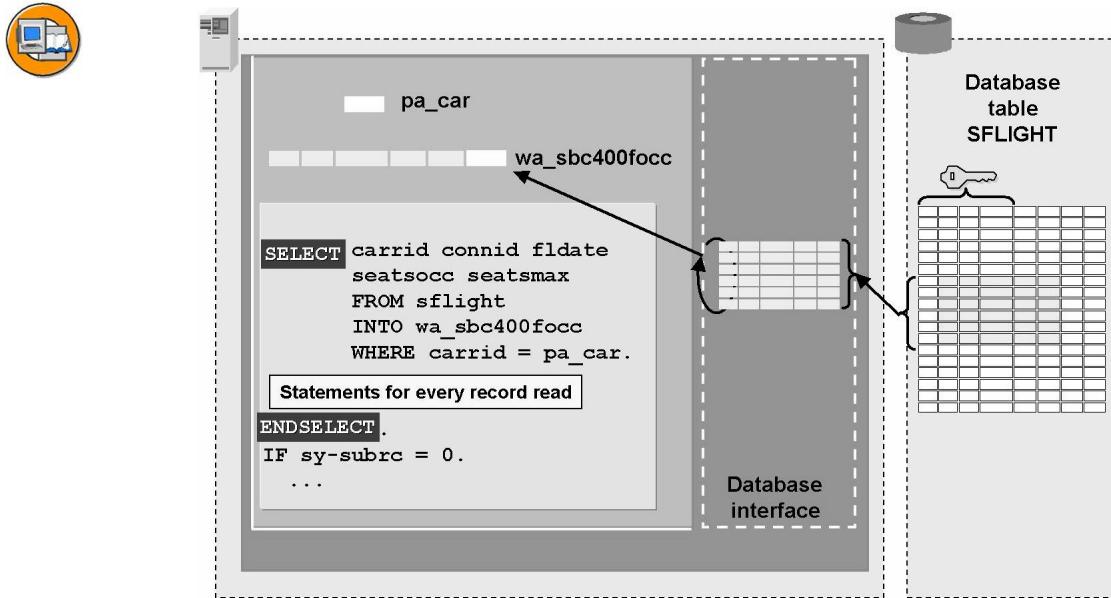


Figure 85: Reading several rows using a loop

If you do not use the SINGLE addition with the SELECT statement, the system reads multiple records from the database. In this example, the field list determines the columns whose data is to be read from the database.

The number of requested **rows** to be read can be restricted using the WHERE clause. In the WHERE clause, you enter **only the field names** of the database table. The name of the database table that is accessed is found in the FROM clause. Multiple logical conditions within the WHERE clause can be logically connected using AND or OR.

The database delivers the data to the *database interface* in packages. The ABAP runtime system copies the data records to the target area row by row using a loop. It also enables sequential processing of all the statements between SELECT and ENDSELECT.

After the ENDSELECT statement, you can check the return code for the SELECT loop. SY-SUBRC has the value 0 if the solution set consists of at least one record.

After the ENDSELECT statement, SY-DBCNT contains the total number of rows read.

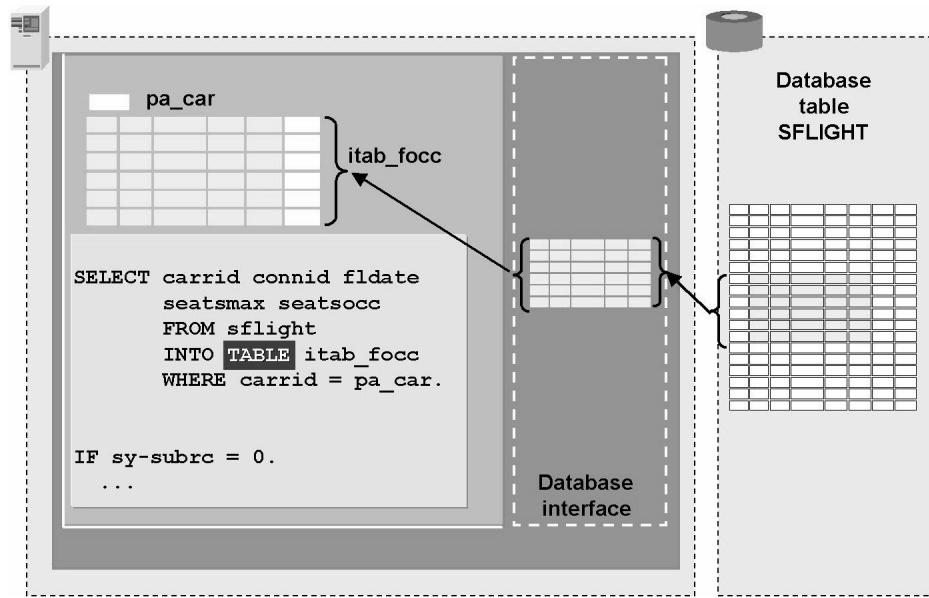


Figure 86: Reading Several Rows Using an Array Fetch

The INTO TABLE addition causes the ABAP runtime system to copy the contents of the *database interface* directly to the specified internal table. This is called an **array fetch**.

Since the array fetch is not executed as a loop, you must not program an ENDSELECT statement. If you want to add rows to the end of an internal table that is already filled, instead of overwriting it, use the APPENDING TABLE addition.

SY-SUBRC has the value 0 if the system read at least one record. Analogous to SELECT-ENDSELECT, SY-DBCNT then contains the total number of rows read.

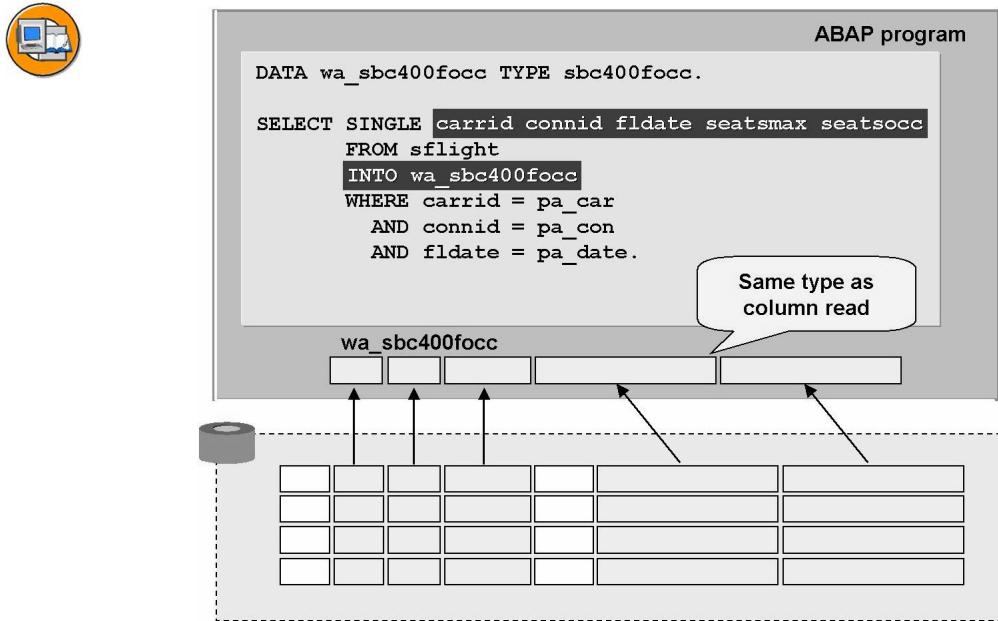


Figure 87: INTO Clause: Target Structure Suitable for Field List

The program must contain a data object with a suitable type for each **column** that is requested from a database table. To make the program easier to maintain, you should use the corresponding global types (*ABAP Dictionary*) to assign types to the data objects. The INTO clause specifies into which data object the data from the database table is placed. There are two possibilities:

“Flat” structure

You define a structure in your program that has the same fields in the same sequence as the field list in the SELECT clause. In this case, you can enter the name of the structure in the INTO clause. The contents are copied left-justified without gaps. The field names within the structure (component names) are **disregarded**.

Single data objects

You can specify a set of data objects in the **INTO** clause.

Example:

```
DATA: gd_carrid TYPE sflight-carrid,
      gd_connid TYPE sflight-connid,
      gd_fldate  TYPE sflight-flddate,
      gd_seatsmax TYPE sflight-seatsmax,
      gd_seatsocc TYPE sflight-seatsocc.

START-OF-SELECTION.

SELECT carrid connid fldate seatsmax seatsocc
FROM sflight
INTO (gd_carrid, gd_connid, gd_fldate,
      gd_seatsmax, gd_seatsocc)
WHERE ... .
```

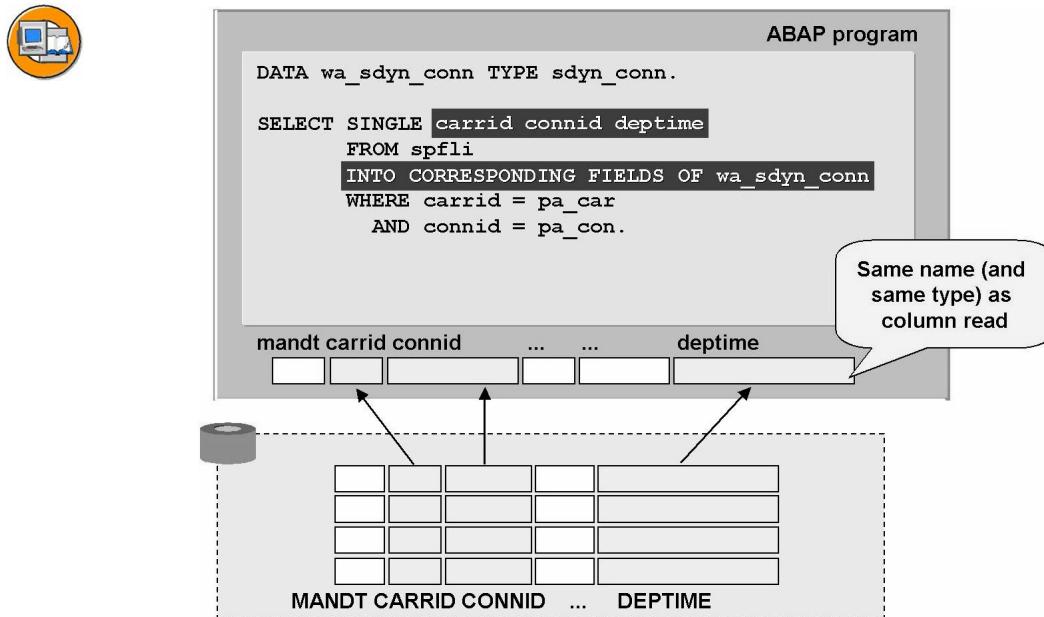


Figure 88: INTO Clause: Same-Name Fields of Field List in the Target Structure

If you use the INTO CORRESPONDING FIELDS OF addition, the data is placed into the identically-named fields in the target structure.

- The target structure does not have to be structured left-justified in the same way as the field list.

- This construction is easy to maintain, since extending the field list does not require other changes to the program, as long as there is a field in the target structure that has the same name and type.

If you want to place data into internal table columns of the same name using an array fetch, use the addition INTO CORRESPONDING FIELDS OF TABLE.

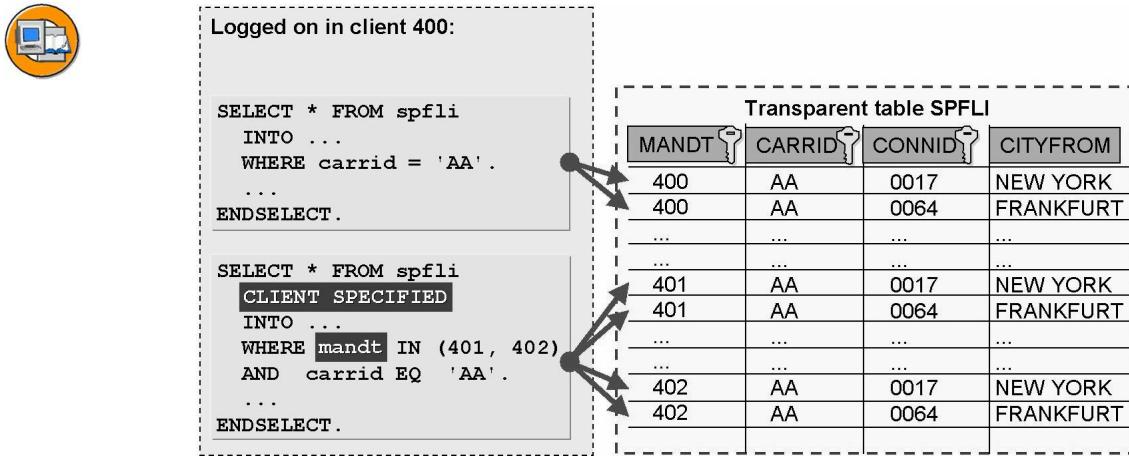


Figure 89: Selecting Data From Client-Specific Tables

With client-specific database tables, the first key field is always MANDT (data type CLNT).

If you select data from client-specific tables, the system **always** reads the data records **of the current client**, unless otherwise specified. (The ABAP runtime system automatically converts the SELECT statement accordingly.)

If you wish to explicitly **include** the client field in the WHERE clause, you must set the CLIENT SPECIFIED addition after the FROM clause.

Since this special case is not relevant for this course, the client field is generally omitted from the graphics.

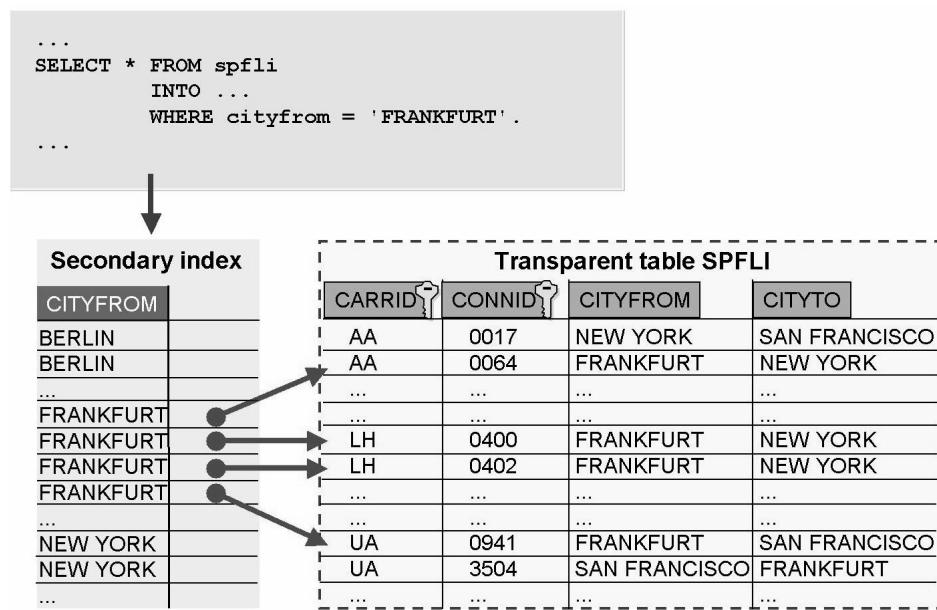


Figure 90: Performance Aspect: Secondary Index

As soon as you navigate to the definition of a database table in the *ABAP Dictionary*, you will see information about all the technical attributes of that database table. The following information is of interest for improving the performance of database accesses:

Key fields

If the rows requested from the database are being retrieved according to key fields, the *Database Optimizer* can perform access using a primary index.

Secondary index

If the rows requested from the database are retrieved according to fields in a secondary index, the *Database Optimizer* can perform access using this secondary index. These are displayed in a dialog box whenever you choose the *Indexes* pushbutton. You can choose an index from the dialog box by simply double-clicking it. The system then displays a screen with additional information about that index. In the case of client-specific tables, it is standard that the client field is included in the index.

Outlook: Reading Multiple Database Tables, Accesses that Initiate Changes

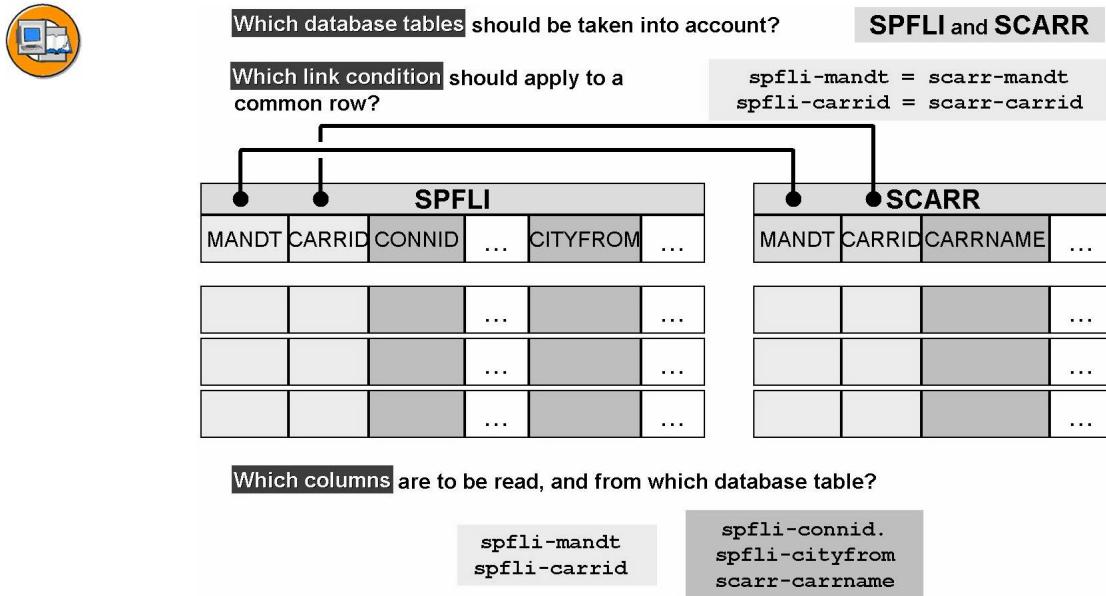


Figure 91: Reading Linkable Database Tables

If you want to read data from two databases that are logically connected, you can access both database tables using **one** database query. To do so, you must enter the following information:

- **Which database tables** are to be accessed?
- What should the **link condition** look like? In this condition, columns from both database tables are linked. A record is placed in a common row in the results table if all the field values of the linked columns match.
- **Which columns** are to be read? If a column appears in both database tables, you must specify the table from which this column is to be read.

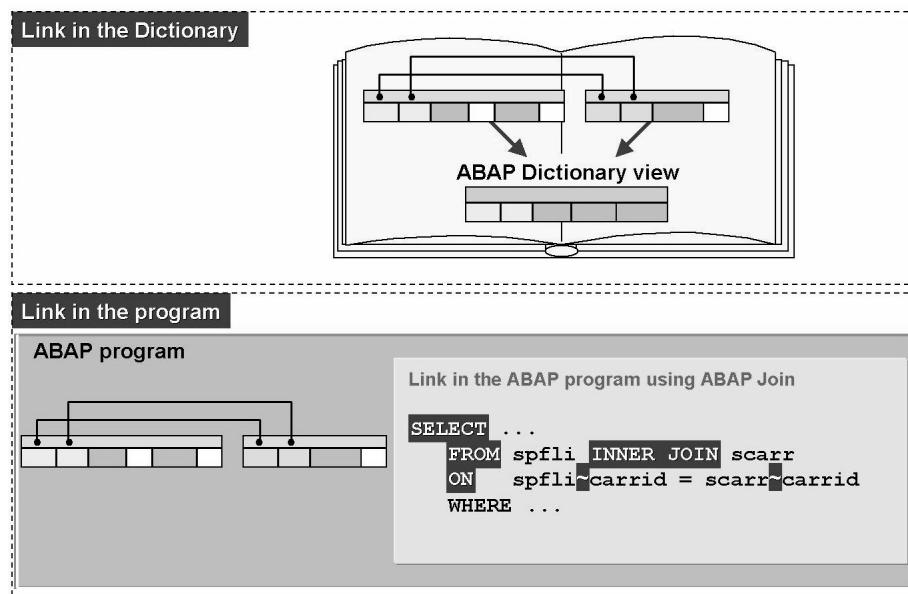


Figure 92: ABAP Dictionary and ABAP Join Views

You can define the link conditions either statically or dynamically:

You can define the **static link** in the *ABAP Dictionary*. It is known as a **view**. There are different kinds of views. You can find additional information in the SAP Library under *Basis → ABAP Workbench → BC ABAP Dictionary → Views*.

You can implement a **dynamic link** using ABAP statements. It is known as an **ABAP join**. At runtime, the system generates an appropriate database query in the database interface. For more information, see the keyword documentation for the FROM clause of the SELECT statement.

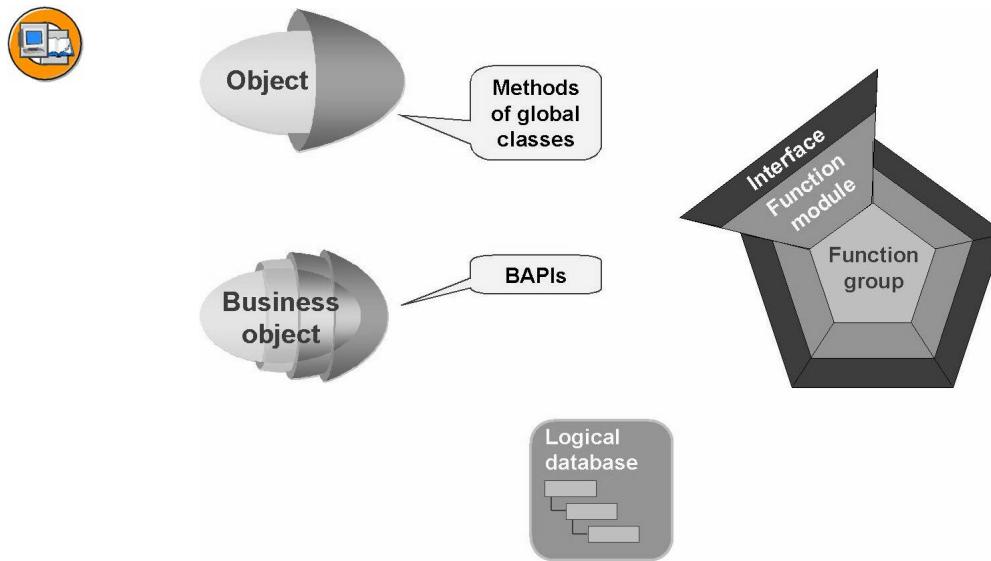


Figure 93: Reuse Components for Data Retrieval

If reusable components that encapsulate complex data retrieval are available, you must use them. There are four possible techniques:

- Calling methods of global classes
- Calling methods of business objects
- Calling function modules
- Including logical databases. Logical databases are data retrieval programs that return logically connected data in a hierarchical sequence.

Information about the first three techniques is available in the relevant units.

For detailed information on how to include logical databases, refer to the SAP library: *Basis → ABAP Programming and Runtime Environment → ABAP Database Accesses → Logical Databases*

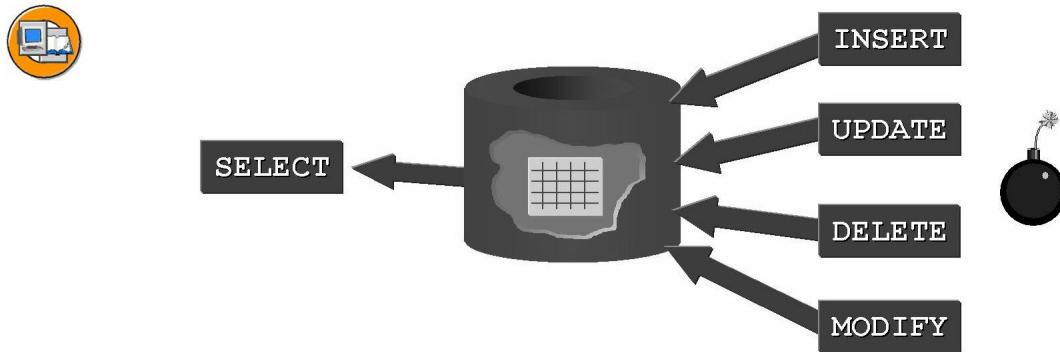


Figure 94: Database Access Initiating Changes

In addition to the SELECT statement, Open SQL also contains the UPDATE, INSERT, DELETE, and MODIFY statements. Possible database accesses that initiate changes in the database should **not** be used unless the user understands the SAP transaction concept.

All Open SQL statements support the return value SY-SUBRC. Therefore, if the access was successful, SY-SUBRC equals 0. If the access was unsuccessful, SY-SUBRC is not 0.

For further information, refer to the relevant keyword documentation.

Exercise 8: Data Retrieval Using a SELECT Loop

Exercise Objectives

After completing this exercise, you will be able to:

- Program loop database accesses
- Limit the quantity of selected data in accordance with an input parameter

Business Example

Enhance your ABAP program to issue flight schedules so that only the schedules of the airlines selected by the user are displayed.

The list should also include the occupancy, in percent, of each flight.

Task 1

Reading several rows using a loop

1. Copy your executable program **ZBC400##GS_LIST** (solution to exercise in lesson “Developing Programs and Organizing Developments”) or the template **SAPBC400WBS_GS_LIST** to the new name **ZBC400##SELECT_SFLIGHT**.
2. Change the work area type (name suggestion: **wa_flight**). Now use the global structure **SBC400FOCC**. Find out information about the subfields of this structure.
3. Define an input parameter for the airline abbreviation.
4. Select the flight departure times from the database table **SFLIGHT** that belong to the selected airline. Select only those fields that have a target field in the work area.
5. What is the first key field in the transparent table **SFLIGHT**? What is the first key field in your WHERE clause? Does the database interface (still) use the primary index?

Task 2

Calculating the percentage

Continued on next page

1. Within the loop statement, calculate the percentage occupancy for each flight. Assign the result to the **PERCENTAGE** field in your work area.
2. Extend your list output to include the percentage occupancy.

Solution 8: Data Retrieval Using a SELECT Loop

Task 1

Reading several rows using a loop

1. Copy your executable program **ZBC400##GS_LIST** (solution to exercise in lesson "Developing Programs and Organizing Developments") or the template **SAPBC400WBS_GS_LIST** to the new name **ZBC400##SELECT_SFLIGHT**.
 - a) Carry out this step as usual.
2. Change the work area type (name suggestion: **wa_flight**). Now use the global structure **SBC400FOCC**. Find out information about the subfields of this structure.
 - a) See source code excerpt in the model solution.

The global structure **SBC400FOCC** has the following components: **CARRID** with the type **S_CARR_ID**, **CONNID** with the type **S_CONN_ID**, **FLDATE** with the type **S_DATE**, **SEATSMAX** with the type **S_SEATSMAX**, **SEATSOCC** with the type **S_SEATSOCC**, and **PERCENTAGE** with the type **S_FLGHTOCC**. With the exception of the last component, all others have the same type as identically-named fields in the transparent table **SFLIGHT**. However, the database table **SFLIGHT** does not contain a percentage occupancy field.

3. Define an input parameter for the airline abbreviation.
 - a) See source code excerpt in the model solution.
4. Select the flight departure times from the database table **SFLIGHT** that belong to the selected airline. Select only those fields that have a target field in the work area.
 - a) See source code excerpt in the model solution.

Continued on next page

5. What is the first key field in the transparent table **SFLIGHT**? What is the first key field in your WHERE clause? Does the database interface (still) use the primary index?
 - a) The **MANDT** field in the transparent table **SFLIGHT** is marked as a key field. However, this field need not be included in the WHERE clause. Despite this fact, the *database interface* can use the primary index in this case for selecting the data records.

Reason: When the ABAP runtime system accesses client-specific tables, it assumes that data should only be taken into account if it belongs to the client under which the user is logged on. Thus, by default, the database interface adds a condition `WHERE mandt = sy-mandt` to the WHERE clause.

Task 2

Calculating the percentage

1. Within the loop statement, calculate the percentage occupancy for each flight. Assign the result to the **PERCENTAGE** field in your work area.
 - a) See source code excerpt in the model solution.

Continued on next page

2. Extend your list output to include the percentage occupancy.
 - a) See source code excerpt in the model solution.

Result

Source code excerpt: **SAPBC400DDS_SELECT_SFLIGHT**

```
REPORT sapbc400dds_select_sflight.

DATA: wa_flight TYPE sbc400focc.
PARAMETERS: pa_car TYPE s_carr_id.

* Select all datasets from database table SFLIGHT
* corresponding to carrier PA_CAR:
SELECT carrid connid fldate seatsmax seatsocc
FROM sflight
INTO CORRESPONDING FIELDS OF wa_flight
WHERE carrid = pa_car.

* Calculate occupation of each flight:
wa_flight-percentage =
100 * wa_flight-seatsocc / wa_flight-seatsmax.

* Create List:
WRITE: / wa_flight-carrid,
       wa_flight-connid,
       wa_flight-fldate,
       wa_flight-seatsmax,
       wa_flight-seatsocc,
       wa_flight-percentage,
       '%'.
ENDSELECT.
```


Exercise 9: Data Retrieval and Buffering in an Internal Table

Exercise Objectives

After completing this exercise, you will be able to:

- Fill an internal table with data by appending data records
- Sort the content of an internal table

Business Example

Extend your ABAP program for the output of flight departure times so that the flights appear in the list sorted by percentage occupancy.

For this purpose, append the data records for the selected airline to an internal table.

Task 1

Data retrieval and filling an internal table

1. Copy your executable program **ZBC400##_SELECT_SFLIGHT** (solution to first exercise in this lesson) or the template **SAPBC400DDS_SELECT_SFLIGHT** to the new name **ZBC400##_SELECT_SFLIGHT_ITAB**.
2. Define an internal table (name proposal: **it_flight**) based on a global table type that has the global structure **SBC400FOCC** as the row type.



Hint: Use the *Where-Used List* in the *ABAP Dictionary* to find a suitable table type.

3. Fill this internal table row by row using an **INSERT** statement within the **SELECT** loop.
4. Remove the statements for list output.



Caution: You can still use the source code later.

Task 2

Sorting and outputting the internal table

Continued on next page

1. Sort the internal table according to occupancy.
2. Use a LOOP to display the contents of the internal table in a list.

Solution 9: Data Retrieval and Buffering in an Internal Table

Task 1

Data retrieval and filling an internal table

1. Copy your executable program **ZBC400_##_SELECT_SFLIGHT** (solution to first exercise in this lesson) or the template **SAPBC400DDS_SELECT_SFLIGHT** to the new name **ZBC400_##_SELECT_SFLIGHT_ITAB**.
 - a) Carry out this step as usual.
2. Define an internal table (name proposal: **it_flight**) based on a global table type that has the global structure **SBC400FOCC** as the row type.



Hint: Use the *Where-Used List* in the *ABAP Dictionary* to find a suitable table type.

- a) Carry out this step as usual.
3. Fill this internal table row by row using an **INSERT** statement within the **SELECT** loop.
 - a) See source code excerpt in the model solution.
4. Remove the statements for list output.



Caution: You can still use the source code later.

- a) See source code excerpt in the model solution.

Task 2

Sorting and outputting the internal table

1. Sort the internal table according to occupancy.
 - a) See source code excerpt in the model solution.

Continued on next page

2. Use a LOOP to display the contents of the internal table in a list.
 - a) See source code excerpt in the model solution.

Result

Source code excerpt: **SAPBC400DDS_SELECT_SFLIGHT_TAB**

```
REPORT  sapbc400dds_select_sflight_tab

DATA: wa_flight TYPE sbc400focc,
      it_flight TYPE sbc400_t_sbc400focc.
PARAMETERS: pa_car TYPE s_carr_id.

* select all datasets from database table SFLIGHT
* corresponding to carrier
SELECT carrid connid fldate seatsmax seatsocc
      FROM sflight
      INTO CORRESPONDING FIELDS OF wa_flight
      WHERE carrid = pa_car.

* calculate occupation of each flight
wa_flight-percentage =
  100 * wa_flight-seatsocc / wa_flight-seatsmax.

* append single line to internal table
INSERT wa_flight INTO TABLE it_flight.

* alternatively, you could use the APPEND statement,
* if you are using standard tables:
*   APPEND wa_flight TO it_flight.
ENDSELECT.

* sort internal table
SORT it_flight BY percentage.

* create list from sorted internal table
LOOP AT it_flight INTO wa_flight.
  WRITE: / wa_flight-carrid,
          wa_flight-connid,
          wa_flight-fldate,
          wa_flight-seatsocc,
          wa_flight-seatsmax,
          wa_flight-percentage, '%'.

```

Continued on next page

ENDLOOP.



Lesson Summary

You should now be able to:

- List various ways of finding database tables
- Program read access to specific columns and rows within a particular database table
- List the different kinds of read access possibilities for multiple database tables

Lesson: Authorization Check

Lesson Overview

In this lesson, you will learn why an authorization check is useful and how to include it in your programs.



Lesson Objectives

After completing this lesson, you will be able to:

- Explain the SAP authorization concept
- Implement authorization checks

Business Example

Authorization checks are necessary in your programs to protect the data from unauthorized access.

Authorization Checks

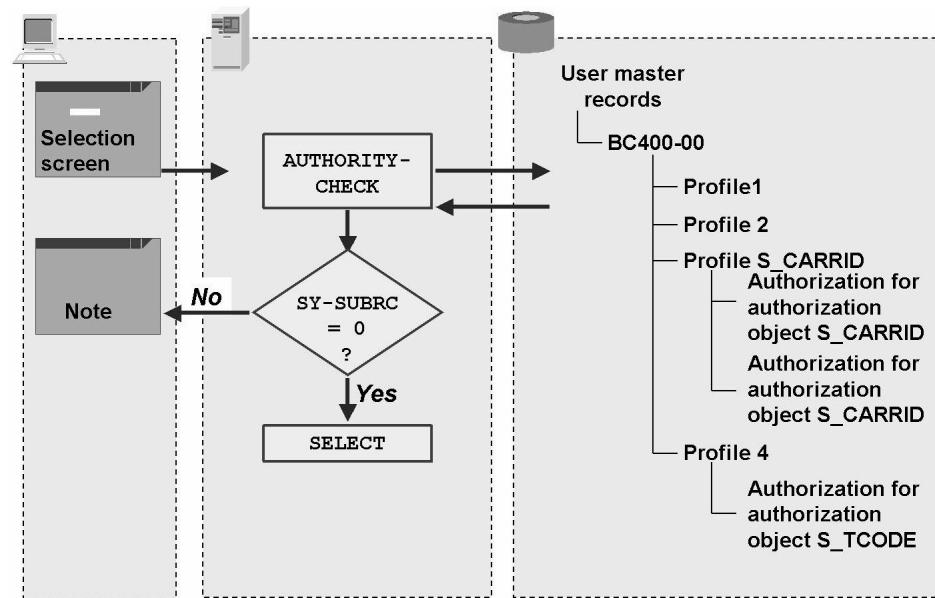


Figure 95: Authorization Checks in ABAP Programs

Critical data and parts of the functional scope of the SAP system must be protected from unauthorized access so that the user can only access those areas for which he has explicit authorization.

You should therefore carry out an authorization check before accessing the database. The AUTHORITY-CHECK statement checks whether the user has the authorization containing all the required values. The system administrator assigns user authorizations when maintaining user master data.

After the AUTHORITY-CHECK, **you must check the return value SY-SUBRC** to be able evaluate the information: If it is 0, this means that the user has the required authorization and the program can be continued. However, if the value is not 0, the user does not possess the required authorization. In this case, you could display a message, for example.

Later in this course, you will learn how to make fields on the selection screen ready for input again if you perform the authorization check right after the selection screen and output a message if the user does not have the required authorization.



Caution: Apart from this type of authorization check **within your program**, there is generally **no other** security check. There are other concepts, such as applying an authorization check to a transaction code, but these are merely extras and not replacements.

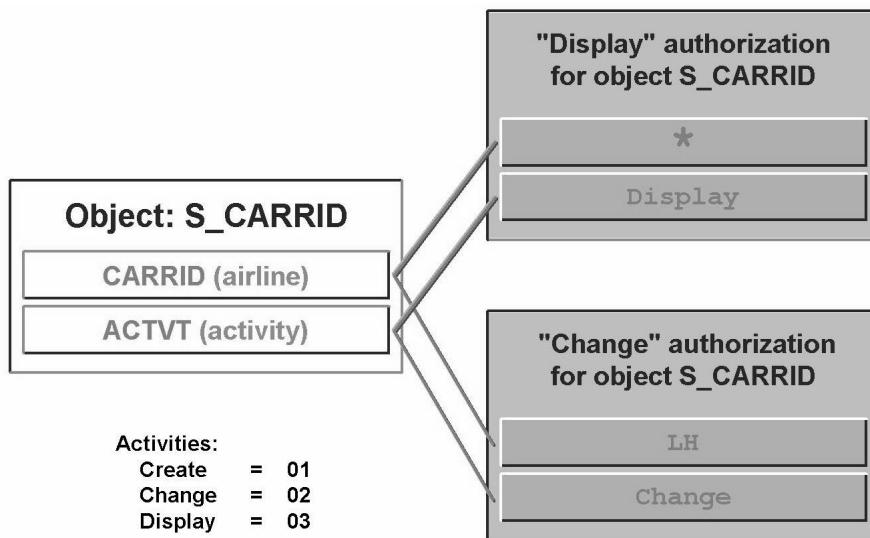


Figure 96: Authorization Objects and Authorizations

When assigning authorizations, you should specify exactly **which data** users are allowed to access and what **kind of access** should be allowed. For example, you might want to allow users to display data for all airlines, but only allow them to change data for certain selected ones. Consequently, the system must look for a **combination** of the fields "Activity" and "Airline" each time it performs an authorization check. Both fields must be filled with values during authorization creation as well (for example, activity "Change" and airline carrier "LH" or activity "Display", Airline carrier "All"). This is ensured by creating an **authorization object** that consists of the fields "Activity" and "Airline". This object must then be addressed both for the authorization assignment within user maintenance and for the authorization check in the program.

Therefore, authorization objects simply define the combination of fields that need to be addressed simultaneously and serve as templates for the authorizations and the authorization checks. The authorization objects are organized into object classes in order to make it easier to find and manage them; one or more classes may be assigned to each application.

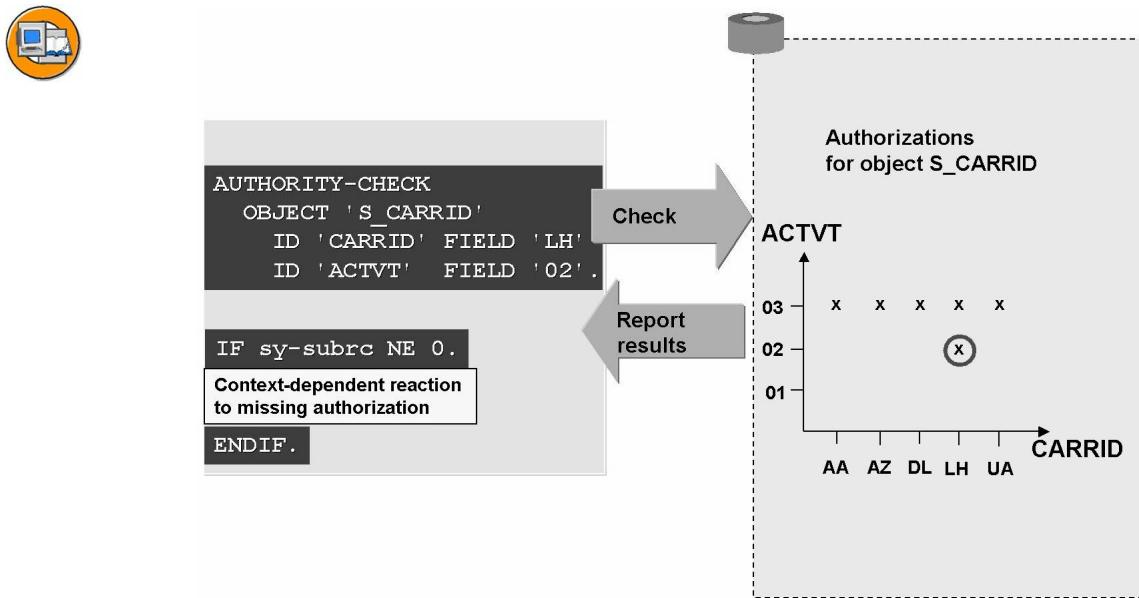


Figure 97: Syntax: Authorization Check

For authorization checks in programs, you specify the object and values the user needs in an authorization to be able to access the object in question.

The above example checks whether the user is authorized for the object **S_CARRID** that has the value **LH** in the field **CARRID** (airline) and the value **02** (change) in the field **ACTVT** (activity). The abbreviations for the activities are documented in the tables **TACT** and **TACTZ** and also in the appropriate objects.



Caution: The AUTHORITY CHECK statement executes the authorization check for the user master record and also issues the return code. This return code enables you to specify the consequences of a missing authorization yourself (for example, program terminates or merely skips some output lines).

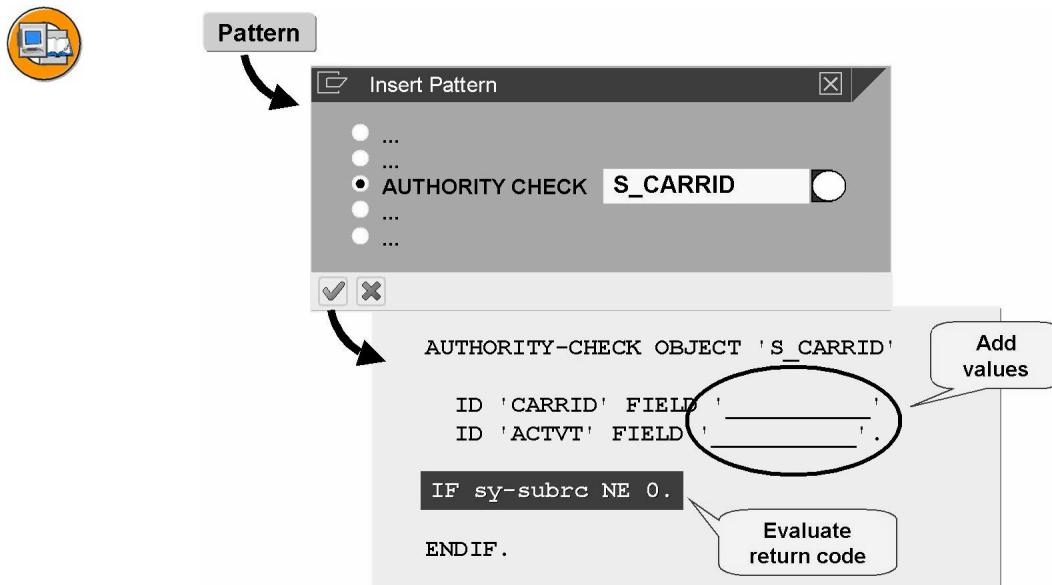


Figure 98: Implementing Authorization Checks

For syntax reasons, you must specify all fields of the object in an AUTHORITY-CHECK statement, otherwise you receive a return code **not equal** to 0. If you do not want to carry out a check for a particular field, enter DUMMY after the authorization field.

Example: When calling a transaction to change flight data, you want to check whether the user is authorized to change the entries for at least one airline carrier:

```

AUTHORITY-CHECK OBJECT 'S_CARRID'
  ID 'CARRID' DUMMY

```

ID 'ACTVT' FIELD '02'.

The most important return codes for the AUTHORITY-CHECK statement are:

0

The user has an authorization containing the required values.

4

The user has **no** authorization containing the required values.

8

Some of the fields of the authorization object were not listed. Therefore, the check could not be executed successfully.

The keyword documentation for AUTHORITY-CHECK contains a complete list of all the return codes.

Exercise 10: Authorization Check

Exercise Objectives

After completing this exercise, you will be able to:

- Implement authorization checks
- Set up the program flow in a variable manner depending on the outcome of the authorization checks

Business Example

Enhance your ABAP program to output the flight departure times such that the flights can only be read and displayed if the user has display authorization for the selected airline.

Task 1

Check for display authorization for airlines

1. Copy your executable program **ZBC400_##_SELECT_SFLIGHT** (solution of first exercise in lesson “Reading Database Tables”) or the template **SAPBC400DDS_SELECT_SFLIGHT** to the new name **ZBC400_##_AUTHORITY_CHECK**.
2. Implement an authorization check for the authorization object **S_CARRID**. Check the combination of selected airline and activity “Display”.



Hint: Use the relevant statement pattern in the *ABAP Editor*. Search for the correct abbreviation of the activity to be checked.

3. Define a constant for the abbreviation of the activity “Display” (name proposal: **actvt_display**). Use the data element **ACTIV_AUTH** to assign the type. Use this constant in the authorization check.
4. Only execute the database access if the user has display authorization for the selected airline. If this is not the case, display an appropriate error message on the list.
5. Execute your programs; select the airline abbreviations **AA** and **UA**.

Continued on next page

Task 2

Check for display authorization for airlines II

1. Copy your executable program **ZBC400_##_SELECT_SFLIGHT_ITAB** (solution of second exercise in lesson “Reading Database Tables”) or the template **SAPBC400DDS_SELECT_SFLIGHT_ITAB** to the new name **ZBC400_##_AUTHORITY_CHECK_2**.
2. Proceed as you did in the previous exercise.

Solution 10: Authorization Check

Task 1

Check for display authorization for airlines

1. Copy your executable program **ZBC400_##_SELECT_SFLIGHT** (solution of first exercise in lesson “Reading Database Tables”) or the template **SAPBC400DDS_SELECT_SFLIGHT** to the new name **ZBC400_##_AUTHORITY_CHECK**.
 - a) Carry out this step as usual.
2. Implement an authorization check for the authorization object **S_CARRID**. Check the combination of selected airline and activity “Display”.



Hint: Use the relevant statement pattern in the *ABAP Editor*. Search for the correct abbreviation of the activity to be checked.

- a) See source code excerpt in the model solution.
3. Define a constant for the abbreviation of the activity “Display” (name proposal: `actvt_display`). Use the data element **ACTIV_AUTH** to assign the type. Use this constant in the authorization check.
 - a) See source code excerpt in the model solution.
4. Only execute the database access if the user has display authorization for the selected airline. If this is not the case, display an appropriate error message on the list.
 - a) See source code excerpt in the model solution.
5. Execute your programs; select the airline abbreviations **AA** and **UA**.
 - a) Carry out this step as usual.

Task 2

Check for display authorization for airlines II

Continued on next page

1. Copy your executable program **ZBC400_##_SELECT_SFLIGHT_ITAB** (solution of second exercise in lesson “Reading Database Tables”) or the template **SAPBC400DDS_SELECT_SFLIGHT_ITAB** to the new name **ZBC400_##_AUTHORITY_CHECK_2**.
 - a) Carry out this step as usual.
2. Proceed as you did in the previous exercise.
 - a) See source code excerpt in the model solution.

Result

Source code excerpt: **SAPBC400DDS_AUTHORITY_CHECK**

```
REPORT sapbc400dds_authority_check.
```

```
CONSTANTS actvt_display TYPE activ_auth VALUE '03'.

DATA wa_flight TYPE sbc400focc.
PARAMETERS pa_car TYPE s_carr_id.

* authority-check:
* Is user authorized to read data for carrier PA_CAR?
AUTHORITY-CHECK OBJECT 'S_CARRID'
  ID 'CARRID' FIELD pa_car
  ID 'ACTVT'  FIELD actvt_display.
CASE sy-subrc.

* user is authorized
WHEN 0.
  SELECT carrid connid flddate seatsmax seatsocc
    FROM sflight
    INTO CORRESPONDING FIELDS OF wa_flight
    WHERE carrid = pa_car.

  wa_flight-percentage =
    100 * wa_flight-seatsocc / wa_flight-seatsmax.

  WRITE: / wa_flight-carrid COLOR COL_KEY,
         wa_flight-connid COLOR COL_KEY,
         wa_flight-flddate COLOR COL_KEY,
         wa_flight-seatsocc,
         wa_flight-seatsmax,
         wa_flight-percentage, '%'.

```

Continued on next page

```
ENDSELECT.  
  
* user is not authorized or  
* other error of authority-check  
WHEN OTHERS.  
    WRITE: / 'Authority-Check Error' (001).  
ENDCASE.
```

Source code excerpt: **SAPBC400DDS_AUTHORITY_CHECK_2**

See source code excerpt: **SAPBC400DDS_AUTHORITY_CHECK**



Lesson Summary

You should now be able to:

- Explain the SAP authorization concept
- Implement authorization checks



Unit Summary

You should now be able to:

- List various ways of finding database tables
- Program read access to specific columns and rows within a particular database table
- List the different kinds of read access possibilities for multiple database tables
- Explain the SAP authorization concept
- Implement authorization checks

Related Information

... Refer to the online documentation for the relevant ABAP statement.

Unit 6

Internal Program Modularization

Unit Overview

Refer to the lesson summaries for an overview of this unit.



Unit Objectives

After completing this unit, you will be able to:

- Define subroutines
- Call subroutines
- Analyze the execution of subroutines in debugging mode

Unit Contents

Lesson: Subroutines.....	176
Exercise 11: Subroutines.....	185

Lesson: Subroutines

Lesson Overview

In this lesson, you will learn how to use subroutines and why there is an interface for subroutines. You will also learn the different transfer types of a main program to the interface of a subroutine: Call By Value, Call By Value and Result, Call By Reference.



Lesson Objectives

After completing this lesson, you will be able to:

- Define subroutines
- Call subroutines
- Analyze the execution of subroutines in debugging mode

Business Example

You need to structure a comprehensive program and encapsulate source code that is to be executed several times in a subroutine.

Internal Program Modularization with Subroutines

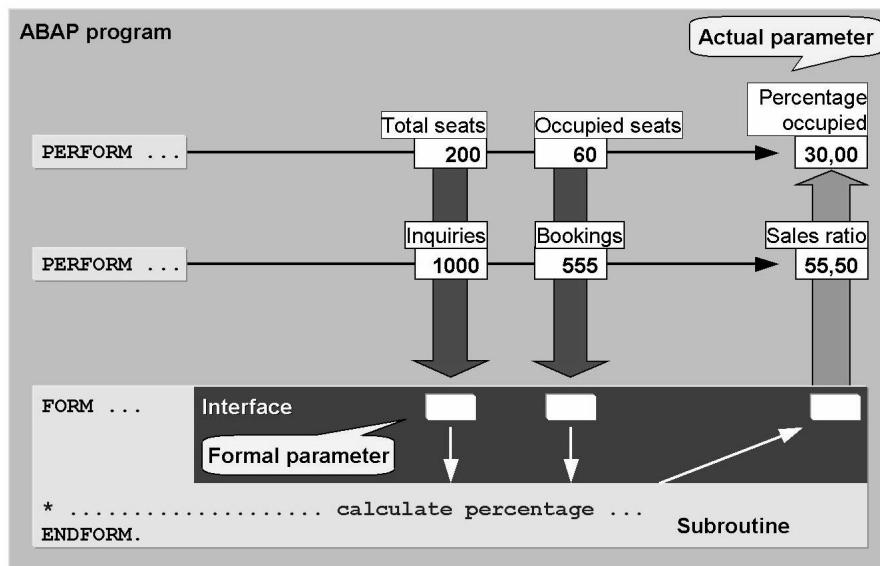


Figure 99: Calls and Interfaces

A subroutine is a **program-internal** modularization unit. You move parts of a program to a subroutine to make the program easier to read and allow you to use these code segments more than once.

You can pass data to the subroutine and back using its interface. This allows you to call the same functions for different data objects. The above example shows a subroutine that calculates a percentage. This subroutine is called several times, even though different data objects are passed to the interface in each case.

Using subroutines makes your program **more function-oriented**: It splits the program's task into subfunctions, so that each subroutine is responsible for one subfunction.

This generally makes programs easier to maintain. In the Debugger, you can run these programs "in the background", so that you only see the result. This usually makes it easier to find the source of the error.

- Each subroutine starts with FORM and ends with ENDFORM.
- The name of the program is followed by the interface definition.
- This is followed in turn by the statements that the subroutine executes.

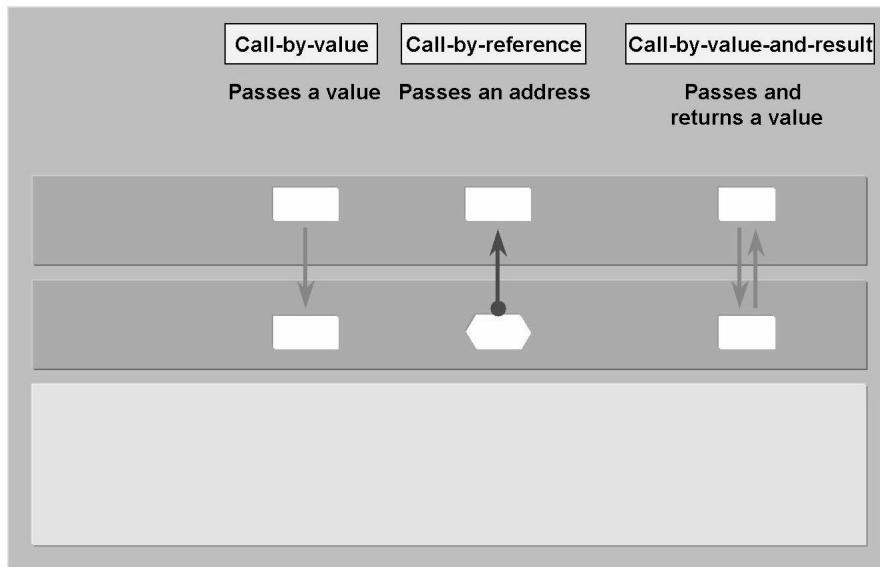


Figure 100: Ways of Passing Interface Parameters

You specify **how** the data is passed from the main program (**actual parameters**) to the subroutine (**formal parameters**) **in the interface**. There are three ways of doing this:

Call-by-value

A **local copy** of the actual parameter is passed to the subroutine. This means that value assignments to the formal parameter have **absolutely no effect** on the actual parameter.

Call-by-reference

The **dereferenced address** of the actual parameter is passed to the subroutine. This means that value assignments to the formal parameter **affect the actual parameter directly**. The value of the actual parameter is physically changed by the subroutine through the address.

Call-by-value-and-result

A **local copy** of the actual parameter is passed to the subroutine and **a value is passed back** to the main program. This means that value assignments to the formal parameter only affect the actual parameter **after leaving the subroutine**. You should choose this option if you want to be sure that the actual parameter will not be changed, should the subroutine be terminated prematurely.

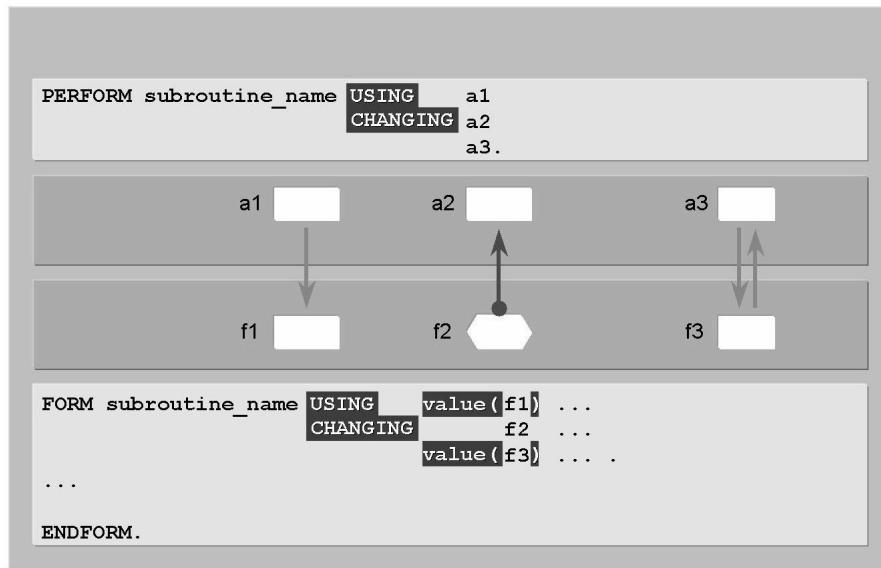


Figure 101: Passing Interface Parameters: Syntax

In the interface definition, you list the **formal parameters** of the different kinds and assign a type to each one. The **sequence** is syntactically fixed: First you name all the **USING** parameters, followed by all the **CHANGING** parameters. Thus within the subroutine, you address the data that has been passed using the formal parameters.

The actual parameters are also grouped by USING and CHANGING when the subroutine is called.

You specify the way in which the parameters are to be passed in the interface definition:

Call-by-value (here F1)

You list these parameters after the USING keyword and complete them with the **value (f1)** addition.

Call-by-reference (here F2)

You list these parameters after the CHANGING keyword.

 **Note:** For documentation purposes, you can also list a parameter for this transfer type after USING. Technically, USING and CHANGING have exactly the same effect **here**. (However, if you change a parameter listed after USING in the subroutine, you receive a warning in the *extended program check*.)

Call-by-value-and-result (here F3)

You list these parameters after the CHANGING keyword and complete them with the **value (f3)** addition.



<pre> TYPES t_perc TYPE p DECIMALS 2. DATA: a1 TYPE ..., a2 TYPE ..., a3 TYPE PERFORM calc_perc USING a1 a2 CHANGING a3. FORM calc_perc USING value(f_part) TYPE i value(f_all) TYPE i CHANGING value(f_pc) TYPE t_perc. CHECK f_all <> 0. f_pc = f_part * 100 / f_all. ENDFORM. </pre>	<pre> DATA: a1 TYPE ..., a2 TYPE ..., a3 TYPE PERFORM calc_perc USING a1 a2 CHANGING a3. FORM calc_perc USING value(f_part) TYPE ANY value(f_all) TYPE ANY CHANGING value(f_pc) TYPE ANY. CHECK f_all <> 0. f_pc = f_part * 100 / f_all. ENDFORM. </pre>
Specified type	Inherited type
	Risk of type conflicts

Figure 102: Interface Parameters: Specified and Inherited Types

The data objects passed to a subroutine (that is, the actual parameters) can be of **any type**. If you use **elementary types**, you can decide whether or not you want to specify the type of the formal parameters.

By specifying the type of the formal parameters, you ensure that only actual parameters of that type can be passed to the subroutine. This makes your program more stable, since the syntax check will find any type conflicts.

If you use the **TYPE ANY** addition, you leave the type unspecified. (For compatibility reasons, you can also omit this addition.) In this case, the formal parameter "inherits" the type from the actual parameter at runtime. Then, if the statements in the subroutine are not suited to the inherited type, a **runtime error** may occur.

If you assign the types P, N, C, or X, the **missing** type attributes are similarly "inherited" at runtime. If you want to specify a type completely, you must use a user-defined type. If you use **STRING** or **XSTRING**, the type is not fully specified until runtime.

Conversely, the data types I, F, D, and T are fully typed.

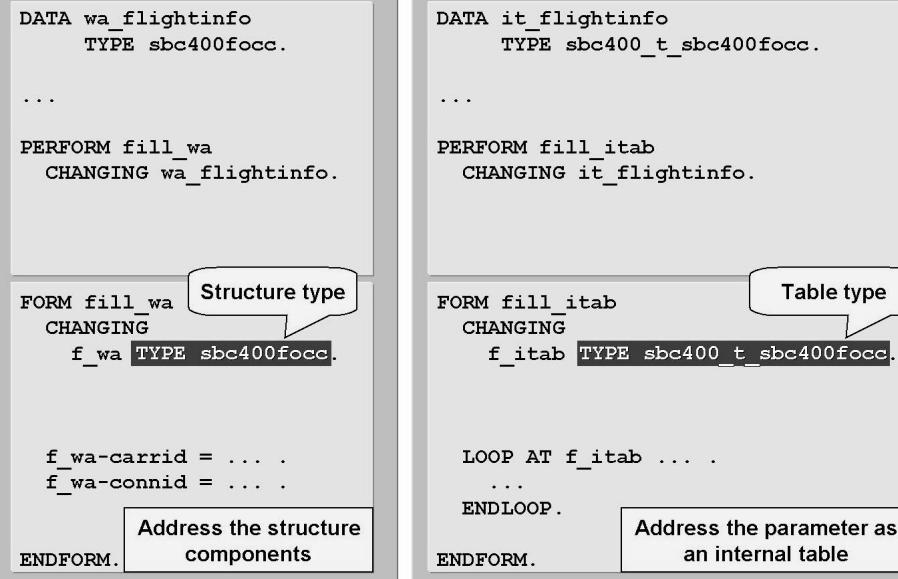


Figure 103: Interface Parameters: Specifying Types for Structures and Internal Tables

If you use structures or internal tables as formal parameters, you **must** type them fully. This allows you to access these formal parameters as normal within the subroutine.

The components of structures are known in the subroutine, as a result of the assigned type, so that you can address these components with the usual syntax. If you use internal tables, the assigned type allows you to address the formal parameter **as an internal table** with the usual syntax.

Note on runtime requirements:

If you use **internal tables** as parameters, you should generally pass them using **call-by-reference**. Otherwise, the system can use considerable resources simply copying them to the subroutine.

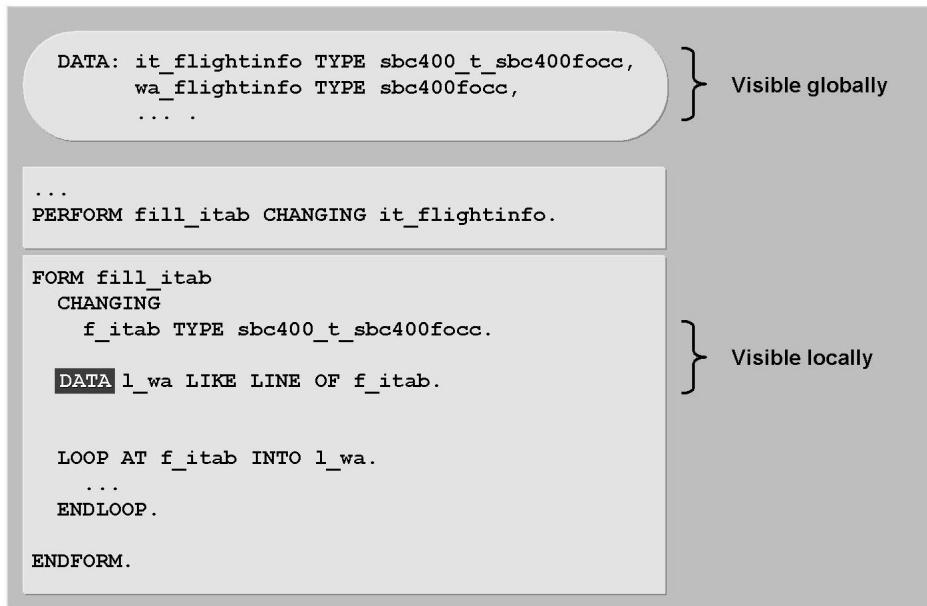


Figure 104: Visibility of Global and Local Data Objects

You can define **local** data within a subroutine. Both the formal parameters and the local data objects are only active at the runtime of the subroutine. This means that memory is only allocated when the subroutine is called and is released as soon as the subroutine has been executed. Thus these parameters and data objects can only be addressed from within the subroutine.

The global data objects from the main program can also be addressed from the subroutine. However, you should avoid doing this wherever possible. Otherwise, you are bypassing the interface, which makes the program more error-prone.

If a local data object or formal parameter has the same name as a global data object, the **ABAP runtime system** addresses the local data object in the subroutine and the global one outside it. They are then known as **locally obscured objects**.

- Address the global data objects in the main program and pass them to the subroutine using the interface.
- Address only the formal parameters and local data objects in the subroutine.
- For clarity, avoid using identically-named variables. Instead, use a simple prefix, such as **f_...** for “formal parameters” and **l_...** for “local data objects” or similar.



```

DATA: it_flightinfo TYPE sbc400_t_sbc400focc,
      wa_flightinfo TYPE sbc400focc,
      lines          TYPE i.
      ...

PERFORM count_lines USING      it_flightinfo
                           'LH'
                           CHANGING lines.

FORM count_lines USING          f_itab    TYPE sbc400_t_sbc400focc
                           f_carr    TYPE sbc400focc-carrid
                           CHANGING value(f_lines) TYPE i.

      DATA l_wa LIKE LINE OF f_itab.

      CLEAR f_lines.
      LOOP AT f_itab INTO l_wa
                     WHERE carrid = f_carr.
      ADD 1 TO f_lines.
      ENDLOOP.
ENDFORM.

```

Figure 105: Syntax Example: Passing an Internal Table

The above example shows a main program calling a subroutine, which specifies the number of lines of an internal table needed to fulfill a specific criterion. The internal table, F_ITAB, and the comparison value, F_CARR, are passed by reference to reduce runtime requirements. The number of lines, F_LINES, is passed by value. To loop through the internal table, you need a work area with a compatible line type, which you define **locally** in the subroutine, by making a reference to the formal parameter.

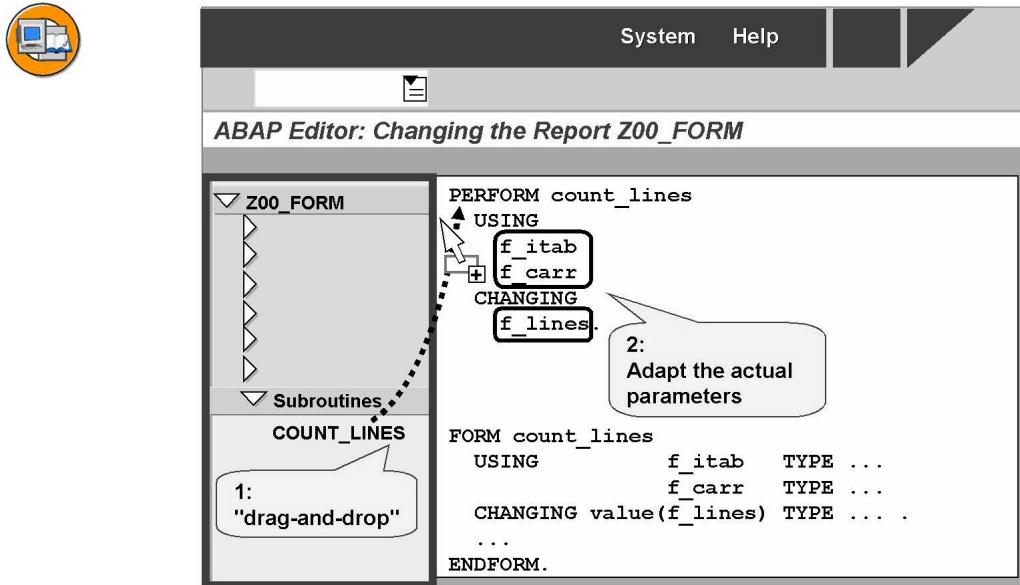


Figure 106: Calling Subroutines

A subroutine is called using the PERFORM statement. When the subroutine is called, parameters are passed to it **in strict sequence**.

For this reason, you should define the subroutine first and then call it. The *Object Navigator* supports you in this respect: You can generate the PERFORM statement by dragging the subroutine from the navigation area to the editor area. Alternatively, you can generate the call using the statement patterns in the *ABAP Editor*.

This prevents you from mixing up or forgetting any parameters. However, you must remember to replace the formal parameters with actual parameters.

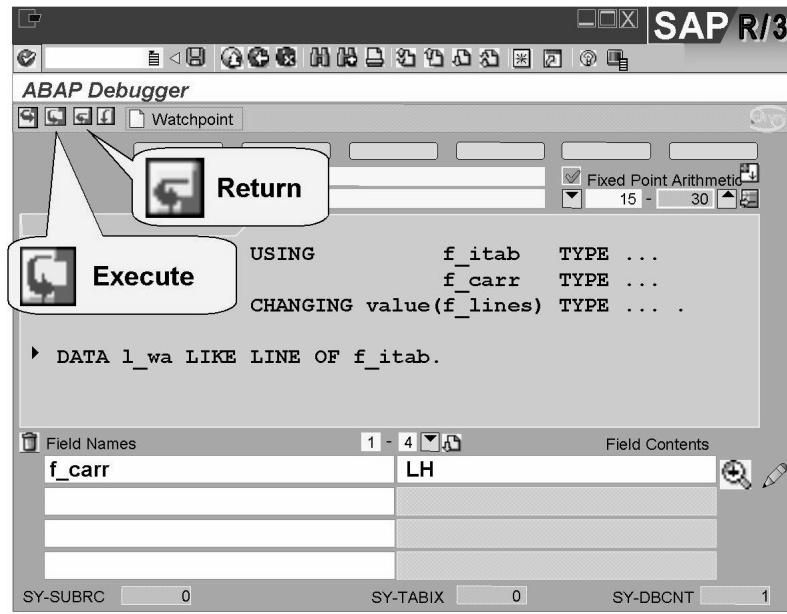


Figure 107: Subroutines in Debugging Mode

In debugging mode, you can make a subroutine run without stopping. The execution of the main program does not stop until after the subroutine has been executed.

Alternatively, you can execute the subroutine statement-by-statement using *Single Step*.

If the current statement is located in a subroutine, you can execute the rest of the subroutine without it stopping by choosing *Return*. The execution of the main program does not stop until after the subroutine has been executed.

Exercise 11: Subroutines

Exercise Objectives

After completing this exercise, you will be able to:

- Create subroutines
- Use the subroutine interface to pass data

Business Example

Change your program so that the display of the data in a list is encapsulated in a subroutine.

Task 1

Create a subroutine with internal table transfer

1. Copy your program **ZBC400_##_SELECT_SFLIGHT_ITAB** or the corresponding model solution **SAPBC400DDS_AUTHORITY_CHECK_2** to the new program **ZBC400_##_SUBROUTINE**.
2. Encapsulate the display of data in a list in a subroutine. To do so, define the subroutine and specify types for the interface parameter (name proposal: **WRITE_LIST**).
3. Display the data from the subroutine using a LOOP. To do this, create the required table work area as a local data object in the subroutine (name proposal: **wa**).



Hint: To specify the type of the local structure, use the ABAP statement **DATA <wa> LIKE LINE OF <itab>**.

Task 2

Call subroutine

1. Call the subroutine after the SELECT loop. Pass the internal table containing the data that has been read using the interface.
2. Execute your program to test the result.

Solution 11: Subroutines

Task 1

Create a subroutine with internal table transfer

1. Copy your program **ZBC400_##_SELECT_SFLIGHT_ITAB** or the corresponding model solution **SAPBC400DDS_AUTHORITY_CHECK_2** to the new program **ZBC400_##_SUBROUTINE**.
 - a) Carry out this step as usual.
2. Encapsulate the display of data in a list in a subroutine. To do so, define the subroutine and specify types for the interface parameter (name proposal: **WRITE_LIST**).
 - a) See source code excerpt in the model solution.
3. Display the data from the subroutine using a LOOP. To do this, create the required table work area as a local data object in the subroutine (name proposal: **wa**).



Hint: To specify the type of the local structure, use the ABAP statement **DATA <wa> LIKE LINE OF <itab>**.

- a) See source code excerpt in the model solution.

Task 2

Call subroutine

1. Call the subroutine after the SELECT loop. Pass the internal table containing the data that has been read using the interface.
 - a) See source code excerpt in the model solution.
2. Execute your program to test the result.
 - a) Carry out this step as usual.

Result

Source code excerpt: **SAPBC400PBS_SUBROUTINE**

```
REPORT  sapbc400pbs_subroutine.
```

```
CONSTANTS actvt_display TYPE activ_auth VALUE '03'.
```

Continued on next page

```

DATA: wa_flight TYPE sbc400focc,
      it_flight TYPE sbc400_t_sbc400focc.
PARAMETERS: pa_car TYPE sflight-carrid.

START-OF-SELECTION.
* Authority-Check:
  AUTHORITY-CHECK OBJECT 'S_CARRID'
    ID 'CARRID' FIELD pa_car
    ID 'ACTVT'   FIELD actvt_display.
CASE sy-subrc.
* User is authorized
  WHEN 0.
    SELECT carrid connid fldate seatsmax seatsocc FROM sflight
      INTO CORRESPONDING FIELDS OF wa_flight
      WHERE carrid = pa_car.
    wa_flight-percentage =
      100 * wa_flight-seatsocc / wa_flight-seatsmax.
    APPEND wa_flight TO it_flight.
  ENDSELECT.
  PERFORM write_list USING it_flight.
* User is not authorized or other error of authority-check
  WHEN OTHERS.
    WRITE: / 'Authority-Check Error'(001).
  ENDCASE.

*&-----*
*&     Form  WRITE_LIST
*&-----*
*     text
*-----*
*     -->P_IT_FLIGHT  text
*-----*
FORM write_list USING p_it_flight TYPE sbc400_t_sbc400focc.
DATA: wa LIKE LINE OF p_it_flight.
LOOP AT p_it_flight INTO wa.
  WRITE: / wa-carrid COLOR COL_KEY,
         wa-connid COLOR COL_KEY,
         wa-fldate COLOR COL_KEY,
         wa-seatsocc,
         wa-seatsmax,

```

Continued on next page

```
        wa-percentage, '%'.  
      ENDLOOP .  
    ENDFORM.                                     " WRITE_LIST
```



Lesson Summary

You should now be able to:

- Define subroutines
- Call subroutines
- Analyze the execution of subroutines in debugging mode



Unit Summary

You should now be able to:

- Define subroutines
- Call subroutines
- Analyze the execution of subroutines in debugging mode

Related Information

... Refer to the online documentation for the relevant ABAP statement.

Unit 7

Introduction to Event Blocks

Unit Overview

Refer to the lesson summaries for an overview of this unit.



Unit Objectives

After completing this unit, you will be able to:

- Describe the event-controlled processes for executing an ABAP program
- Use the event block LOAD-OF-PROGRAM or INITIALIZATION
- Use the event block START-OF-SELECTION

Unit Contents

Lesson: Event Blocks	192
----------------------------	-----

Lesson: Event Blocks

Lesson Overview

This lesson explains the advantages of event controlling using LOAD-OF-PROGRAM and START-OF-SELECTION as examples.



Lesson Objectives

After completing this lesson, you will be able to:

- Describe the event-controlled processes for executing an ABAP program
- Use the event block LOAD-OF-PROGRAM or INITIALIZATION
- Use the event block START-OF-SELECTION

Business Example

Implement your processes for specific events during the program execution.

Event Blocks

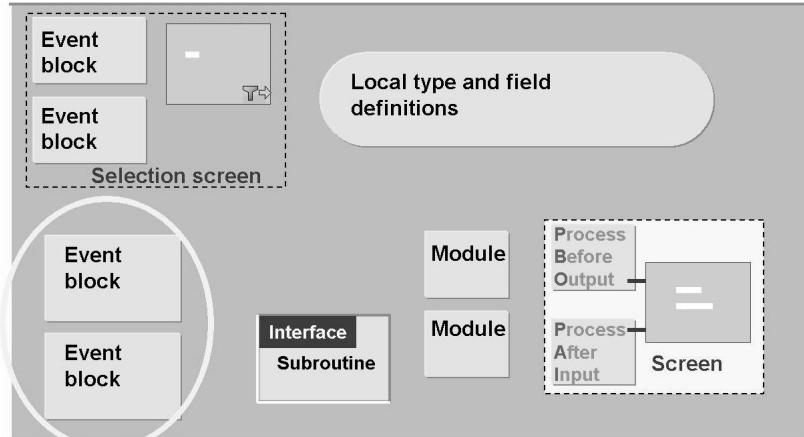


Figure 108: Possible Elements in an ABAP Program

An ABAP program is a collection of processing blocks. A processing block consists of passive source code sections that are processed sequentially when called.

Processing blocks are the smallest units in ABAP. They cannot be split, which also means that they cannot be nested.

There are various kinds of ABAP processing blocks:

- **Event blocks** are processing blocks that are called **by the ABAP runtime system** whenever the corresponding event has been triggered. Such events can be initiated by the ABAP runtime system itself or by the user.
- **Subroutines** are processing blocks that are called **by ABAP statements** - that is, the developer controls them.
- **Modules** are special ABAP processing blocks that are called by **screen statements**. These are only required for screen processing.

When a program is started, all the global data objects of the program are created in the working memory first. Declarative ABAP statements are therefore **not** components of ABAP processing blocks, but are collected by the ABAP runtime system when the overall source code is generated. However, for the sake of clarity, you should place all declarative statements together at the beginning of the source code, with the exception of local declarations within subroutines.

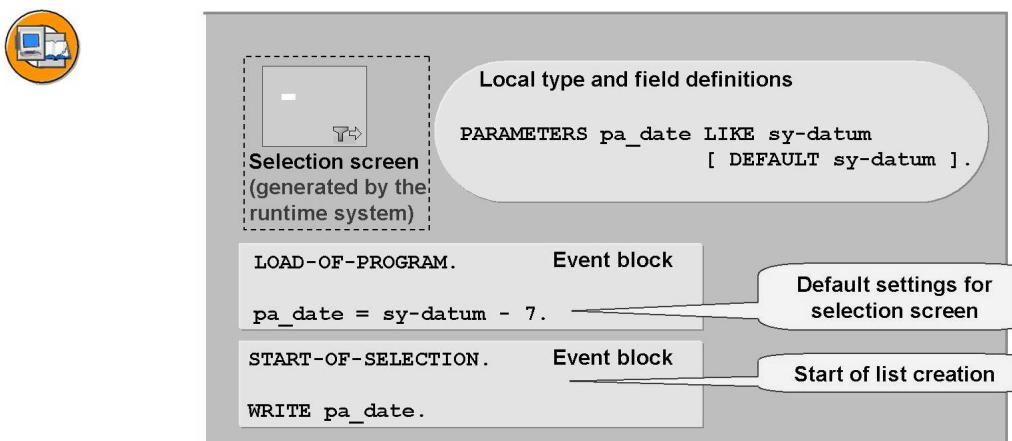


Figure 109: Example: ABAP Program with Several Event Blocks and a Selection Screen

In all of the programs that we have seen so far in this course, only one event block has been programmed in addition to the declarative statements. In this case, there is no need to declare the event block explicitly. However, in more complex programs, we will require several different event blocks.

The example given here contains a selection screen with an input field for a date. The date of the same weekday in the previous week should appear as the default value. The PARAMETERS statement is one of the declarative language elements. However, a calculation is required in order to preset the date. (The DEFAULT addition to the PARAMETERS statement only ensures

that the data object is filled with a default value.) The ABAP runtime system fills the system field SY-DATUM with the current date at the start of the program. To be able to change variables **at runtime**, you must therefore program a processing block. This should be processed **before** the standard selection screen is transmitted. You can therefore use the appropriate event block **LOAD-OF-PROGRAM**.

You can use the standard event block **START-OF-SELECTION** to create the list.

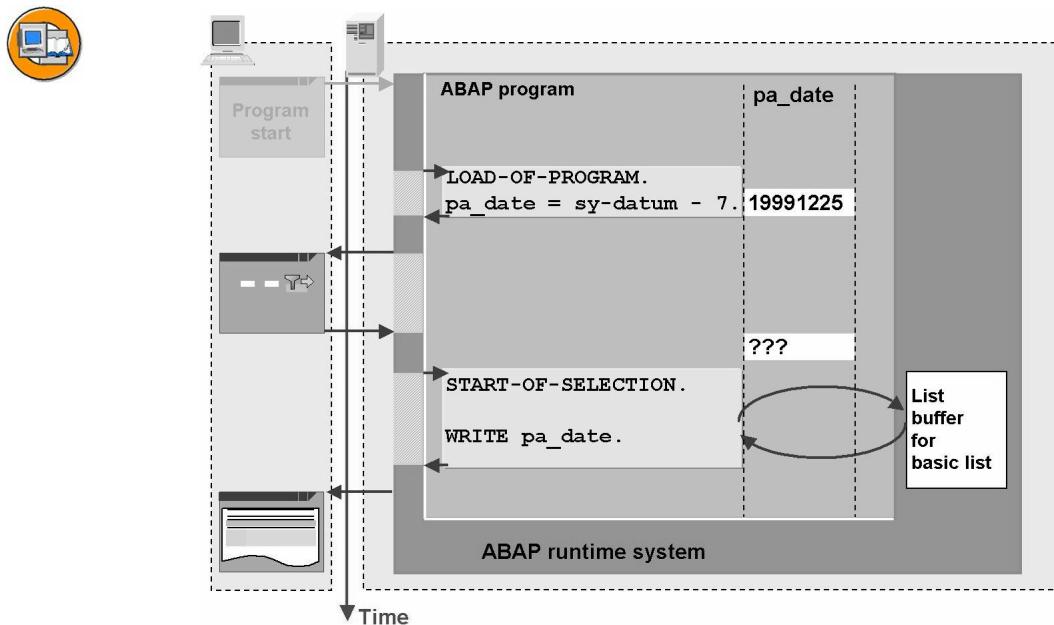


Figure 110: Example Program Runtime Behavior

The above flow diagram clearly shows the sequence of processing steps by the ABAP runtime system .

Event blocks are called in a **fixed sequence** designed for list processing:

1. The first event block to be called is **LOAD-OF-PROGRAM**. In the above example, the date “a week ago today” is calculated and placed into the **PA_DATE** input field. Afterwards, the systems sends the selection screen with this new value to the presentation server. The user has the option of changing the value.
2. Then the selection screen is sent to the presentation server.
3. After the user leaves the selection screen, the event block **START-OF-SELECTION** is called.

4. If the **START-OF-SELECTION** event block contains ABAP statements like WRITE, SKIP, or ULINE, a list buffer is filled.
5. As soon as **START-OF-SELECTION** has been **processed completely**, the list buffer is sent to the presentation server as a list.

Both events presented here are initiated by the ABAP runtime system **exactly once** each time a program is executed.



- **Introduced by an event keyword**
- **Ended by the next processing block**
- **Event blocks for different tasks**
- **Processing sequence for event blocks is defined by the runtime system - sequence of event blocks in the source code is irrelevant**
- **Standard event block in executable program: START-OF-SELECTION**

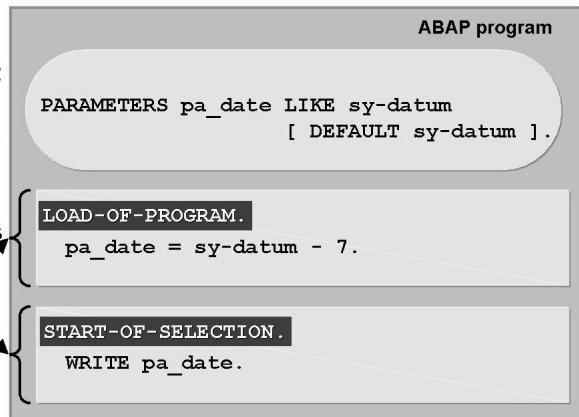


Figure 111: Event Block Characteristics

An event block is introduced with an **event keyword**. It ends **implicitly** when the next processing block starts.

Event blocks are called by the ABAP runtime system. Therefore, the sequence in which the event blocks are written in the program is **not important**.

LOAD-OF-PROGRAM is the first event block called in **all** ABAP program types. This event block allows you to set default values that can only be determined at runtime, for example.

→ **Note:** For compatibility reasons, the event block **INITIALIZATION** is available as an alternative. However, it is available for **executable programs only**.

START-OF-SELECTION is the first event block for creating a list. It is called by the ABAP runtime system as soon as you have left the standard selection screen. It is also only available for executable programs.

In executable programs, there are further event blocks available for different tasks that are designed for creating lists. A complete list is available in the ABAP documentation: First, choose *ABAP Overview* followed by *ABAP Overview Diagrams* → *Classification of Keywords by Type* → *Event Keywords*.



Lesson Summary

You should now be able to:

- Describe the event-controlled processes for executing an ABAP program
- Use the event block LOAD-OF-PROGRAM or INITIALIZATION
- Use the event block START-OF-SELECTION



Unit Summary

You should now be able to:

- Describe the event-controlled processes for executing an ABAP program
- Use the event block LOAD-OF-PROGRAM or INITIALIZATION
- Use the event block START-OF-SELECTION

Related Information

... Refer to the article “Event Blocks” in the online documentation and to the keyword documentation for the relevant ABAP statement.

Unit 8

User dialogs

Unit Overview

Refer to the lesson summaries for an overview of this unit.



Unit Objectives

After completing this unit, you will be able to:

- Describe the attributes and benefits of ABAP lists
- Implement interactive lists
- Describe the event-controlled processes for the AT LINE-SELECTION event when executing an ABAP program
- List the properties and benefits of selection screens
- Implement the options for restricting selections using selection screens
- Implement input checks on the selection screen
- List attributes and benefits of screens
- Write programs that display data on screens
- Write programs that allow users to change some data on screens
- Write programs that allow users to affect subsequent program flow using pushbuttons
- Create and set a GUI title
- Maintain menu bars
- Maintain standard toolbars
- Maintain application toolbars
- Maintain function key assignments
- Create and set a GUI status for lists and screens

Unit Contents

Lesson: List.....	201
-------------------	-----

Exercise 12: User Dialogs: Lists	215
Lesson: Selection Screen.....	223
Exercise 13: Selection Screen.....	239
Lesson: Screen	245
Exercise 14: Creating a Screen	275
Exercise 15: Screens and Data Transport	281
Exercise 16: Screens: Specifying the Next Screen Dynamically.....	287
Lesson: User interface	295
Exercise 17: Creating an Interface	307

Lesson: List

Lesson Overview

In this lesson, you will learn about the benefits of using the dialog form known as the ABAP list. You will also create a detail list using an ABAP event.



Lesson Objectives

After completing this lesson, you will be able to:

- Describe the attributes and benefits of ABAP lists
- Implement interactive lists
- Describe the event-controlled processes for the AT LINE-SELECTION event when executing an ABAP program

Business Example

You need to write a program that displays line-specific detailed information when the user chooses a row in an interactive list.

The List

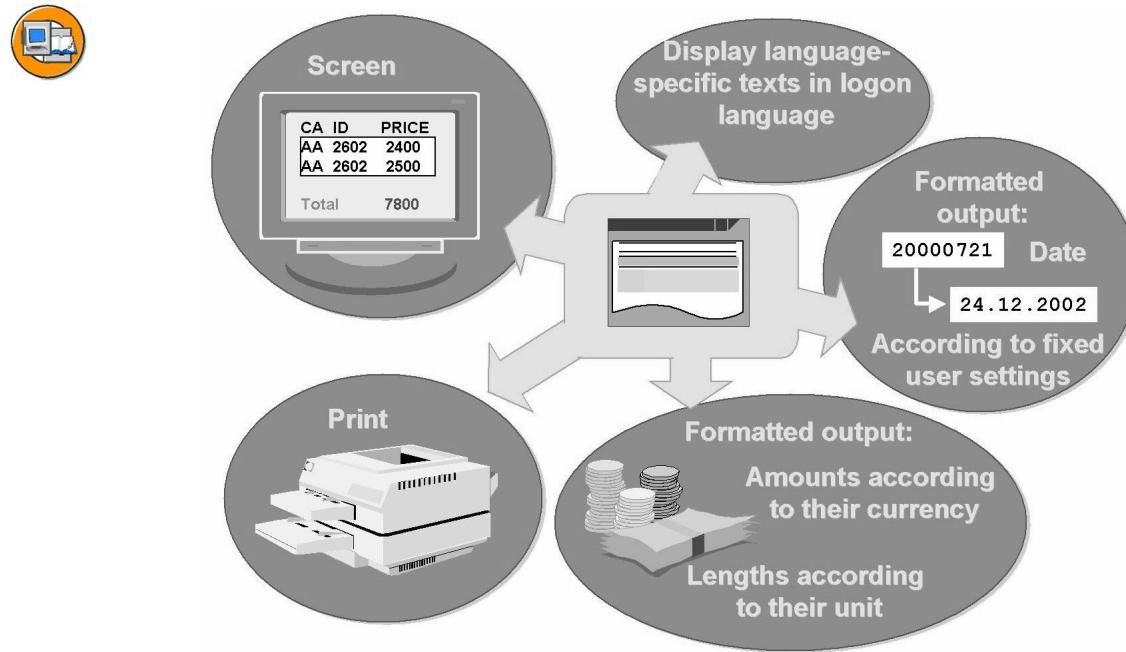


Figure 112: Features of Lists

The main purpose of a list is to display data for the minimum amount of programming. Lists also take the special requirements of business data into account:

- Lists are language-independent. Texts and headers appear in the logon language whenever a translation is available.
- Lists can display monetary values in the appropriate currency.

The following options are available to the user outputting a list:

- Screen: You can add colors and icons.
- Printers
- Internet/Intranet: The system automatically converts to HTML.
- Saving: You can save lists within the *SAP R/3* as well as outside for further processing – for example, using spreadsheet programs.

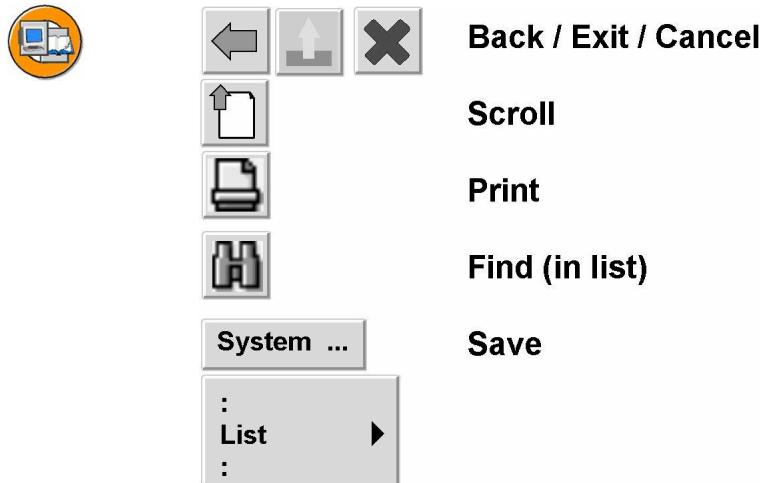


Figure 113: Standard List Functions

The standard list interface offers the user several navigation features:

- *Back*
- *Exit*
- *Cancel*
- *Print*
- *Find... in a list*
- *Save*: As a file, in the menu, or to a buffer
- *Send*: Sends the list in e-mail form

You can use the *Menu Painter* to adapt the default list interface to your own needs.

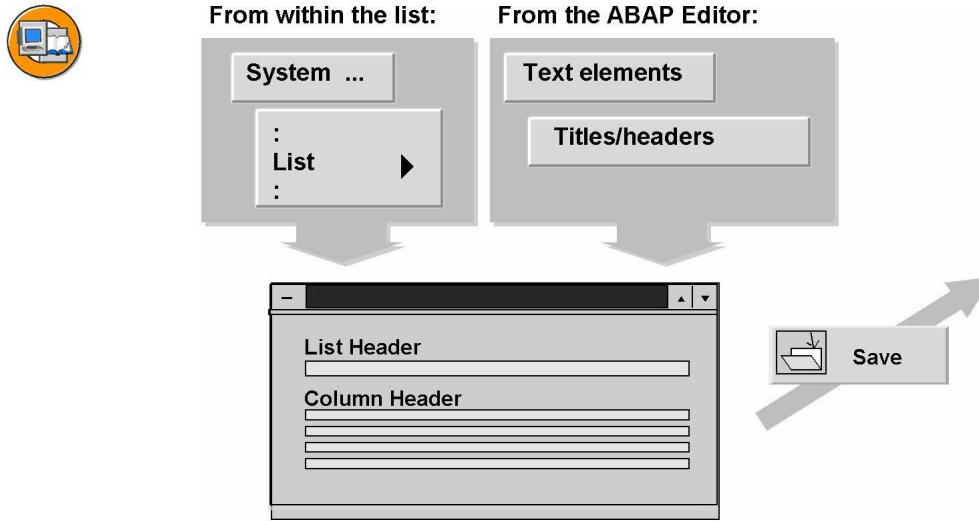


Figure 114: List Headers in the Default Page Header

Each list can have a **list header** and up to four lines of **column headers**. There are two different ways to do this:

From the ABAP Editor

The text element maintenance screen provides a ruler.

From the list:

Save the **program**, **activate** it, and then by generate the list by choosing **Execute** for the program. You can then open a maintenance screen by choosing **System → List → List Header**. The main advantage of using this method is that the list is still displayed on the screen. This makes it easier to position column headers.

The next time you start the program, the new headers will appear in the list automatically. If there is no header text entered, the program title is inserted in the header.

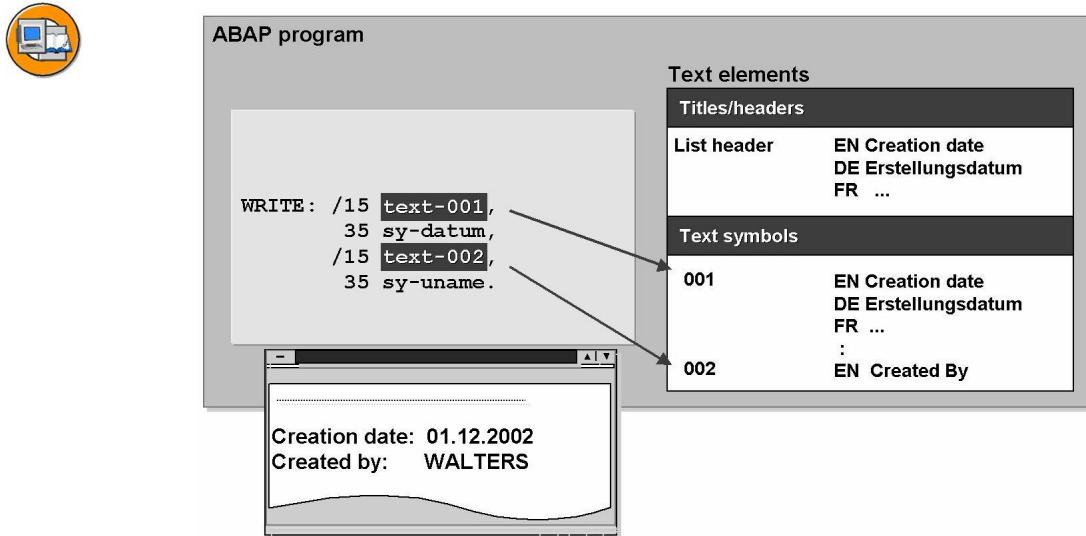


Figure 115: Multilingual Capabilities of Lists

Titles and headers are part of the program's text elements. You can translate all text elements into other languages. The logon language setting on the logon screen specifies the language in which text elements will be displayed.

Text symbols are another kind of text element. Text symbols are text literals that can be translated and are assigned to the program. They are frequently used to make lists independent of a specific language.

You can write text symbols into your program in either of the following ways:

- **TEXT-ccc** (where ccc is a string three characters long)
- **'...' (ccc)** (where ... is any text and ccc is a string three characters long)

In the *ABAP Editor*, you can navigate to text symbol maintenance either by choosing *Goto→Text Elements→Text Symbols* or by double-clicking the number of a text symbol.

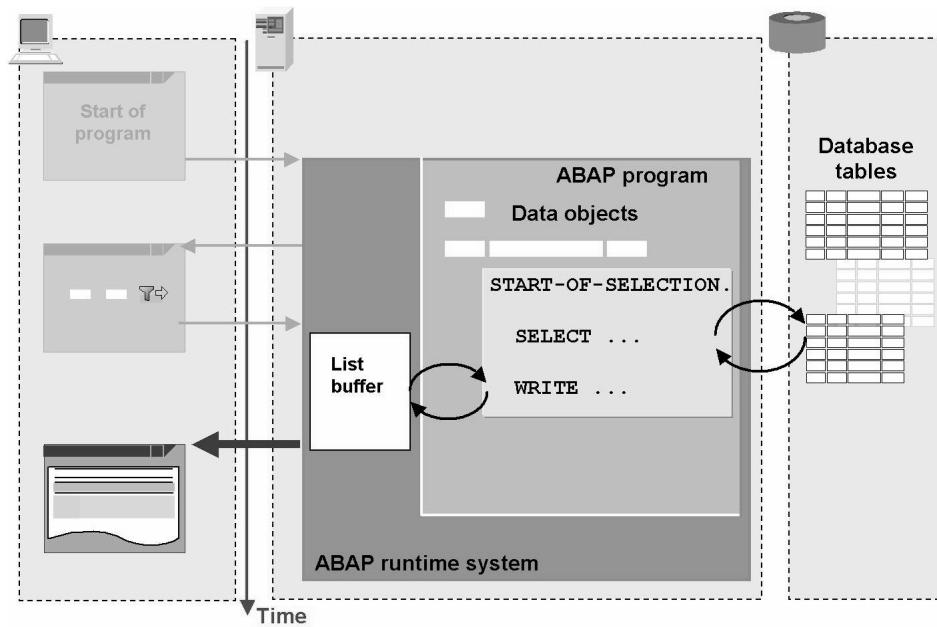


Figure 116: Lists in Executable Programs

In executable programs, lists are automatically displayed after their corresponding event blocks have been processed. These processing blocks must, however, contain a statement that writes to the list buffer, such as **WRITE**, **SKIP**, or **ULINE**.

Event blocks are called in a sequence designed for list processing:

- Before the selection screen: **LOAD-OF-PROGRAM**.
- After the user leaves the selection screen: **START-OF-SELECTION**.

All output from the event blocks that create the lists and from the modularization units processed before a list is sent, is temporarily stored in the list buffer. (The above example shows only the **START-OF-SELECTION** event). Once all list creation event blocks (here only **START-OF-SELECTION**) have been processed, all data from the list buffer is displayed as a list.

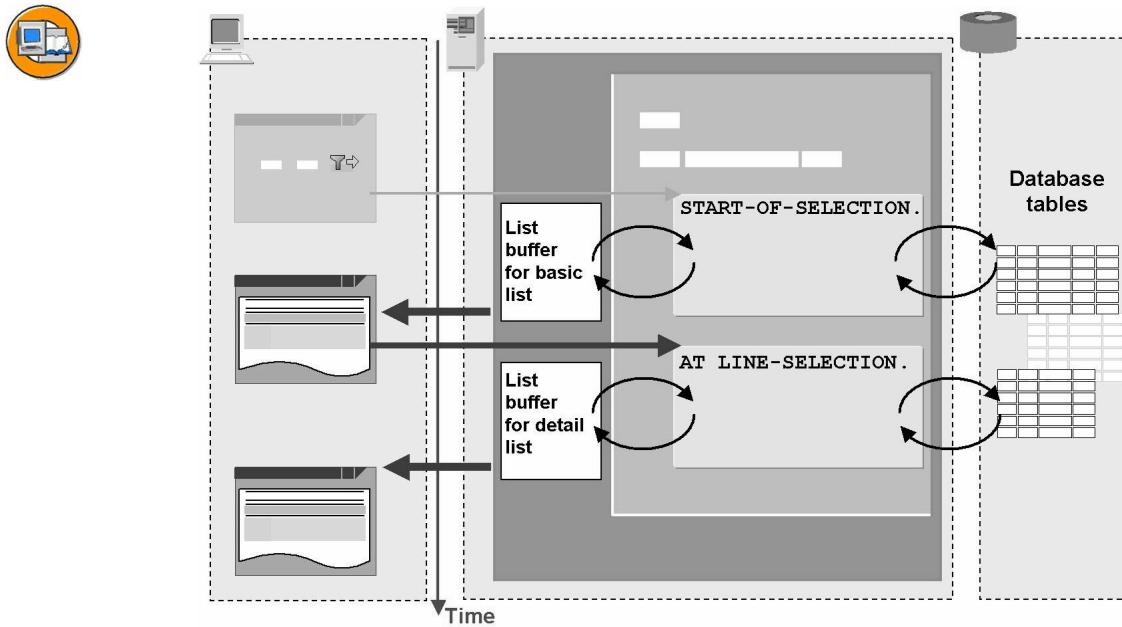


Figure 117: Execution of Detail Lists

In executable programs, you can use the event block **AT LINE-SELECTION** to create detail lists.

The ABAP runtime system ensures that the following take place:

- The basic list is sent after the event blocks responsible for creating the basic list have been processed (for example, after **START-OF-SELECTION**). The system field **SY-LSIND** contains the value 0.
- The event block **AT LINE-SELECTION** is processed using the **PICK** function code each time you double-click an entry or choose an action for the system to perform. If you are using a default status, this happens automatically every time you choose an icon, the menu function *Choose*, or the function key **F2**.
- The detail list is sent after the **AT LINE-SELECTION** event block has been processed and the value contained in **SY-LSIND** is increased by one.
- After every user action with the function code **BACK** performed on a detail list at level n, the detail list at level n-1 is sent.

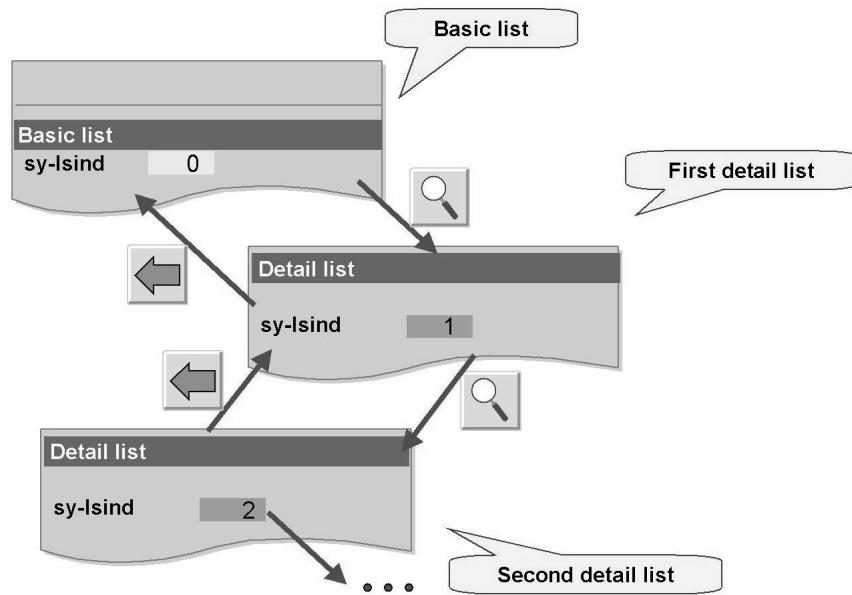


Figure 118: Navigation in Detail Lists

The lists in the example program should function as follows:

- The basic list should display the text “Basic List” and the system field SY-LSIND.
- The user can navigate to a detail list by using any of the following:
 - Double-clicking
 - The *Choose* button in the application toolbar
 - A menu function
 - The **F2** function key Then the “detail list” appears and the system field SY-LSIND has the value 1.
- Repeating this action calls the second detail list. The system field SY-LSIND contains the value 2 instead (representing the current detail list level).
- Repeating the action should consecutively increase SY-LSIND’s value by one up to a value of twenty (the total number of detail lists supported).
- Choosing *Back* takes the user back a single detail list level at a time until the basic list is reached.



```
...  
START-OF-SELECTION.  
    WRITE: / text-001 COLOR col_heading,  
           / 'sy-lsind',  
             sy-lsind color 2.  
  
AT LINE-SELECTION.  
    WRITE: / text-002 COLOR col_heading.  
    ULINE.  
    WRITE: / 'sy-lsind',  
          sy-lsind color 4.  
  
Text symbols:  
001 Basic list  
002 Detail list
```

Figure 119: Syntax Example: Simple Detail List

You program a detail list as follows:

- You create a basic list by filling the basic list buffer at an appropriate event block (here **START-OF-SELECTION**) using either WRITE, SKIP, or ULINE.
- When programming detail lists, use the event block **AT LINE-SELECTION**. Whenever you use WRITE, SKIP, or ULINE with this event block, you fill the detail list buffer for the next level (the detail list buffer with a level value one greater than the level on which the user performed the action).
- You can pre-determine navigation between details lists by querying system field SY-LSIND at the event block **AT LINE-SELECTION**.



Figure 120: Example of a Detail List in an Application

We will now write a program using both basic lists and detail lists:

The basic list in your program should contain flight data, such as carrier ID and flight numbers, departure city and airport, destination city and airport, as well as departure and arrival times. This data can be found in the database table **SPFLI**.

The user should be able to access information about any particular flight by double-clicking its carrier ID and flight number. Flight date and occupancy should be displayed. This data can be found in the database table **SFLIGHT**. You must use the **SPFLI** key fields in this details list in order to read the appropriate data in **SFLIGHT**. The following graphics demonstrate how this is done.

The sample program is **SAPBC400UDD_DETAIL_LIST**, part of development class **BC400**.

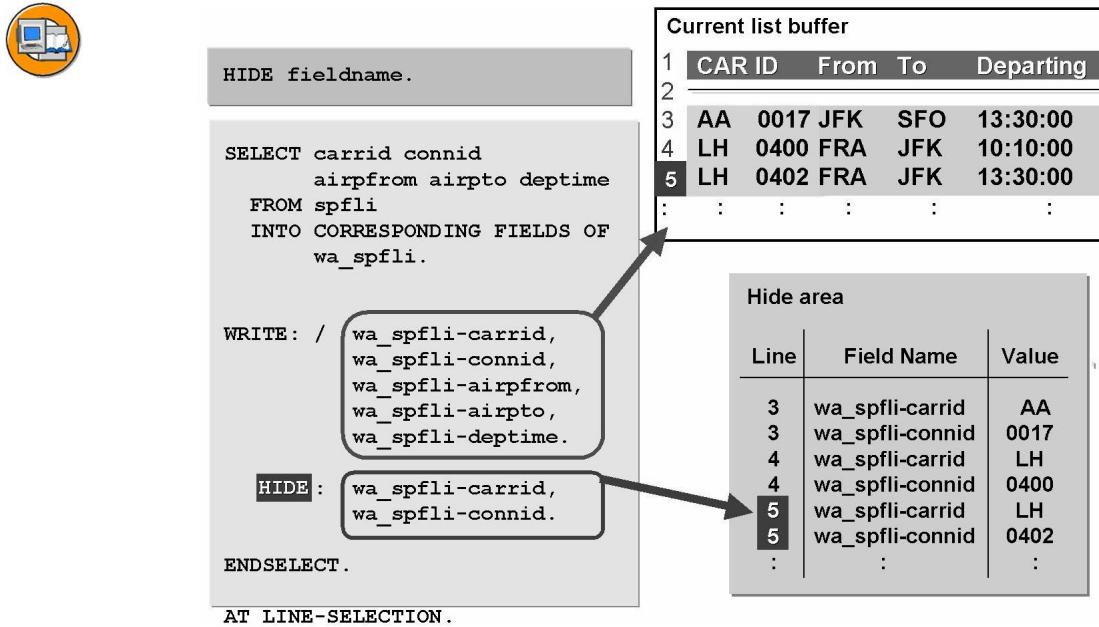


Figure 121: Buffering Global Data in the Hide Area

When the event **AT LINE-SELECTION** is processed, a program's data objects contain the same values as before basic list display. A detail list, however, often needs data selected within the basic list itself. It would be good if you could store row-specific data from the row you have selected and then automatically insert it in the corresponding data objects. You can implement this using the **HIDE** area. You can predetermine which row-specific information should be stored when you are creating a basic list.

To do this, you use the ABAP keyword **HIDE** followed by a list of the data objects you require. The runtime system automatically records the name and contents of the data object in relation to its line position in the list currently being created.

The **HIDE global_field.** statement buffers the content of the global data field `global_field` with reference to the current display line. You do **not** have to display the field beforehand using the **WRITE** statement. You merely have to program the **HIDE** statements after the last **WRITE** statement in the current line.

The data field can also be a "flat" structure.

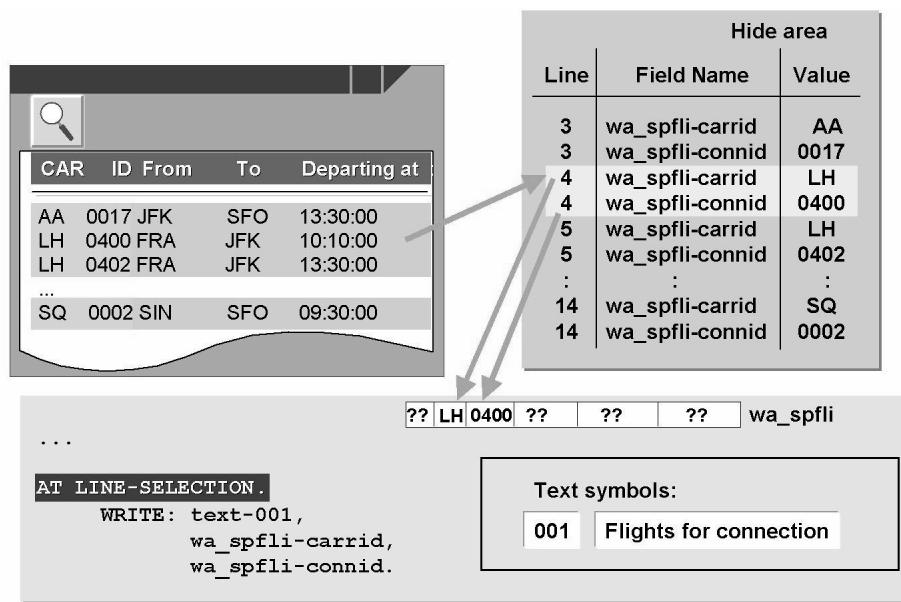


Figure 122: Selecting a Row

Selecting a row (by positioning the cursor followed by PICK or double-clicking) triggers the **AT LINE-SELECTION** event.

First the values for this line buffered in the HIDE area are inserted **back into the corresponding global data objects**.

Then the system processes the relevant event block in the ABAP program.



```

...
AT LINE-SELECTION.
IF sy-lsind = 1.
  WRITE: text-001,
         wa_spfli-carrid,
         wa_spfli-connid.

  SELECT fldate seatsmax seatsocc
    FROM sflight
    INTO CORRESPONDING FIELDS OF wa_sflight
    WHERE carrid = wa_spfli-carrid
      AND connid = wa_spfli-connid.

  WRITE:/ wa_sflight-fldate,
         wa_sflight-seatsmax,
         wa_sflight-seatsocc.
ENDIF.

```

Text symbols:

001	Flights for connection
-----	------------------------

Figure 123: Syntax Example: Additional Data for Selected Row

You create a detail list by filling the detail list buffer in the AT LINE-SELECTION event block using either WRITE, SKIP, or ULINE.

The key fields of the airline will be displayed in the example program. The flights available for this airline in the database table **SFLIGHT** are read using a SELECT loop.

Note that the line-specific information on the airline is only available by double-clicking in the data objects if the relevant data objects have been placed in the HIDE area when the basic list was created.

Exercise 12: User Dialogs: Lists

Exercise Objectives

After completing this exercise, you will be able to:

- Create a detail list in a program

Business Example

Extend the program **ZBC400_##_SELECT_SFLIGHT**

or the corresponding model solution as follows:

Once the user has selected a flight on the basic list (double-click or **F2** on the relevant row in the list), display a detail list containing all of the bookings for the selected flight.

Task 1

Prepare the detail list

1. Copy your program **ZBC400_##_SELECT_SFLIGHT** or the corresponding template **SAPBC400DDS_AUTHORITY_CHECK** to the new program **ZBC400_##_DETAIL_LIST**.
2. Make sure that the key fields of the database table **SFLIGHT** are available to you for creating the detail list when the user selects a flight from the basic list (double-click or **F2** on the corresponding row in the list).
3. Add the AT LINE-SELECTION event to your program to allow you to construct a detail list.

Task 2

Create the detail list

1. In the first line of the detail list, display key information from the selected flight. Under this line, display a horizontal line and a blank line.
2. Read all of the bookings from database table **SBOOK** for the selected flight. Make sure that you only read fields from the database table that you want to display in the list. To display the following fields of the database table **SBOOK** in the detail list:

**BOOKID, CUSTOMID, CUSTTYPE, CLASS, ORDER_DATE,
SMOKER, CANCELLED.**

Continued on next page

3. **Optional:** Display the fields **LOCCURAM** and **LOCCURKEY** on the detail list: Ensure that the currency amount **LOCCURAM** is displayed with the appropriate formatting for the currency **LOCCURKEY**. Use the addition **CURRENCY <currency_key>** in the **WRITE** statement.



Hint: `WRITE: wa_sflight-price CURRENCY
wa_sflight-currency, wa_sflight-currency.`

4. **Optional:** Display the **BOOKID** field in the color **COL_KEY**.

Solution 12: User Dialogs: Lists

Task 1

Prepare the detail list

1. Copy your program **ZBC400_##_SELECT_SFLIGHT** or the corresponding template **SAPBC400DDS_AUTHORITY_CHECK** to the new program **ZBC400_##_DETAIL_LIST**.
 - a) Carry out this step as usual.
2. Make sure that the key fields of the database table **SFLIGHT** are available to you for creating the detail list when the user selects a flight from the basic list (double-click or **F2** on the corresponding row in the list).
 - a) See source code excerpt in the model solution.
3. Add the **AT LINE-SELECTION** event to your program to allow you to construct a detail list.
 - a) See source code excerpt in the model solution.

Task 2

Create the detail list

1. In the first line of the detail list, display key information from the selected flight. Under this line, display a horizontal line and a blank line.
 - a) See source code excerpt in the model solution.
2. Read all of the bookings from database table **SBOOK** for the selected flight. Make sure that you only read fields from the database table that you want to display in the list. To display the following fields of the database table **SBOOK** in the detail list:

**BOOKID, CUSTOMID, CUSTTYPE, CLASS, ORDER_DATE,
SMOKER, CANCELLED.**

- a) See source code excerpt in the model solution.

Continued on next page

3. **Optional:** Display the fields **LOCCURAM** and **LOCCURKEY** on the detail list: Ensure that the currency amount **LOCCURAM** is displayed with the appropriate formatting for the currency **LOCCURKEY**. Use the addition CURRENCY <currency_key> in the WRITE statement.



Hint: `WRITE: wa_sflight-price CURRENCY
wa_sflight-currency, wa_sflight-currency.`

- a) See source code excerpt in the model solution.
4. **Optional:** Display the **BOOKID** field in the color **COL_KEY**.
- a) See source code excerpt in the model solution.

Result

Model solution without optional exercises: **SAPBC400UDS_DETAIL_LIST**
`REPORT sapbc400uds_detail_list.`

```
CONSTANTS: actvt_display TYPE activ_auth VALUE '03'.

DATA: wa_flight TYPE sbc400focc,
      wa_sbook  TYPE sbook.

PARAMETERS: pa_car TYPE s_carr_id.

START-OF-SELECTION.
  AUTHORITY-CHECK OBJECT 'S_CARRID'
    ID 'CARRID' FIELD pa_car
    ID 'ACTVT'   FIELD actvt_display.
  CASE sy-subrc.
    WHEN 0.
      SELECT carrid connid fldate seatsmax seatsocc FROM sflight
        INTO CORRESPONDING FIELDS OF wa_flight
        WHERE carrid = pa_car.
      wa_flight-percentage =
        100 * wa_flight-seatsocc / wa_flight-seatsmax.
      WRITE: / wa_flight-carrid,
              wa_flight-connid,
              wa_flight-fldate,
              wa_flight-seatsocc,
```

Continued on next page

```

        wa_flight-seatsmax,
        wa_flight-percentage, '%'.
* Hide key field values corresponding to the actual line
HIDE: wa_flight-carrid, wa_flight-connid,
       wa_flight-fldate.
ENDSELECT.
WHEN OTHERS.
  WRITE: / 'Authority-Check Error' (001).
ENDCASE.
CLEAR wa_flight.
* Program continues here, if a line is selected on basic list

AT LINE-SELECTION.
IF sy-lsind = 1.

* Key fields transported back from hide area to ABAP dataobjects
  WRITE: / wa_flight-carrid,
         wa_flight-connid,
         wa_flight-fldate.

  ULINE.
  SKIP.
* Selection of bookings, which depend on selected flight
  SELECT bookid customid custtype class order_date
    smoker cancelled loccuram loccurkey
  FROM sbook INTO CORRESPONDING FIELDS OF wa_sbook
  WHERE carrid = wa_flight-carrid
    AND connid = wa_flight-connid
    AND fldate = wa_flight-fldate.

* Creation of detail list
  WRITE: / wa_sbook-bookid,
         wa_sbook-customid,
         wa_sbook-custtype,
         wa_sbook-class,
         wa_sbook-order_date,
         wa_sbook-smoker,
         wa_sbook-cancelled.

  ENDSELECT.
ENDIF.
CLEAR wa_flight.

```

Model solution with optional exercises: **SAPBC400UDS_DETAIL_LIST1**

Continued on next page

```

REPORT  sapbc400uds_detail_list1.

CONSTANTS: actvt_display TYPE activ_auth VALUE '03'.

DATA: wa_flight TYPE sbc400focc,
      wa_sbook  TYPE sbook.

PARAMETERS: pa_car TYPE s_carr_id.

START-OF-SELECTION.

  AUTHORITY-CHECK OBJECT 'S_CARRID'
    ID 'CARRID' FIELD pa_car
    ID 'ACTVT'   FIELD actvt_display.

CASE sy-subrc.
  WHEN 0.
    SELECT carrid connid fldate seatsmax seatsocc FROM sflight
      INTO CORRESPONDING FIELDS OF wa_flight
      WHERE carrid = pa_car.
    wa_flight-percentage =
      100 * wa_flight-seatsocc / wa_flight-seatsmax.

    WRITE: / wa_flight-carrid,
            wa_flight-connid,
            wa_flight-fldate,
            wa_flight-seatsocc,
            wa_flight-seatsmax,
            wa_flight-percentage, '%'.
  * Hide key field values corresponding to the actual line
  HIDE: wa_flight-carrid, wa_flight-connid, wa_flight-fldate.
ENDSELECT.

  WHEN OTHERS.
    WRITE: / 'Authority-Check Error' (001).
ENDCASE.

  CLEAR wa_flight.

* Program continues here, if a line is selected on basic list
AT LINE-SELECTION.

  IF sy-lsind = 1.
* Key fields transported back from hide area to ABAP dataobjects
  WRITE: / wa_flight-carrid, wa_flight-connid, wa_flight-fldate.
  ULINE.

```

Continued on next page

```
SKIP.  
* Selection of bookings, which depend on selected flight  
SELECT bookid customid custtype class order_date  
      smoker cancelled loccuram loccurkey  
  FROM sbook INTO CORRESPONDING FIELDS OF wa_sbook  
 WHERE carrid = wa_flight-carrid  
 AND connid = wa_flight-connid  
 AND fldate = wa_flight-fldate.  
* Creation of detail list  
WRITE: / wa_sbook-bookid COLOR COL_KEY,  
        wa_sbook-customid,  
        wa_sbook-custtype,  
        wa_sbook-class,  
        wa_sbook-order_date,  
        wa_sbook-smoker,  
        wa_sbook-cancelled,  
        wa_sbook-loccuram CURRENCY wa_sbook-loccurkey,  
        wa_sbook-loccurkey.  
ENDSELECT.  
ENDIF.  
CLEAR wa_flight.
```



Lesson Summary

You should now be able to:

- Describe the attributes and benefits of ABAP lists
- Implement interactive lists
- Describe the event-controlled processes for the AT LINE-SELECTION event when executing an ABAP program

Lesson: Selection Screen

Lesson Overview

In this lesson, you will learn about the benefits of using the dialog form known as the selection screen. We will also deal with the ABAP statements that generate selection screens, PARAMETERS and SELECT-OPTIONS. There is also an ABAP event (AT SELECTION-SCREEN), which you will use to perform an authorization check.



Lesson Objectives

After completing this lesson, you will be able to:

- List the properties and benefits of selection screens
- Implement the options for restricting selections using selection screens
- Implement input checks on the selection screen

Business Example

You need to write a program that allows users to restrict the amount of data it reads by entering value ranges on a selection screen.

Advantages of Selection Screens

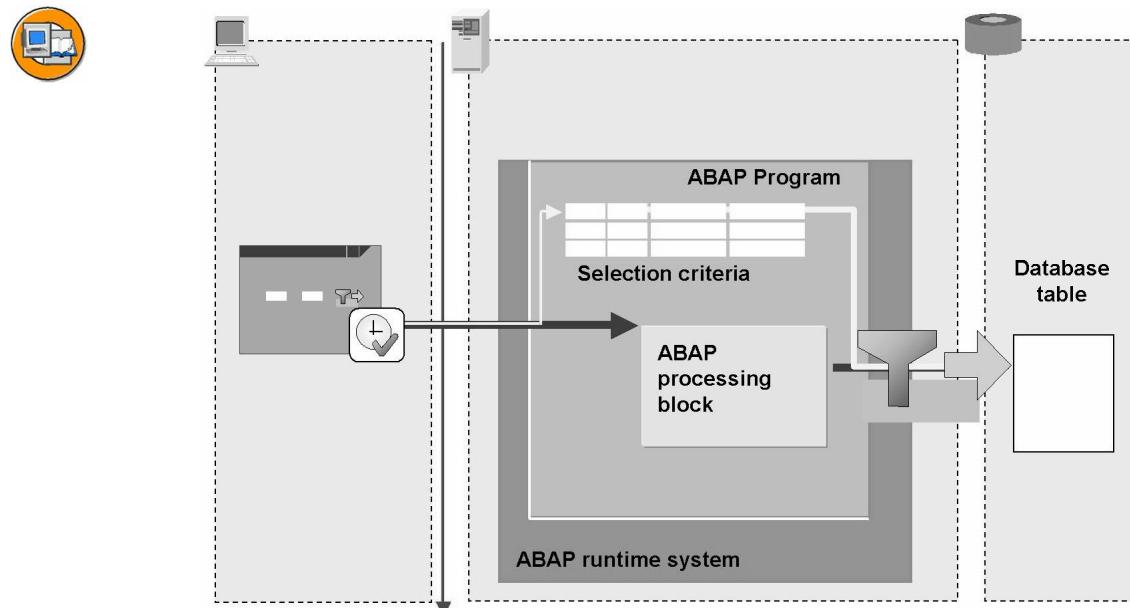


Figure 124: Purpose of Selection Screens

Selection screens allow users to enter selection criteria required by the program for it to continue.

For example, if you create a list containing data from a very large database table, you can use a selection screen to restrict the amount of that data that is selected. At runtime, the user can select intervals for one of the key fields, and only data that corresponds to this selection is read from the database and displayed in the list. This considerably reduces the load on the network.

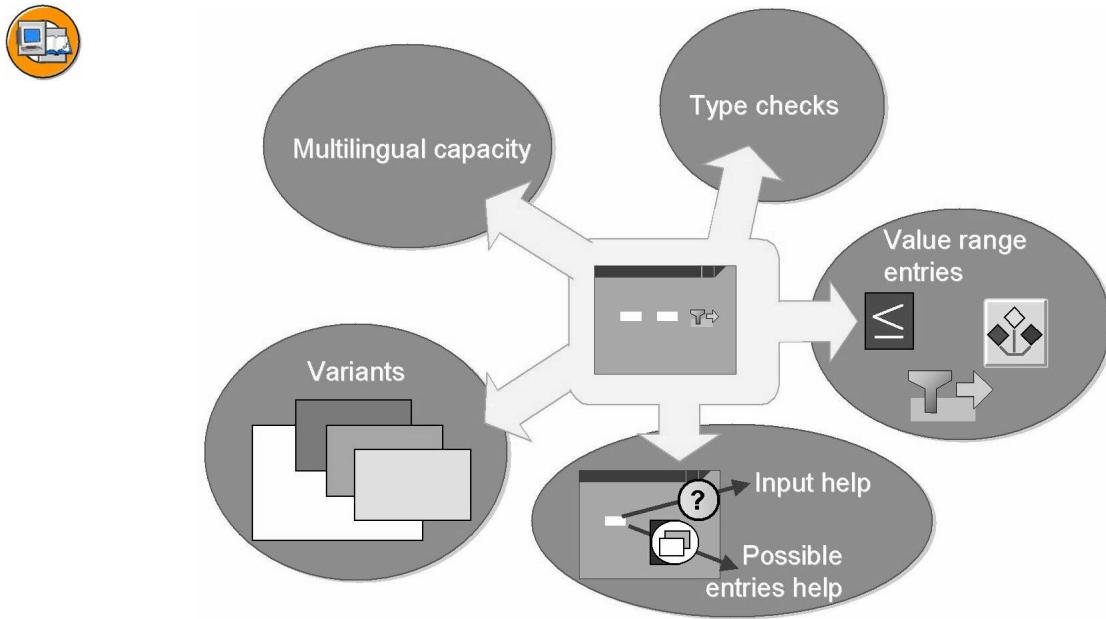


Figure 125: Selection Screen Attributes

Selection screens are input masks for value sets that restrict the amount of data read from the database. The user has the following options:

- Entries in single fields
- **Complex entries**: Intervals, operations, and patterns
- Saving selections fields filled with values as **variants**
- Possible entries help, respectively **search helps**, are available by choosing the **F4** function key or the *Possible Entries* button

You can translate **selection texts** into other languages so that they are then displayed in the language in which the user is logged on.

The system performs a **type check**: If you enter a value with an incorrect type, the system displays an error message and makes the field ready for new input.

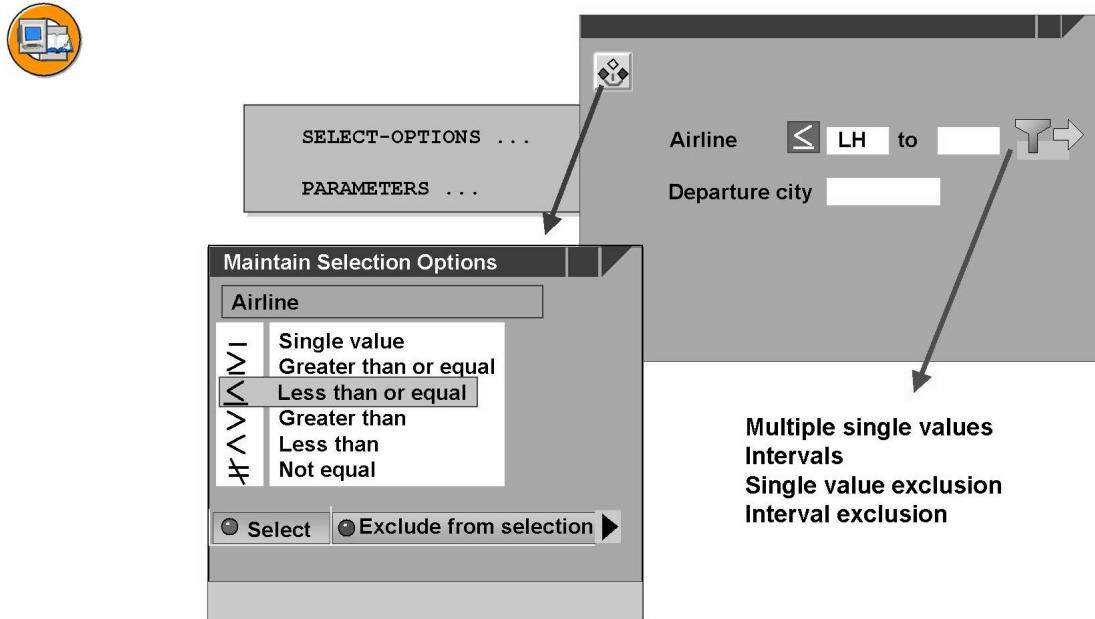


Figure 126: Entering Value Sets

Selection screens allow you to define **selection options** for complex entries as well as simple input fields.

- Setting selection options
- Entering multiple values or intervals
- Defining a value set that is not to be selected and is to be subtracted from the value set to be selected at the end.

Every selection screen contains an information icon (also known as a *Help on Screen* icon). Choose this icon to display additional information.

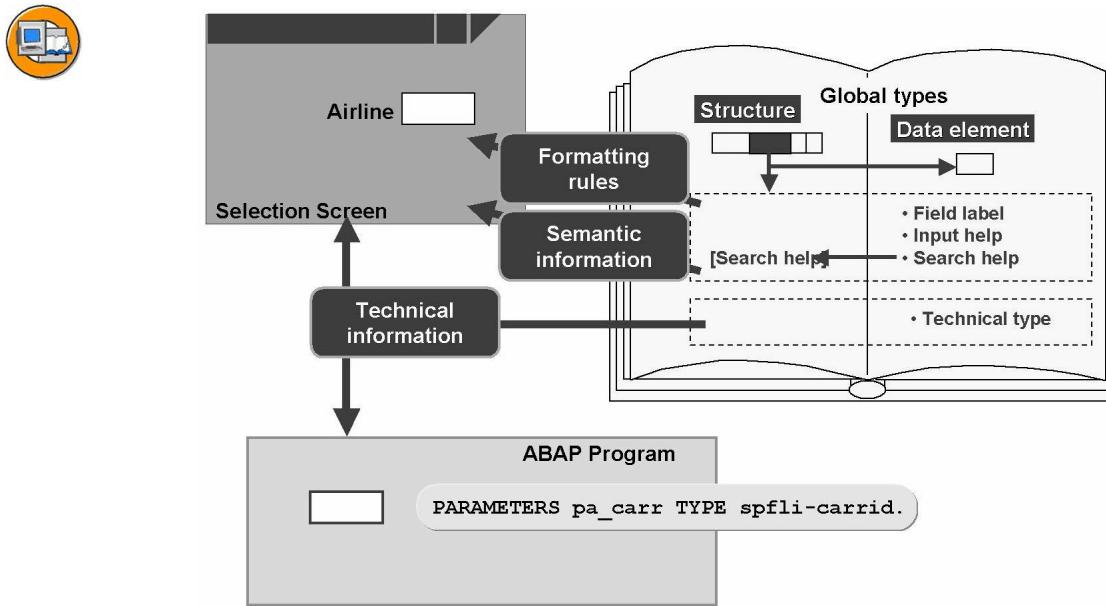


Figure 127: Using the Semantic Information of Global Types

If an input field is typed with a **data element**, the following additional semantic information is available:

- You can use a **field name** as a selection text.
- **Input help** (F1 help) is available automatically from the data element.
- **Possible entries help** (F4 help) is available automatically, provided the data element has been linked with a search help. A search help is a standalone object defined in the *ABAP Dictionary*. It controls the dialog with, and data retrieval for, the possible entries help.

If an input field is typed with a **structure field** that uses a data element, the following semantic information is available:

- **Field names** and **input help** (F1 help) are copied from the data element that has been used to type the *ABAP Dictionary* structure field.
- If a structure field is coupled with a search help, then this is the search help that is used for the **possible entries** help (F4 help) – that is, it obscures the data element search help. If there is no search help coupled with the structure field, the system uses the data element search help.

Bear in mind that the global type you choose to provide a type for an input field affects the semantic information available to the user.

For more information, refer to the online documentation for the *ABAP Dictionary*.

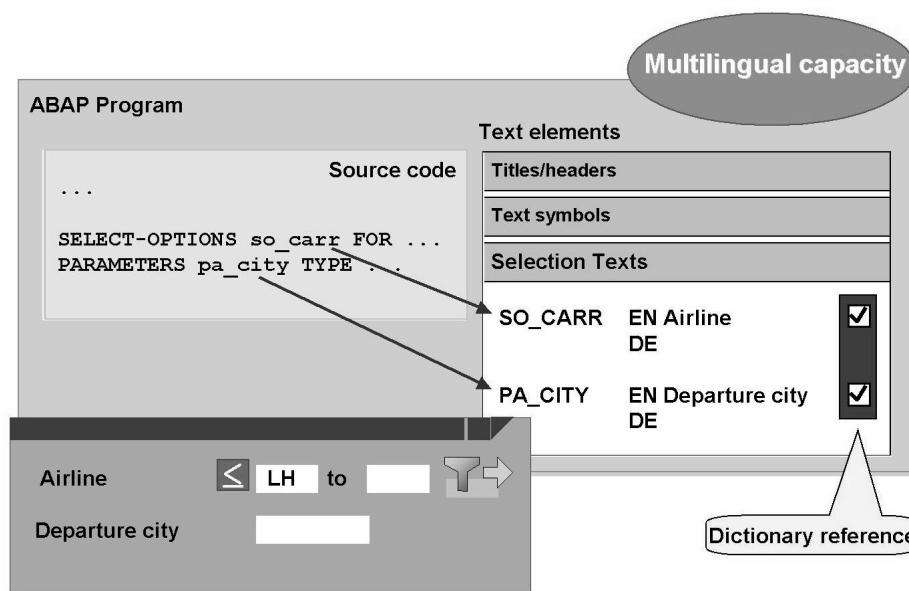
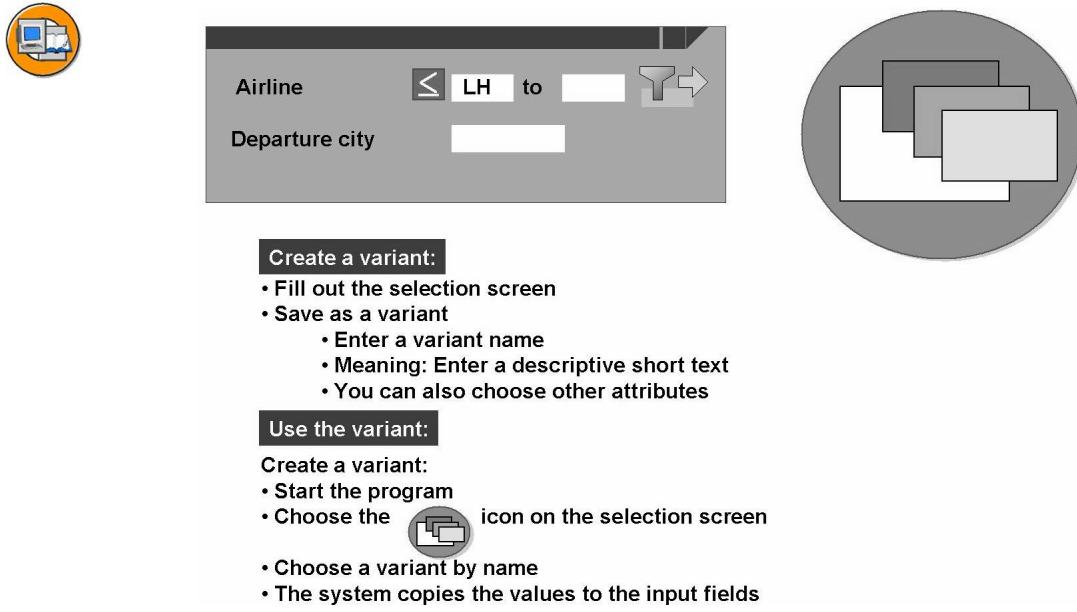


Figure 128: Selection Texts

On the selection screen, the field names appear as text next to the input fields. However, you can replace these with selection texts, which you can then translate into any further languages you require. Selection texts are displayed in the user's logon language.

If the input field is typed directly or indirectly with a data element, you can copy the field name from one of the texts stored in the *ABAP Dictionary*.

**Figure 129: Variants**

If you want to save the values (or some of the values) on a selection screen that you have filled out, you can do so by creating a **variant**. When you start the program again later, you can get these values from the variant and display them in the selection screen.

Users can define and save variants for any program that contains a selection screen. They do this by starting the program and then creating a variant by choosing *Variants → Save as Variant*.

Variants allow users to tailor selection screens to suit their needs. They can:

- Preassign values for input fields
- Hide input fields

A single variant can refer to more than one selection screen in the same program.

Variants are client-specific.

Every *Save as Variant* screen contains an information icon (with the tool tip "Information About Variant Attributes"). Choose this icon to display additional information.

Single Fields (PARAMETERS Statement)

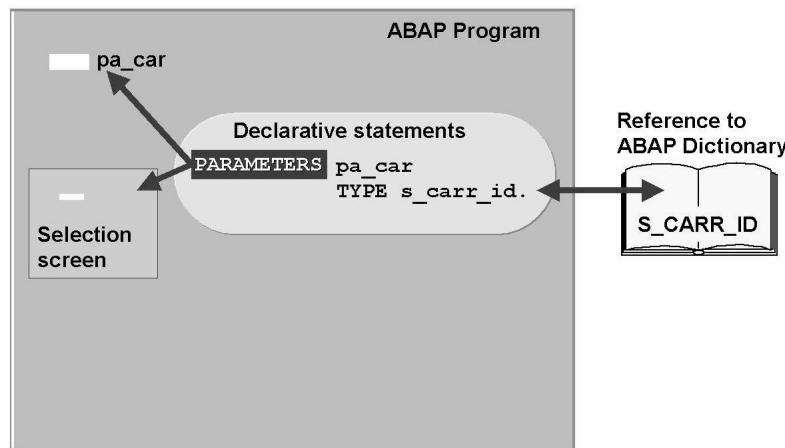


Figure 130: Effect of the PARAMETERS Statement

In an executable program, a single PARAMETERS statement is sufficient to generate a standard selection screen. Simply using the statements

PARAMETERS name TYPE data_type.

or

PARAMETERS name LIKE data_object.

an elementary input field is generated on the selection screen and an identically-typed data object NAME is declared.

If the user enters a value and chooses *Execute*, that value is placed in the internal data object NAME in the program. The system will only permit entries with the correct type.

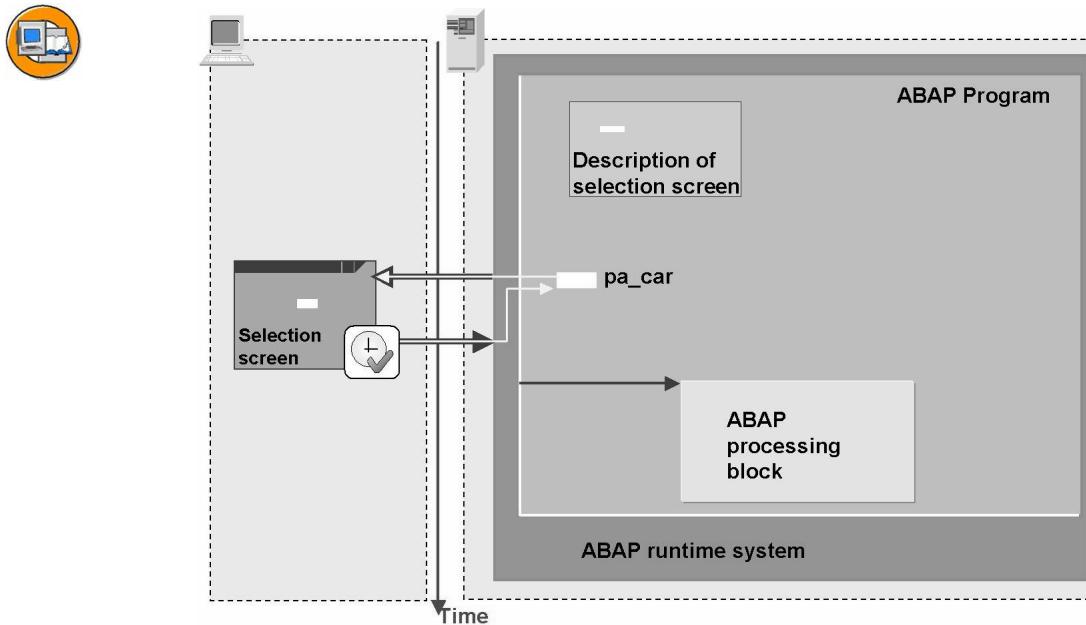


Figure 131: Runtime Behavior and Data Transport: Parameters

Once the LOAD-OF-PROGRAM event block has been processed, the selection screen is sent to the presentation server. The runtime system transports the data object values that are defined using PARAMETERS to the selection screen input fields of the same name.

The user can then change the values in the input fields. If the user then chooses the *Execute* function, the input field values are transported to the program data objects with the same name and can be evaluated in the ABAP processing blocks.

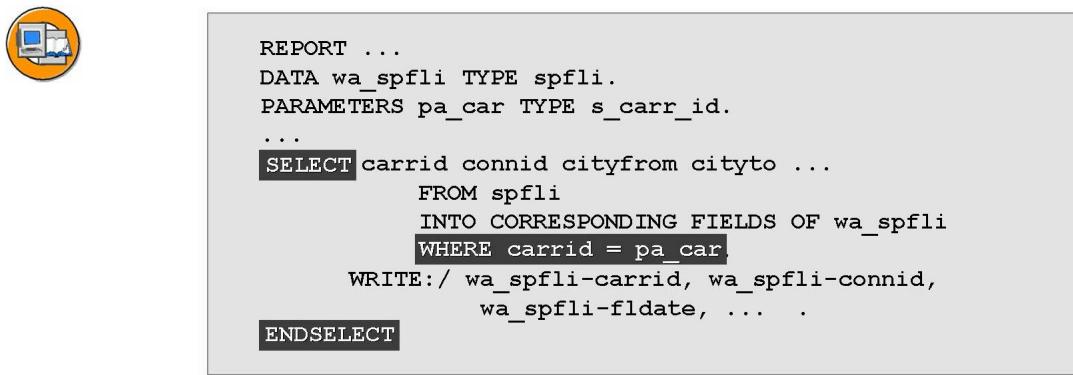


Figure 132: Using Parameters When Accessing the Database

If you have used the PARAMETERS statement to program an input field as a key field for a database table, you can use a WHERE clause at the SELECT statement to limit data selection to this value.

In the example above, the program reads only those data records from the database table SPFLI where the value in the key field CARRID value matches that in the data object **pa_car** at runtime.

Value Sets (the SELECT-OPTIONS Statement)

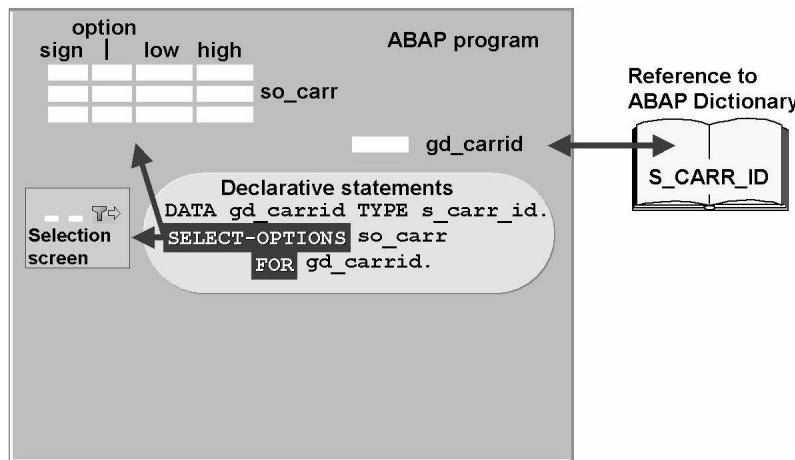


Figure 133: Effect of the SELECT-OPTIONS Statement

Selection options are defined when you use the

SELECT-OPTIONS name FOR data_object.

statement: Two input fields with the same type as the data object appear on the selection screen, allowing you to enter range limits. Additionally, a button appears that allows you to realize complex set limits.

The statement also declares an internal table name within the program, with the following four columns:

SIGN

Specifies whether the value or interval should be included (I) in or excluded (E) from the selection.

OPTION

Specifies the range, depending on the range limit operator chosen (such as BT or EQ). For a list of possible operators, refer to the keyword documentation for the SELECT-OPTIONS statement.

LOW

Contains the lower limit of a range, or a single value.

HIGH

Contains the upper limit of an interval.

The selection table NAME always refers to a data **object** that has already been defined. The data object is used as a target field during database selection, while the selection table is a set of possible values. For this reason a special version of the WHERE clause exists for database selection. It determines whether or not the database contains the corresponding field within the **value set**.

 **Hint:** Similarly, an inclusion test is also defined for the IF statement.

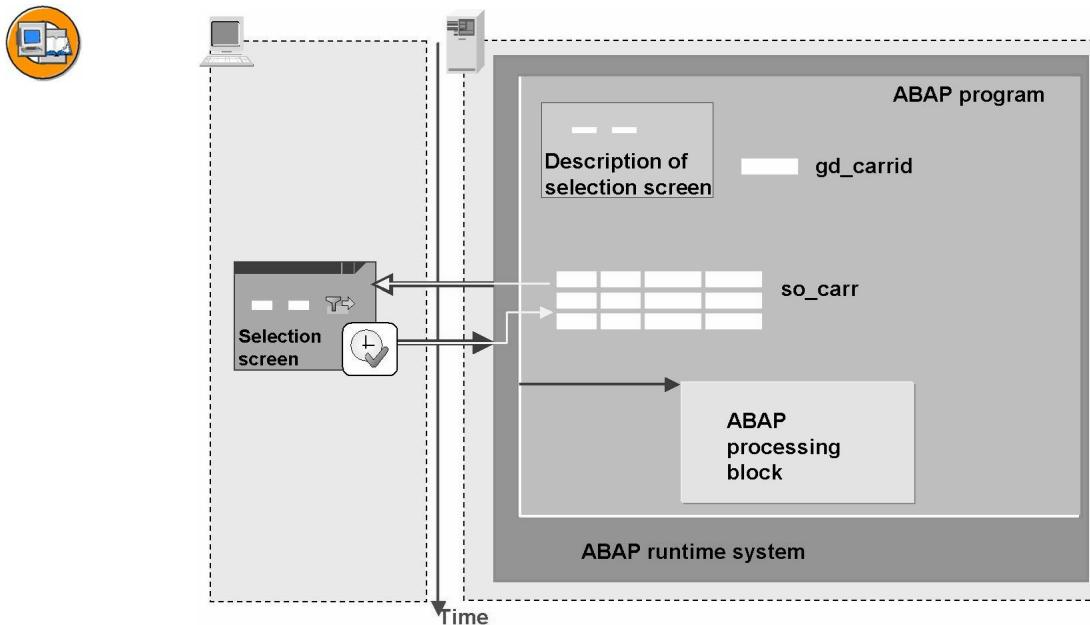


Figure 134: Runtime Behavior and Data Transport: Selection Options

If the user enters several values or intervals for a selection option and chooses *Execute*, the system places them in the internal table.



```
REPORT ...  
DATA wa_spfli TYPE spfli.  
SELECT-OPTIONS so_carr FOR wa_spfli-carrid.  
...  
SELECT carrid connid cityfrom cityto ...  
    FROM spfli  
    INTO CORRESPONDING FIELDS OF wa_spfli  
    WHERE carrid IN so_carr.  
    WRITE:/ wa_spfli-carrid, wa_spfli-connid,  
          wa_spfli-cityfrom, wa_spfli-cityto, ... .  
ENDSELECT.
```

Figure 135: Using Selection Options When Accessing the Database

The above example shows how you can restrict database selection to a certain range using a selection table. Conditions in an internal table that has been declared using SELECT-OPTIONS are interpreted as follows:

- If the internal table `selname` is empty, the condition field `IN selname` is always true.
- If the internal table contains only inclusive elementary conditions, such as `i_1, ... i_n`, the result is the composite condition (`i_1 OR ... OR i_n`)
- If the internal table contains only exclusive elementary conditions, such as `i_1, ... i_n`, the result is the composite condition (`NOT e_1 AND ... AND (NOT e_m)`).
- If the internal table contains only inclusive elementary conditions, such as `i_1, ... i_n`, the result is the composite condition (`i_1 OR ... OR i_n) AND (NOT e_1) AND ... AND (NOT e_m)`.

Selection Screen Events

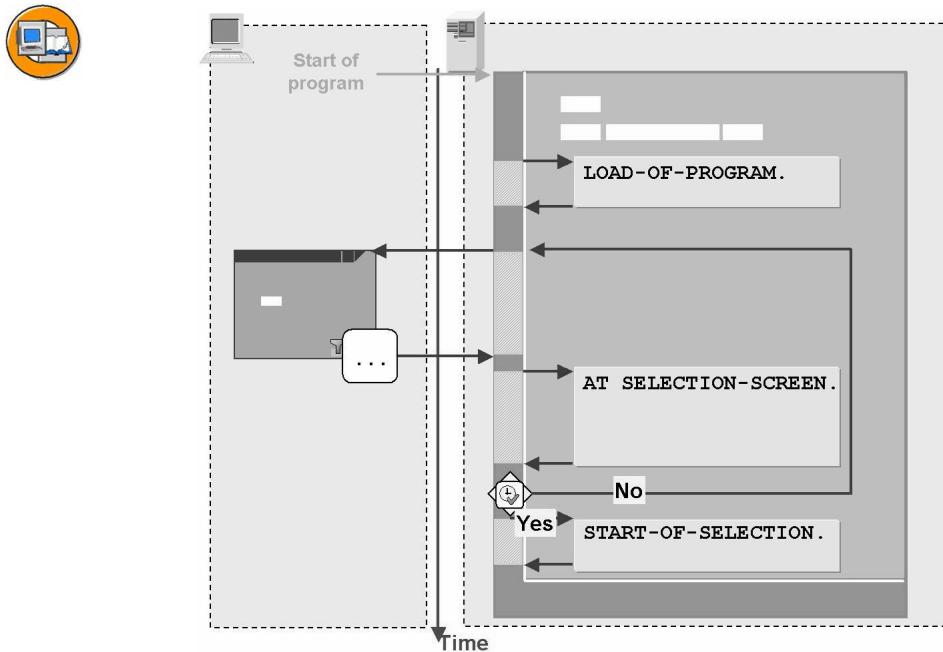


Figure 136: Selection Screen Event

In an executable program, the ABAP runtime system generates a standard selection screen as long as you have written at least one PARAMETERS or SELECT-OPTIONS statement. The event block AT SELECTION-SCREEN belongs to the selection screen. The selection screen is sent after the event block LOAD-OF-PROGRAM.

Each time the user chooses a pushbutton, function key, a menu function, or **Enter**, the system initially carries out a type check. If the entries do not have the correct type, the system displays an error message, and makes the fields ready for input again. When the entries have the correct type, the system processes the event block AT SELECTION-SCREEN.

Subsequent program flow depends on the user action:

- If the user chooses **F8** or **Execute**, the next event block is called: in this case START-OF-SELECTION.
- If the user chose any other function, the selection screen is displayed again.

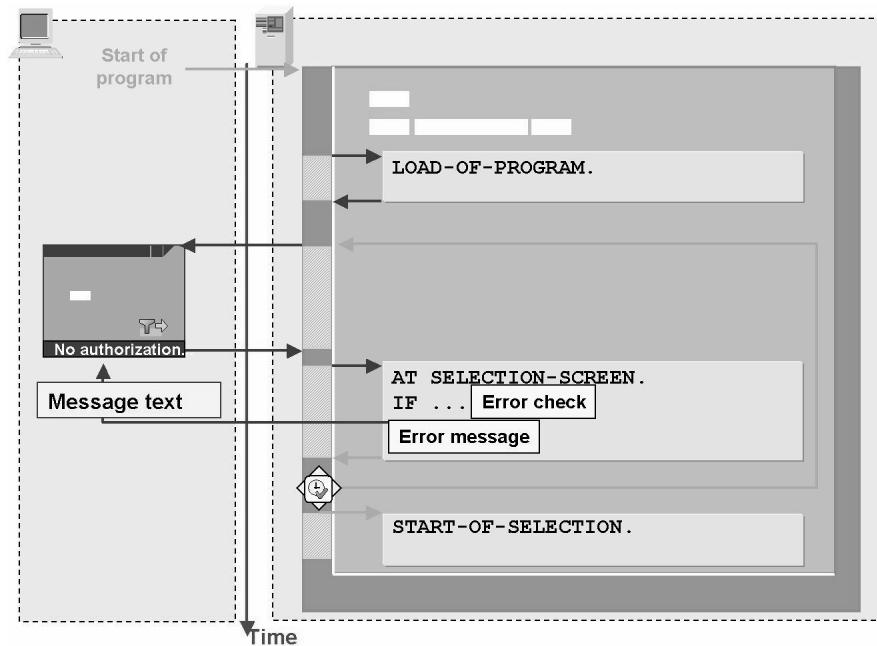


Figure 137: Error Dialogs in AT SELECTION-SCREEN

Use the event block **AT SELECTION-SCREEN** if you want to program additional input checks for a standard selection screen. This event block is processed after every user action (see above).

If an error dialog is triggered by the MESSAGE statement, the system resends the selection screen and automatically resets all input fields to ready for input. For more detailed information on the MESSAGE statement, refer to the keyword documentation.

Additional information can be found in the keyword documentation for AT SELECTION-SCREEN.



```
PARAMETERS pa_car TYPE s_carr_id.  
  
* event processed after leaving the selection screen  
AT SELECTION-SCREEN.  
  AUTHORITY-CHECK OBJECT 'S_CARRID'  
    ID 'CARRID' FIELD pa_car  
    ID 'ACTVT'   FIELD actvt_display.  
  IF sy-subrc NE 0.  
* show selection screen again with message in status bar  
  MESSAGE e045(bc400) WITH pa_car.  
ENDIF.
```

Figure 138: Syntax Example: Authorization Check for a Selection Screen

As an example of an additional input check with error dialog, an input field for the airline ID needs to be added to the program. An authorization check is to be carried out on the selection screen.

- If the user has authorization to display the specified airline, the program continues.
- If the user does not have display authorization, then the selection screen is displayed again and an error message appears in the status bar.

Exercise 13: Selection Screen

Exercise Objectives

After completing this exercise, you will be able to:

- Use the ABAP statement **SELECT-OPTIONS** to enter complex values on a standard selection screen
- Include complex values in a database selection
- Program an error message for a standard selection screen.

Business Example

Extend your program **ZBC400_##_DETAIL_LIST** or the corresponding model solution as follows:

Extend the standard selection screen to allow users to define a complex value set for limiting the connecting flight number. The value set should be taken into account when the user is selecting a flight.

Additionally, change your program so that the user can only progress from the selection screen if the authorization check for the selected airline is successful.

Task 1

Define and evaluate restriction of values entered on a selection screen.

1. Copy your program **ZBC400_##_DETAIL_LIST** or the corresponding template **SAPBC400UDS_DETAIL_LIST** to the new program **ZBC400_##_SEL_SCREEN**.
2. Extend your selection screen to allow the user to enter a complex value range for the flight number **CONNID**.



Hint: When you are assigning a type to the airline and flight number, make sure you refer to the same structure so that the search helps are context-specific.

3. Use the complex value set to restrict the amount of data selected from the database table **SFLIGHT** accordingly.

Task 2

Perform the authorization check **at the selection screen event**.

Continued on next page

1. Change your program so that the user **cannot** progress from the selection screen if the authorization check against the authorization object **S_CARRID** fails. If the authorization check does fail, display a suitable error message from message class **BC400** in the status bar, and allow the user to enter a different value on the selection screen.
2. Remove the error text for the failed authorization check from the list; it is no longer needed.

Solution 13: Selection Screen

Task 1

Define and evaluate restriction of values entered on a selection screen.

1. Copy your program **ZBC400_##_DETAILED_LIST** or the corresponding template **SAPBC400UDS_DETAILED_LIST** to the new program **ZBC400_##_SEL_SCREEN**.
 - a) Carry out this step as usual.
2. Extend your selection screen to allow the user to enter a complex value range for the flight number **CONNID**.



Hint: When you are assigning a type to the airline and flight number, make sure you refer to the same structure so that the search helps are context-specific.

- a) See source code excerpt in the model solution.
3. Use the complex value set to restrict the amount of data selected from the database table **SFLIGHT** accordingly.
 - a) See source code excerpt in the model solution.

Task 2

Perform the authorization check **at the selection screen event**.

1. Change your program so that the user **cannot** progress from the selection screen if the authorization check against the authorization object **S_CARRID** fails. If the authorization check does fail, display a suitable error message from message class **BC400** in the status bar, and allow the user to enter a different value on the selection screen.
 - a) See source code excerpt in the model solution.

Continued on next page

2. Remove the error text for the failed authorization check from the list; it is no longer needed.
 - a) See source code excerpt in the model solution.

Result

Source code extract SAPBC400UDS_SEL_SCREEN

```

REPORT  sapbc400uds_sel_screen.

CONSTANTS: actvt_display TYPE activ_auth VALUE '03'.

DATA: wa_flight TYPE sbc400focc,
      wa_sbook  TYPE sbook.

PARAMETERS: pa_car LIKE wa_flight-carrid.
* Data field for complex restrictions applied to connection id
SELECT-OPTIONS: so_con FOR wa_flight-connid.

* First event processed after leaving the selection screen
AT SELECTION-SCREEN.
  AUTHORITY-CHECK OBJECT 'S_CARRID'
    ID 'CARRID' FIELD pa_car
    ID 'ACTVT'   FIELD actvt_display.
  IF sy-subrc <> 0.
* Return to selection screen again and display message in status bar
  MESSAGE ID 'BC400' TYPE 'E' NUMBER '045' WITH pa_car.
ENDIF.

START-OF-SELECTION.
  SELECT carrid connid fldate seatsmax seatsocc FROM sflight
    INTO CORRESPONDING FIELDS OF wa_flight
    WHERE carrid = pa_car
    AND   connid IN so_con.
  wa_flight-percentage =
    100 * wa_flight-seatsocc / wa_flight-seatsmax.

  WRITE: / wa_flight-carrid,
          wa_flight-connid,
          wa_flight-fldate,
          wa_flight-seatsocc,

```

Continued on next page

```
        wa_flight-seatsmax,  
        wa_flight-percentage, '%'.  
HIDE: wa_flight-carrid, wa_flight-connid, wa_flight-fldate.  
ENDSELECT.  
  
CLEAR wa_flight.  
  
AT LINE-SELECTION.  
IF sy-lsind = 1.  
  WRITE: / wa_flight-carrid, wa_flight-connid, wa_flight-fldate.  
  ULINE.  
  SKIP.  
  SELECT bookid customid custtype class order_date  
        smoker cancelled loccuram loccurkey  
    FROM sbook INTO CORRESPONDING FIELDS OF wa_sbook  
  WHERE carrid = wa_flight-carrid  
    AND connid = wa_flight-connid  
    AND fldate = wa_flight-fldate.  
  WRITE: / wa_sbook-bookid color col_key,  
        wa_sbook-customid,  
        wa_sbook-custtype,  
        wa_sbook-class,  
        wa_sbook-order_date,  
        wa_sbook-smoker,  
        wa_sbook-cancelled,  
        wa_sbook-loccuram CURRENCY wa_sbook-loccurkey,  
        wa_sbook-loccurkey.  
  ENDSELECT.  
ENDIF.  
  
CLEAR: wa_flight, wa_sbook.
```



Lesson Summary

You should now be able to:

- List the properties and benefits of selection screens
- Implement the options for restricting selections using selection screens
- Implement input checks on the selection screen

Lesson: Screen

Lesson Overview

In this lesson, you will: Generate a screen with input and output fields, which transport data between the ABAP program and the screen using the TABLES statement; place buttons on the screen; and finally program functions for them in the ABAP program.



Lesson Objectives

After completing this lesson, you will be able to:

- List attributes and benefits of screens
- Write programs that display data on screens
- Write programs that allow users to change some data on screens
- Write programs that allow users to affect subsequent program flow using pushbuttons

Business Example

You need to develop a program that displays modifiable data on screens, and enables users to navigate dynamically using pushbuttons.

The Screen

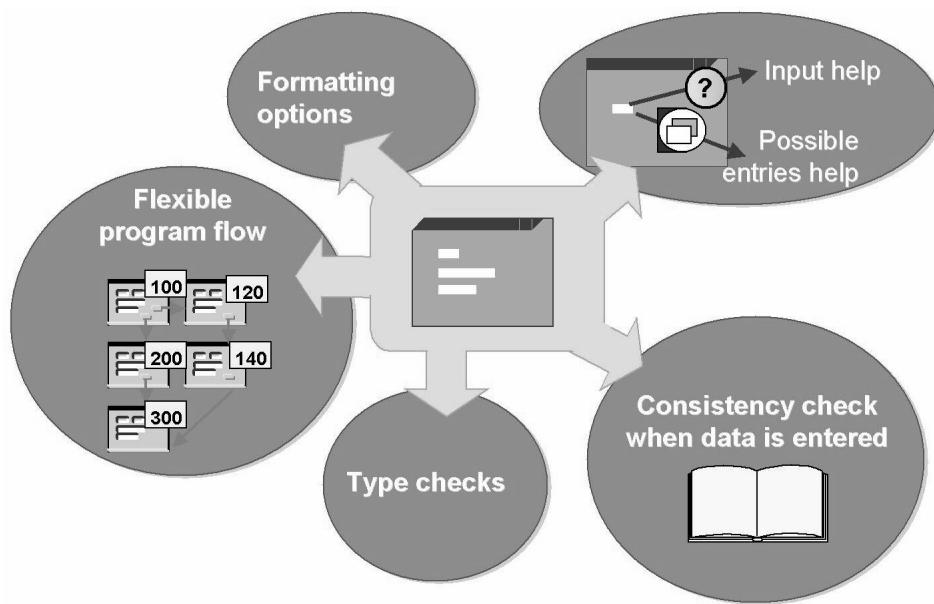


Figure 139: Features of Screens

Screens are made up of more than just a screen layout with input and output fields. They also have their own processing logic.

The fact that the *ABAP Dictionary* is integrated in the system means that automatic consistency checks for screen input fields are provided. These checks include type checks, foreign key checks, and fixed value checks. All of these checks are automatically supplied with information from the *ABAP Dictionary*.

Checks like the ones above can be complemented by other program-specific checks. There are techniques available for screens that allow you to control the order in which checks are performed and, if errors occur, to make the fields input-ready again, where appropriate.

You can also set up screen layout in a very flexible way. Input fields, output fields, radio buttons, check boxes, and even pushbuttons can be placed on screens. They allow users to affect the direction in which the program will proceed.

The same editing options are available for the screen as for list display and the selection screen: the fixed point numbers and the date are set according to the user defaults, the time according to the template hh:mm:ss, amounts according to the setting in a currency field, and lengths and weights according to the content of a unit of measurement field.

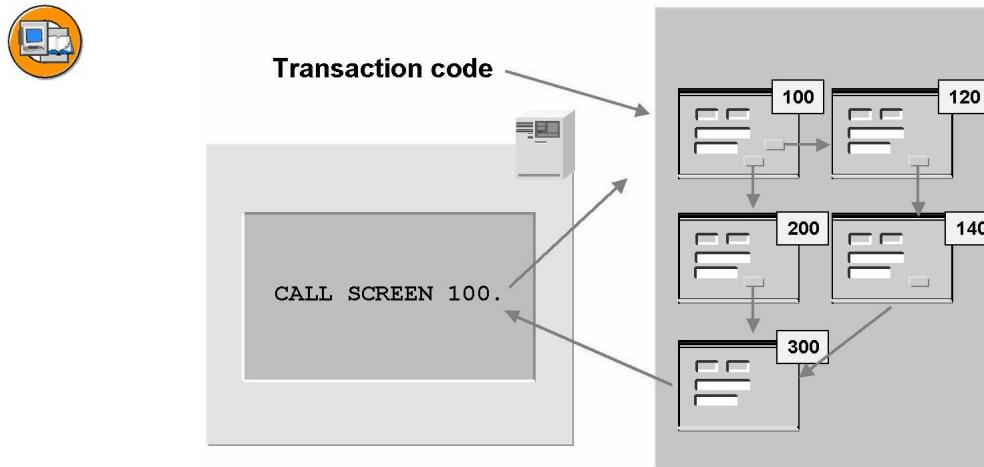


Figure 140: Options for Calling Screens

To start a screen sequence:

- Specify the first screen as the start screen of a transaction code of the type *dialog transaction*.
- Call the first screen from any one ABAP processing block in the program.

In this unit, we will concentrate on the last option mentioned.

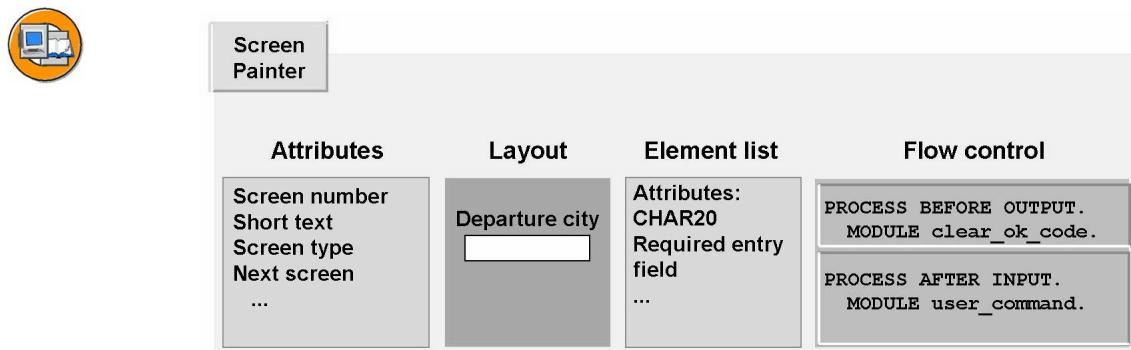


Figure 141: Components for Screen Processing

Each screen has the following information:

Attributes

Contain, for example, a four-digit number as the screen “name”, a short text, information on the screen type (for example, *Normal* for full screen size).

Layout

You can place elements on the screen. Elements that are displayed on the screen are called **screen elements**.

Element list

This contains the attributes of the screen elements, such as the position, size, and data type.

Flow control

This contains the processing logic that is to be processed through before the screen is sent to the presentation server (**PBO**) and the processing logic that is to be processed through after a user action has taken place (**PAI**).

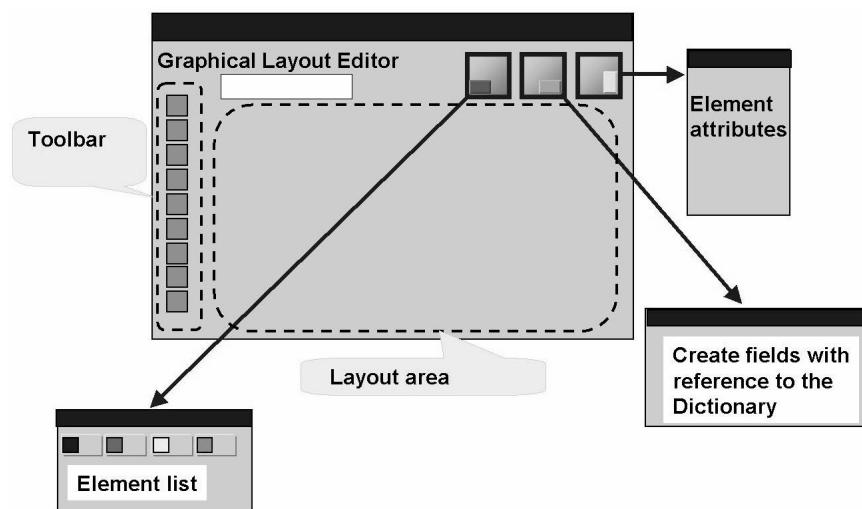


Figure 142: The Editing Window in the Graphical Layout Editor

From the *Graphical Layout Editor*, you can start other functions:

Element attributes

In this dialog box, all the attributes of a screen element are displayed. You can change some of the attributes here in the dialog. Example: you can specify whether or not an input/output field is to be input-ready.

Get from Dictionary/Get from program:

Allows you to generate fields that have either a global type or the same type as a data object in the program.

Element list

Shows all the elements displayed on the screen with their attributes.
You can also change attributes here.

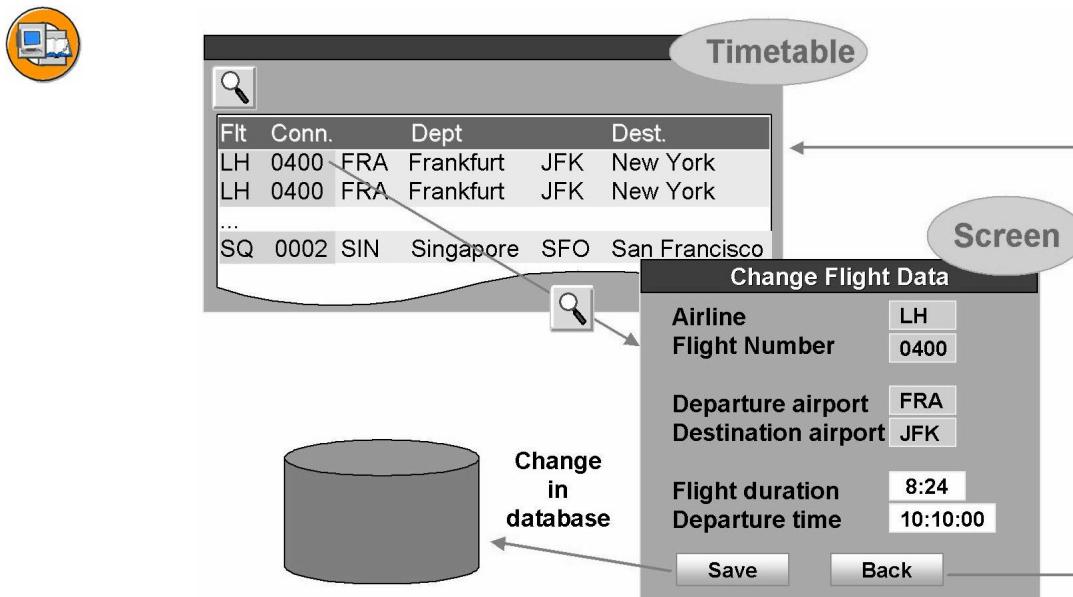


Figure 143: Application Example: Objective

In the following sections, we will develop a program step by step. This program will enable users to change the master data of the flight timetable.

If users double-click an entry in the basic list containing the flight timetable, the program displays another screen. This screen displays data from the line they selected, as well as additional information about the flight connection. Users can change the flight duration and the departure time.

Choosing *Back* takes them back to the basic list without changing any data.

Choosing *Save* changes the data in the database. There is a function module available for database changes. We will be implementing it in a later unit.

Creating a Screen

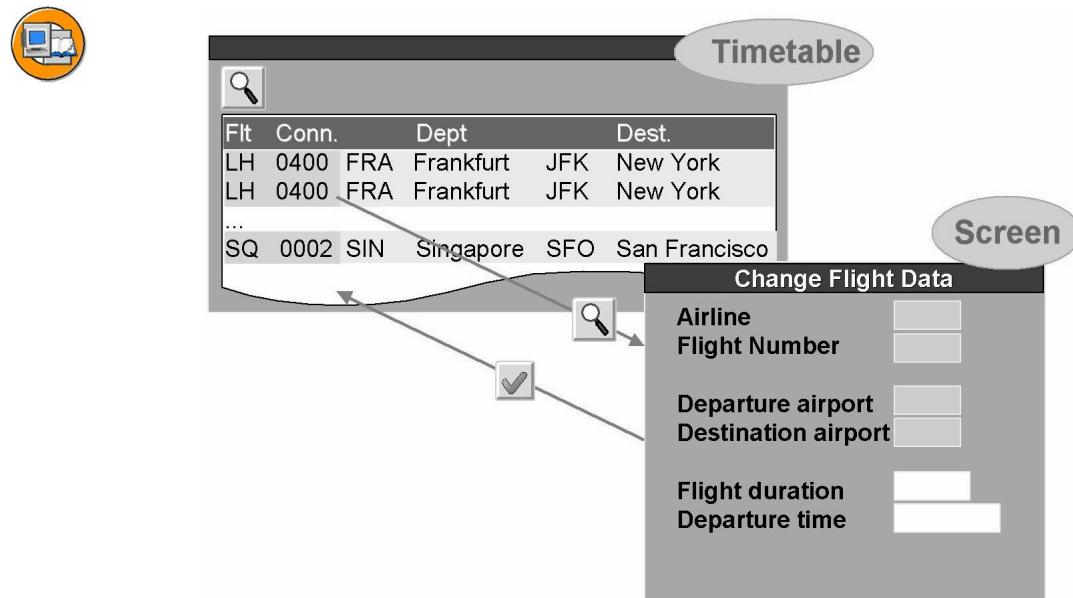


Figure 144: Example, Step 1: Creating a Screen

Your first step is to create a screen, specify its layout, and define its field attributes. The fields *Airline Carrier*, *Flight Number*, *Departure Airport*, and *Destination Airport* are to appear as output fields. The fields *Flight Duration* and *Departure Time* are to be made available as input-ready fields.

Users should be able to call the screen by double-clicking a line within the basic list and should be able to return to the basic list by choosing the appropriate function on the screen.

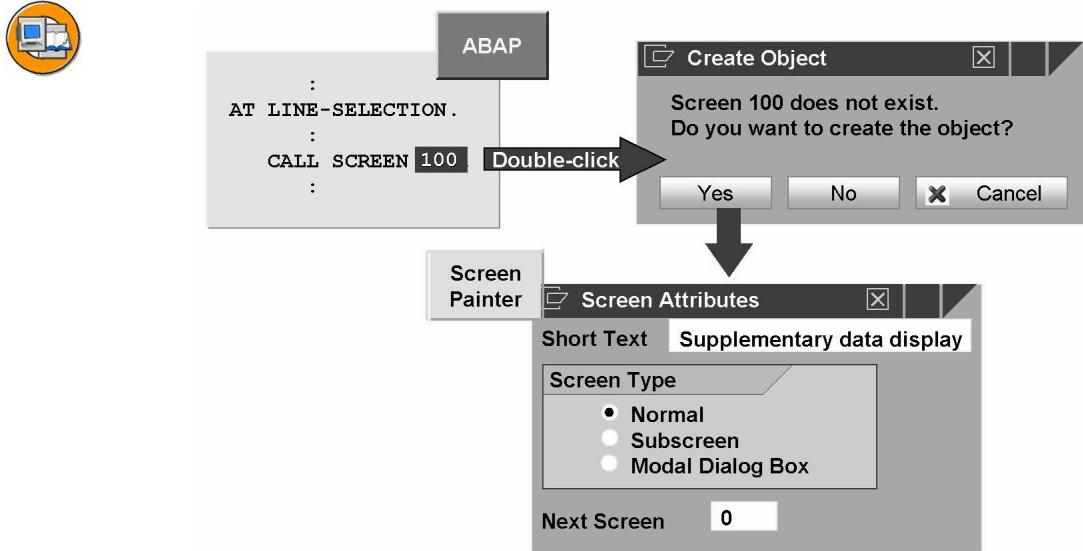


Figure 145: Creating a Screen: Screen Attributes

There are several ways to create a screen:

Forward navigation

In the *ABAP Editor*, double-click the screen number. The system opens the *Screen Painter* automatically.

Object Navigator

You can create a new program object *screen* for your program directly from the object list in the navigation area.

When you are creating a screen, the system will ask you to enter **screen attributes**. Enter a **short description** of the screen, select **screen type Normal**, and enter the number of the **subsequent screen**.

If you enter **0** for the subsequent screen, the system first processes your screen completely and then returns to processing at the point where the screen call is set.

Caution: Note that "0" in the entry field for the subsequent screen is not displayed because this is actually the initial value.

In this example, the screen the user creates is supposed to be called from within the basic list. This means that `CALL SCREEN...` must belong to the event block `AT LINE-SELECTION`.

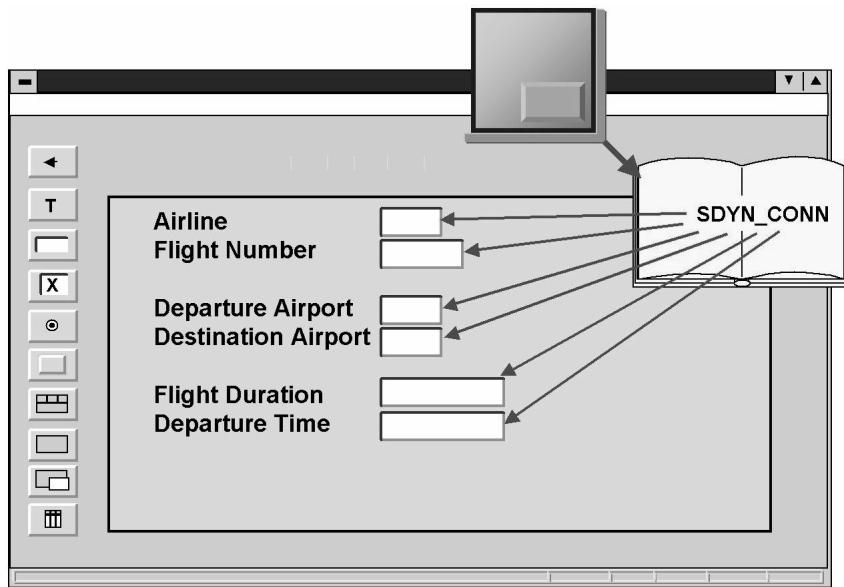


Figure 146: Input Fields with Reference to Fields of an *ABAP Dictionary* Structure

There are two ways of assigning field attributes to the required screen fields:

Get from Dictionary:

You can adopt types and field attributes from existing *ABAP Dictionary* structures. This makes all information about the object available to you, including semantic information about its data elements and foreign key dependencies. The field names are also supplied by the *ABAP Dictionary*.

Get from program:

You can adopt field attributes from data objects already defined within a program. For you to do this, however, a generated version of the program must already exist. The name of the data object is used as the field name.

The *Graphical Screen Painter* allows you to define different screen elements (such as input and output fields, keyword texts, borders) with ease. Choose the desired screen element and then place it on the screen using the mouse.

You can delete screen elements simply by selecting them with the mouse and then choosing *Delete*.

You can move screen elements by holding down the left mouse button and dragging them to a new position.

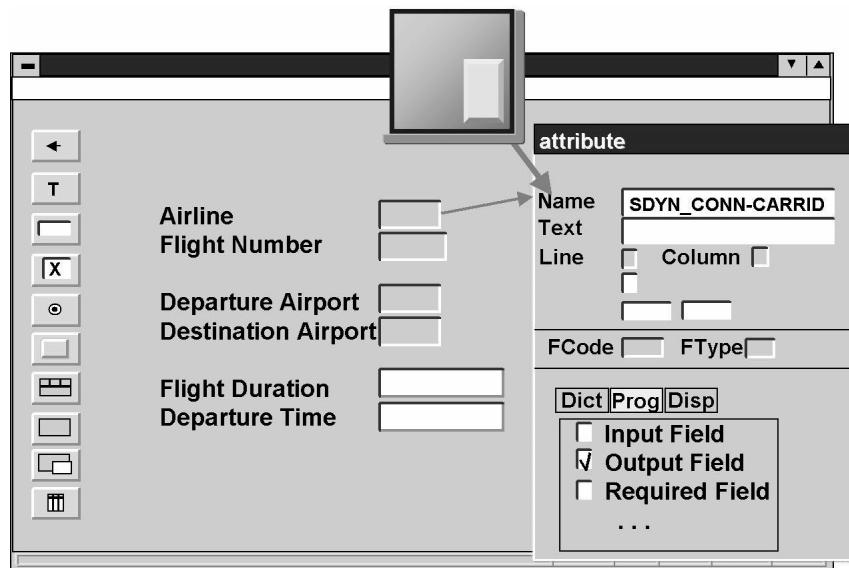


Figure 147: Changing the Element Attributes of a Field: The Attributes Dialog

To change the attributes of a screen field, select the field and choose *Attributes*.

To make input mandatory, assign the attribute *Obligatory* to a screen field. At runtime, the field will be marked accordingly if it is blank.

If not all required fields have been filled at runtime and a user action is performed, an error dialog is triggered by the ABAP runtime system, and all input fields are once again displayed ready for input.

Data Transports Between the ABAP Program and the Screen

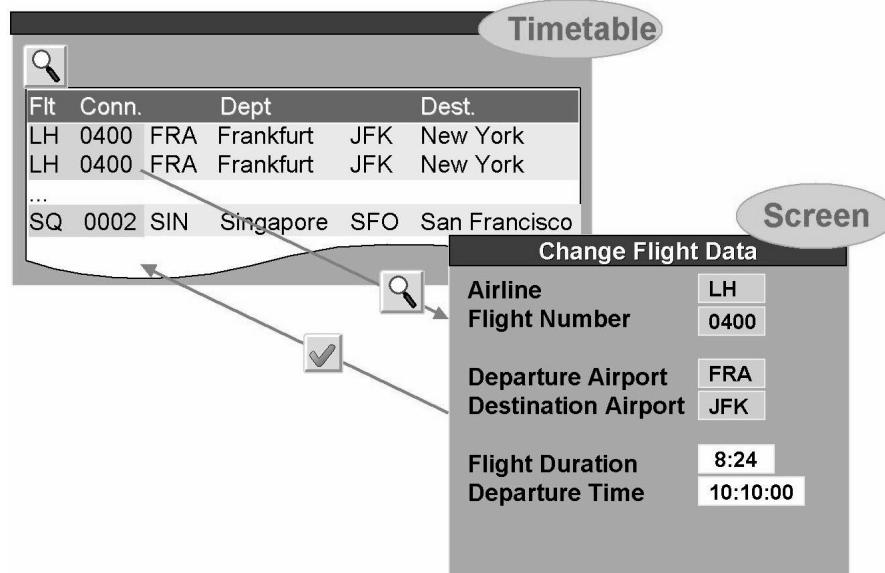


Figure 148: Example, Step 2: Displaying Data

In step two you will learn how to write the code that transports the data from the basic list to your screen.

For the user, the program works as follows:

- When the user double-clicks a line in the basic list, the system displays the screen. On this screen the most important items of information for the flight connection he or she has chosen are displayed. The flight duration and departure time are displayed in a field that is ready for input and hence can be changed.
- The user can return to the basic list in one of several ways.

With this in mind, this part of the unit will deal with:

- Prerequisites for automatic data transport between programs and screen fields
- Defining the screen interface and programming data transport to the data objects of the interface

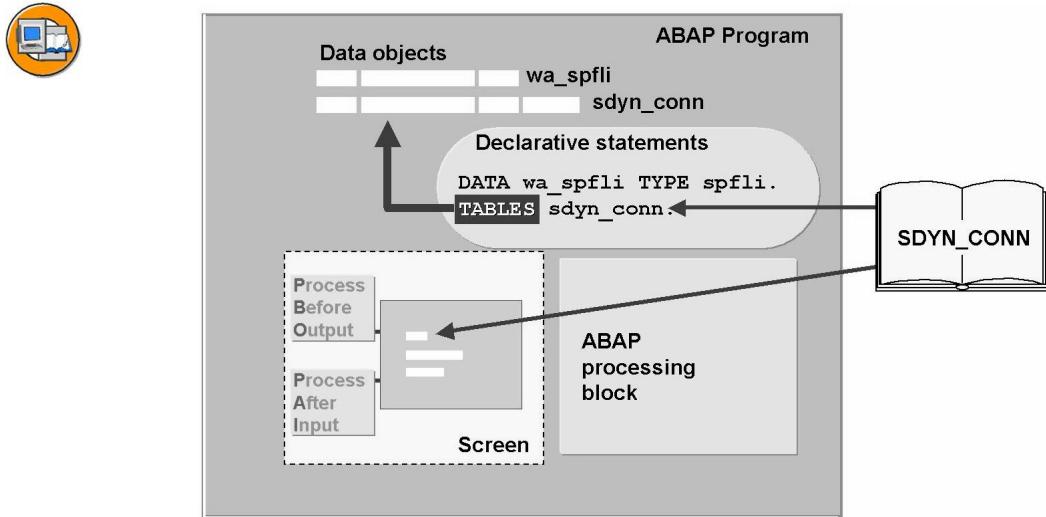


Figure 149: The Interface with the Screen

The **TABLES** statement defines an internal data object that provides an interface for the screen. TABLES always refers to an *ABAP Dictionary* object (for example, a transparent table or structure).

If the screen fields and the TABLES statement refer to the same Dictionary object, this data object's data is automatically transported to the screen fields every time the screen is transmitted. Any new entries or changes that the user makes on the screen are then transferred back into this data object.

As a rule, structures are created in the *ABAP Dictionary* that contain fields of the same type for the different tables. The flight data programs being created in this course use one structure for master data maintenance (**SDYN_CONN**) and another for bookings data (**SDYN_BOOK**). Using your own structures as interfaces makes the program easier to understand and helps you avoid errors.

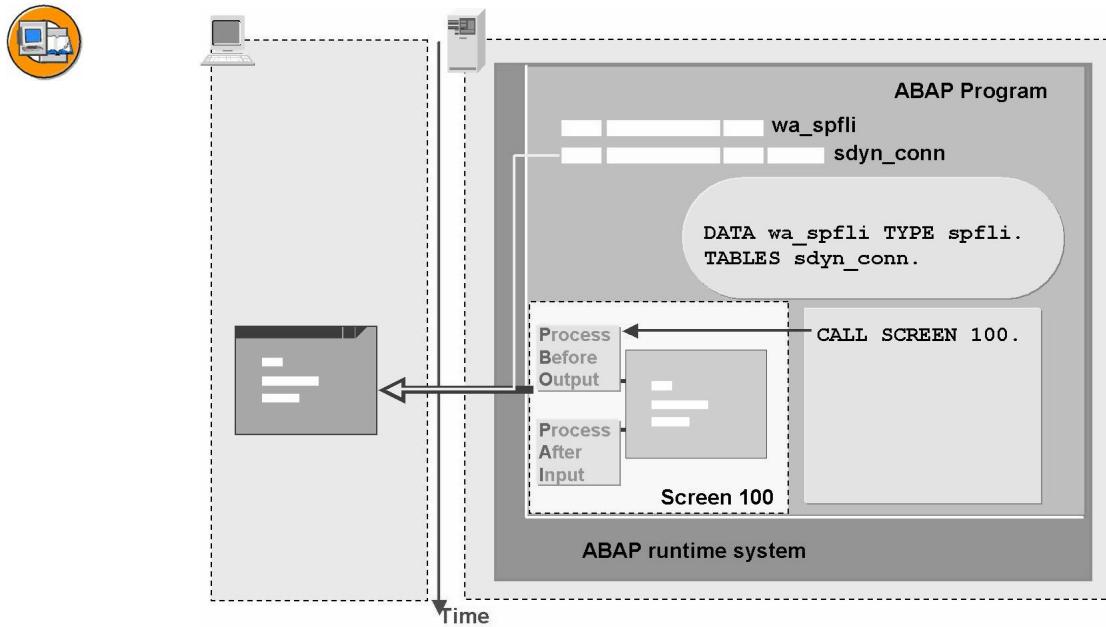


Figure 150: Data Transport from the Program to the Screen

The data transport between program data objects and the identically-named screen fields takes place automatically. Immediately before the screen is sent to the presentation server – that is, **after** the **PBO** event has been fully processed, the contents from fields with the same names are copied from the ABAP work area into the screen fields.

ABAP statements facilitate data transport between ABAP program data objects and the work area designated as the screen interface.

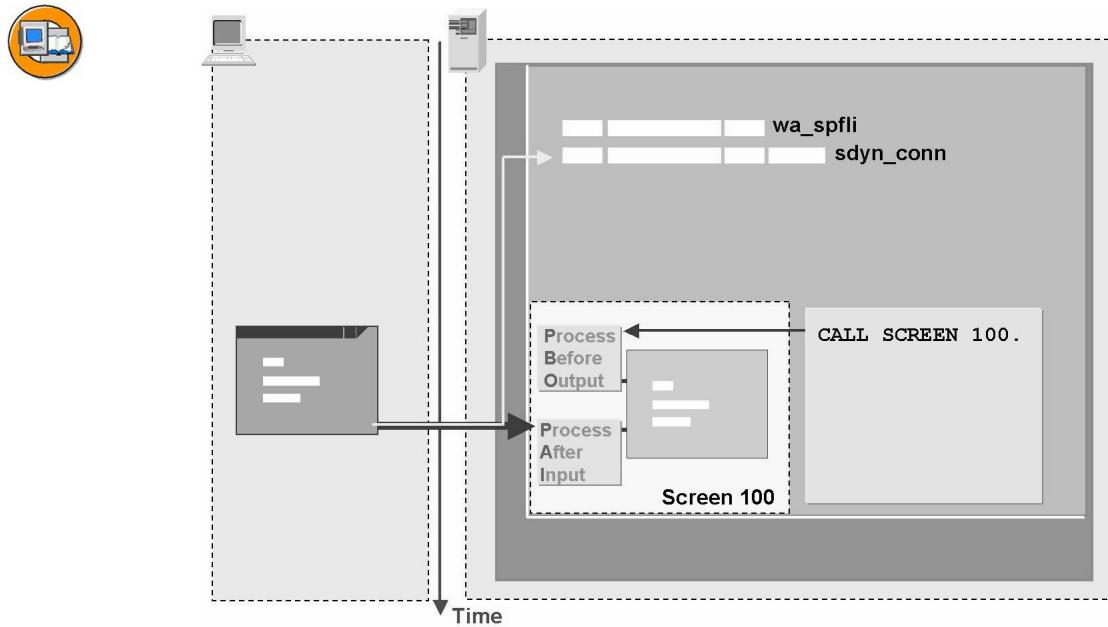


Figure 151: Data Transport from the Screen to the Program

The data transport between screen fields and program data objects with the same name takes place automatically. Immediately after a user performs an action on the screen – that is, **before** the PAI event is fully processed, the contents of the fields with the same name are copied from the screen fields into the ABAP work area.

ABAP statements facilitate data transport between program data objects and the work area designated as the screen interface.

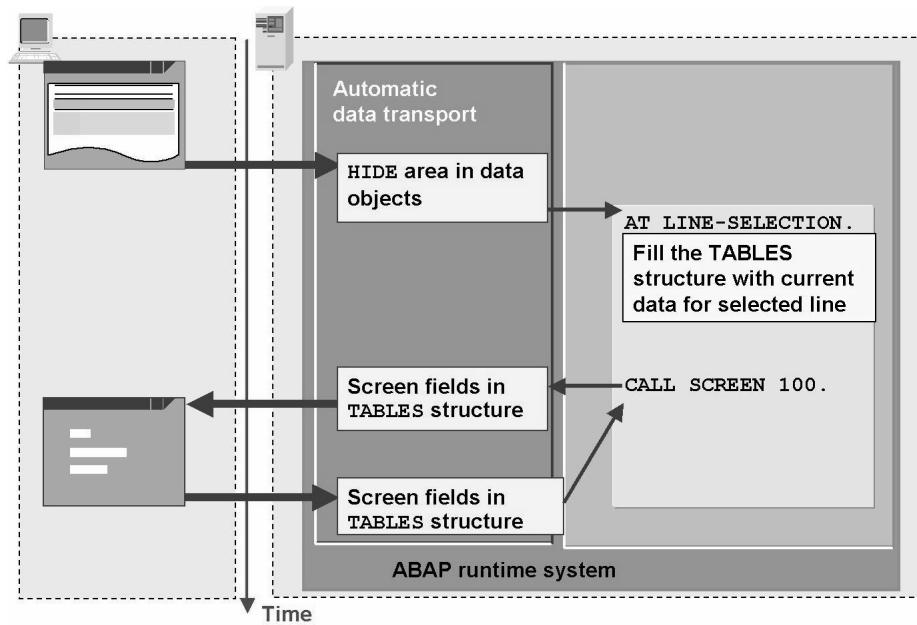


Figure 152: Transporting Data in the Example Program

The sample program should display screen data that matches the line selected in the basic list.

If data objects and their values were stored in the HIDE area when the basic list was created, the data belonging to the selected line will be put back into the corresponding data objects.

You must copy the data to be displayed on the screen to a TABLES structure. You can do this in any ABAP processing block that was processed before the screen is sent to the presentation server. There are various ways of doing this:

- You start by reading all the data to be displayed on the screen before retrieving the basic list from the database. You then place all the necessary data in the HIDE area. Then, at the AT LINE-SELECTION event, you only need to copy the data into the TABLES structure.

Disadvantage:

You have to read data from the database in which the user may not be interested. If the detailed data has changed between creating the basic list and displaying the screen, the system will display the wrong data.

- Before the screen is called using the SELECT SINGLE statement, you read the additional data for the selected key (HIDE area) from the database.



Structure:	wa_spfli		sdyn_conn	
	Basic list	HIDE area	Screen:	
Fields:			Output Field	Input Field
MANDT	key			
CARRID	key	✓	✓	
CONNID	key	✓	✓	
COUNTRYFR				
CITYFROM				
AIRPFROM	✓		✓	
COUNTRYTO				
CITYTO				
AIRPTO	✓		✓	
FLTIME	-		✓	✓
DEPTIME	-		✓	✓
ARRTIME				
DISTANCE				
DISTID				
FLTYPE				

Before the screen is called:
SELECT SINGLE * FROM spfli ...

Figure 153: Implementing the Sample Program: A Possible Approach

The last step in development should extend the program to enable the user to change data in the database. He or she should be able to change the fields **FLTIME** and **DEPTIME**. To enable the user to change data for several airlines, a basic list of the airlines for which the user is allowed to change data should be displayed. The user reaches the change screen by double-clicking. Once the changes have been made successfully, the user returns to the basic list. However, the basic list is not refreshed. Therefore, the fields that can be changed should not usually appear on the basic list.

To ensure that the database data displayed on the screen is up-to-date, the record is read again from the database at the beginning of AT LINE-SELECTION.

Advantages of this method:

- For the basic list, only those columns of the database table that are displayed in the list need to be read. If the list is large, this substantially reduces the runtime resources needed.
- The data that is displayed on the screen is always up-to-date, even if the data record selected has only just been changed.
- Changes made to the database using the screen do not lead to incorrect values in the basic list, as the modifiable fields are not contained in that list.



```

START-OF-SELECTION.

SELECT carrid connid airpfrom cityfrom airpto cityto
      INTO CORRESPONDING FIELDS OF wa_spfli
      FROM spfli.
      WRITE: / wa_spfli-carrid COLOR COL_KEY,
             wa_spfli-connid COLOR COL_KEY,
             ...
* buffering key fields
HIDE: wa_spfli-carrid, wa_spfli-connid.

ENDSELECT.

AT LINE-SELECTION.

SELECT SINGLE * FROM spfli
      INTO wa_spfli
      WHERE carrid = wa_spfli-carrid
            AND connid = wa_spfli-connid.
MOVE-CORRESPONDING wa_spfli TO sdyn_conn.
CALL SCREEN 100.

```

Figure 154: Syntax: Sample Program with Data Transport

To display data on the screen, the TABLES structure must be filled with current data before the screen is sent to the presentation server. The example above shows one way of doing this.

The HIDE statement is used to place key fields of database tables with reference to the list line in the HIDE area. Then the current data for the line selected is available in fields **WA_SPFLI-CARRID** and **WA_SPFLI-CONNID** at event AT LINE-SELECTION.

The data record is read from the database using SELECT SINGLE. This ensures that the data is up-to-date.

The corresponding fields of the **WA_SPFLI** work area are copied to the TABLES structure **SDYN_CONN** using MOVE-CORRESPONDING. The system transports the structure data to the screen fields automatically.

Alternatively, you could also place the data in the TABLES structure directly when accessing the database, using the INTO CORRESPONDING FIELDS OF statement.

Pushbuttons

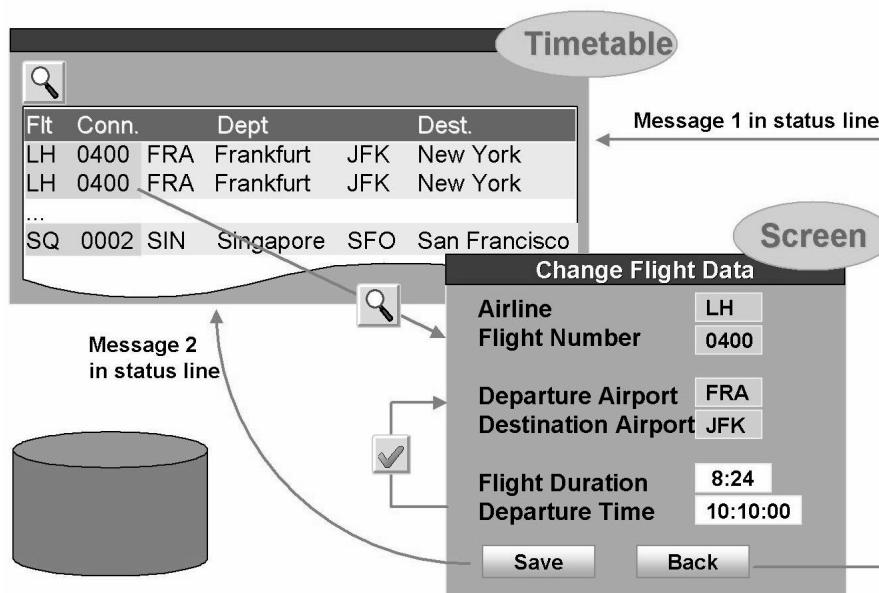


Figure 155: Example, Step 3: Assigning Actions to Buttons

In step three of our example, you will learn how to assign pushbutton functions. These functions allow different kinds of program logic to be processed according to user choice.

For the user, the program works as follows:

- By double-clicking a line, the user navigates to a screen. This screen displays additional information for the connection he or she has chosen. The flight duration and departure time can be changed.
- By choosing the *Back* pushbutton, the user returns to the basic list without writing any changes to the database table. The message "You left the screen without saving" is displayed in the status bar of the basic list.
- Choose *Save* to write all of your changes to the database. The changes are not implemented in the program until later. The pushbutton is already prepared in the following section. The user should return to the basic list after the pushbutton is chosen and a message should be displayed in the status bar.

- After each other user activity, the screen is displayed once again.

With this in mind, this part of the unit deals with:

- Flow logic in PBO and PAI event blocks
- Using PBO and PAI modules as ABAP processing blocks for screen programming
- Evaluating information on which pushbutton was selected at the time of PAI

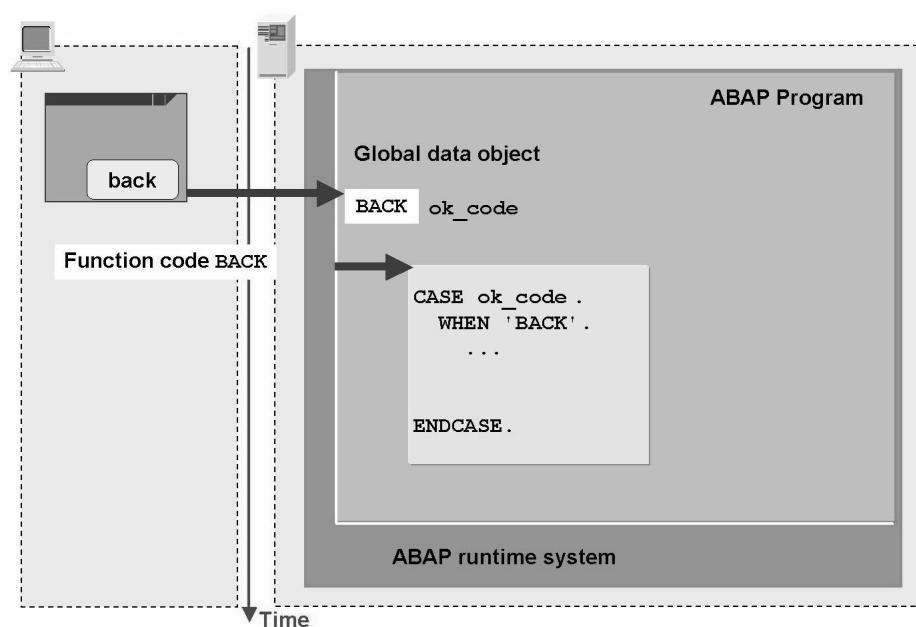


Figure 156: Runtime Behavior When the User Chooses a Pushbutton

If the user chooses a pushbutton, the runtime system copies the associated **function code** to a special screen field. This screen field is usually called **OK_CODE**.

The content of this special screen field is then automatically transported if there is an ABAP data object of the same name.

The content of this OK-CODE field can then be evaluated in an ABAP processing block.

This mechanism allows you to create a program flow that depends on the user's actions. The following sections deal with: How you declare this OK-CODE field, how you create pushbuttons and assign function codes to them, and how you evaluate user actions – for example, by setting the next screen.

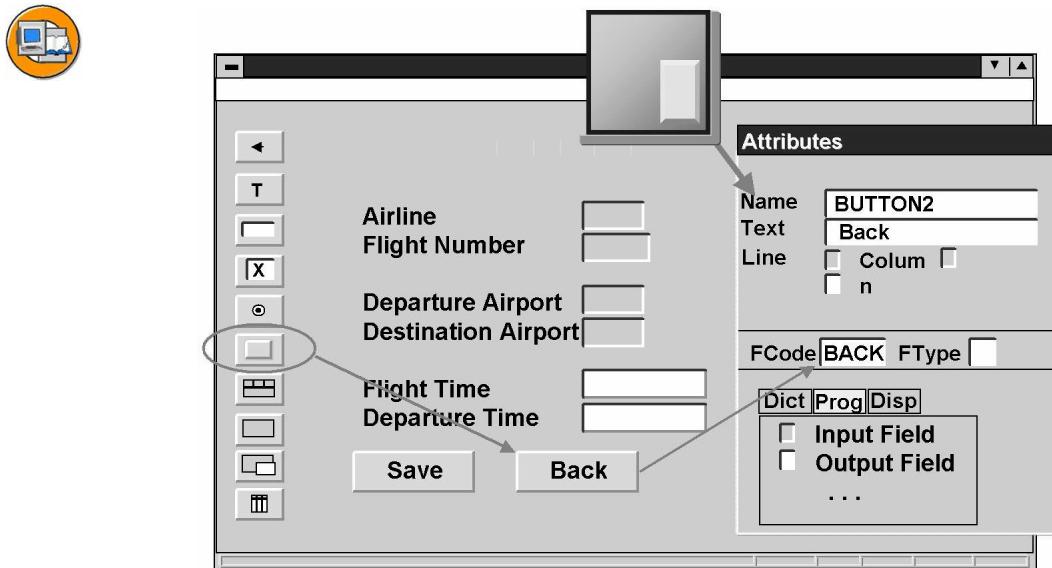


Figure 157: Defining Pushbuttons and Assigning Function Codes

To be able to tell the program which pushbutton the user chose, you must assign a function code to each pushbutton. You can do this either in the *Graphical Layout Editor* using the maintenance function for field attributes or from the element list.

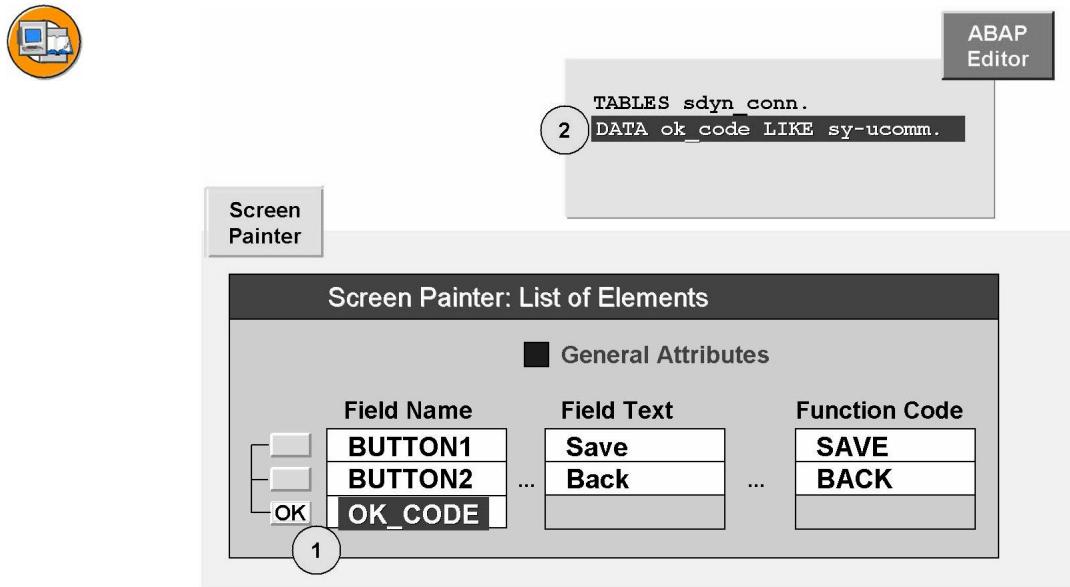


Figure 158: Making the Command Field Usable

The command field is a special screen field into which the associated function code is placed after every user action.

This field must be supplied with a name in the last line of the list of elements in **every screen**. Generally, the name **OK_CODE** is used.

In the declaration part of the program, define a corresponding data object with the same name. At runtime, whenever a user action is triggered, the function code of the pushbutton chosen is placed in the data object. For best results, assign the system field **SY-UCOMM** as the type for this field.

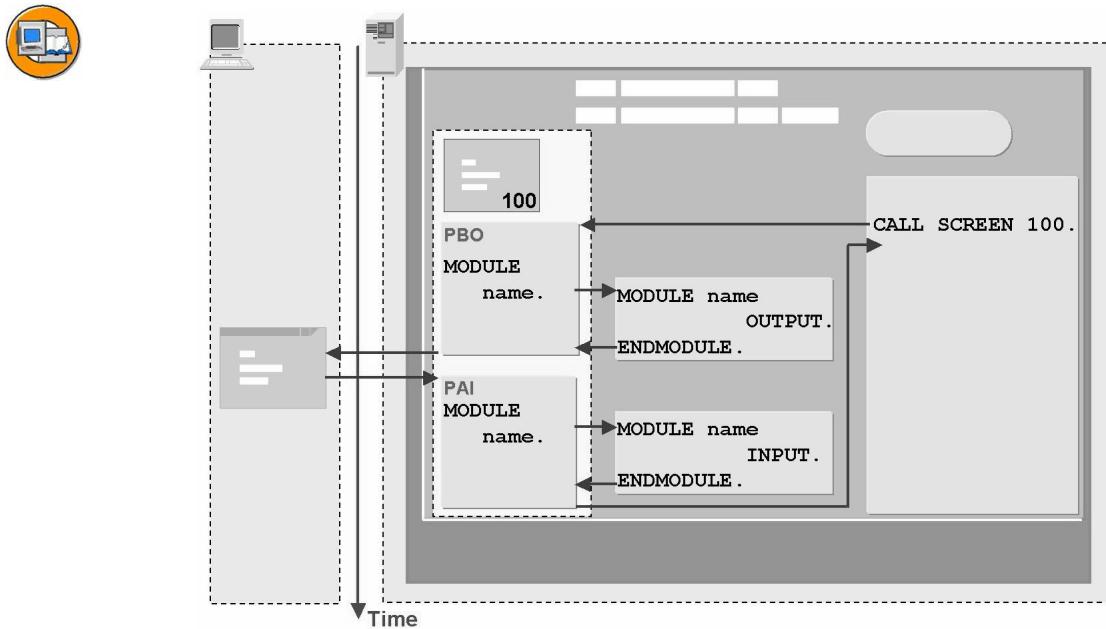


Figure 159: Calling Modules

The ABAP statement CALL SCREEN interrupts the processing of the processing block and calls a screen.

Each screen has two event blocks:

- **PROCESS BEFORE OUTPUT (PBO)** is processed immediately before a screen is displayed. At this time, modules are called that take care of tasks such as presetting values in screen fields.
- **PROCESS AFTER INPUT (PAI)** is processed after a user action. All program logic that is influenced by user action must be written in the PAI event.

Caution: The code for the PBO and PAI events is written using the *Screen Painter*, **not** the *ABAP Editor*. These two event blocks make up the **flow logic** of a screen. There is a special small set of commands available (**not** ABAP statements) for programming flow logic. The most important statement in this set is the **MODULE module_name** statement. This calls the ABAP processing block `module_name`. Within this block you can code in ABAP as usual.

Modules are ABAP processing blocks with no interface that can only be called from within the flow logic. Modules are enclosed within the ABAP statements MODULE and ENDMODULE.

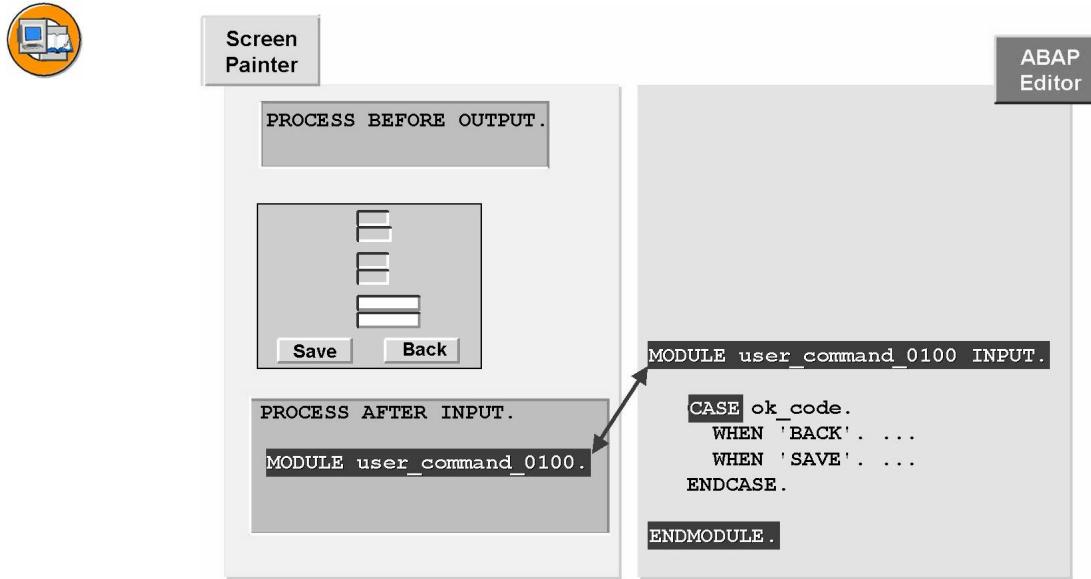


Figure 160: Evaluating the Function Code at the PAI Event

You can evaluate user actions in a PAI module. This PAI module is usually given the name **USER_COMMAND_nnnn**. (nnnn stands for the screen number.) In this module, you evaluate the function code in the field **OK_CODE**.



Caution: For reasons of compatibility, modules have no interface and no local variables. You can access all the ABAP program's **global data** in modules.

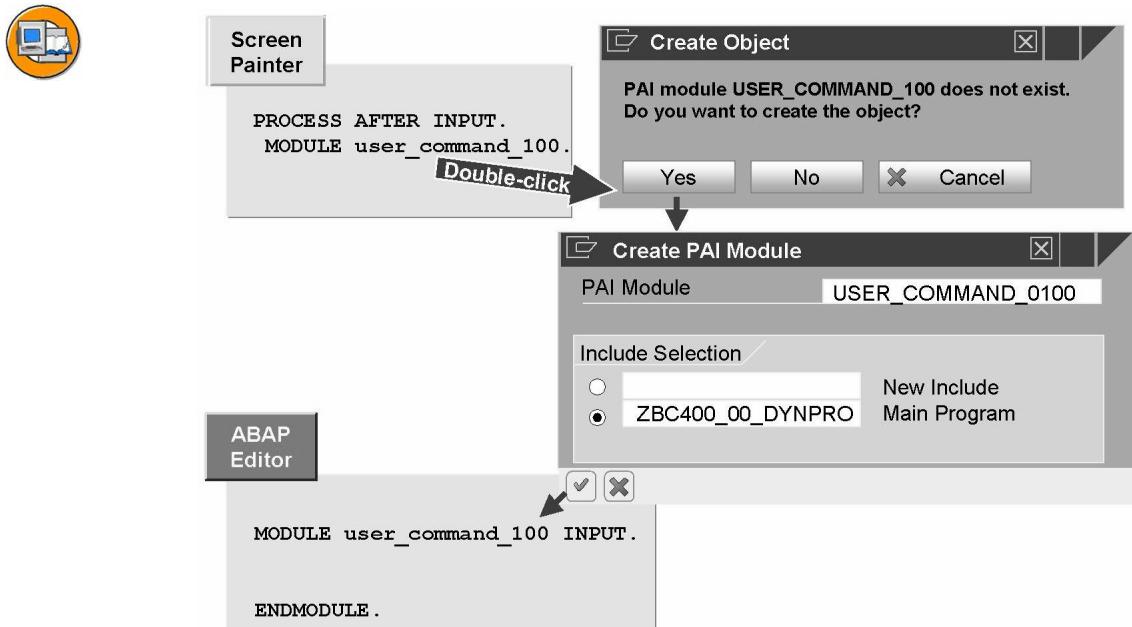


Figure 161: Creating Modules Using Forward Navigation

Within a screen's flow control (PBO and PAI events), you usually implement MODULE calls. The modules themselves are created using ABAP.

There are two ways to create a module:

Forward navigation

In the *Screen Painter*, double-click the module name to create the module.

Navigation area

If you want to create a module from within the object list of the program, choose a new program object, *PBO module* or *PAI module*.

A module can be called from more than one screen (reusability).

Make sure that modules called at PBO events are defined using the **MODULE ... OUTPUT** statement. Modules created in ABAP using the **MODULE ... INPUT** statement can only be called at the PAI event.

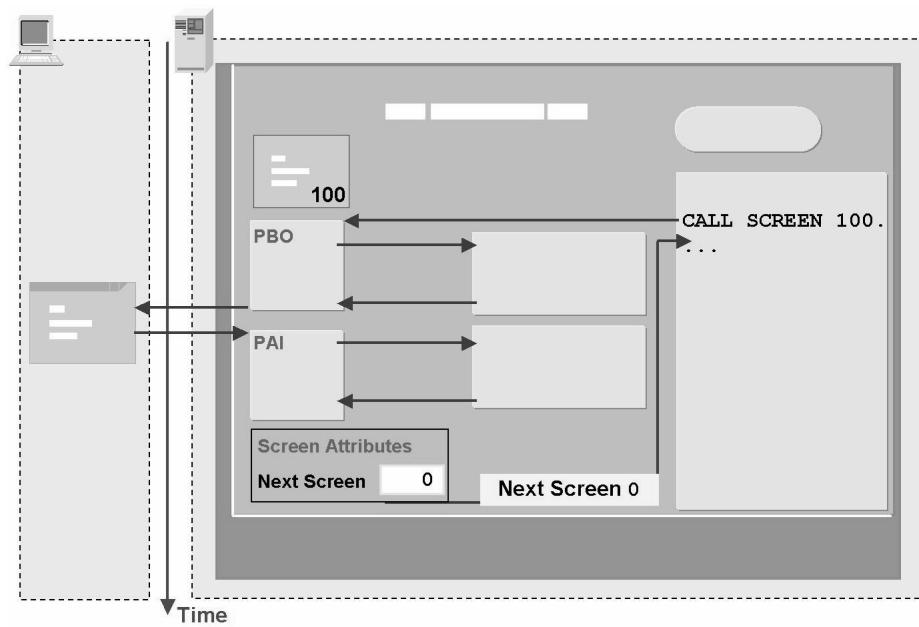


Figure 162: Next Screen: Starting-Point

If you enter 0 for the subsequent screen, the system first processes your screen completely and then returns to processing at the point where the screen call is set.

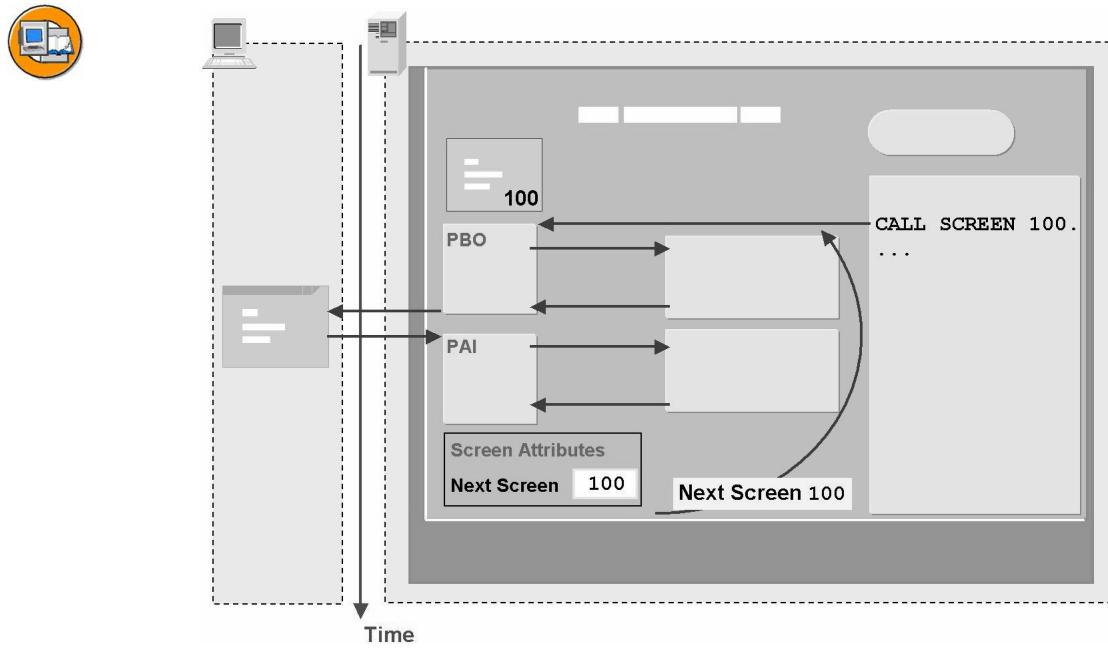


Figure 163: Setting the Next Screen Static

If you set the next screen for screen 100 to 100, the system processes the screen again after it has finished processing the PAI module.

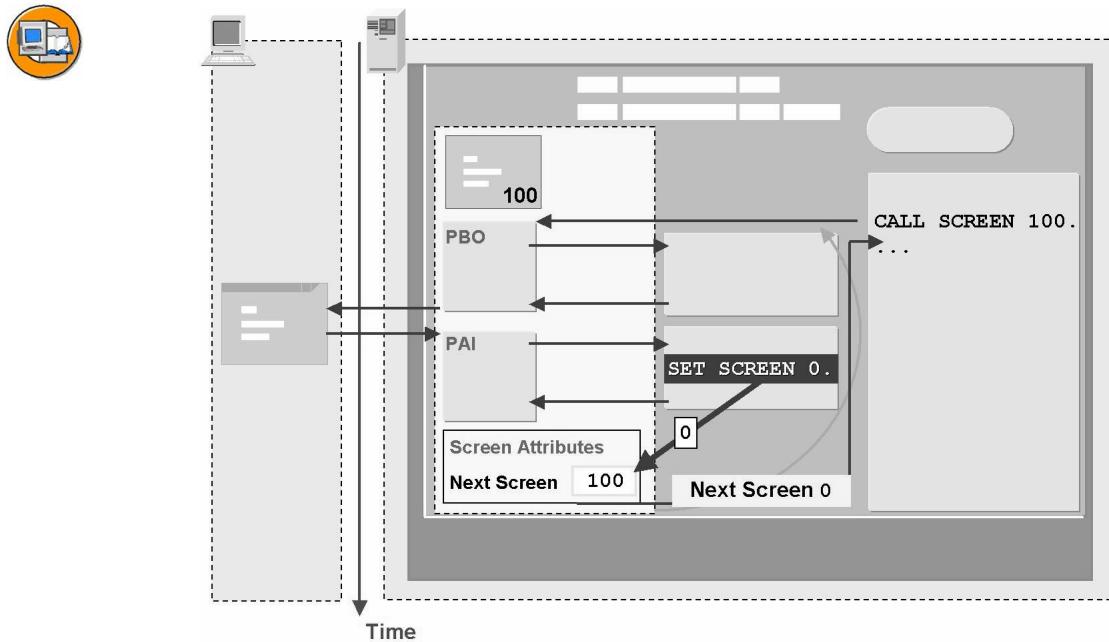


Figure 164: Setting the Next Screen Dynamically

Using the **ABAP statement SET SCREEN in a PAI module**, you can have the next screen set **dynamically**. The static entry is then suppressed.

The number of the current screen is entered by the system as the standard entry for the next screen. This ensures that the *Enter* key leads to a field check only, and then afterwards the screen is transmitted once again, possibly with an appropriate error message. For the user to be able to leave the screen, an appropriate pushbutton must be defined that is handled in the PAI module.

Note that, if the system processes the same screen again, it also runs through all the PBO modules again. If you decide to fill the TABLES structure in a PBO module, you must make sure that data changes made by the user are not overwritten on the screen if the module gets called twice.



```
DATA ok_code LIKE sy-ucomm.  
  
.  
  
MODULE user_command_0100 INPUT.  
CASE ok_code.  
    WHEN 'BACK'.  
        SET SCREEN 0.  
        MESSAGE s057(BC400).  
    WHEN 'SAVE'.  
        * For teaching purposes, we have omitted calling a  
        * function module that saves the changes  
        SET SCREEN 0.  
        MESSAGE s057(BC400).  
ENDCASE.  
ENDMODULE.
```

Figure 165: Syntax Example: Evaluating the Function Codes

For this example, two pushbuttons are to be handled:

- Choosing **Back (function code BACK)** dynamically sets the value of *Next Screen* to zero. This sends the user back to the point where the screen was called. In our case, this means the user returns to the basic list. Message 057 appears in the status bar of the next screen.
- If the user chooses **Save (function code SAVE)**, the program behaves in the same way as *Back*. The user returns to the basic list; however, there is a new message in the status bar. We will extend the program to include changing data records in a later lesson.

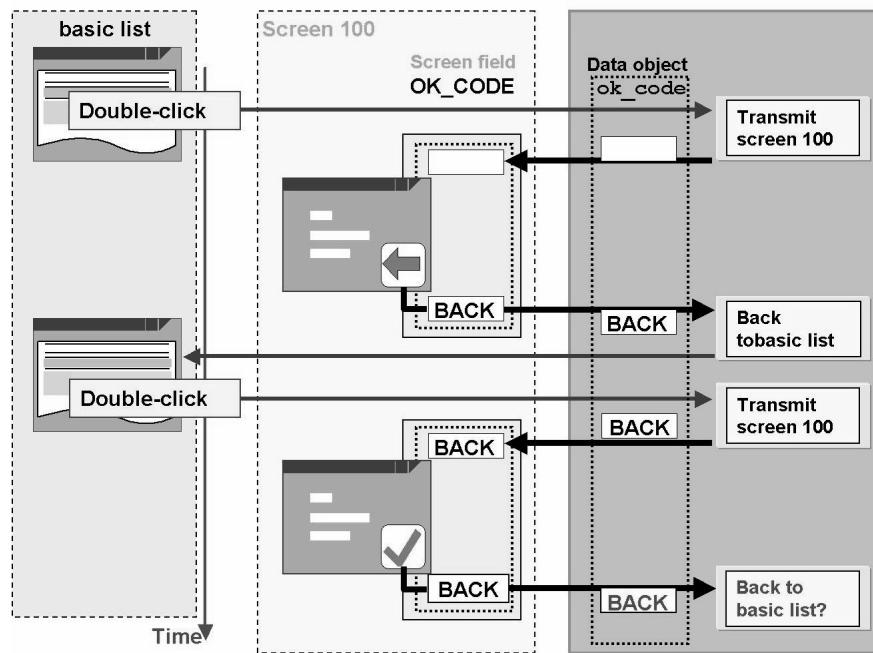


Figure 166: Runtime Behavior in Special Cases: No Function Code Is Assigned to the ENTER Key

In the section below, we need to analyze a very particular scenario, to point out the need for a further program step.

The user starts the program and double-clicks to display detailed information on the screen. There the user sees that all the data is correct, so he or she returns to the basic list. In this way, the function code BACK, which is assigned to the standard pushbutton *Back*, is placed into the command field. Afterwards it is transported automatically to the global data object **OK_CODE** of the ABAP program. This function code is then evaluated in a PAI module. The next screen is set to zero.

In our example program, the AT LINE-SELECTION event block is processed further. Since there is no WRITE statement, the system displays the basic list again.

The user then displays details for another record by double-clicking it. Through automatic field transport, the content of the global data object **OK_CODE** is transported into the screen field with the same name, and the screen is displayed.

If the user now chooses *Enter*, the screen is displayed once again, provided no function code has been assigned to the **Enter** key. (Note: This is the norm.) Therefore, the command field is **not** overwritten. The function code BACK remains in the command field and is copied to the global data object **OK_CODE** at the beginning of the PAI event.

This function code is then evaluated in a PAI module. Consequently, as described above, the system goes back to the basic list, instead of re-displaying the screen.

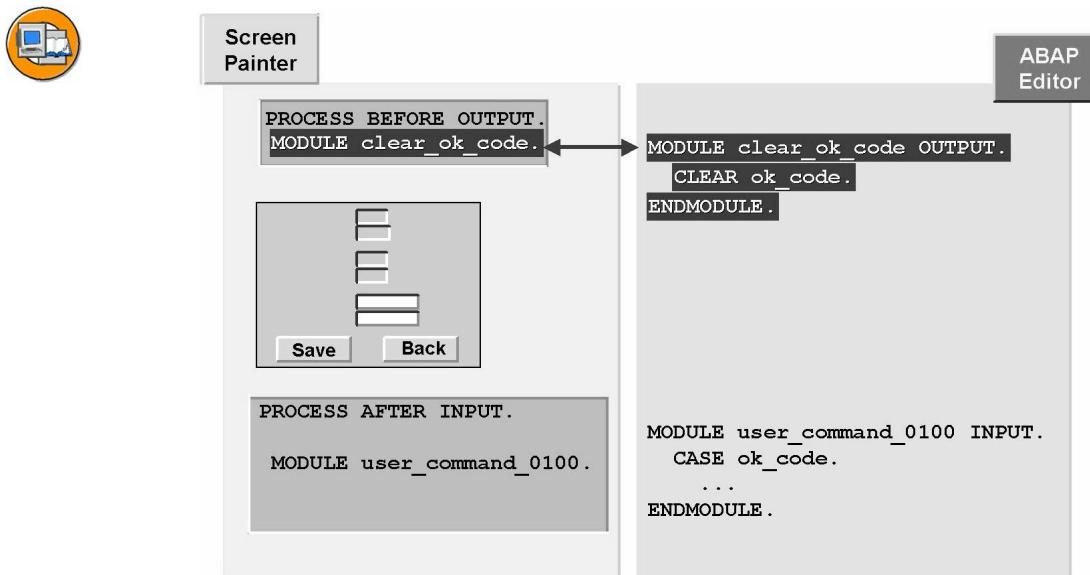


Figure 167: Possible Solution: Deleting the Command Field at the PBO Event

Errors can occur if the command field is not initialized, since not every pushbutton is required to have a function code assigned to it. There are two common techniques for avoiding this problem:

- You initialize the OK-CODE field in a **PBO** module. Then it is always set to the initial value at PAI, unless the user has carried out a user action to which a function code is assigned. If so, the OK-CODE field contains the current function code.
- You use an **auxiliary field** and copy the content of the OK-CODE field into the auxiliary field in a **PAI** module, and then initialize the OK-CODE field. In this case, the auxiliary field must be queried in the PAI module for the function code evaluation.

Exercise 14: Creating a Screen

Exercise Objectives

After completing this exercise, you will be able to:

- Create screens
- Call existing screens from the program

Business Example

The program **SAPBC400UDT_DYNPRO_A** displays all **bookings** made by a specific agency as a list. As a first step, change the program as follows: Call a screen whenever the user double-clicks a line in the basic list. Make sure this screen contains input-ready fields for some of the booking data, which will not be displayed in the list.

In addition, make sure there are output-ready fields for **booking information**, which has already been displayed in the list. Any user action should result in the basic list being displayed again

Task 1

Call and create a screen

1. Copy the template **SAPBC400UDT_DYNPRO_A** to your new program **ZBC400_##_DYNPRO**.
2. Get to know the program. Test the program using the travel agency number **1##** (where **##** is your group number).
3. Change the program so that when the user selects a line on the basic list (by double-clicking or using **F2**), it calls a screen. Create this screen (screen number 100) using forward navigation.
4. When specifying the attributes, assign screen number **0** as the number of the next screen, so that after any user action on screen 100, the user returns to the basic list.

Task 2

Place elements on the screen

1. Create input/output fields on the screen. When you are assigning field types, use the *ABAP Dictionary* structure **SDYN_BOOK**.

Continued on next page

The booking table key fields **CARRID**, **CONNID**, **FLDATE**, and **BOOKID** should be copied **with their field labels**.

The customer name **NAME** should be copied **without a field label** and displayed next to the customer number.

The booking table key fields **CUSTOMID**, **CUSTTYPE**, **SMOKER**, **CLASS**, **LOCCURAM**, and **LOCCURKEY** should be copied **with their field labels**.

2. Maintain the screen field attributes:

The fields **CARRID**, **CONNID**, **FLDATE**, **BOOKID** and **CUSTOMID** should appear as output fields (attribute *Output Field*).

The customer name **NAME** should be displayed next to the customer number without text (*Output Only* attribute).

The fields **CARRID**, **CONNID**, **FLDATE**, **BOOKID** and **CUSTOMID** should appear as output fields (attribute *Output Field*).

Solution 14: Creating a Screen

Task 1

Call and create a screen

1. Copy the template **SAPBC400UDT_DYNPRO_A** to your new program **ZBC400_##_DYNPRO**.
 - a) Carry out this step as usual.
2. Get to know the program. Test the program using the travel agency number **1##** (where **##** is your group number).
 - a) Carry out this step as usual.
3. Change the program so that when the user selects a line on the basic list (by double-clicking or using **F2**), it calls a screen. Create this screen (screen number 100) using forward navigation.
 - a) See source code excerpt in the model solution.
 - b) Create a screen using forward navigation: Double-click 100 in the **CALL SCREEN 100** statement.)
4. When specifying the attributes, assign screen number **0** as the number of the next screen, so that after any user action on screen 100, the user returns to the basic list.
 - a) Maintain the screen attributes: Enter a short text. Set the next screen to **0**.

Task 2

Place elements on the screen

1. Create input/output fields on the screen. When you are assigning field types, use the **ABAP Dictionary** structure **SDYN_BOOK**.

The booking table key fields **CARRID**, **CONNID**, **FLDATE**, and **BOOKID** should be copied **with their field labels**.

The customer name **NAME** should be copied **without a field label** and displayed next to the customer number.

Continued on next page

The booking table key fields **CUSTOMID**, **CUSTTYPE**, **SMOKER**, **CLASS**, **LOCCURAM**, and **LOCCURKEY** should be copied **with their field labels**.

- a) Navigate to the Graphical Layout Editor
 - b) Use the *Dict/Program fields* key
 - c) Enter **SDYN_BOOK**
 - d) Choose the *Get from Dictionary* pushbutton
 - e) Choose the fields you want: First block, for key fields: Transfer the fields **CARRID**, **CONNID**, **FLDATE** and **BOOKID** with field labels. Second block: Transfer the customer name **NAME** field (choosing the *Without Text* radio button). Third block: Transfer the fields **CUSTOMID**, **CUSTTYPE**, **SMOKER**, **CLASS**, **LOCCURAM** and **LOCCURKEY** with field labels.
 - f) Choose *Enter* to confirm, and drag these fields to the screen
2. Maintain the screen field attributes:
- The fields **CARRID**, **CONNID**, **FLDATE**, **BOOKID** and **CUSTOMID** should appear as output fields (attribute *Output Field*).
- The customer name **NAME** should be displayed next to the customer number without text (*Output Only* attribute).
- The fields **CARRID**, **CONNID**, **FLDATE**, **BOOKID** and **CUSTOMID** should appear as output fields (attribute *Output Field*).
- a) To navigate to the field attributes you can either double-click the screen element or use the Element List pushbutton.

Result

Source code extract: **SAPBC400UDS_DYNPRO_A**

```
REPORT  sapbc400uds_dynpro_a.
CONSTANTS actvt_display TYPE activ_auth VALUE '03'.

* Definition of selection screen
PARAMETERS pa_anum TYPE sbook-agencynum.

* workarea for select
DATA: wa_booking TYPE sbc400_booking.

START-OF-SELECTION.
```

Continued on next page

```
* selecting data using a dictionary view to get the data from
* sbook and the customer name from scustom
SELECT carrid connid fldate bookid customid name
      FROM sbc400_booking
      INTO CORRESPONDING FIELDS OF wa_booking
      WHERE agencynum = pa_anum.

AUTHORITY-CHECK OBJECT 'S_CARRID'
  ID 'CARRID' FIELD wa_booking-carrid
  ID 'ACTVT'   FIELD actvt_display.
  IF sy-subrc = 0.
* Output
  WRITE: / wa_booking-carrid COLOR col_key,
         wa_booking-connid COLOR col_key,
         wa_booking-fldate COLOR col_key,
         wa_booking-bookid COLOR col_key,
         wa_booking-name.

  ENDIF.
ENDSELECT.

AT LINE-SELECTION.
CALL SCREEN 100.
```


Exercise 15: Screens and Data Transport

Exercise Objectives

After completing this exercise, you will be able to:

- Fill the screen fields with data from the program

Business Example

Change your program **ZBC400_##_DYNPRO** so that double-clicking on a line of the basic list displays details of the selected booking on the screen. If the user changes data on the screen, then these changes should be available in the program once the user has left the screen.

Task 1

Prepare for the field transport from the ABAP program to the screen and back

1. Extend your program **ZBC400_##_DYNPRO** or copy the template **SAPBC400UDS_DYNPRO_A** to the new program **ZBC400_##_DYNPRO_B**.
2. In your program, use the TABLES statement to create a work area with the same name. This work area will then serve as the screen interface.
3. Ensure that the key fields of the **SBOOK** database table and the customer name are still available in the AT LINE-SELECTION event block, even after a line has been selected on the basic list by double-clicking or choosing **F2** (using **HIDE: . . .**).

Task 2

Perform the authorization check and call the screen

1. Later, you should change your program appropriately so that data in the database can be changed. Ensure that the screen can only be processed if the user has change authorization for the airline selected. If the user does not have this authorization, display the success message 047 from the message class **BC400**. (You use a success message to avoid displaying an error dialog).

Continued on next page

Before the screen is called – and provided the user has authorization – send the data of the selected posting from the **SBOOK** database table to a suitable work area (we suggest the name: wa_sbook) to make sure that the data is up-to-date.

2. Immediately before the screen is called, copy the relevant data from your work area for database access into the TABLES work area.

Solution 15: Screens and Data Transport

Task 1

Prepare for the field transport from the ABAP program to the screen and back

1. Extend your program **ZBC400_##_DYNPRO** or copy the template **SAPBC400UDS_DYNPRO_A** to the new program **ZBC400_##_DYNPRO_B**.
 - a) Carry out this step as usual.
2. In your program, use the TABLES statement to create a work area with the same name. This work area will then serve as the screen interface.
 - a) Carry out this step as usual.
3. Ensure that the key fields of the **SBOOK** database table and the customer name are still available in the AT LINE-SELECTION event block, even after a line has been selected on the basic list by double-clicking or choosing **F2** (using **HIDE: . . .**).
 - a) See source code excerpt in the model solution.

Task 2

Perform the authorization check and call the screen

1. Later, you should change your program appropriately so that data in the database can be changed. Ensure that the screen can only be processed if the user has change authorization for the airline selected. If the user does not have this authorization, display the success message 047 from the message class **BC400**. (You use a success message to avoid displaying an error dialog).

Before the screen is called – and provided the user has authorization – send the data of the selected posting from the **SBOOK** database table to a suitable work area (we suggest the name: **wa_sbook**) to make sure that the data is up-to-date.

- a) See source code excerpt in the model solution.

Continued on next page

2. Immediately before the screen is called, copy the relevant data from your work area for database access into the TABLES work area.
 - a) See source code excerpt in the model solution.

Result

Source code extract: **SAPBC400UDS_DYNPRO_B**

```

REPORT  sapbc400uds_dynpro_b.
CONSTANTS: actvt_display TYPE activ_auth VALUE '03',
            actvt_change TYPE activ_auth VALUE '02'.

* Definition of selection screen
PARAMETERS pa_anum TYPE sbook-agencynum.

* workarea for list
DATA wa_booking TYPE sbc400_booking.

* workarea for single booking to be changed
DATA wa_sbook TYPE sbook.

* workarea for dynpro
TABLES sdyn_book.

START-OF-SELECTION.

* selecting data using a dictionary view to get the data from
* sbook and the customer name from scustom
SELECT carrid connid fldate bookid customid name
      FROM sbc400_booking
      INTO CORRESPONDING FIELDS OF wa_booking
      WHERE agencynum = pa_anum.

AUTHORITY-CHECK OBJECT 'S_CARRID'
  ID 'CARRID' FIELD wa_booking-carrid
  ID 'ACTVT'   FIELD actvt_display.
  IF sy-subrc = 0.

* Output
  WRITE: / wa_booking-carrid COLOR col_key,
         wa_booking-connid COLOR col_key,
         wa_booking-fldate COLOR col_key,
         wa_booking-bookid COLOR col_key,
         wa_booking-name.

```

Continued on next page

```

HIDE:      wa_booking-carrid,
          wa_booking-connid,
          wa_booking-fldate,
          wa_booking-bookid,
          wa_booking-name.

ENDIF.

ENDSELECT.

AT LINE-SELECTION.

AUTHORITY-CHECK OBJECT 'S_CARRID'
    ID 'CARRID' FIELD wa_booking-carrid
    ID 'ACTVT'   FIELD actvt_change.

IF sy-subrc = 0.

    SELECT SINGLE *
        FROM sbook
        INTO wa_sbook
        WHERE carrid = wa_booking-carrid
            AND connid = wa_booking-connid
            AND fldate = wa_booking-fldate
            AND bookid = wa_booking-bookid.

    IF sy-subrc = 0.
        MOVE-CORRESPONDING wa_sbook TO sdyn_book.
        MOVE wa_booking-name TO sdyn_book-name.
        CALL SCREEN 100.
    ENDIF.

ELSE .
    MESSAGE ID 'BC400' TYPE 'S' NUMBER '047'
        WITH wa_booking-carrid.
ENDIF.

```


Exercise 16: Screens: Specifying the Next Screen Dynamically

Exercise Objectives

After completing this exercise, you will be able to:

- Create pushbuttons on screens
- Process the function code triggered when the user clicks on a pushbutton and thus control the program flow
- Set subsequent screens dynamically

Business Example

Change your program **ZBC400##_DYNPRO** so that the user should be able to control program flow through two pushbuttons on the screen.

Task 1

Define the pushbuttons

1. Extend your program **ZBC400##_DYNPRO** or copy the template **SAPBC400UDS_DYNPRO_B** to the new program **ZBC400##_DYNPRO_C**.
2. Define two pushbuttons on the screen that allow the user to either return to the basic list (**PUSH_BACK**) or to save changes to data (**PUSH_SAVE**):

Name of pushbutton	Text	Function code
PUSH_BACK	<i>Back</i>	BACK
PUSH_SAVE	<i>Save or icon ICON_SYSTEM_SAVE</i>	SAVE

Task 2

Implement dynamic control of subsequent screens

1. Name the **OK_CODE** field on the screen and define a data object of the same name (and corresponding type) in the program. (We suggest the name **OK_CODE**.)

Continued on next page

2. Navigate in the flow logic. Create a module for function code processing (using forward navigation) at PROCESS AFTER INPUT:

Function code	Action	Next screen
BACK	None	List
SAVE	First: Information message060 (BC400)	List
Any other action	None	Screen 100

3. Ensure that choosing *Enter* always displays screen 100, regardless of the navigation history. To do this, initialize the ok code field in a **PBO** module.

Solution 16: Screens: Specifying the Next Screen Dynamically

Task 1

Define the pushbuttons

1. Extend your program **ZBC400_##_DYNPRO** or copy the template **SAPBC400UDS_DYNPRO_B** to the new program **ZBC400_##_DYNPRO_C**.
 - a) Carry out this step as usual.
2. Define two pushbuttons on the screen that allow the user to either return to the basic list (**PUSH_BACK**) or to save changes to data (**PUSH_SAVE**):
 - a) Carry out this step as described in the training material.

Name of pushbutton	Text	Function code
PUSH_BACK	<i>Back</i>	BACK
PUSH_SAVE	<i>Save or icon ICON_SYSTEM_SAVE</i>	SAVE

- a) Carry out this step as described in the training material.
2. Implement dynamic control of subsequent screens
 1. Name the **OK_CODE** field on the screen and define a data object of the same name (and corresponding type) in the program. (We suggest the name **OK_CODE**.)
 - a) Carry out this step as described in the training material.
 - b) See source code excerpt in the model solution.
 2. Navigate in the flow logic. Create a module for function code processing (using forward navigation) at PROCESS AFTER INPUT:

Function code	Action	Next screen
BACK	None	List

Continued on next page

SAVE	First: Information message060 (BC400)	List
Any other action	None	Screen 100

- a) See source code excerpt in the model solution.
3. Ensure that choosing *Enter* always displays screen 100, regardless of the navigation history. To do this, initialize the ok code field in a **PBO** module.
- a) See source code excerpt in the model solution.

Result

Source code extract: **SAPBC400UDS_DYNPRO_C**

```

REPORT  sapbc400uds_dynpro_c.
CONSTANTS: actvt_display TYPE activ_auth VALUE '03',
            actvt_change TYPE activ_auth VALUE '02'.

*  Definition of selection screen
PARAMETERS pa_anum TYPE sbook-agencynum.

* workarea for list
DATA wa_booking TYPE sbc400_booking.

* workarea for single booking to be changed
DATA wa_sbook TYPE sbook.

* workarea for dynpro
TABLES sdyn_book.
* variable for function code of user action
DATA ok_code LIKE sy-ucomm.

START-OF-SELECTION.

* selecting data using a dictionary view to get the data from
* sbook and the customer name from scustom
SELECT carrid connid fldate bookid customid name
      FROM sbc400_booking
      INTO CORRESPONDING FIELDS OF wa_booking
      WHERE agencynum = pa_anum.

```

Continued on next page

```

AUTHORITY-CHECK OBJECT 'S_CARRID'
  ID 'CARRID' FIELD wa_booking-carrid
  ID 'ACTVT'   FIELD actvt_display.
  IF sy-subrc = 0.
    * Output
    WRITE: / wa_booking-carrid COLOR col_key,
            wa_booking-connid COLOR col_key,
            wa_booking-fldate COLOR col_key,
            wa_booking-bookid COLOR col_key,
            wa_booking-name.

    HIDE:   wa_booking-carrid,
            wa_booking-connid,
            wa_booking-fldate,
            wa_booking-bookid,
            wa_booking-name.

  ENDIF.
ENDSELECT.

CLEAR wa_booking.

AT LINE-SELECTION.

AUTHORITY-CHECK OBJECT 'S_CARRID'
  ID 'CARRID' FIELD wa_booking-carrid
  ID 'ACTVT'   FIELD actvt_change.
  IF sy-subrc = 0.

    SELECT SINGLE *
      FROM sbook
      INTO wa_sbook
      WHERE carrid = wa_booking-carrid
        AND connid = wa_booking-connid
        AND fldate = wa_booking-fldate
        AND bookid  = wa_booking-bookid.
    IF sy-subrc = 0.
      MOVE-CORRESPONDING wa_sbook TO sdyn_book.
      MOVE wa_booking-name TO sdyn_book-name.
      CALL SCREEN 100.
    ENDIF.
  ENDIF.

```

Continued on next page

```

ENDIF.

ELSE .
MESSAGE ID 'BC400' TYPE 'S' NUMBER '047'
WITH wa_booking-carrid.
ENDIF.

CLEAR: wa_sbook, wa_booking.

*-----*
*&      Module  CLEAR_OK_CODE  OUTPUT
*-----*
*&      text
*-----*
MODULE clear_ok_code OUTPUT.

CLEAR ok_code.

ENDMODULE.           " CLEAR_OK_CODE  OUTPUT

*-----*
*&      Module  USER_COMMAND_0100  INPUT
*-----*
*&      text
*-----*
MODULE user_command_0100 INPUT.

CASE ok_code.
WHEN 'BACK'.
LEAVE TO SCREEN 0.
WHEN 'SAVE'.
MOVE-CORRESPONDING sdyn_book TO wa_sbook.
MESSAGE ID 'BC400' TYPE 'I' NUMBER '060'.
LEAVE TO SCREEN 0.

ENDCASE.
ENDMODULE.           " USER_COMMAND_0100  INPUT

```

Flow logic for screen 100:

PROCESS BEFORE OUTPUT.

Continued on next page

```
MODULE clear_ok_code.
```

```
PROCESS AFTER INPUT.
```

```
MODULE user_command_100.
```



Lesson Summary

You should now be able to:

- List attributes and benefits of screens
- Write programs that display data on screens
- Write programs that allow users to change some data on screens
- Write programs that allow users to affect subsequent program flow using pushbuttons

Lesson: User interface

Lesson Overview

In this lesson, you will create a status with standard list functions for your list, and a status for a screen, which makes reference to the components of the list status. You will also assign a title to the list and screen.



Lesson Objectives

After completing this lesson, you will be able to:

- Create and set a GUI title
- Maintain menu bars
- Maintain standard toolbars
- Maintain application toolbars
- Maintain function key assignments
- Create and set a GUI status for lists and screens

Business Example

You need to provide user interfaces for your program screens.

User interface

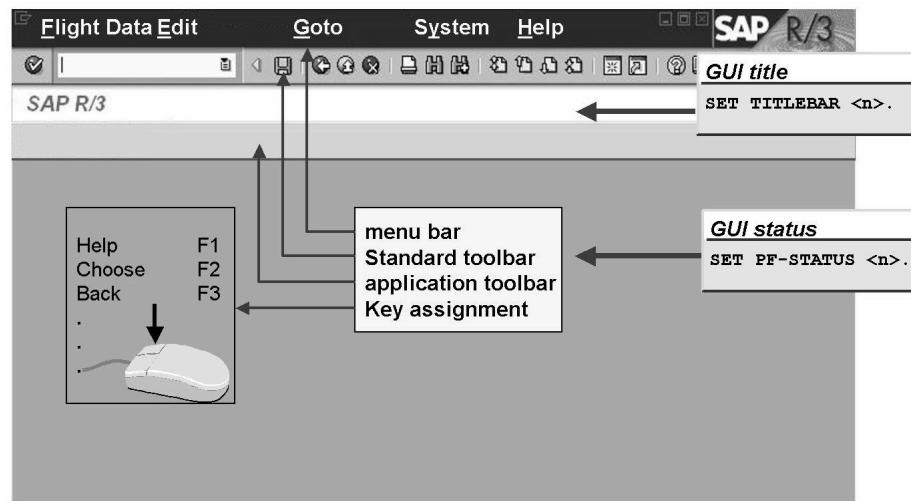


Figure 168: Overview: Screen

The screen contains the following elements:

- The **title bar** contains the title of the current screen.
- The **menu bar** contains menus, which users can expand.

These **menus** contain the executable functions for the current program. Menus can also contain submenus (cascading menus). The *System* and *Help* menus can be found on every screen in *SAP R/3* and always contain the same functions. Neither of these menus may be changed or hidden.

On every screen in the *SAP R/3 System*, the **standard toolbar** always contains the same standard pushbuttons for functions that are used a lot. The pushbuttons for functions that cannot be executed on the current screen are grayed out.

The **function key assignment** can be displayed by right-clicking. Ideally, you should be able to execute all menu functions using a function key as well as from the menu.

The **application toolbar** contains pushbuttons for frequently used functions on a particular screen.

Each program is created with an interface containing all of the above tools and objects. Different views of this interface (GUI statuses) are then created for a program's individual screens, selection screens, and lists. These views are known as **GUI statuses**.

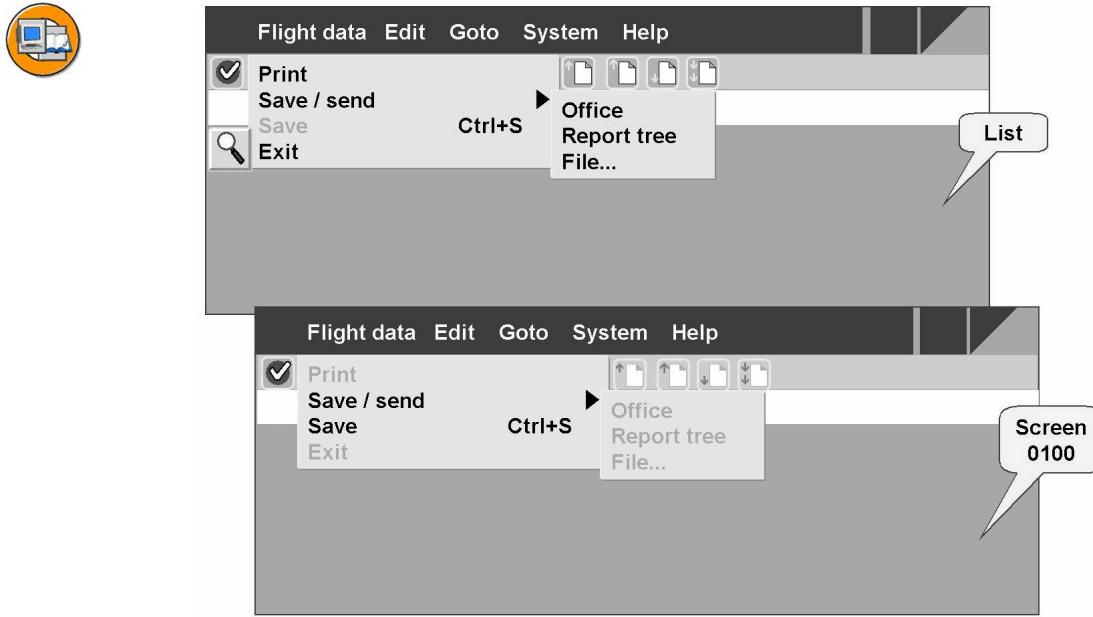


Figure 169: The Objective: User Interface for the Example Program

We will extend the user interface for the entire example program. The various statuses will have the following attributes:

- The **same** menu bar will be available on both the list and the screen. Only those menu functions that can be executed will appear in black; functions that cannot be executed will be grayed out.
- In the standard toolbar, each pushbutton will appear in color only if the function associated with it can be executed. The set of features associated with the screen will not be in any way.

The screen will have its own title.



```
SET PF-STATUS 'LIST_STATUS'.
```

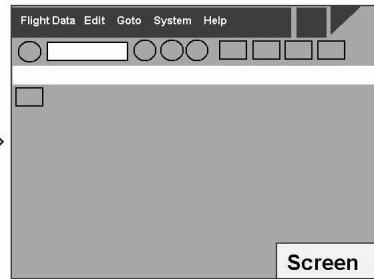
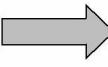
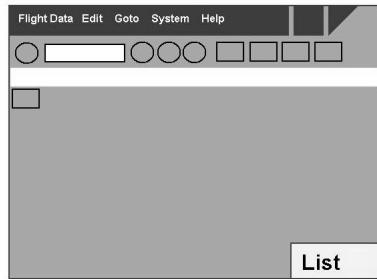
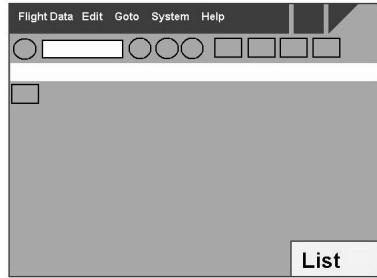


Figure 170: Setting a Status

You set a status using the SET PF-STATUS statement before the screen is displayed. It will remain valid for all subsequent screens provided you do not set a new status.



```
SET PF-STATUS 'LIST_STATUS'.
```



```
SET PF-STATUS 'DYNP_100'.
```

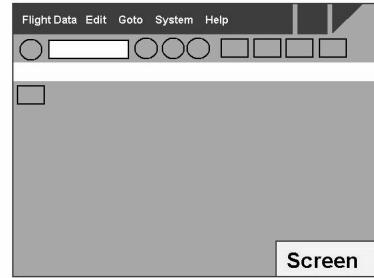


Figure 171: Setting a Status for Each Screen

In the example, we will set a separate status for each screen. However, the same functions are to be available in every status. Not all functions are appropriate on both screens – for example the *Print* function is only useful on the list.

Depending on whether the program is currently displaying a list or a screen, the functions should be black (active – that is, executable) or gray (inactive – that is, not executable).

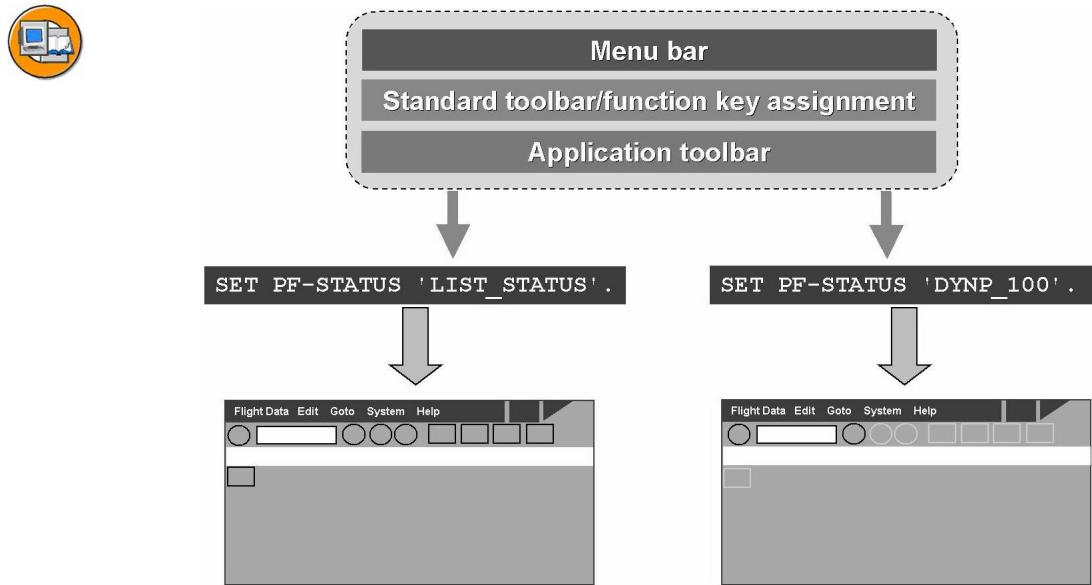


Figure 172: References to Subobjects

To make maintenance easier, we want to avoid creating the subobjects menu bar and standard toolbar with function keys and application toolbar for the screen status again. For this reason, we simply reference the subobjects already created in the context of the list status and specify which functions are active or inactive, from the point of view of this status.

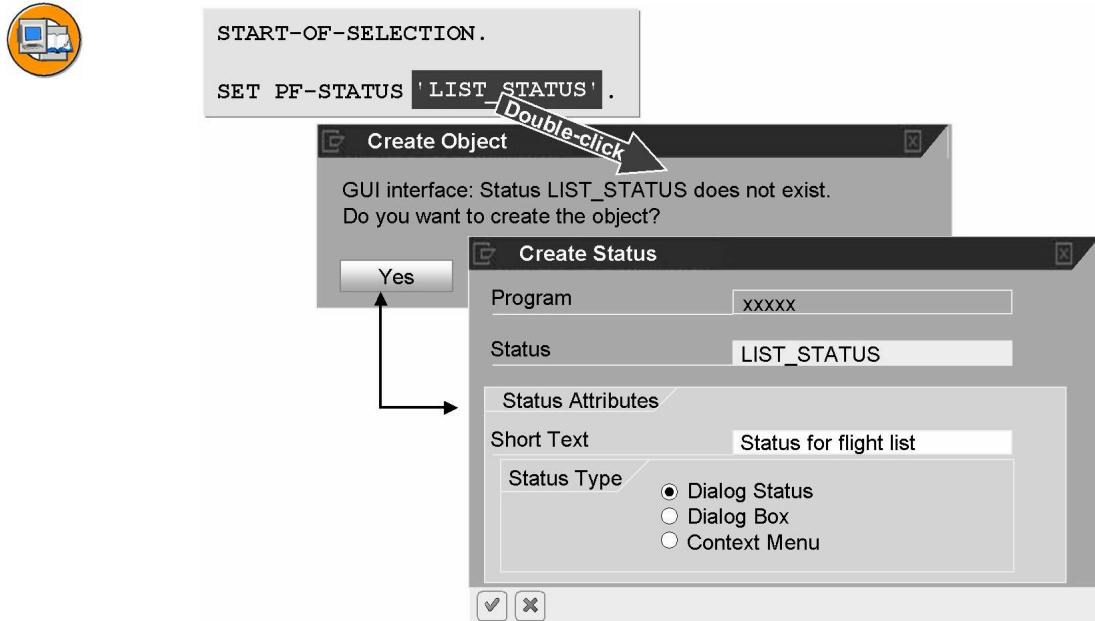


Figure 173: Creating GUI Statuses for Lists

You can create or maintain statuses in three different ways:

- By using the **object list** in the *Object Navigator*
- By using **forward navigation** in the *ABAP Editor*
- By using the *Menu Painter* itself

When creating a status you can create a new menu bar, application toolbar, and key assignment (top down), adopt existing objects (bottom up), or use a combination of both methods.

Status names can have a maximum of 20 characters.

By choosing a status type, you determine whether, for instance, a status refers to a normal screen or a dialog box. Depending on the status type, you have a choice of different types of functions.

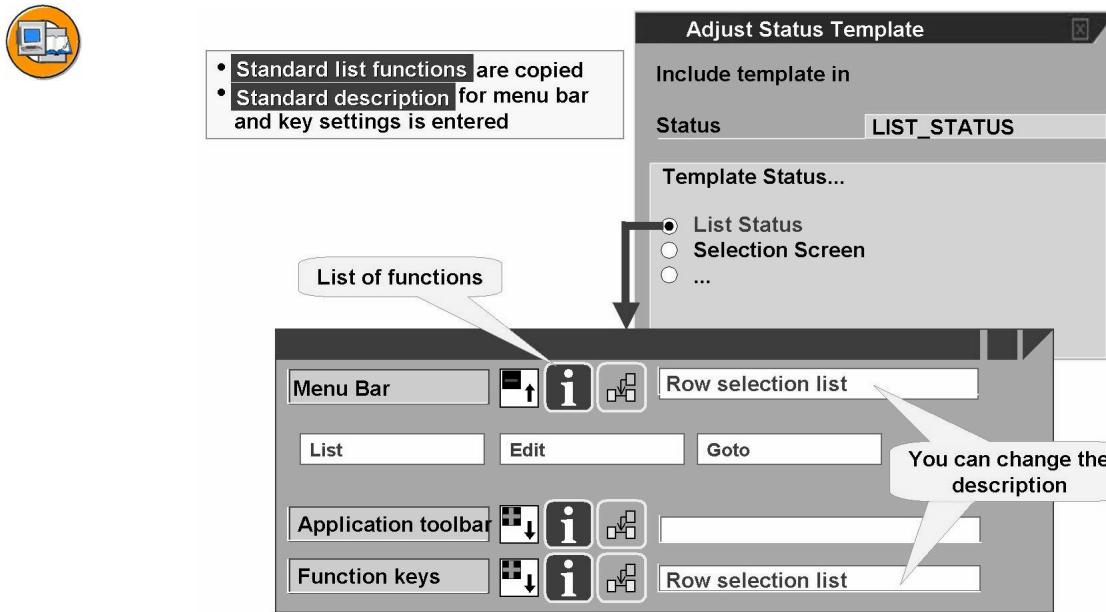


Figure 174: Adjusting Statuses

The *Adjust Template* function in the *Extras* menu allows you to include standardized function codes in your status. This function also allows you to include objects from a status of your choice in the existing status. In particular, you can choose standard values for list and selection screens or use any status you want from another ABAP program.

If you choose *List Status*, the system generates standard list functions and the standard entries for the menu, application toolbar, and the assignment of functions to function keys. Each menu bar, application toolbar, and function key setting is also given a default name, which you should replace with an explanatory short text.

You can display the generated functions by choosing the Information icon.

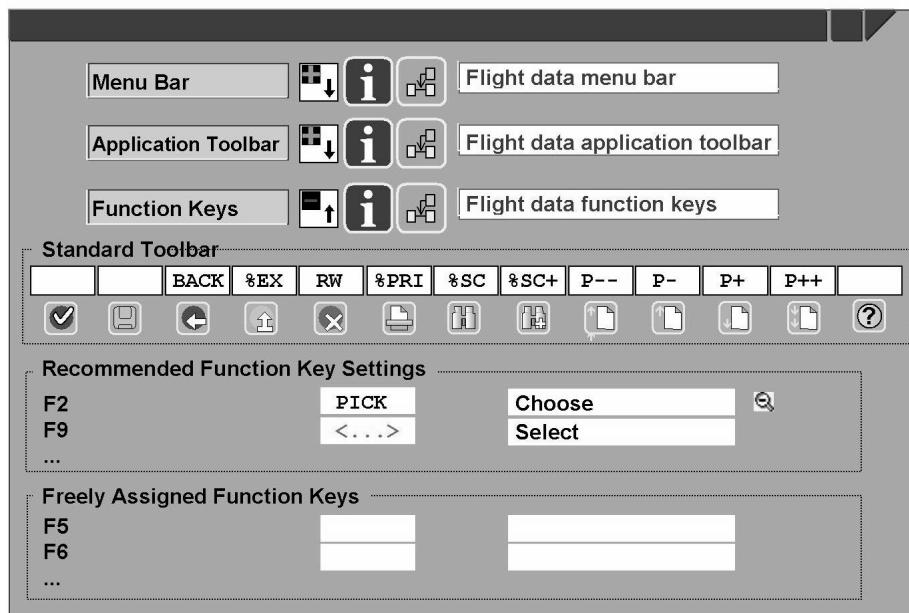


Figure 175: Status in the Menu Painter: Function Keys

Function key assignment is split into three areas:

Standard Toolbar

Certain function codes are predefined for the functions *Save*, *Back*, *Exit Program*, *Cancel*, *Print*; for the *Scroll Icons*; and for the *Enter* pushbutton. Simply assign function codes to the standard toolbar icons and they will be automatically assigned to their corresponding function keys.

Recommended Function Key Settings

The system proposes the functions that you should generally assign to specific function keys. Users are accustomed to these assignments, so you should not change them without good reason.

Freely Assigned Function Keys

Here all the remaining function codes, that are not assigned to a standard function key, are listed. From this list, you can choose appropriate function keys for program-specific functions.

You can also define buttons on an application toolbar for frequently-used function keys. These pushbuttons can have icons or text or both.



Caution: Do not confuse these pushbuttons (on the application toolbar) with the *Pushbutton* element on a screen.

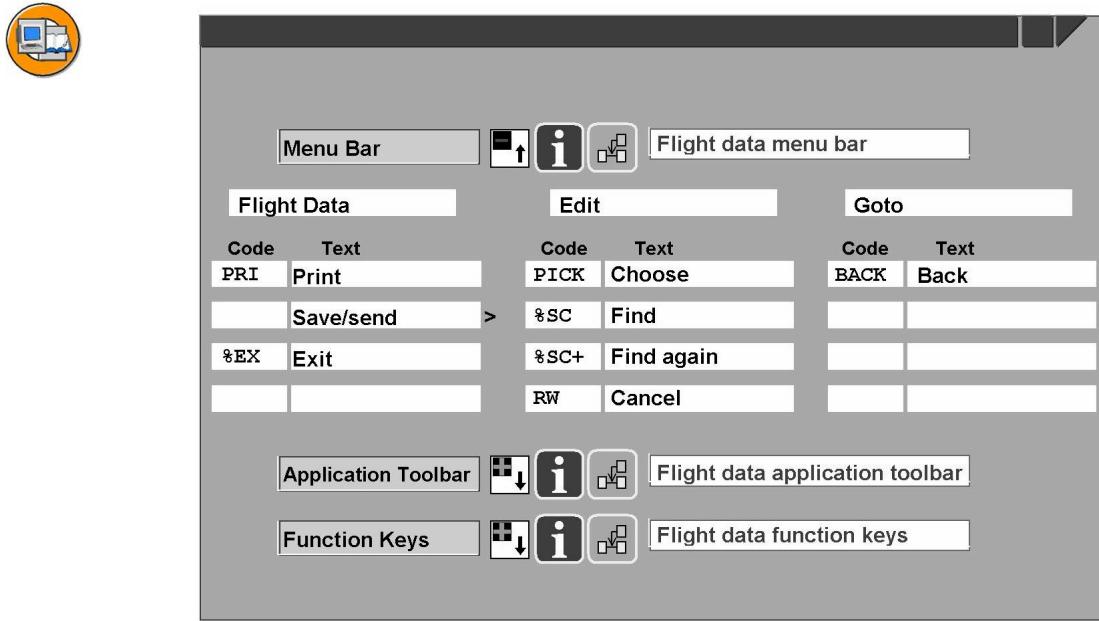


Figure 176: Statuses in the Menu Painter: Menu Bar

If so desired, the system will suggest standard values for your menu bar, which you can overwrite.

Menu bars can contain up to eight menus. You can define up to six of these yourself, but the *System* and *Help* menus are standard and are added **automatically** by the system.



Hint: If necessary, create cascading menus.

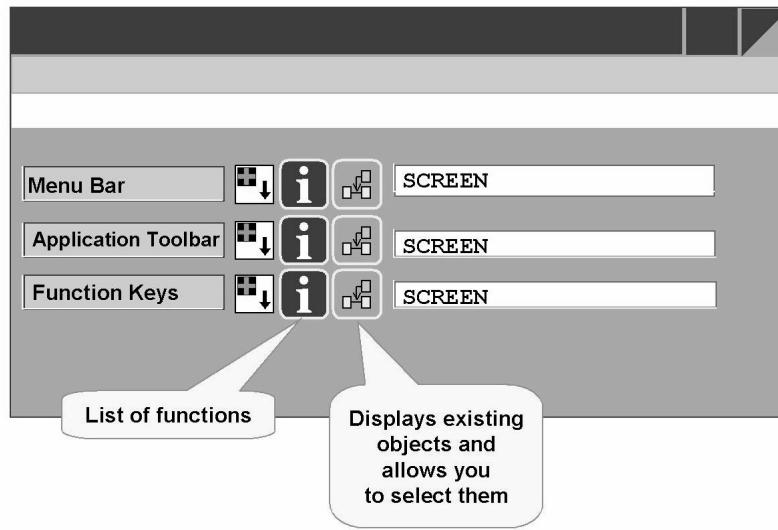


Figure 177: Incorporating Existing Elements

Create a new status for the screen and use the icon shown above (or choose **F4**) to display all the menu bars and function key assignments already defined.

The example program displays only one menu bar or function key assignment. Double-click this to reference the same menu bar and function key assignment for the screen status as for the list status.

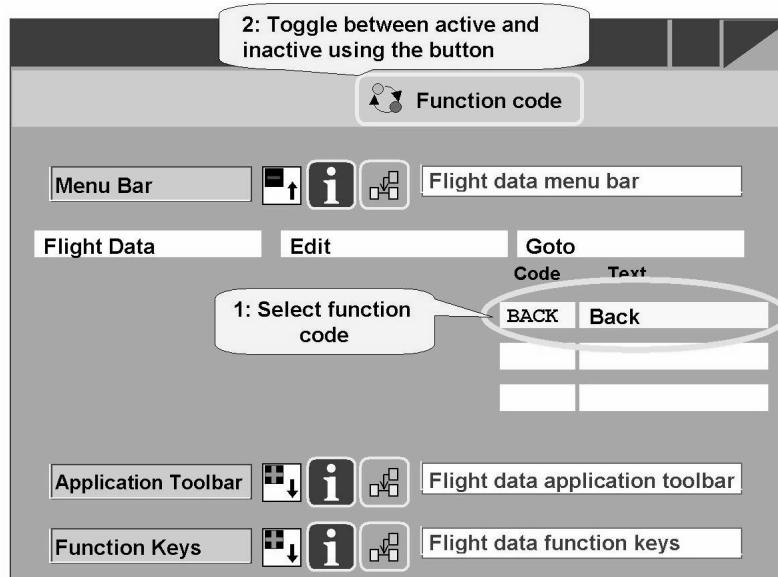


Figure 178: Setting Functions to Active or Inactive in the Status

To set status functions to active or inactive, proceed as follows:

- Switch to *Change* mode in the status maintenance screen by choosing *Display <-> Change*.
- Place the cursor on the function code you wish to activate and choose the *Active <-> Inactive* icon on the application toolbar.

In our example, we are adding the **SAVE** function, which is not a standard list function, to the standard toolbar and the *Flight Data* menu. The system informs us that we are referencing the menu bar and standard toolbar. If we confirm the dialog box, the system automatically adds the **SAVE** function to the menu bar and standard toolbar in the list status and sets it to inactive.

The **BACK** and **SAVE** functions in our example are evaluated in the **PAI module USER_COMMAND_0100**.

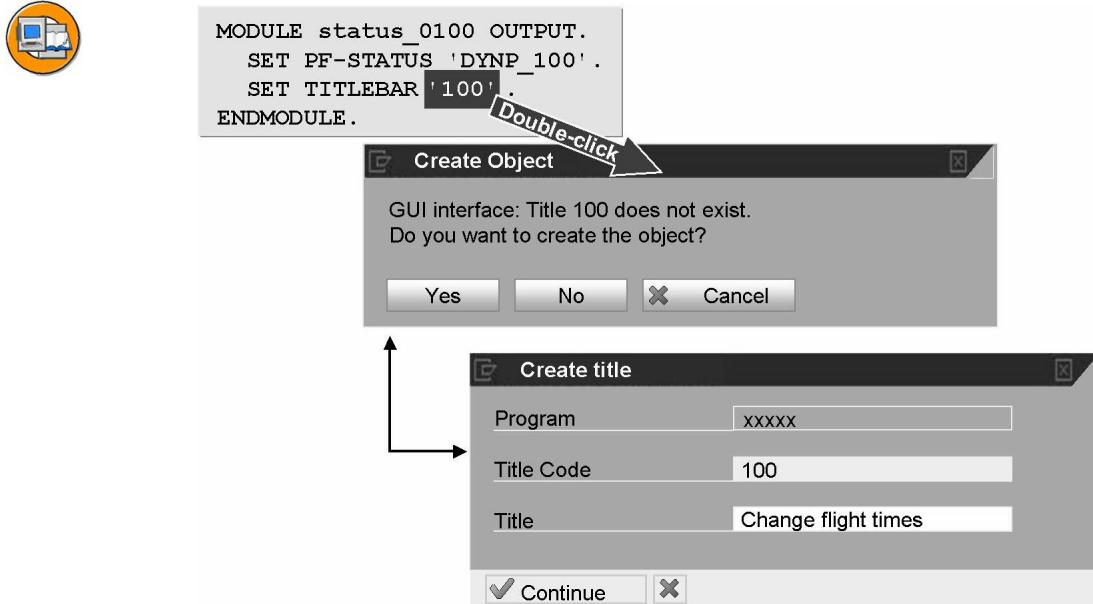


Figure 179: Creating a GUI Title for a Screen

You can create or maintain titles in three different ways:

- By using the **object list** in the *Object Navigator*
- By using **forward navigation** in the *ABAP Editor*
- By using the *Menu Painter* itself

A title can have a maximum of 20 characters.



Hint: If you choose the pregenerated call of the PBO module STATUS_nnnn using forward navigation, the **SET PF-STATUS 'xxxxxxxx'**. and **SET TITLEBAR 'xxx'**. statements are generated in the module as comments. You can use this suggestion by erasing the comment asterisks and entering the names of the objects you want. You can then create the status and title using forward navigation.

Exercise 17: Creating an Interface

Exercise Objectives

After completing this exercise, you will be able to:

- Create user interfaces for a program
- Include a GUI status and GUI title in a program

Business Example

Extend your program **ZBC400_##_DYNPRO**:

The system is to display texts corresponding to the list and screen in the title bar. You also specify a suitable status for the list and screen.

Task 1

Create and set a GUI status for the list

1. Extend your program **ZBC400_##_DYNPRO** or copy the template **SAPBC400UDS_DYNPRO_C** to the new program **ZBC400_##_DYNPRO_D**.
2. Create the status (type: *Dialog Status*) using forward navigation. Add the relevant ABAP statement to the **START-OF-SELECTION** processing block

The status you create must correspond to the standard list status. Add the appropriate default functions for function key assignment, the application toolbar, and the menu bar. Change the name of the first menu item from *List* to *Bookings*.

3. Add the function code **SAVE**. Users should be able to trigger this by choosing the *Save* icon in the standard toolbar. Users should also be able to choose *Save* from an **item in the Bookings menu**. In this status, however, *Save* should be **inactive**.
4. Optional:
Define and set a title to the list and screen (using forward navigation).

Task 2

Create and set GUI Statuses for a screen

Continued on next page

1. Create the status (type: *Dialog Status*) using forward navigation. Add the relevant ABAP statement to a module at the **PROCESS BEFORE OUTPUT event**.
2. Make a reference to the status you created previously for the list. Activate the function codes **BACK** and **SAVE** in this status.
3. Optional:
Define and set a title for the list and screen (using forward navigation).

Solution 17: Creating an Interface

Task 1

Create and set a GUI status for the list

1. Extend your program **ZBC400_##_DYNPRO** or copy the template **SAPBC400UDS_DYNPRO_C** to the new program **ZBC400_##_DYNPRO_D**.
 - a) Carry out this step as usual.
2. Create the status (type: *Dialog Status*) using forward navigation. Add the relevant ABAP statement to the **START-OF-SELECTION** processing block

The status you create must correspond to the standard list status. Add the appropriate default functions for function key assignment, the application toolbar, and the menu bar. Change the name of the first menu item from *List* to *Bookings*.

- a) In the program source code, add the `SET PF-STATUS 'LIST'` statement to the **START-OF-SELECTION** event.

 **Caution:** Text between the two apostrophes **must** be in upper case.
- b) In the program line `SET PF-STATUS 'LIST'`, **double-click** the **LIST** status name.
- c) The system displays a dialog box containing the words "The GUI interface status 'LIST' does not exist. Do you want to create it?"
Choose Yes to confirm.
- d) **Enter** the *Short Text*.
- e) **Choose** the status type *Dialog Status*.
- f) **Choose:** *Extras → Adjust Template → List Status*
- g) **Change** the **List** entry for the first menu to **Bookings** and confirm this in the dialog box that appears.

Continued on next page

3. Add the function code **SAVE**. Users should be able to trigger this by choosing the *Save* icon in the standard toolbar. Users should also be able to choose *Save* from an **item in the *Bookings* menu**. In this status, however, *Save* should be **inactive**.
 - a) In the standard toolbar, enter the function code **SAVE** above the *Save* icon and choose *Enter*.
 - b) Double-click to open the *Bookings* menu, enter **SAVE** as a function code, and assign the text “Save bookings” to it.
 - c) Choose *Active <-> Inactive* to deactivate **SAVE** for this status.
 - d) Activate the **whole status** and the program.
4. Optional:
Define and set a title to the list and screen (using forward navigation).
 - a) In the program source code, add the `SET TITLEBAR 'LIST'` statement to the **START-OF-SELECTION** event.
 - b) Create the title using forward navigation and enter the text in the dialog box.
 - c) Activate the interface and the program.

Task 2

Create and set GUI Statuses for a screen

Continued on next page

1. Create the status (type: *Dialog Status*) using forward navigation. Add the relevant ABAP statement to a module at the **PROCESS BEFORE OUTPUT** event.
 - a) Display screen 100 in change mode (for example, by selecting the screen number in the program object list and using the alternate mouse button). In the flow logic of the PROCESS BEFORE OUTPUT event, add the statement **MODULE set_status_0100**.
 - b) Double-click the name of the module; choose *Yes* to confirm the settings in the dialog box. In the next dialog box, choose *Main Program* and add the following source text:

```
MODULE set_status_100 OUTPUT. SET PF-STATUS  
'DYNPRO'. ENDMODULE.
```
 - c) Create the status using forward navigation. (In the **SET PF-STATUS 'DYNPRO' .** statement, double-click '*DYNPRO*').
Choose *Yes* to confirm the dialog box that appears.
In the next dialog box that appears, enter a short text and choose *Dialog Status*.
2. Make a reference to the status you created previously for the list. Activate the function codes **BACK** and **SAVE** in this status.
 - a) Place the cursor on the *Menu Bar* and choose the reference icon (or **F4**). Create the reference to the existing menu bar by double-clicking.
 - b) Place the cursor on *Function Keys* and choose the reference icon (or **F4**). Create the reference to the existing function key assignment by double-clicking.
 - c) Activate the two functions **SAVE** and **BACK** by placing the cursor on the function code in the menu or on the function key assignment and choosing the *Active <-> Inactive* pushbutton.
 - d) Activate the status.
Activate the screen.
3. Optional:

Continued on next page

Define and set a title for the list and screen (using forward navigation).

- In the program source code, add the `SET TITLEBAR 'DYNPRO'` statement to the `SET_STATUS_0100` module.
- Create the title using forward navigation and enter the text in the dialog box.
- Activate the interface and the program.

Result

Source code extract: **SAPBC400UDS_DYNPRO_D:**

```
REPORT  sapbc400uds_dynpro_d.

.

START-OF-SELECTION.

  SET PF-STATUS 'LIST'.
  SET TITLEBAR 'LIST'.

* selecting data using a dictionary view to get the data from
* sbook and the customer name from scustom
  SELECT carrid connid fldate bookid customid name
    FROM sbc400_booking
   INTO CORRESPONDING FIELDS OF wa_booking
  WHERE agencynum = pa_anum.

.

AT LINE-SELECTION.

.

*-----*
*&     Module  STATUS_0100  OUTPUT
*-----*
*      text
*-----*
*-----*

MODULE STATUS_0100 OUTPUT.
  SET PF-STATUS 'DYNPRO'.
  SET TITLEBAR 'DYNPRO'.      "optional
ENDMODULE.                      " STATUS_0100  OUTPUT
```

Continued on next page

```

*&-----*
*&      Module  CLEAR_OK_CODE  OUTPUT
*&-----*
*&      text
*&-----*
MODULE clear_ok_code OUTPUT.
CLEAR ok_code.
ENDMODULE.          " CLEAR_OK_CODE  OUTPUT

*&-----*
*&      Module  USER_COMMAND_0100  INPUT
*&-----*
*&      text
*&-----*
MODULE user_command_0100 INPUT.
CASE ok_code.
WHEN 'BACK'.
    LEAVE TO SCREEN 0.
WHEN 'SAVE'.
    MOVE-CORRESPONDING sdyn_book TO wa_sbook.
    MESSAGE ID 'BC400' TYPE 'I' NUMBER '060'.
    LEAVE TO SCREEN 0.

ENDCASE.
ENDMODULE.          " USER_COMMAND_0100  INPUT

```

Flow logic for screen 100:

```

PROCESS BEFORE OUTPUT.

MODULE status_0100.

MODULE clear_ok_code.

PROCESS AFTER INPUT.

MODULE user_command_0100.

```



Lesson Summary

You should now be able to:

- Create and set a GUI title
- Maintain menu bars
- Maintain standard toolbars
- Maintain application toolbars
- Maintain function key assignments
- Create and set a GUI status for lists and screens



Unit Summary

You should now be able to:

- Describe the attributes and benefits of ABAP lists
- Implement interactive lists
- Describe the event-controlled processes for the AT LINE-SELECTION event when executing an ABAP program
- List the properties and benefits of selection screens
- Implement the options for restricting selections using selection screens
- Implement input checks on the selection screen
- List attributes and benefits of screens
- Write programs that display data on screens
- Write programs that allow users to change some data on screens
- Write programs that allow users to affect subsequent program flow using pushbuttons
- Create and set a GUI title
- Maintain menu bars
- Maintain standard toolbars
- Maintain application toolbars
- Maintain function key assignments
- Create and set a GUI status for lists and screens

Related Information

... Refer to the article “Screens” in the online documentation and to the keyword documentation for the relevant ABAP statement.

Unit 9

Outlook: Developing Internet Applications

Unit Overview

Refer to the lesson summaries for an overview of this unit.



Unit Objectives

After completing this unit, you will be able to:

- Create a transaction code for a Web transaction
- Create an Internet service
- Publish an Internet service
- Execute a Web transaction using the Web browser

Unit Contents

Lesson: Developing Internet Applications	318
Procedure: Creating a Web Transaction	323

Lesson: Developing Internet Applications

Lesson Overview

This lesson shows how a SAP R/3 single screen transaction is displayed as a Web transaction on the Internet.



Lesson Objectives

After completing this lesson, you will be able to:

- Create a transaction code for a Web transaction
- Create an Internet service
- Publish an Internet service
- Execute a Web transaction using the Web browser

Business Example

Your project leader wants to know whether or not the *Internet Transaction Server* is the concept best suited to a specific application requirement.

Internet Applications

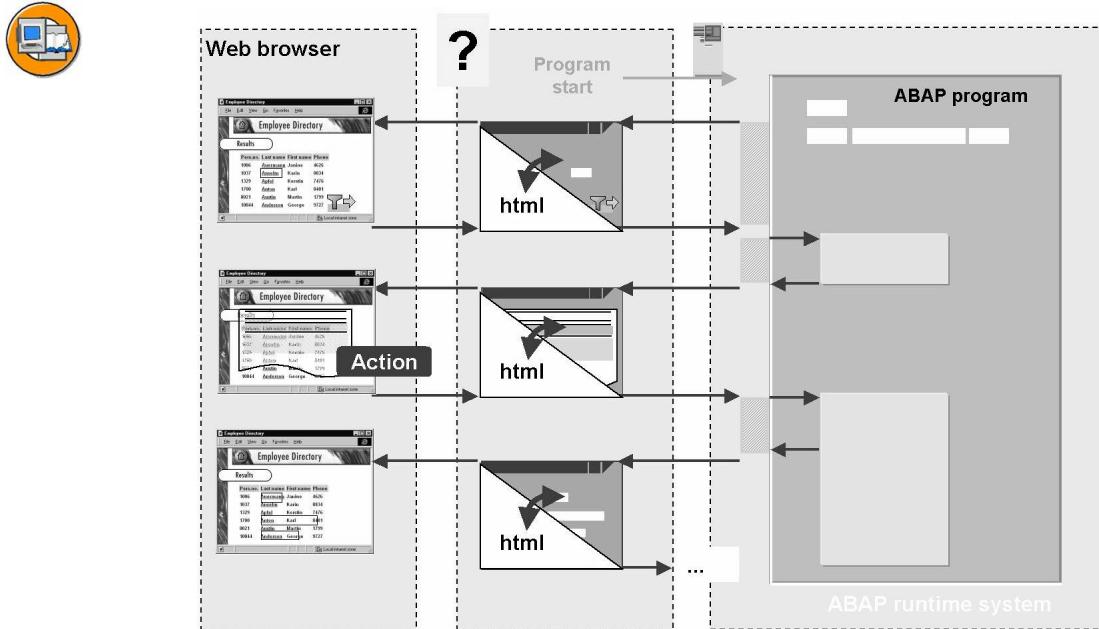


Figure 180: Aim: ABAP Program User Dialogs in a Web Browser

To launch an SAP program from a Web browser, you have to generate HTML pages for the screens. You can generate these automatically.

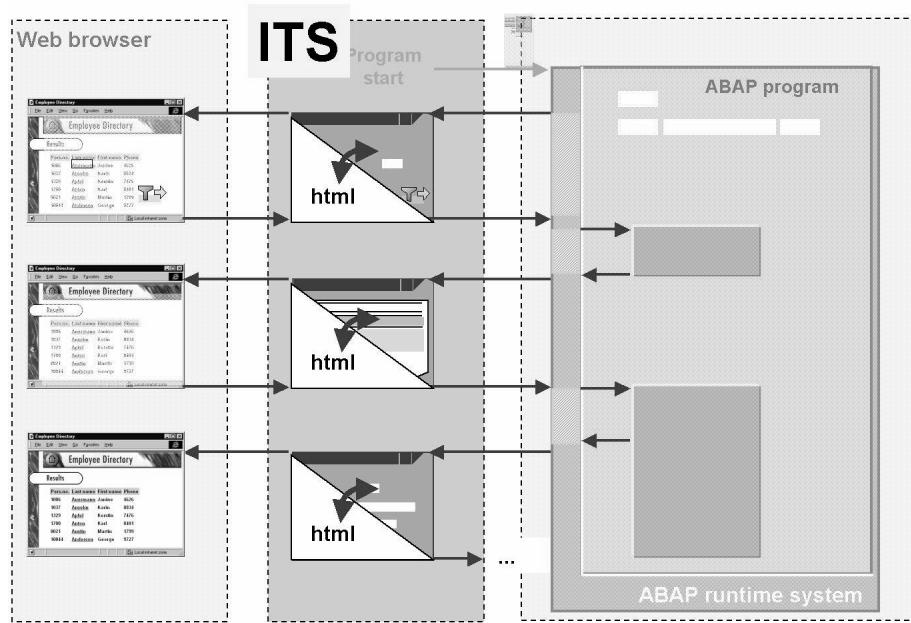


Figure 181: Internet Transaction Server

This task is carried out by the *Internet Transaction Server*.

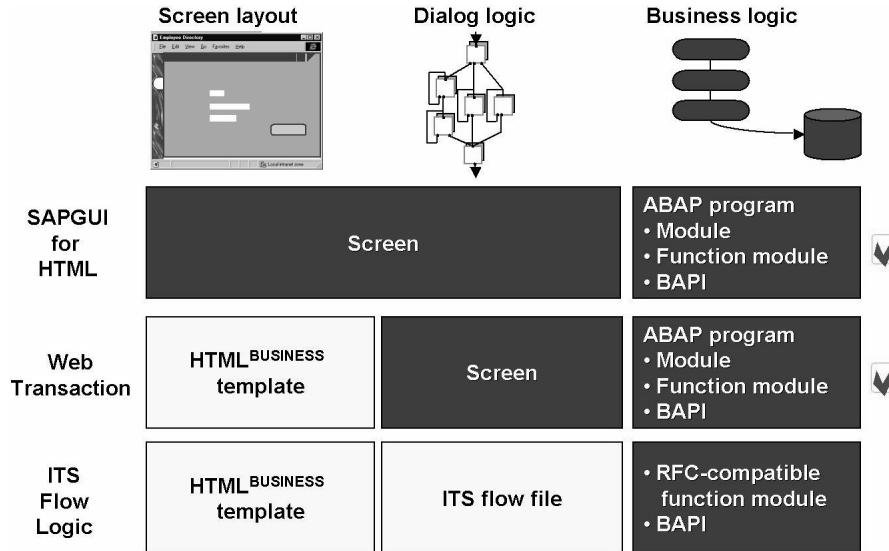


Figure 182: ITS Programming Models

There are three programming models for creating internet applications using the *Internet Transaction Server*:

SAPGUI for HTML

Here an ABAP program with user dialogs provides the basis for the display, dialog logic, and business logic. The layout is generated automatically from the screen information. Therefore, the layout in the Web Browser contains exactly the same elements as the *SAP R/3* screen.

Web Transaction

Only the business logic and the dialog logic are copied from the ABAP program. You can then change or add to the appearance of the page.

ITS Flow Logic

Only the pure business logic is executed in an ABAP program, which can be called "externally". This is usually a BAPI or an RFC-compatible function module. The dialog logic and layout are defined on the *ITS*.

The following pages discuss the programming models SAPGUI for HTML and Web Transaction in more detail.

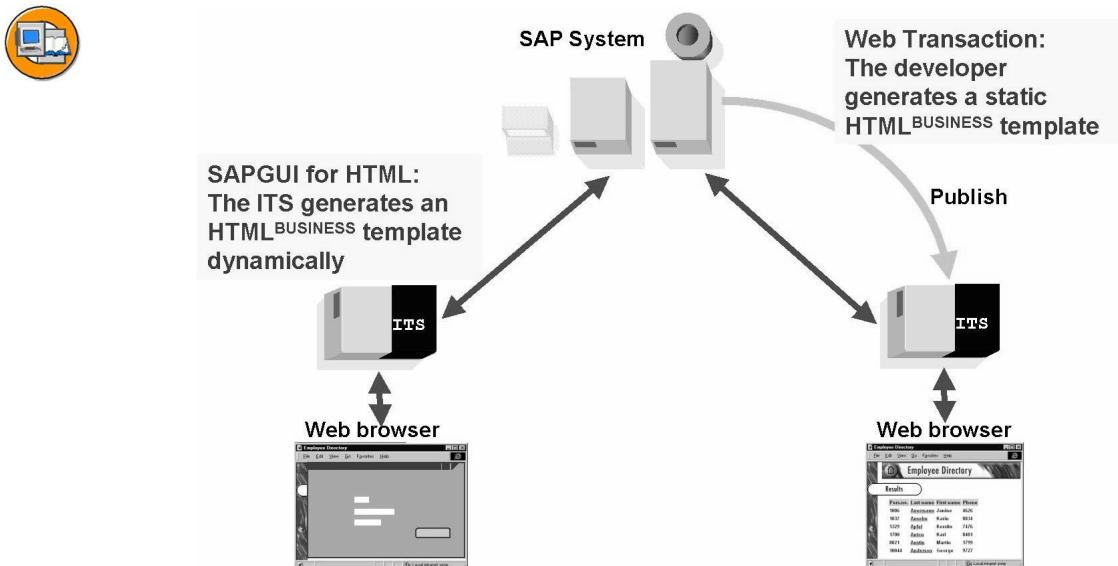


Figure 183: SAPGUI for HTML and Web Transaction

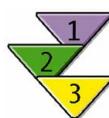
SAP GUI for HTML generates **HTML** pages dynamically from the relevant SAP screens. Features of this process are:

- Simple, screen-based layout
- Screen elements (such as text or input fields) are mapped 1:1 to the fields on the generated HTML page.
- No other HTML elements are available.

- No further development is necessary.
- The user must enter the path
`http://<server>[:<port>]/<path>/wgate/we-bgui/! [?~transaction=...]` in the Web browser. You could create a simple user interface with an appropriate link.

With **Web Transaction** the **developer** creates the static **HTML^{BUSINESS} templates** himself. Features of this process are:

- Flexible design of the layout. The screen information serves as a basis for the template; however, you can adjust the template to suit individual needs (for example, hide complex functions or include additional MIME objects).
- You can add additional functions
- Additional development necessary:
 - You must create, edit, and publish HTML templates
 - Additional functions must be implemented manually



Creating a Web Transaction

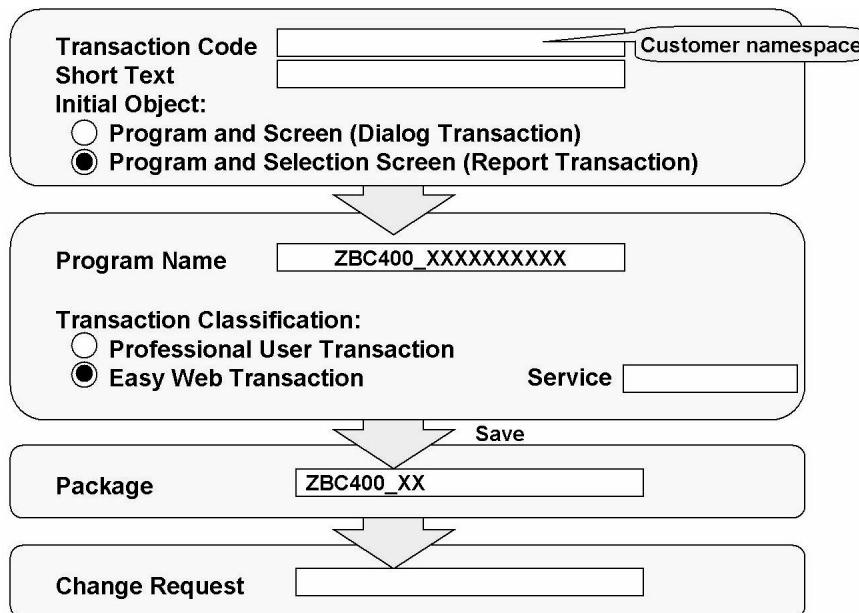


Figure 184: Creating a Transaction Code for a Web Transaction

1. In the *Object Navigator*, open the object list of the program you wish to create a Web Transaction for. In the context menu for the program name, choose *Create → Transaction*.

Enter a transaction code. Comply with the customer namespace conventions. Enter a short text. If the program is an *executable program*, choose *Program and Selection Screen (Report Transaction)*.

On the next screen, enter the name of the program. Choose *Easy Web Transaction* followed by *Save*.

On the next screen, enter the package to which the transaction code is to be assigned.

On the next screen, assign the change request.

Continued on next page

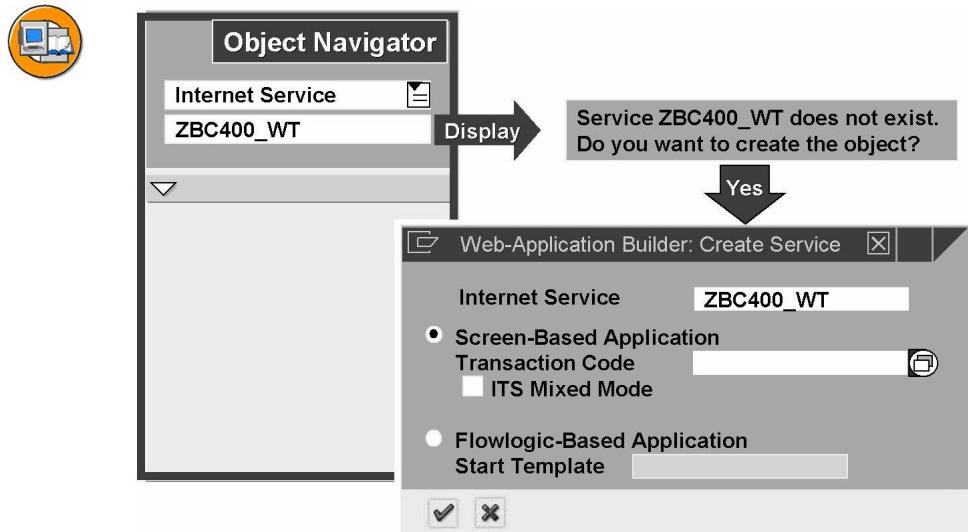


Figure 185: Creating an Internet Service for Web Transaction

2. One way of making a transaction available over the *ITS* directly is to create a service file. The service file specifies the information, which transaction is to be started.

You can create a service for a transaction in the *Object Navigator*. Choose *Internet Services* and enter a name for your service. Make sure you comply with the customer namespace conventions. Internet services are often given the same name as the corresponding transaction. However, you can use a different name, particularly if you are creating several services for the same transaction.

Choose *Enter* or *Display*. If there is no service saved under the name you have entered, you can confirm the query with *Enter*.

Save the service.



Hint: If you choose *ITS Mixed Mode*, you do not need to generate a static HTML page for each screen. At runtime, the SAPGUI for HTML dynamically generates an HTML page for those screens that do not have a static HTML page.

Continued on next page

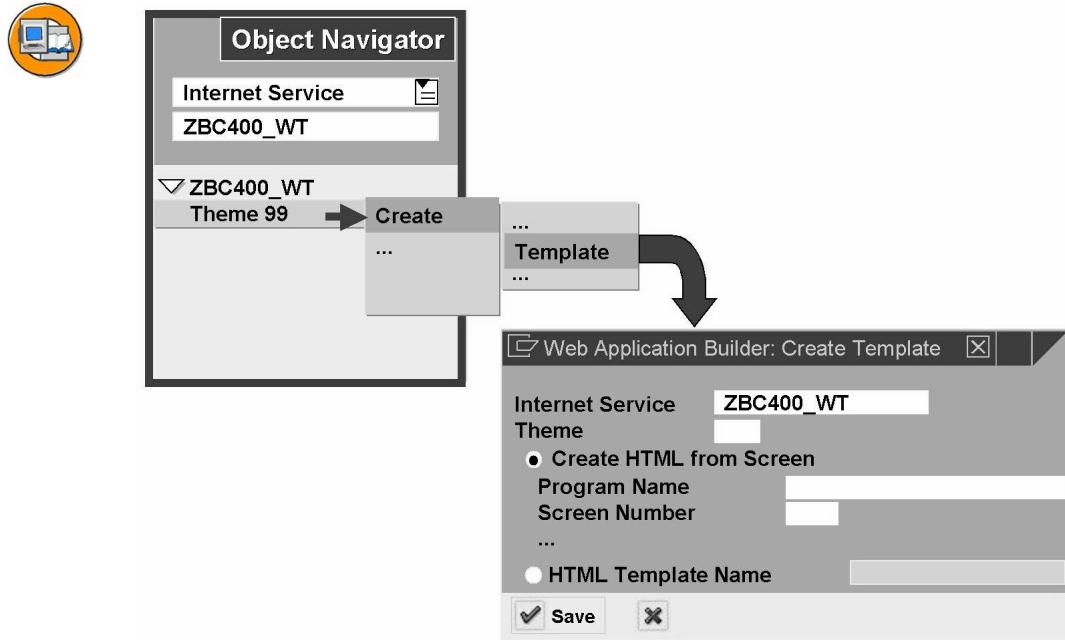


Figure 186: Creating Templates

3. To create templates for a service, choose *Create* → *Template* in the context menu for a theme.

In the dialog box for your service and the selected theme, enter the program name and screen number, which will serve as a basis for the template.

Save the template.

Continued on next page

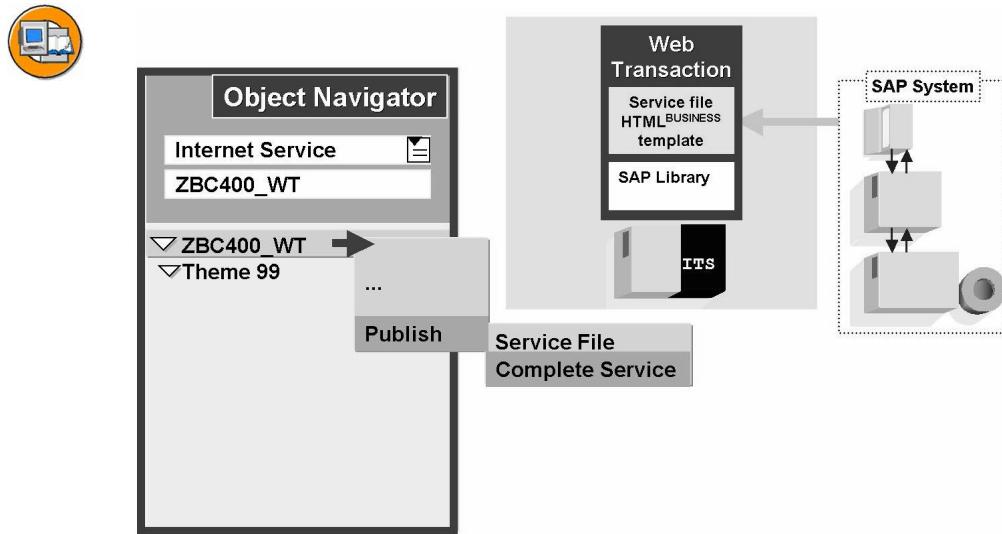


Figure 187: Publishing an Internet Service

4. You must save the Internet service in a file that the *ITS* can access. You can publish the service in the *Web Application Builder* in the *Object Navigator*:
Display the object list for the Internet service in the *Object Navigator*.
In the context menu for the service, choose *Publish* → *Complete Service*.



Hint: Your user-specific *ITS* settings must be correct before you can publish a service. You can check these settings in the *Object Navigator* by choosing *Utilities* → *Settings* followed by the *ITS* tab in the dialog box.

Continued on next page

Result

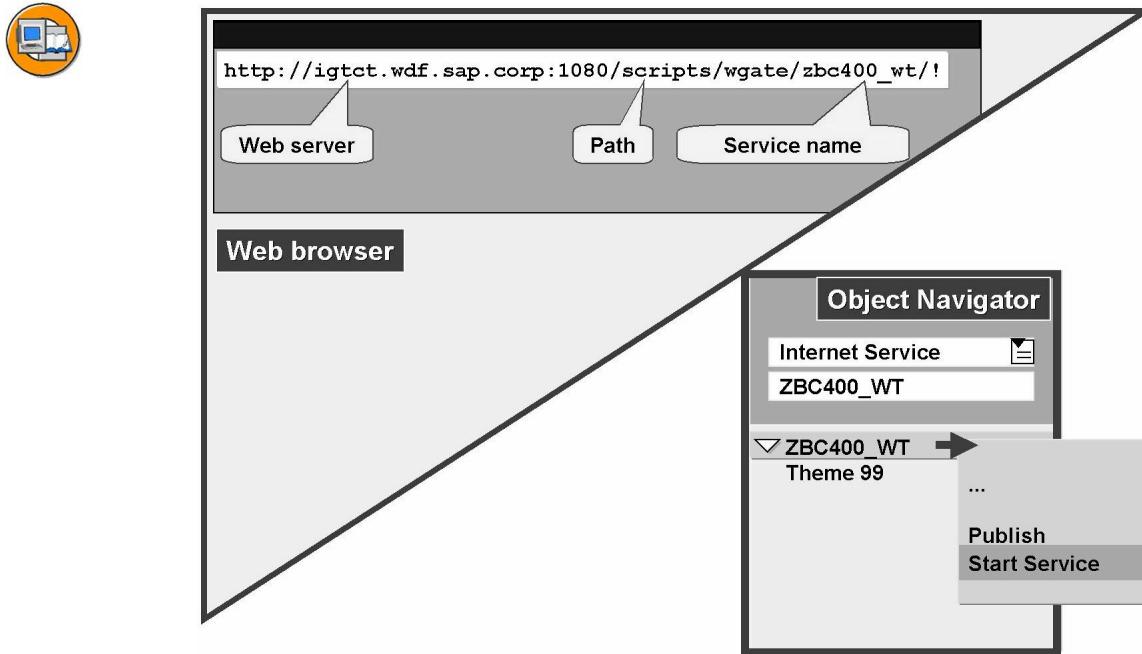


Figure 188: Testing the Web Transaction

To test a transaction after publishing it:

- Choose *Start Service* from the context menu in the Internet service object list.
- Enter the path `http://<server>[:<port>]/<path>/wgate/<service_name>/!` in a Web browser.



Lesson Summary

You should now be able to:

- Create a transaction code for a Web transaction
- Create an Internet service
- Publish an Internet service
- Execute a Web transaction using the Web browser



Unit Summary

You should now be able to:

- Create a transaction code for a Web transaction
- Create an Internet service
- Publish an Internet service
- Execute a Web transaction using the Web browser

Related Information

... Refer to the article "Web Application Builder" in the online documentation.

Unit 10

Cross-Program Modularization

Unit Overview

Refer to the lesson summaries for an overview of this unit.



Unit Objectives

After completing this unit, you will be able to:

- Acquire information on function modules
- Include function module calls in your program
- Create object instantiations and call methods
- Use the SAP Grid Control for displaying data in a fixed screen area
- Acquire information on BAPIs
- Include BAPI calls in your program

Unit Contents

Lesson: Working with Function Modules	332
Exercise 18: Using Function Modules	343
Lesson: Working with Methods	349
Procedure: Displaying the Contents of an Internal Table on a Screen with the SAP Grid Control.....	358
Exercise 19: Using Methods	363
Lesson: Working with BAPIs	369

Lesson: Working with Function Modules

Lesson Overview

In this lesson, you will learn how to search for function modules. Furthermore, you will call a function module for the encapsulation of a standard dialog and one for a database change in your program. You will also learn how to handle function module exceptions.



Lesson Objectives

After completing this lesson, you will be able to:

- Acquire information on function modules
- Include function module calls in your program

Business Example

You need to use standard functions in your program that are encapsulated in a function module.

Working with Function Modules

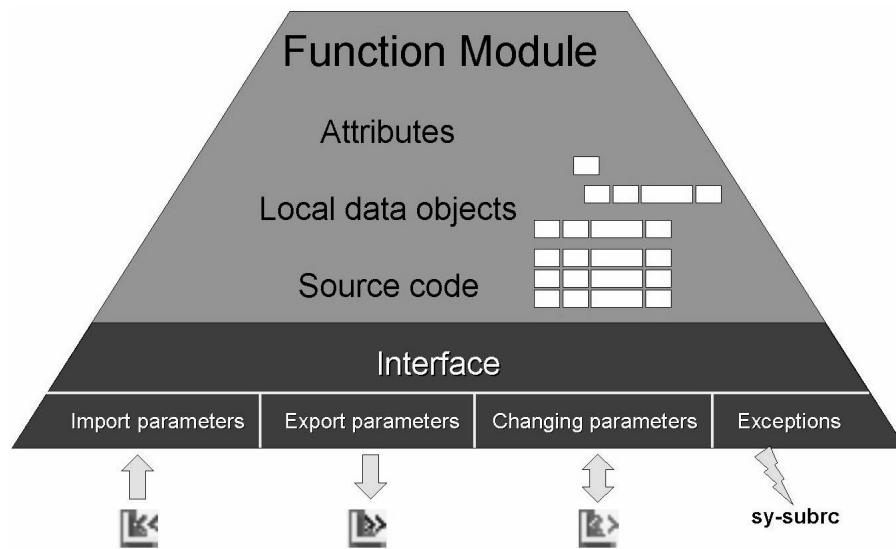


Figure 189: Function Module

Function modules are actively integrated modularization units with an interface. They can therefore be called primarily from within programs or other function modules. The interface of a function module can contain the following elements:

- **Import parameters** are parameters that are passed to the function module. Import parameters can also be characterized as optional.
- **Export parameters** are parameters that are received by function modules. Export parameters are always optional.
- **Changing parameters** are passed to the function module and can be changed by it.
- **Exceptions** provide information about error situations. If an exception is raised in a function module, processing of the function module is interrupted. You can set a return code in the calling program, if necessary.

In general, the interface parameters are assigned types from the *ABAP Dictionary*.

As is the case with subroutines, a function module can contain local type and data object definitions.

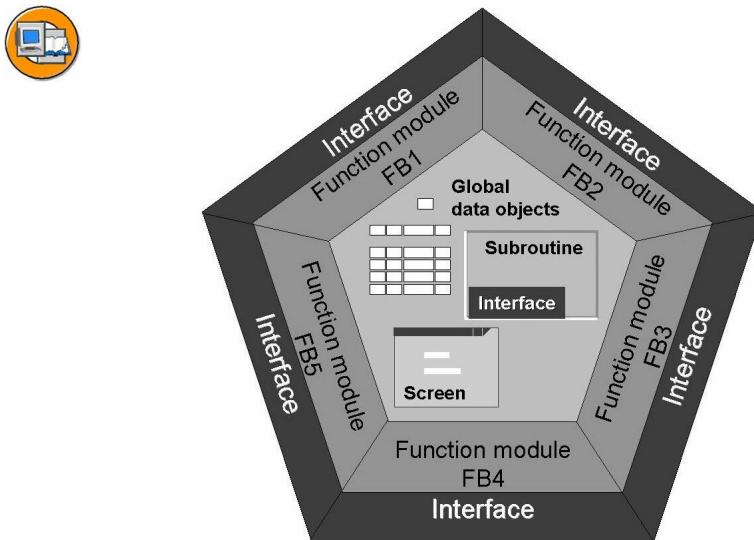


Figure 190: Function group

A **function group** represents the main program for function modules. Several function modules that operate on the same data content are therefore combined to form a function group.

The “life-span” of a function group corresponds to the “life-span” of the calling program.

For example, if an executable program calls a function module, its entire function group is loaded. It remains active until the executable program is completed.

A function group can contain the same components as an executable program. These include:

Data objects

These are global in relation to the function group, that is, they are visible to and changeable by all function modules within the group. The validity period is the same as for the function group.

Subroutines

These can be called from all function modules in the group.

Screens

These can be transmitted from all function modules in the group.

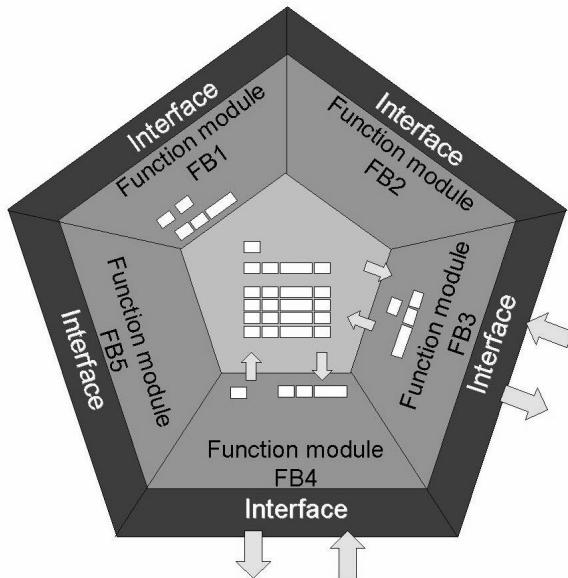


Figure 191: Data Flow Within a Function Group

The global data of the function module is retained until the program that contained the first call of a function module in the function group is finished.

Thus, if a function module that writes values to the global data is called, other function modules in the same function group can access this data when they are called by the program.

Seen from the outside, the global data is encapsulated, - that is, it is **not possible to access it directly**. Therefore, you must have function modules that allow "orderly" access from the outside.

The same applies to all the other components of the function group (screens, subroutines).

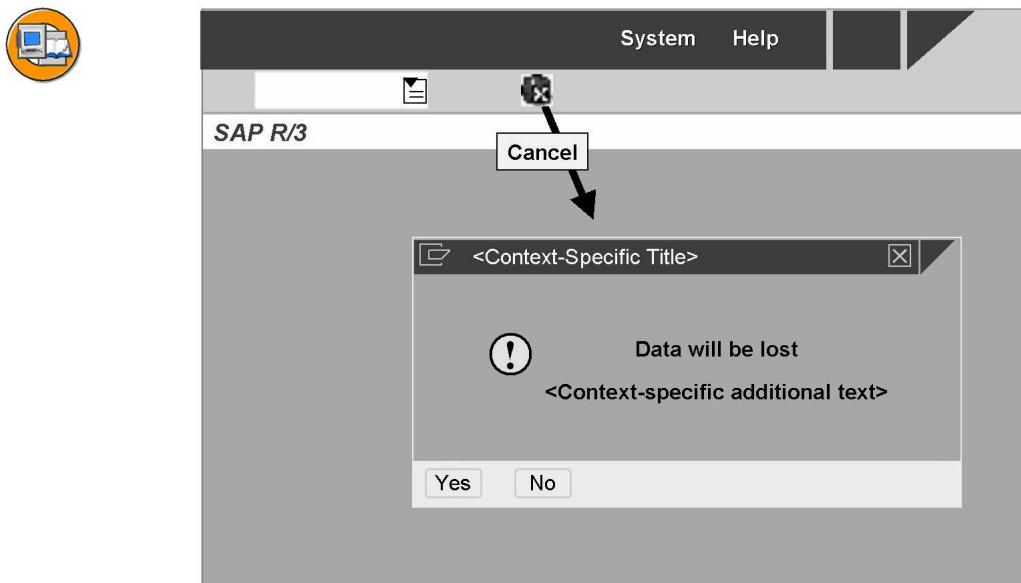


Figure 192: Example: Transmitting a Standard Dialog

In many ABAP programs, a standard dialog is transmitted as soon as the user selects *Cancel*. The processed screen always contains the text "Data will be lost". The title and the two lines following it are context-specific. The user can choose from either *Yes* or *No*.

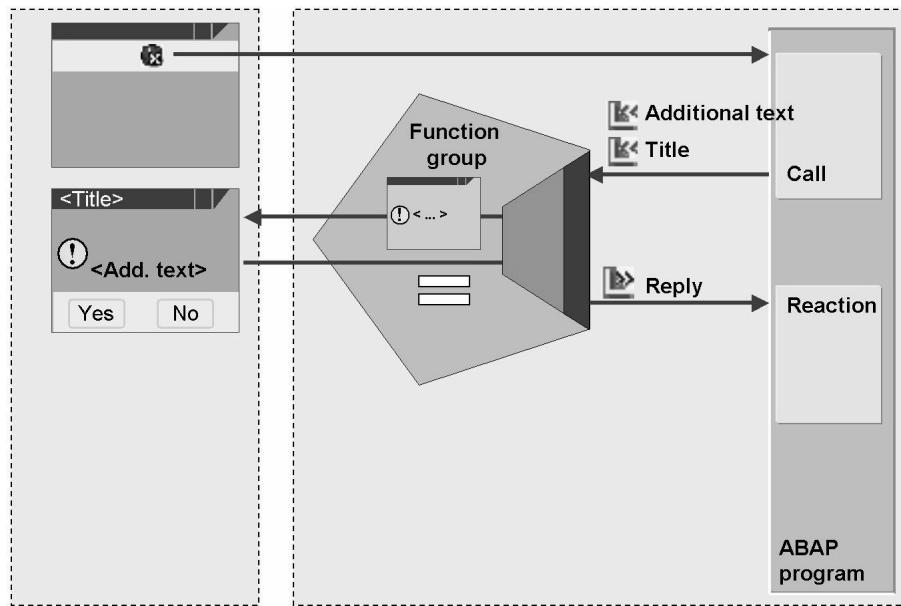


Figure 193: Integration of Standard Dialog

This dialog is encapsulated in a function module.

If you call this function module, the system can pass the additional and title texts to the function module using the import parameters.

The function module passes a screen of the function group that displays the text lines and the title and contains the Yes and No buttons.

An export parameter then tells the calling program which button the user has chosen.

You can integrate this dialog in your program in the following manner: Call the function module as soon as the user has chosen *Cancel*. After the call, evaluate the answer and react accordingly.



**Searching within a particular application component:
*Application hierarchy.***

Searching with the help of a program that uses the function module being searched for:

- **Search in the ABAP Editor for the CALL FUNCTION statement**
- **Change to debugging mode during program execution and set breakpoint at the CALL FUNCTION statement**
- **If the function module transmits a screen, display the program name and the screen number using <F1> and technical information, or navigate directly (double-click)**

Figure 194: Searching for Function Modules

During your search for function modules (and all other Repository objects), you will be supported by the *ABAP Workbench* in various ways.

In the *Repository Information System*, you can use parts of the technical name, such as ***CONFIRM***, or parts of the short text, such as ***Confirmation Prompt***, during your search.

If you know the application component, you should always use the *Application hierarchy*. You use this to call the packages in question.

If you know of a program that already uses the function module, you can search in this program for the call using the CALL FUNCTION statement.

In the case of a modal dialog box, the standard navigation method (*System → Status*) is not available for ascertaining the screen and program names. In such cases, position the cursor on a text field and choose **F1**. Afterwards, choose *Technical Information*.



Caution: Before you integrate into your program, make sure that the function module is released (*Attributes tab*).

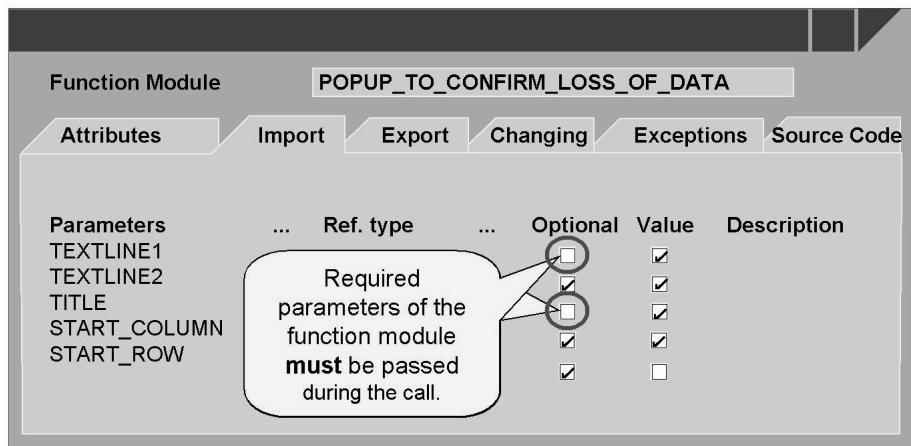


Figure 195: Analyzing a Function Module Interface

Once you have found a function module, you must find out more about its interface.

The required parameters must be supplied with data during the call. To find out which additional parameters are to be passed and when, refer to the documentation of the function module and the documentation on interface parameters.

You can also navigate directly to the source code of the function module.

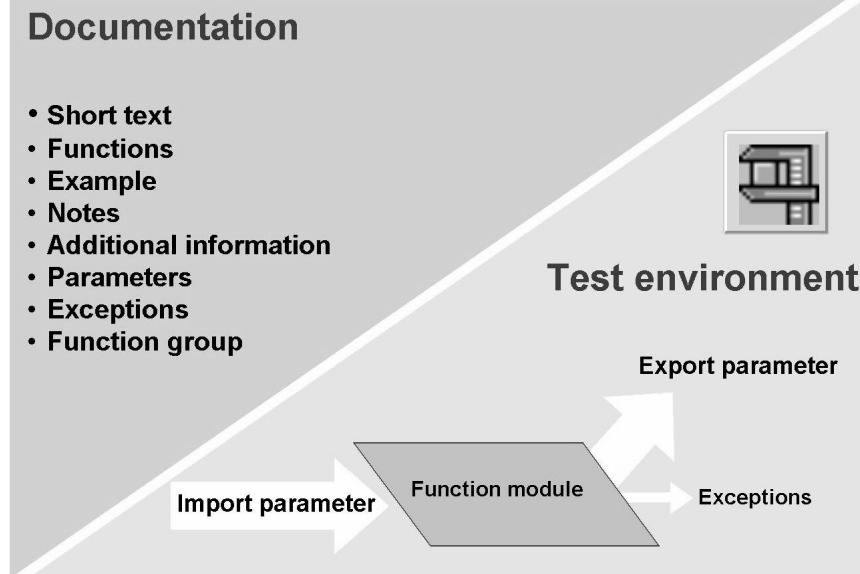


Figure 196: Documentation and Testing

You can test function modules using the test environment. In an input template, you can specify values for the **Import** and **Changing** parameters. You can also switch to debugging mode. The values of the **EXPORT** and **CHANGING** parameters are listed after execution.

If an exception occurs, the system records the raised exception, possibly with a message text.

The time required for executing the function module is displayed. These values are subject to the same restrictions as the *Runtime analysis*. You should therefore repeat the tests several times using the same data.

You can store test data in a test data directory and create test sequences.

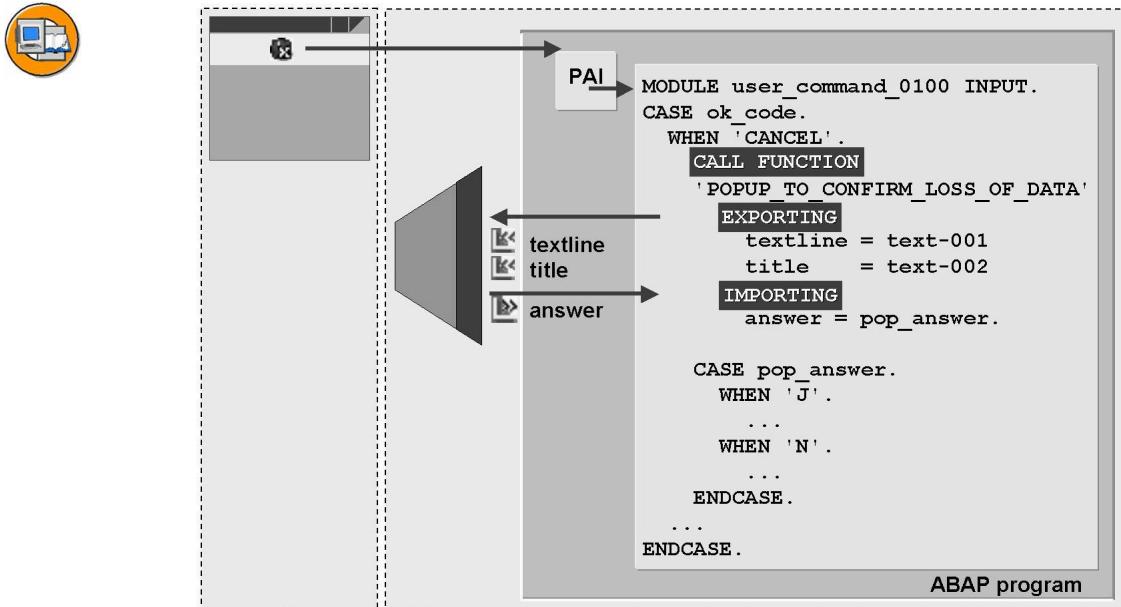


Figure 197: Calling a Function Module

You call function modules using the ABAP statement **CALL FUNCTION**. The name of the function module follows in capital letters enclosed in single quotation marks.

After **EXPORTING** has taken place, the system passes values to the **Import** parameters of the function module.

After **IMPORTING** has taken place, the function module returns the result using its **Export** parameters.

From the point of view of the calling program, the parameters that are passed to the function module are exported, and those passed from the function module to the program are imported.

On the left side of the parameter assignment list, the system displays the names of the interface parameters of the function module (formal parameters), while the right side of the screen displays the data objects of the calling program (actual parameters).

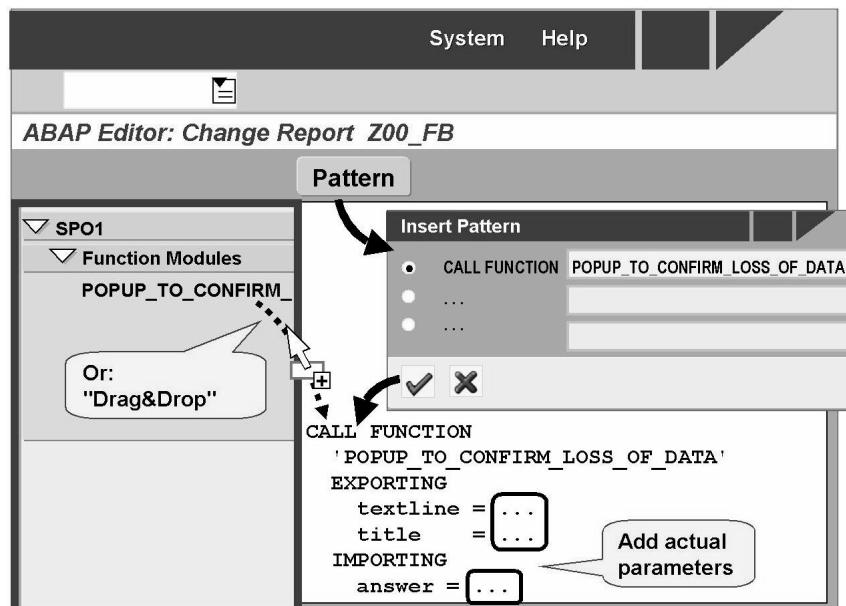


Figure 198: Implementing a Function Module Call

To program the function module call, always use the *Pattern* function.

Alternatively, you can display the function group in the navigation area and then place the function module call in the Editor area using "Drag&Drop".

The ABAP statement **CALL FUNCTION 'FUNCTION_MODULE_NAME'** is inserted, together with the entire list of interface parameters and exceptions, at the current cursor position in the source code of the program. Optional parameters and exceptions are inserted as comments.

In this way, you avoid typing errors and omissions.

Afterwards, you must fill in the current parameters and, if necessary, program the exception handling.

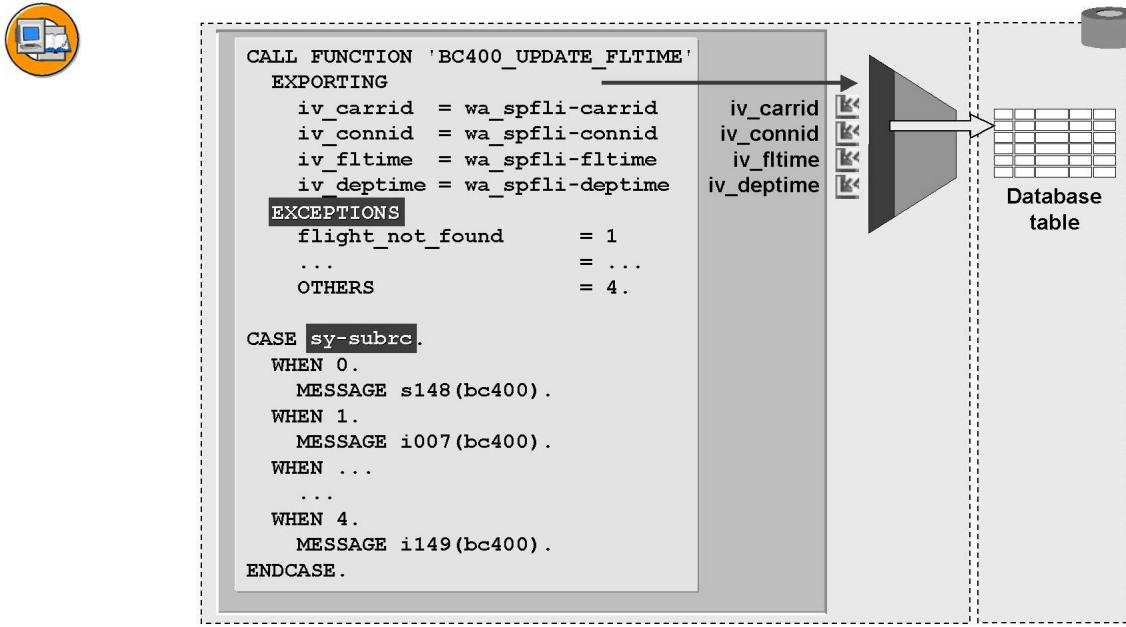


Figure 199: Handling Exceptions

With most function modules, exceptions are triggered in error situations. In such cases, the exceptions are assigned to the number values after EXCEPTIONS. If such an exception was raised by the function module, the respective value is placed in the system field SY-SUBRC. By evaluating this field, you can then react accordingly. In the example given here, suitable messages are transmitted.

If you do not wish to list the triggered exception under EXCEPTIONS, or you cannot do so, the exception is assigned to the OTHERS case, provided this addition is listed.

If no exception was triggered or neither the exception nor OTHERS was listed under EXCEPTIONS, the system sets SY-SUBRC to 0.

Exercise 18: Using Function Modules

Exercise Objectives

After completing this exercise, you will be able to:

- Search for function modules
- Incorporate function module calls in a program

Business Example

Change your program **ZBC400_##_DYNPRO** or the respective template as follows:

Whenever the *Cancel* function is chosen on a screen, a standard dialog that is encapsulated in a function module is to be processed. Additionally, when saving, the data record is to be updated in the database table as well.

Both requirements are to be implemented using function module calls.

Task 1

Standard dialog for *Cancel* function

1. Extend your program **ZBC400_##_DYNPRO** or copy the template **SAPBC400UDS_DYNPRO_D** to the new program **ZBC400_##_DYNPRO_E**.
2. Using one of the methods outlined during the course, search for the function module that encapsulates the standard dialog that is usually triggered when the user chooses *Cancel*.
3. Find out about the function module interface, read the documentation, and test the function module using the test environment.
4. In the GUI status of the screen, activate the *Cancel* function.
5. Extend the **USER_COMMAND_0100** module to evaluate the function code for the *Cancel* function. Then insert the function module call using the *Pattern* function of the *ABAP Editor*. React to the user input returned by the function module as follows:

If the user wants to cancel, set the next screen to 0 dynamically.

If the user does not want to cancel, branch to screen 100 dynamically.

Continued on next page

Task 2

Update the data record in the database table according to user input

1. Change the function code evaluation SAVE so that the function module **BC400_UPDATE_BOOK** is called.
2. Pass the changed data to the function module in the **correct type**.
3. If the change was carried out successfully, display success message 148 of message class BC400 and then return to the list.
If the change is not successful, display error message 149 of message class BC400.

Solution 18: Using Function Modules

Task 1

Standard dialog for *Cancel* function

1. Extend your program **ZBC400_##_DYNPRO** or copy the template **SAPBC400UDS_DYNPRO_D** to the new program **ZBC400_##_DYNPRO_E**.
 - a) Carry out this step as usual.
2. Using one of the methods outlined during the course, search for the function module that encapsulates the standard dialog that is usually triggered when the user chooses *Cancel*.
 - a) Carry out this step as described in the training material.
3. Find out about the function module interface, read the documentation, and test the function module using the test environment.
 - a) Carry out this step as described in the training material. The required import parameters are as follows: **TEXTLINE1** (maximum of 70 characters): First line of dialog box. **TITEL** (maximum of 35 characters): Title line of dialog box.
The optional import parameters are: **TEXTLINE2** (maximum of 70 characters): Second line of dialog box. **START_COLUMN** (type **SY-CUCOL**): Start column of dialog box. **START_ROW** (type **SY-CUROW**): Start line of dialog box. Export parameter: **ANSWER** (type **C**): User's input: **J** = User has confirmed the processing step. **N** = User has cancelled the processing step.
4. In the GUI status of the screen, activate the *Cancel* function.
 - a) Carry out this step as usual. The function code for the *Cancel* function is **RW**.
5. Extend the **USER_COMMAND_0100** module to evaluate the function code for the *Cancel* function. Then insert the function module call using the *Pattern* function of the *ABAP Editor*. React to the user input returned by the function module as follows:

If the user wants to cancel, set the next screen to 0 dynamically.

If the user does not want to cancel, branch to screen 100 dynamically.

 - a) See source code excerpt in the model solution.

Continued on next page

Task 2

Update the data record in the database table according to user input

1. Change the function code evaluation SAVE so that the function module **BC400_UPDATE_BOOK** is called.
 - a) See source code excerpt in the model solution.
2. Pass the changed data to the function module in the **correct type**.
 - a) See source code excerpt in the model solution.
3. If the change was carried out successfully, display success message 148 of message class BC400 and then return to the list.

If the change is not successful, display error message 149 of message class BC400.

- a) See source code excerpt in the model solution.

Result

Source code excerpt: **SAPBC400UDS_DYNPRO_E**, module **USER_COMMAND_0100**

```
MODULE user_command_0100 INPUT.

CASE ok_code.
  WHEN 'BACK'.
    SET SCREEN 0.
  WHEN 'RW'.
    CALL FUNCTION 'POPUP_TO_CONFIRM_LOSS_OF_DATA'
      EXPORTING
        textline1 = text-001
        titel      = text-002
      IMPORTING
        answer     = answer.
  CASE answer.
    when 'N'.
      LEAVE TO SCREEN 100.
    when 'J'.
      LEAVE TO SCREEN 0.
  ENDCASE.
WHEN 'SAVE'.
  MOVE-CORRESPONDING sdyn_book TO wa_sbook.
  CALL FUNCTION 'BC400_UPDATE_BOOK'
    EXPORTING
```

Continued on next page

```
        iv_book = wa_sbook
EXCEPTIONS
        book_not_found          = 1
        update_sbook_rejected   = 2
        book_locked              = 3
        currency_conversion_error = 4
        OTHERS                   = 5.

IF sy-subrc = 0.
MESSAGE s148(bc400).
SET SCREEN 0.
ELSE.
MESSAGE e149(bc400).
ENDIF.
ENDCASE.
ENDMODULE.           " USER_COMMAND_0100  INPUT
```



Lesson Summary

You should now be able to:

- Acquire information on function modules
- Include function module calls in your program

Lesson: Working with Methods

Lesson Overview

In this lesson, you will learn some basic object-oriented syntax elements of ABAP Objects using the SAP List Viewer as an example. In this context, references to instances of the standard classes CL_GUI_CUSTOM_CONTAINER and CL_GUI_ALV_GRID are created. The instance of the SAP List Viewer calls the method SET_TABLE_FOR_FIRST_DISPLAY to display a formatted internal table in the SAP List Viewer.



Lesson Objectives

After completing this lesson, you will be able to:

- Create object instantiations and call methods
- Use the SAP Grid Control for displaying data in a fixed screen area

Business Example

You want to use standard functions in your program that are encapsulated in global classes.

Working with Methods

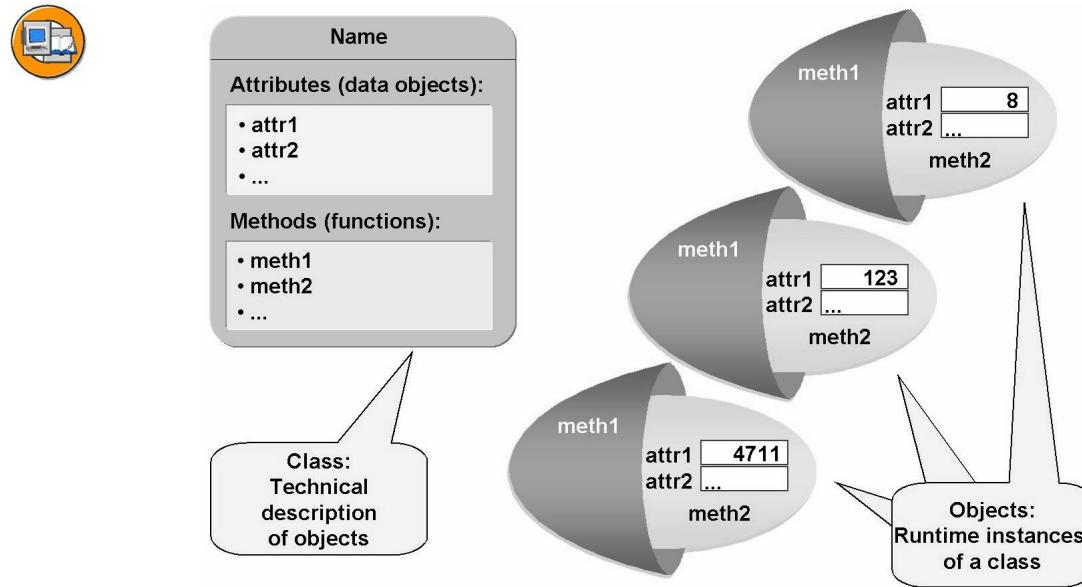


Figure 200: Classes and Objects

At first glance, a class is very much like a function group since it contains data objects (**attributes**) and functions (**methods**). These components can be protected against outside access. This is usually done with the attributes and some methods. Then the protected components can only be accessed by public methods.

Unlike function groups, classes can have **multiple instances**. This means that more than one runtime object can be created for each program in a class. In other words, a class contains the technical description of objects (instances). Therefore, all these objects have the same attributes and provide the same methods. However, they can be distinguished physically. Their attributes can have different features.

Classes can be defined either locally within an executable program or globally.

As of SAP R/3 4.6A, the ABAP runtime system provides support for standard object-oriented syntax elements. However, we cannot deal with these adequately during this course. For more details, refer to the SAP Online Library.

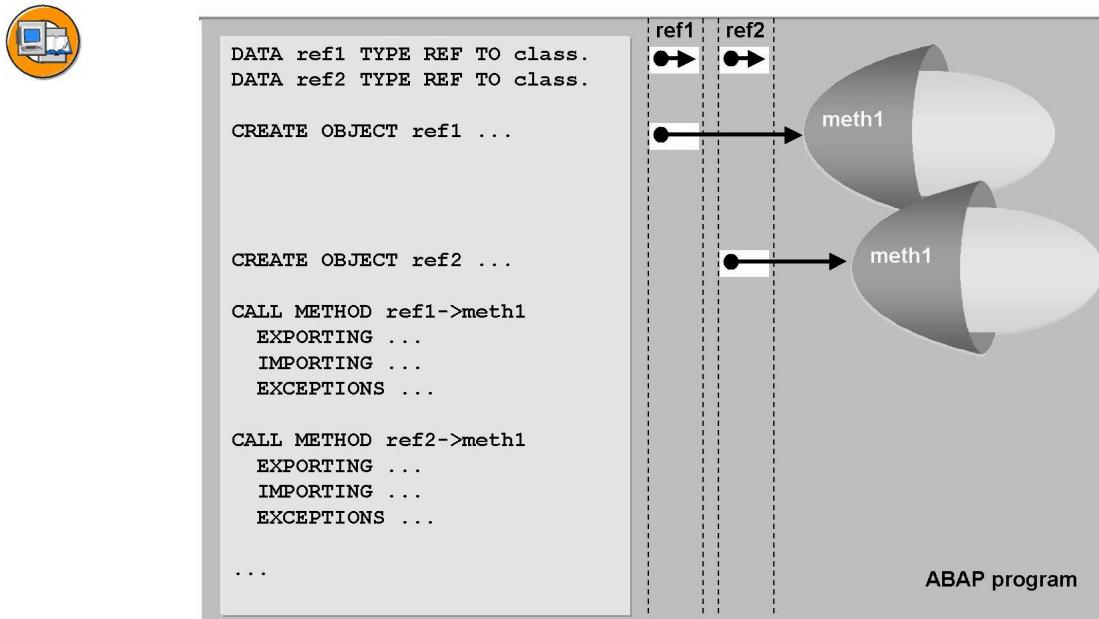


Figure 201: Creating Objects and Calling Methods

To create instances of classes and to be able to address them, you need **reference variables**. These are pointers to these instances. Reference variables are defined using:

`DATA reference_name TYPE REF TO class_name.`

When the program starts, the data objects are created in the memory. However, since no instances have been created yet, the reference variables are still empty.

At runtime, you can create any number of instances of classes using:

`CREATE OBJECT reference_name.`

If necessary, the import parameters of a special method (**constructor**) must be filled with values. This special method is executed immediately after the instance is created. This generally creates basic settings for the actual instance.

You call methods of an instance using:

`CALL METHOD reference_name->method_name.`

In contrast to a function module call, the function name is not sufficient here since, generally, several instances of a class exist for each program. Using the reference variable, here `reference_name`, followed by the object component selector `->`, you instruct a **specific** instance to execute **its** method.

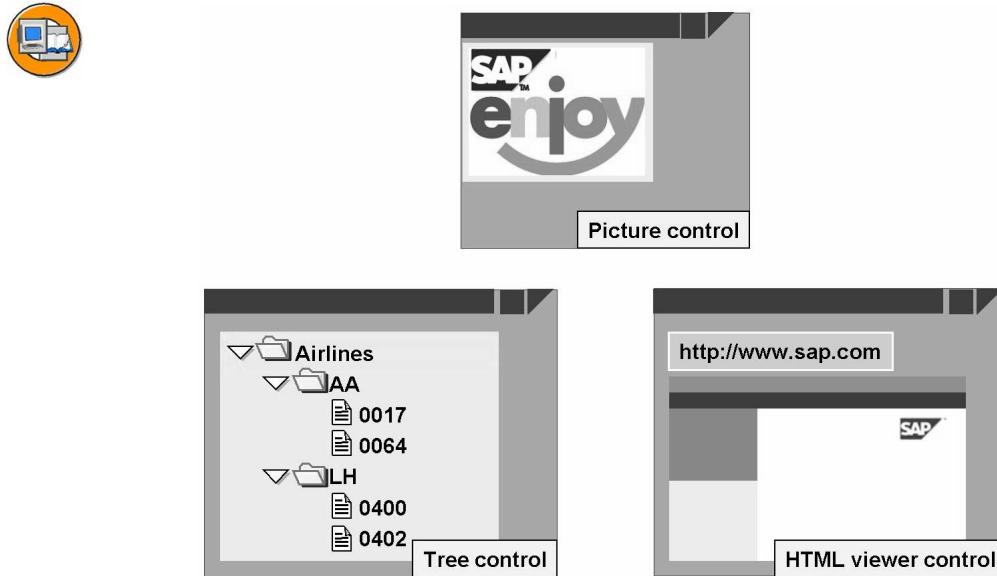


Figure 202: Examples of Standard Classes: EnjoySAP Controls

As of SAP R/3 4.6A, the ABAP Workbench offers several new **EnjoySAP controls** in order to move dialog functions from the application server to the presentation server. These controls are called on an object-oriented basis using classes - that is, the **SAP Control Framework**.

Using method calls, you regulate the controls from your ABAP program. The Control Framework sends your requests to the presentation server, where they are converted on a platform-specific basis.

You can give the user the option of triggering events on the presentation server. These are then converted by the Control Framework and you can react to them in your ABAP program.

For more details on the EnjoySAP controls and the Control Framework, refer to the SAP Online Library.

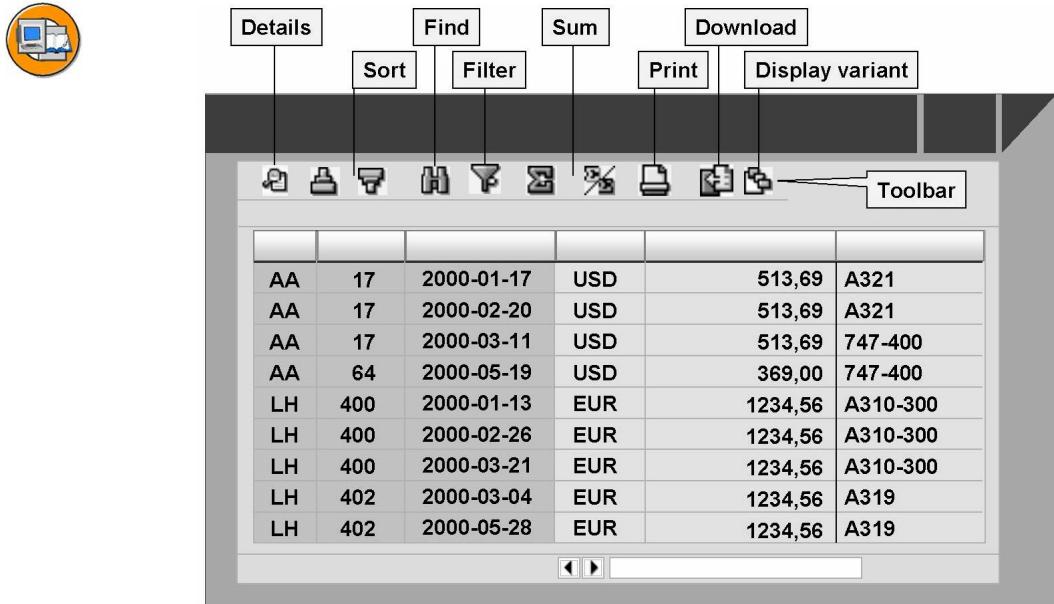


Figure 203: Application Example: SAP Grid Control

The SAP Grid Control is used for displaying non-hierarchical data in table form. Several user functions are already implemented:

On the screen, you can increase or decrease the column width or use the 'Optimum Width' function to adjust the column width to suit the data currently displayed. You can also change the column sequence by selecting a column and dragging it to a new position.

Standard functions are available in the application toolbar for the control. The detail view displays the fields of the row, on which the cursor is positioned, in a modal dialog box.

The sort function in the SAP Grid Control is available for as many columns as required. You can set complex sort criteria and sort columns in either ascending or descending order.

You can use the *Search* function to search for a string (generic search without *) in rows or columns within a selected area.

You can use the *Sum* function to request totals for one or more numeric columns. You can use the *Subtotal* function to structure control level lists: Select the columns (non-numeric fields only) that you want to use and the corresponding control level totals are displayed.

For *Print* and *Download* the whole list is always processed, not just the sections displayed on the screen.

You also have the option of setting display variants.

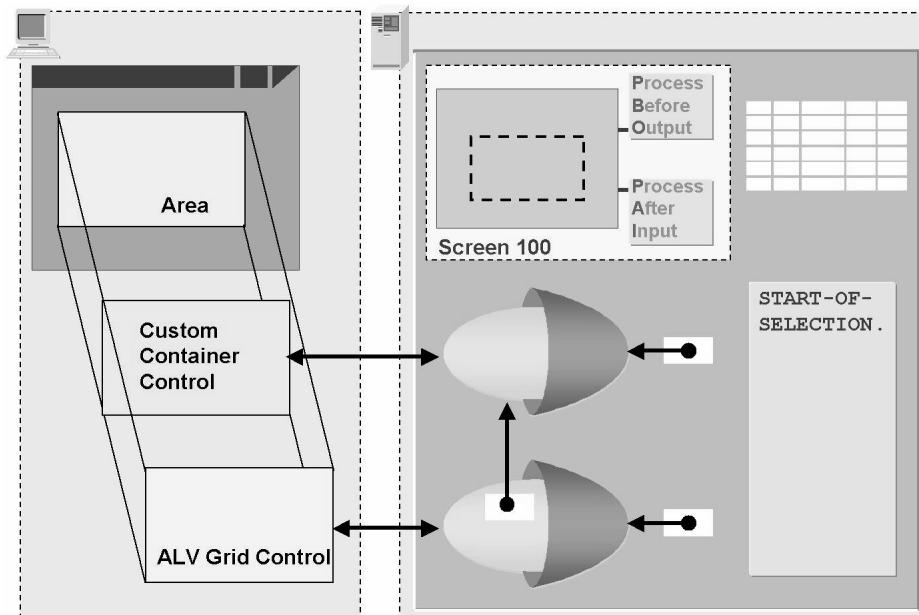


Figure 204: Program with SAP Grid Control

An Enjoy SAP control must always be embedded in an SAP container control. This manages the embedded controls and provides a physical area for displaying them. The container becomes the **parent** of the embedded control. Since containers are themselves controls, they can be nested within one another. In this way, you can work with controls in a flexible and modular manner.

To be able to use an SAP Grid Control in your program, you require at least one container and one SAP grid instance.

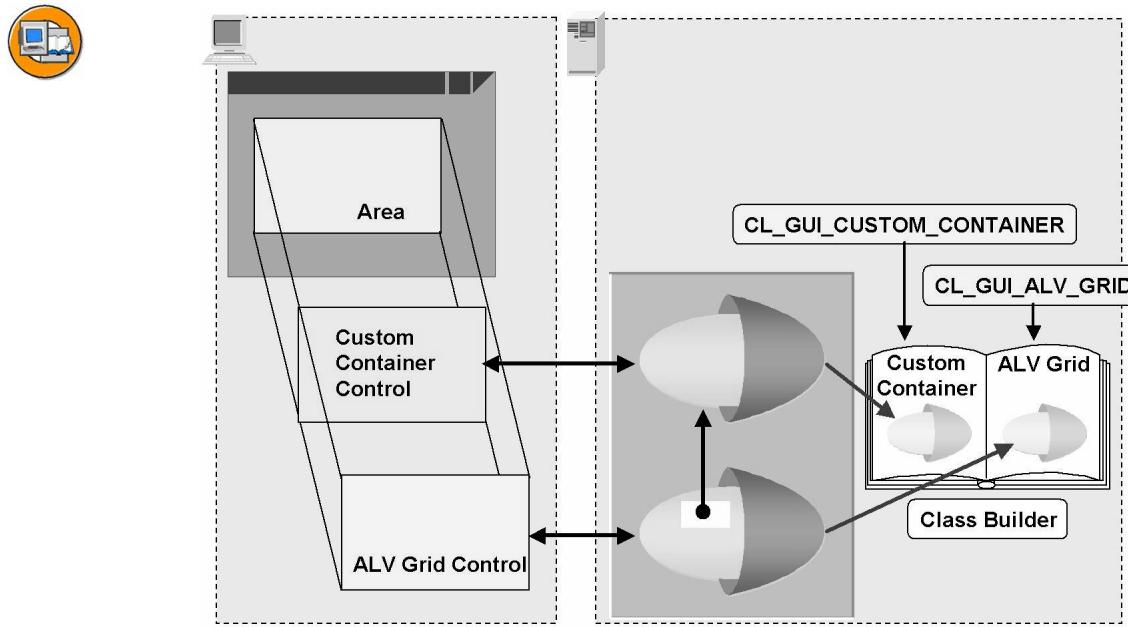


Figure 205: Objects and Classes for SAP Grid Control

To call the custom container controls and the SAP Grid Control, the following global classes (object types) are available in the *Class Builder*:

CL_GUI_CUSTOM_CONTAINER and **CL_GUI_ALV_GRID**

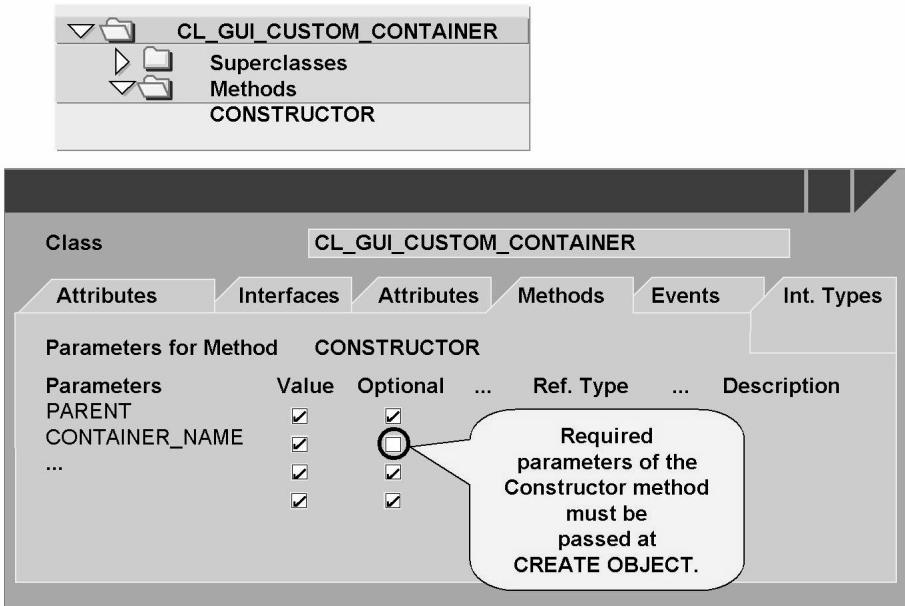


Figure 206: Custom Control Container: Constructor Method

To obtain detailed information on a global class, you can navigate to the *Class Builder*:

Display the object list of the class in the navigation area of the *Object Navigator*. Then display the entire class or individual components in the *Editor*. Alternatively, you can also go to the *Class Builder* by double-clicking the class names from within an ABAP program. Depending on the display mode, you can then display detailed information for the interface parameters by choosing *Parameters* or *Signature*.

The global class **CL_GUI_CUSTOM_CONTAINER** contains the **CONSTRUCTOR** method. This is the constructor mentioned above. When you create an instance in a program using **CREATE OBJECT**, you must pass at least the required parameter **CONTAINER_NAME**. The name of a custom container area on a screen must be passed to this parameter.

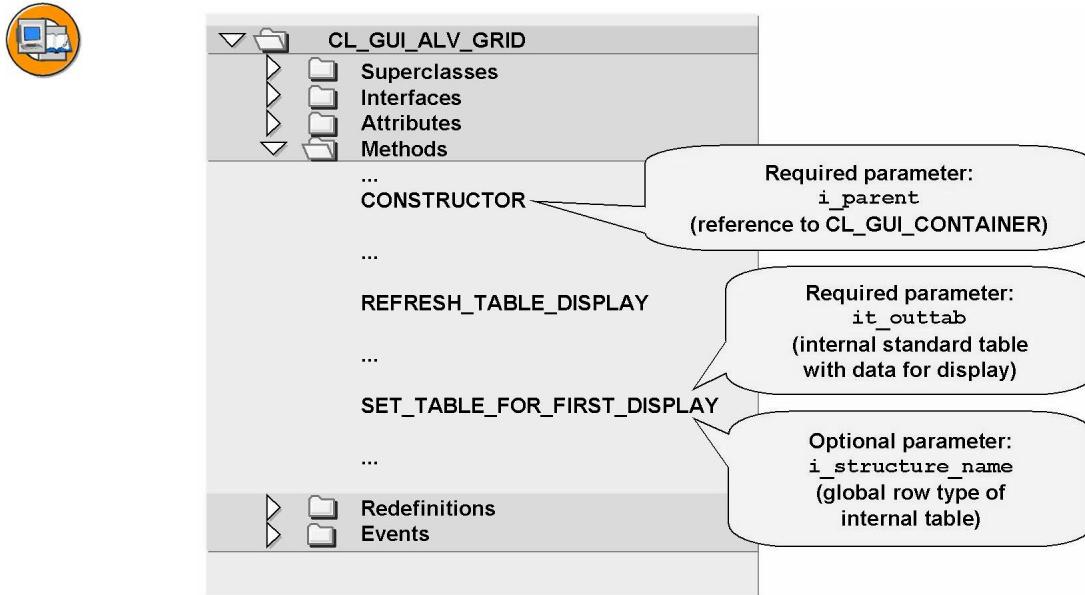


Figure 207: SAP Grid Control: Important Methods

The global class **CL_GUI_ALV_GRID** contains a large number of methods. To display the contents of an internal table with an SAP Grid Control, all you need to do is to use the following three methods:

CONSTRUCTOR

The pointer to the container control instance must be passed to the constructor.

SET_TABLE_FOR_FIRST_DISPLAY

The internal table with the data to be displayed must be passed to the parameter `IT_OUTTAB`. This must be a standard table so that the user can make use of the sort functions.

Furthermore, technical information is required for editing the grid columns. The easiest approach is to use a flat structure or a transparent table — that is, a global type — as the line type for the internal table. In this case, you need only pass the name of this line type to the `I_STRUCTURE_NAME` parameter. Alternatively, you can also set up a field catalog and pass it to `IT_FIELDCATALOG`.

REFRESH_TABLE_DISPLAY

You only need to call this method if you have changed the contents of the internal table after the first display.



Displaying the Contents of an Internal Table on a Screen with the SAP Grid Control

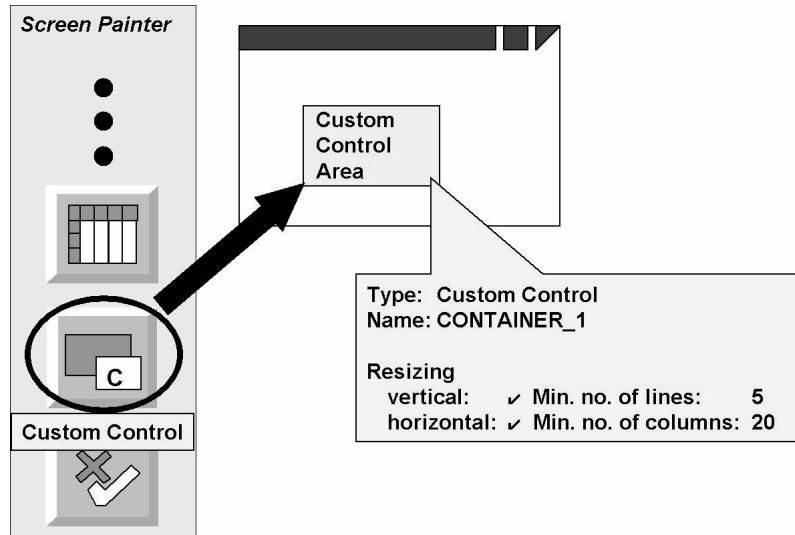


Figure 208: Creating a Screen Element "Custom Control Area"

1. To reserve an area of the screen for an EnjoySAP control, create a custom container area using the *Screen Painter* (choose *Layout*).

On the toolbar to the left of the editing area, choose *Custom Control* and specify the size and position of the area on the screen.

Proceed as follows: Define the top left corner of the area by holding down the left mouse button. Drag the cursor diagonally down to the right to where you want the bottom right corner. Once you release the mouse button, the bottom right corner is fixed in position.

You can change the size and position of the area at any time using the “drag symbols” that appear at the edge of the area.

Enter a name for the new screen element (here: **CONTAINER_1**)

Use the *Resizing Vertical* and *Resizing Horizontal* attributes to specify whether or not the area of the custom control should be resized when the main screen is resized. When you set these attributes, you can also set the minimum values for the area using the additional attributes *Min. Rows* and *Min. Columns*. The upper limit is defined by the size of the area.

Continued on next page



```

DATA gdt_spfli TYPE sbc400_t_spfli.

DATA: container_r TYPE REF TO cl_gui_custom_container,
      grid_r      TYPE REF TO cl_gui_alv_grid.

DATA ok_code TYPE sy-ucomm.

START-OF-SELECTION.
* fill internal table ...

CALL SCREEN 100.

```

Diagram on the right shows variable declarations:

- gdt_spfli**: A grid icon with 4 columns and 3 rows.
- container_r**: A pointer icon pointing to a large grey oval labeled **container**.
- grid_r**: A pointer icon pointing to a smaller grey oval labeled **grid**.
- ok_code**: A pointer icon pointing to a white rectangle.

Figure 209: Defining Reference Variables

- Two reference variables are required in the ABAP program:

The reference variable that is to point to the container control instance is called **CONTAINER_R** in this case, and is typed with the global class **CL_GUI_CUSTOM_CONTAINER**.

The reference variable that is to point to the grid control instance is called **GRID_R** in this case, and is typed with the global class **CL_GUI_ALV_GRID**.



```

MODULE create_control OUTPUT.
  IF container_r IS INITIAL.

    CREATE OBJECT container_r
      EXPORTING container_name = 'CONTAINER_1'.

    CREATE OBJECT grid_r
      EXPORTING i_parent = container_r.

  ...
ENDIF.
ENDMODULE.

```

Diagram on the right shows object creation:

- A pointer icon **container_r** points to a large grey oval labeled **container**.
- A pointer icon **grid_r** points to a smaller grey oval labeled **grid**.

Figure 210: Creating Instances

Continued on next page

3. You create control instances using the CREATE OBJECT statement. Make sure you let the *ABAP Editor* generate the statement (*Pattern statement*). In this way, you avoid typing errors, mistakes, and omissions. As is the case with function module calls, the complete interface is implemented in your program. Optional parameters are entered as comments. CREATE OBJECT is the interface of the Constructor method.

The control processing must be complete **before** the corresponding screen is displayed. In addition, it should be linked modularly to the processing of the corresponding screen. Therefore, we recommend that you implement a PBO module for this purpose.

Creating an instance for the controls should take place only once, before the screen is **first** displayed. This is easily done by querying the content of one of the two reference variables.

Since here the instantiation of the container control is implemented in the corresponding PBO module, you need only pass the name of the custom container area to the interface. In other cases, you can also enter the program name and the screen number.

When you instantiate the grid control, all you need to do is to pass the pointer to the container instance. In this way, the container control manages the technical link of the SAP Grid Control to the screen.



```
MODULE create_control OUTPUT.
  IF container_r IS INITIAL.

  ...

  CALL METHOD
    grid_r->set_table_for_first_display
      EXPORTING i_structure_name = 'SPFLI'
      CHANGING  it_outtab      = gdt_spfli.

  ...

END IF.
ENDMODULE .
```

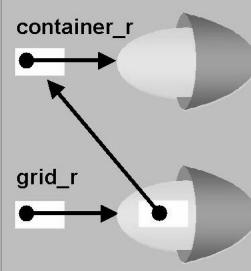


Figure 211: Calling Methods

4. To pass the contents of the internal table and the line type to the SAP Grid Control, you must call the method **SET_TABLE_FOR_FIRST_DISPLAY**. Pass the name of the internal table to the **IT_OUTTAB** parameter.

Continued on next page

Here the internal table has been typed with the transparent table **SPFLI** as the line type. Therefore, you need only pass this name to the **I_STRUCTURE_NAME** parameter. Additionally, you can also define basic settings for displaying the SAP Grid Control on the screen.

If the **contents of the internal table** change during the program, you must call the **REFRESH_TABLE_DISPLAY** method **before** the next screen is displayed.

Exercise 19: Using Methods

Exercise Objectives

After completing this exercise, you will be able to:

- Display simple lists using an SAP Grid Control

Business Example

Write a program that displays the contents of the database table **SPFLI** using an SAP Grid Control.

Task 1

Preparing a screen to include a custom container control

1. Copy the program **SAPBC400RUT_ALV_GRID** to the new name **ZBC400_##_ALV_GRID**.
2. Get to know the program.



Hint: In the template, there is an internal table that matches the database table **SPFLI** and also a screen with the number 100.

3. Fill the internal table with data records from the data table **SPFLI** using an array fetch.
4. Create a container control area on the screen 100 (name proposal: **CONTAINER_1**).

Task 2

Instantiating controls and calling a method

1. Define two reference variables, one for the **CL_GUI_CUSTOM_CONTAINER** class and one for the **CL_GUI_ALV_GRID** class.
2. At **PROCESS BEFORE OUTPUT** of screen 100, call a module in which to create the objects using the **CREATE OBJECT** statement. Also, fill the required parameters for the relevant constructors.

Continued on next page

3. At **PROCESS BEFORE OUTPUT**, call the **SET_TABLE_FOR_FIRST_DISPLAY** method, pass the name of the line type of the internal table to the **I_STRUCTURE_NAME** parameter, and pass the internal table to the parameter **IT_OUTTAB**.
4. Use a query to ensure that the object is only generated and the method is only called when **PROCESS BEFORE OUTPUT** runs for the first time.

Solution 19: Using Methods

Task 1

Preparing a screen to include a custom container control

1. Copy the program **SAPBC400RUT_ALV_GRID** to the new name **ZBC400_##_ALV_GRID**.
 - a) Carry out this step as usual.
2. Get to know the program.



Hint: In the template, there is an internal table that matches the database table **SPFLI** and also a screen with the number 100.

- a) Carry out this step as usual.
3. Fill the internal table with data records from the data table **SPFLI** using an array fetch.
 - a) See source code excerpt in the model solution.
4. Create a container control area on the screen 100 (name proposal: **CONTAINER_1**).
 - a) Carry out this step as described in the training material.

Task 2

Instantiating controls and calling a method

1. Define two reference variables, one for the **CL_GUI_CUSTOM_CONTAINER** class and one for the **CL_GUI_ALV_GRID** class.
 - a) See source code excerpt in the model solution.
2. At **PROCESS BEFORE OUTPUT** of screen 100, call a module in which to create the objects using the **CREATE OBJECT** statement. Also, fill the required parameters for the relevant constructors.
 - a) See source code excerpt in the model solution.

Continued on next page

3. At PROCESS BEFORE OUTPUT, call the SET_TABLE_FOR_FIRST_DISPLAY method, pass the name of the line type of the internal table to the I_STRUCTURE_NAME parameter, and pass the internal table to the parameter IT_OUTTAB.
 - a) See source code excerpt in the model solution.
4. Use a query to ensure that the object is only generated and the method is only called when PROCESS BEFORE OUTPUT runs for the first time.
 - a) See source code excerpt in the model solution.

Result

Source code excerpt: SAPBC400RUS_ALV_GRID

Addition to declaration:

```
DATA:
  container_r TYPE REF TO CL_GUI_CUSTOM_CONTAINER,
  grid_r      TYPE REF TO CL_GUI_ALV_GRID.
```

Addition to START-OF-SELECTION:

```
START-OF-SELECTION.

* fill internal table
  SELECT * FROM spfli
    INTO TABLE gdt_spfli.
*
* WHERE ...

CALL SCREEN 100.
```

Flow logic for screen 100:

```
PROCESS BEFORE OUTPUT.
MODULE status_0100.
MODULE create_control.
*
PROCESS AFTER INPUT.
MODULE copy_ok_code.
MODULE user_command_0100.
```

Continued on next page

Source code excerpt: Module CREATE_CONTROL

```
MODULE create_control OUTPUT.
  IF container_r IS INITIAL.
    CREATE OBJECT container_r
      EXPORTING container_name = 'CONTAINER_1'.

    CREATE OBJECT grid_r
      EXPORTING i_parent = container_r.

    CALL METHOD grid_r->set_table_for_first_display
      EXPORTING i_structure_name = 'SPFLI'
      CHANGING it_outtab      = gdt_spfli.
  ENDIF.
ENDMODULE.                                     " CREATE_CONTROL OUTPUT
```



Lesson Summary

You should now be able to:

- Create object instantiations and call methods
- Use the SAP Grid Control for displaying data in a fixed screen area

Lesson: Working with BAPIs

Lesson Overview

This lesson deals with the significance of BAPIs. You will also learn how to find information on a BAPI in the BAPI Explorer.



Lesson Objectives

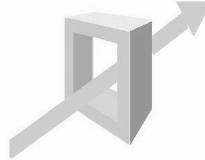
After completing this lesson, you will be able to:

- Acquire information on BAPIs
- Include BAPI calls in your program

Business Example

You want to use standard functions in your program that are encapsulated in a BAPI.

Working with BAPIs



A BAPI is a well-defined interface to processes and data of a business application system, implemented as a method of a business object in the Business Object Repository (BOR).



Figure 212: Business Application Programming Interfaces

A BAPI can be viewed as the “door” to the SAP system - that is, the SAP system thus provides access to business data and processes from the outside.

Each object in the *BOR* can have several methods, one or more of which can be implemented as BAPIs.

BAPIs usually exist for basic functions of a business object, such as:

- Creating objects
- Retrieving the attributes of an object
- Changing the attributes of an object

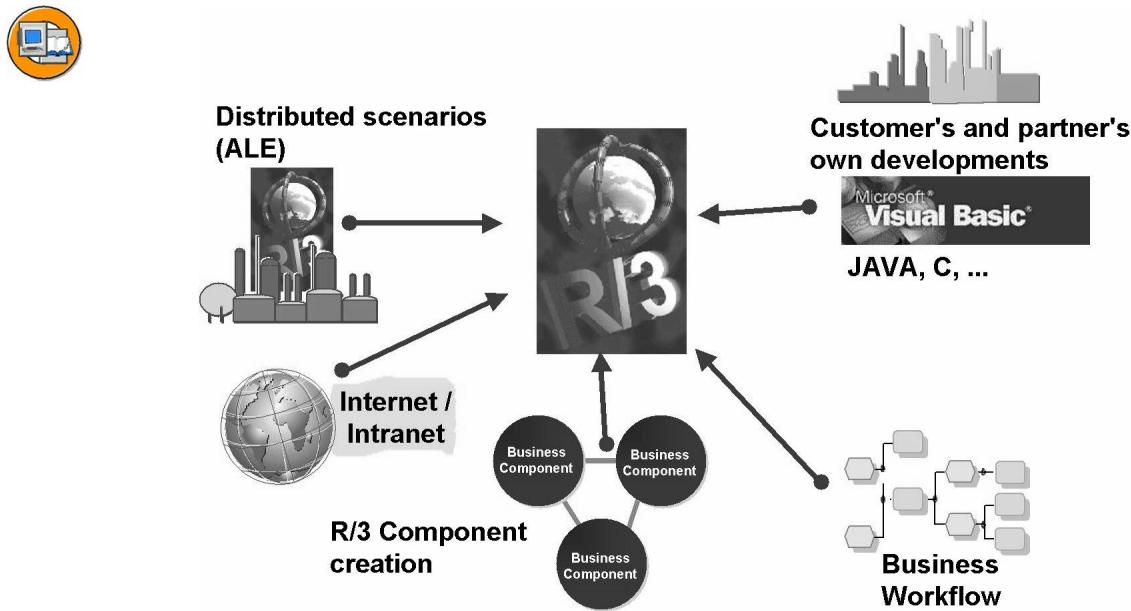


Figure 213: BAPI Use

There are standard methods in the form of BAPIs with standardized names. Some of the most important standard BAPIs are listed here.

Standard BAPIs



GetList

Returns a list of object key fields that satisfy specific selection criteria (search function)

GetDetail

Returns detailed information (attributes) for an object (the complete key must be specified)

Create, Change, Delete, Cancel

Allows you to create, change, and delete objects

AddItem, RemoveItem

Allows you to add and remove subobjects (for example, item for an order)

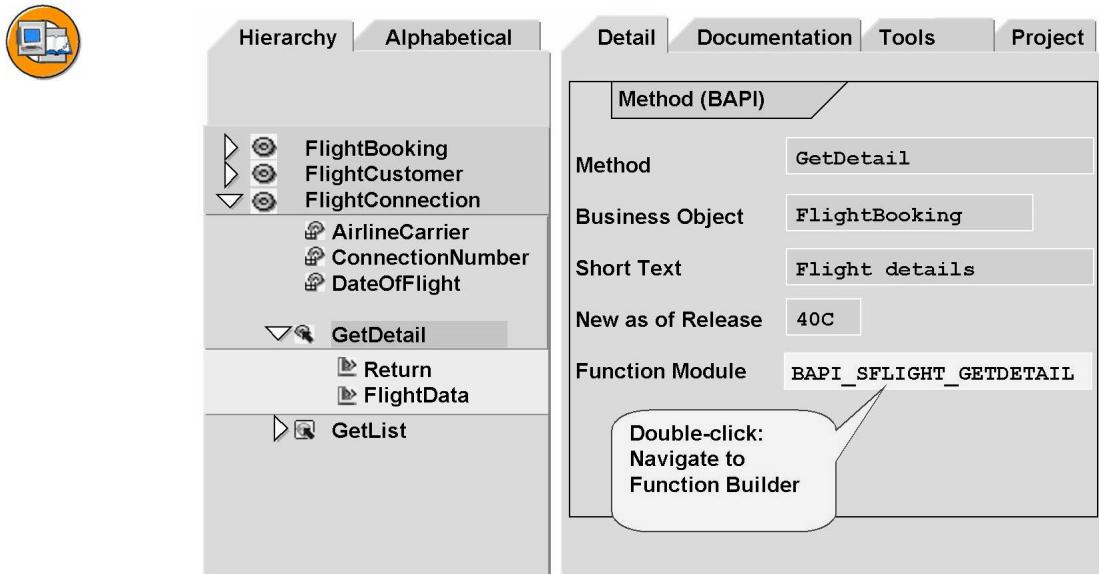


Figure 214: Finding BAPI Function Modules

For the time being, BAPIs are technically implemented in the form of function modules.

In the SAP Easy Access menu, choose *Tools* → *Business Framework* → *BAPI Explorer* to call the overview and search tool for business objects. Using the *BAPI Explorer*, you can navigate to the function module display for the selected BAPI in the *Function Builder*.

- Select the BAPI in the hierarchy section.
- In the detailed display window, choose the *Detail* tab.
- By double-clicking the name of the function module, you can display it in the *Function Builder*.

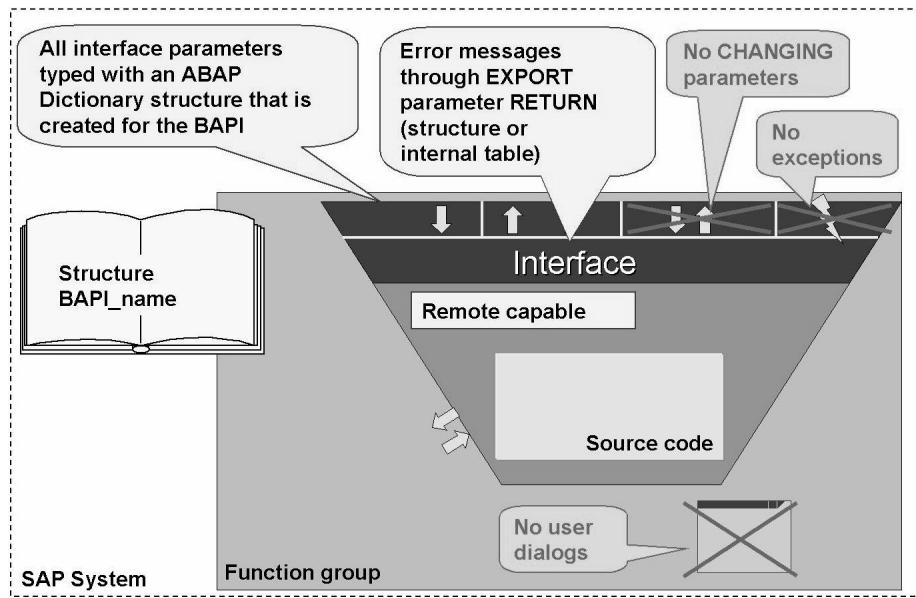


Figure 215: Attributes of a BAPI Function Module

Function modules for BAPIs must fulfill the following technical requirements:

- Naming convention **BAPI_<business_object_name>_<method_name>**
 - Remote capability
 - Contain neither user dialogs nor messages
 - Interface parameters are typed with components of structures from the *ABAP Dictionary* that were created for this BAPI
 - Naming convention **BAPI_<structure_name>**
 - Contain no **changing** parameters
 - Do not trigger any exceptions
- Errors are reported to the caller using special export parameters.

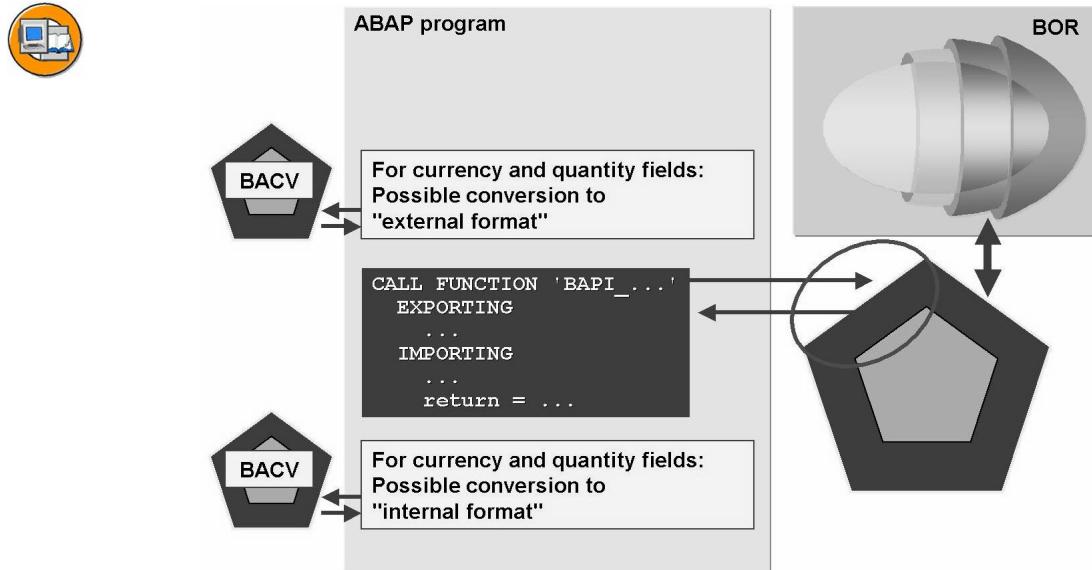


Figure 216: Calling a BAPI Function Module from an ABAP Program

If you would like to use a BAPI within *SAP R/3*, you can directly call the function module containing it. Note the restrictions already mentioned.



Hint: BAPI interfaces are created according to the requirements for the “external” call - that is, from an external system. Therefore, amounts are expected in an external format with 4 or 9 decimal places. The amounts must be transferred to the interface in an appropriately converted format, even if the corresponding currency has no decimal places.

For this conversion you can use function modules from the BACV function group (package **SBF_BAPI**).



Lesson Summary

You should now be able to:

- Acquire information on BAPIs
- Include BAPI calls in your program



Unit Summary

You should now be able to:

- Acquire information on function modules
- Include function module calls in your program
- Create object instantiations and call methods
- Use the SAP Grid Control for displaying data in a fixed screen area
- Acquire information on BAPIs
- Include BAPI calls in your program

Related Information

... Refer to the online documentation for each tool.

Unit 11

Software Logistics and Software Adjustment

Unit Overview

Refer to the lesson summaries for an overview of this unit.



Unit Objectives

After completing this unit, you will be able to:

- Explain the concepts for orderly development
- Use the tools for orderly development
- Explain the terms original, copy, correction, repair, and modification
- Describe how to avoid modifications to SAP objects

Unit Contents

Lesson: Outlook: Organization of Development and Transport.....	378
Lesson: Outlook: Enhancing the Standard Software	386

Lesson: Outlook: Organization of Development and Transport

Lesson Overview

This lesson provides an overview of the *Transport Organizer*, which supports work groups and project-oriented software development.



Lesson Objectives

After completing this lesson, you will be able to:

- Explain the concepts for orderly development
- Use the tools for orderly development

Business Example

You need to carry out your developments in an organized and coordinated fashion as part of orderly projects.

Organization of Development and Transport

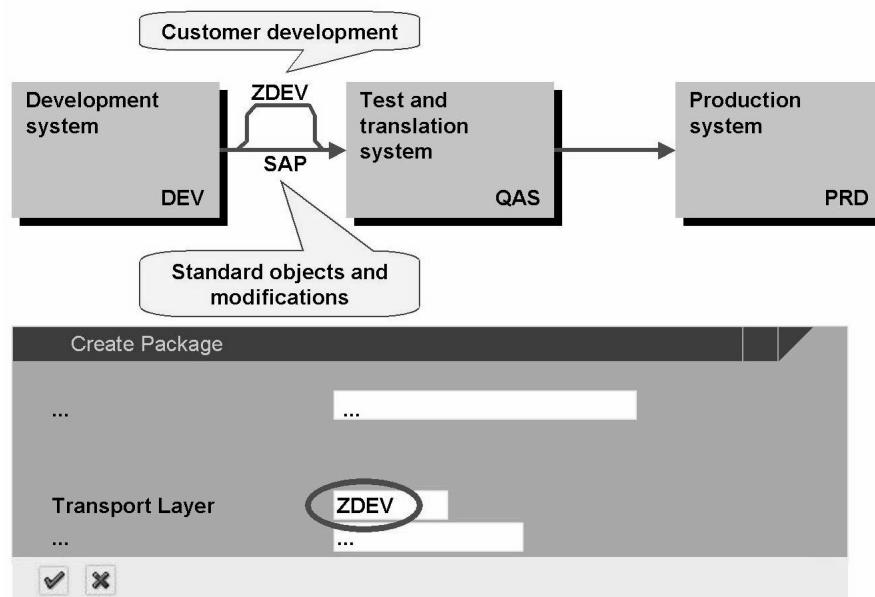


Figure 217: Assigning Transport Routes

If you create a package for your own developments, the system automatically assigns the correct transport layer. This sets the transport route for the Repository objects that are assigned to this package.

You can check the current system configuration in the *Transport Management System* (*Tools → Administration → Transport → Transport Management System*).

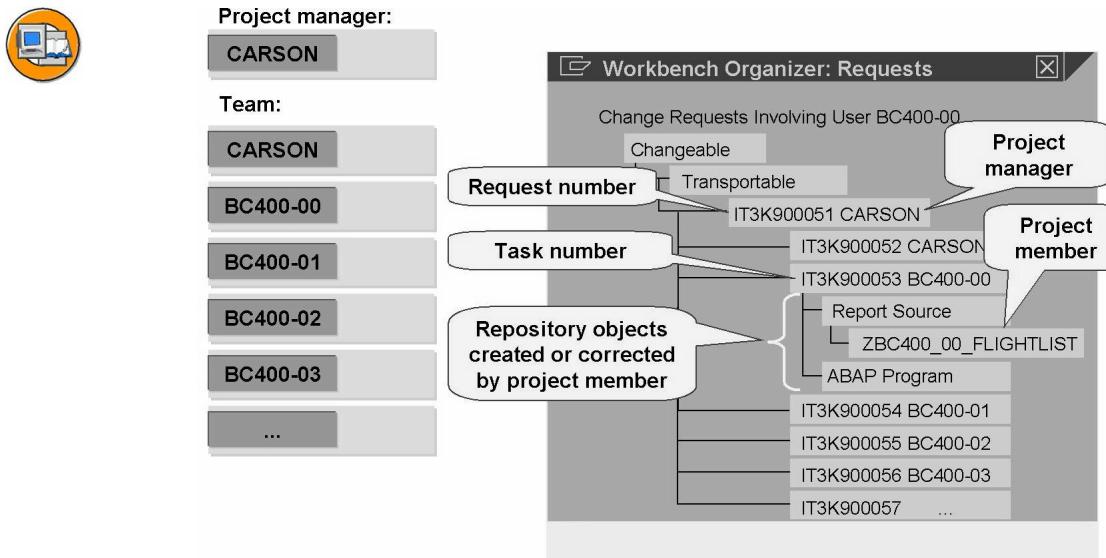


Figure 218: Joint Transport Assignment

At the beginning of a development project, the **project manager** creates a **change request**. The project manager assigns all project team members to the change request. The *Transport Organizer* then creates a **task** for each project developer.

Whenever an employee assigns a repository object to this change request, the repository object is automatically entered as that employee's **task**. Thus, all repository objects that an employee works on during a development project are collected within his or her task. The repository objects are therefore locked for all developers who do not belong to this particular project.

Unlike the logical functional divisions that separate Repository objects using packages, change requests are project-related and therefore have a limitation with regard to the **time schedule** involved. For example, although a program always belongs to only one package, it can belong to different change requests at different times.

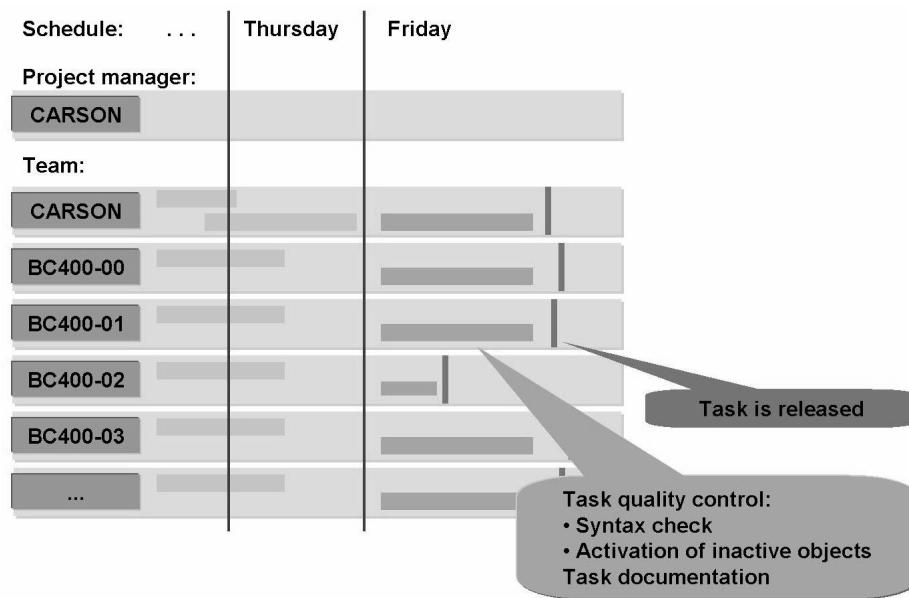


Figure 219: Developer's Actions at End of Development Process

When development is finished, the **developer** carries out a final quality control and releases the **task**.

The object entries and object locks are passed from the task to the request. However, all employees assigned to the request can still make changes to the objects because the *Transport Organizer* will automatically create a new task should the need arise.

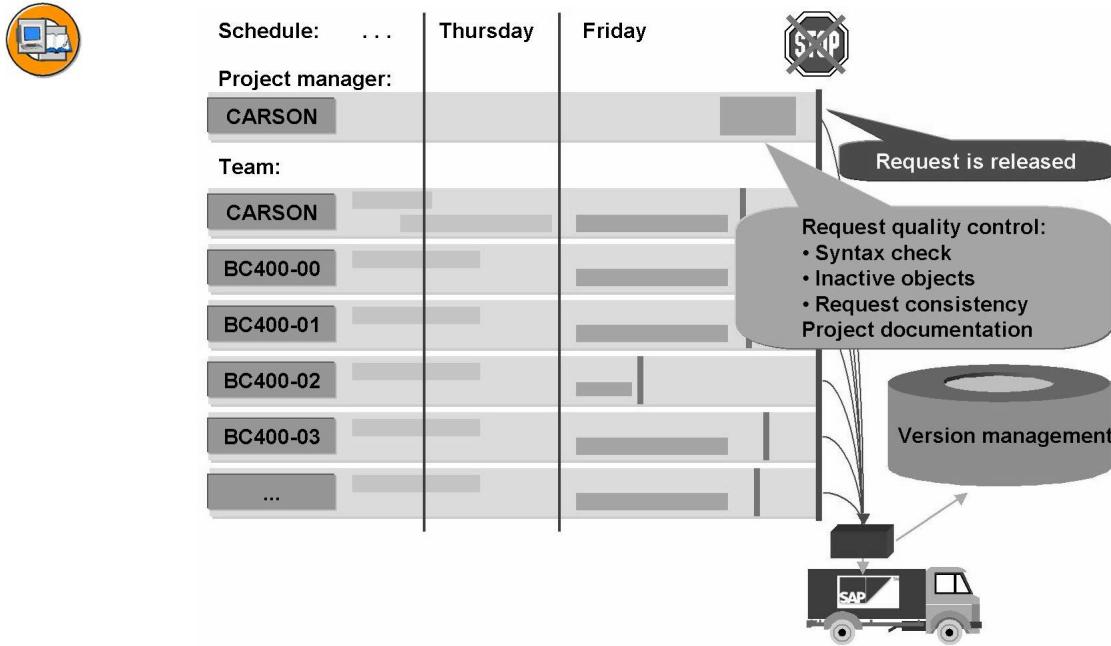


Figure 220: Project Manager's Actions at End of Development Process

When the project is complete, the **project manager** checks the consistency and releases the **request**. The locks on the objects in the change request are released. The development status of every affected Repository object is stored in the *version management*.

Copies of the Repository objects are then exported to the central transport directory.

The objects are not imported into the target system automatically, but this is initiated by the **system administrator**.

Afterwards, the **developer** checks the import logs.

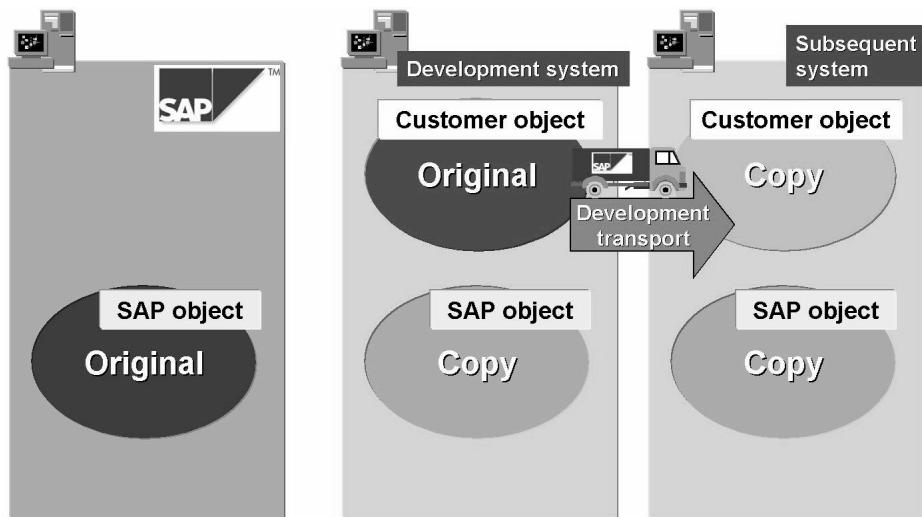


Figure 221: Originals and Copies

When you create a Repository object, the *Transport Organizer* “remembers” the system in which you created it. We say that the **original** of a Repository object is in an SAP system. The original version of an object can only exist in one system. All other systems contain copies of this object.

The idea of having an original system is to ensure that Repository objects can only be changed in the development system. The development system is where you carry out your development work, so all of the objects in it are originals. This means that there is one unique location for changing all Repository objects.

If you now transport your Repository object into a subsequent system, it exists there as a copy. Although it is possible to change copies in exceptional cases, you should always try to make the changes to the original in the development system and transport the new version to the subsequent system from there. This ensures that the state of objects remains consistent across systems.

Originals are never overwritten in transports.

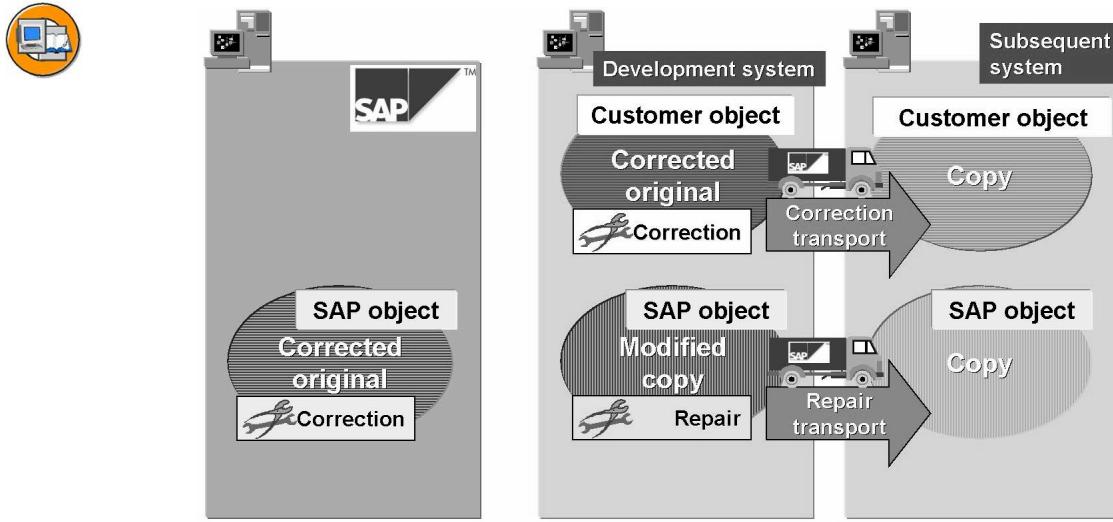


Figure 222: Corrections and Repairs

Changing an original is called a **correction**. The system records these changes in a request containing tasks of type *development/correction*.

However, if a copy is changed (that is, if an object is not changed in its original system), this change is called a **repair**. Accordingly, this change is recorded in a task with type *repair*. A repair of an SAP object is called a **modification**.

The repairs made to your own objects (for example, due to an emergency in the production system) can also be made immediately to the originals in the development system.



Caution: It is imperative that you immediately make the changes you made to the copies to the original as well.

This is not possible for SAP objects because the originals are not in any of your systems.

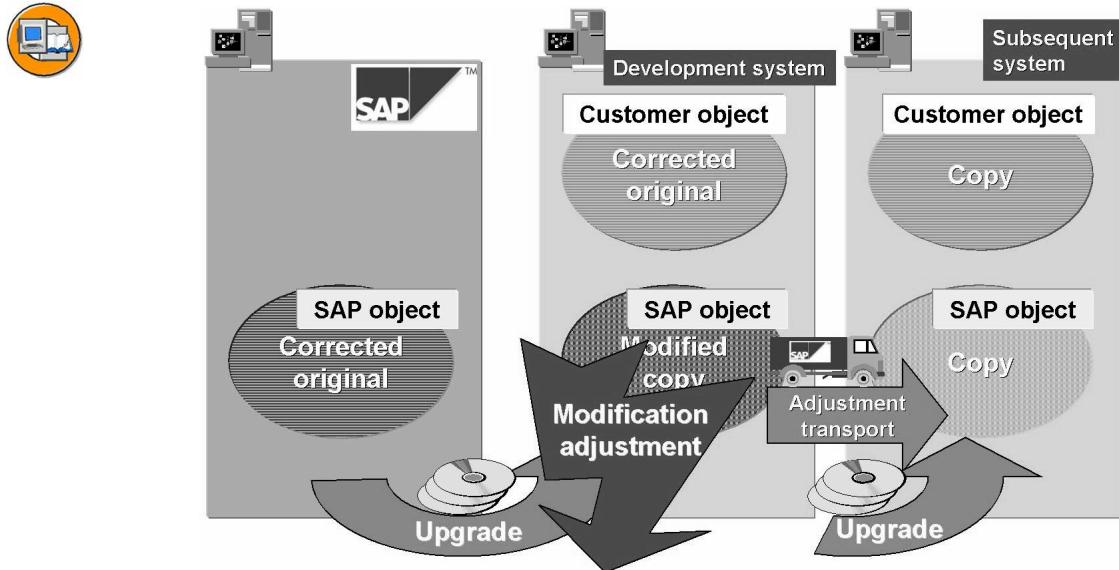


Figure 223: Modifications During Upgrade

A conflict could occur if you import an upgrade, a *support package*, or some other transport request from SAP into your system. Such a conflict arises if an SAP object modified by you is newly shipped by SAP: The object newly transported by SAP becomes an active object in the repository of your system.

If you want to “save” your changes, you have to carry out a **modification adjustment** for the relevant objects. Many modified SAP objects can cause substantial delays when importing an upgrade.

To ensure that the development system and the subsequent system are consistent, you should **only make the modification adjustment in the development system**. The objects of the adjustment are then transported to the subsequent systems.



Lesson Summary

You should now be able to:

- Explain the concepts for orderly development
- Use the tools for orderly development

Lesson: Outlook: Enhancing the Standard Software

Lesson Overview

In this lesson, you will learn about the different ways of changing SAP standard software.



Lesson Objectives

After completing this lesson, you will be able to:

- Explain the terms original, copy, correction, repair, and modification
- Describe how to avoid modifications to SAP objects

Business Example

You want to make changes to the SAP standard software but avoid any adjustment after the new standard version has been shipped.

Enhancing the Standard Software

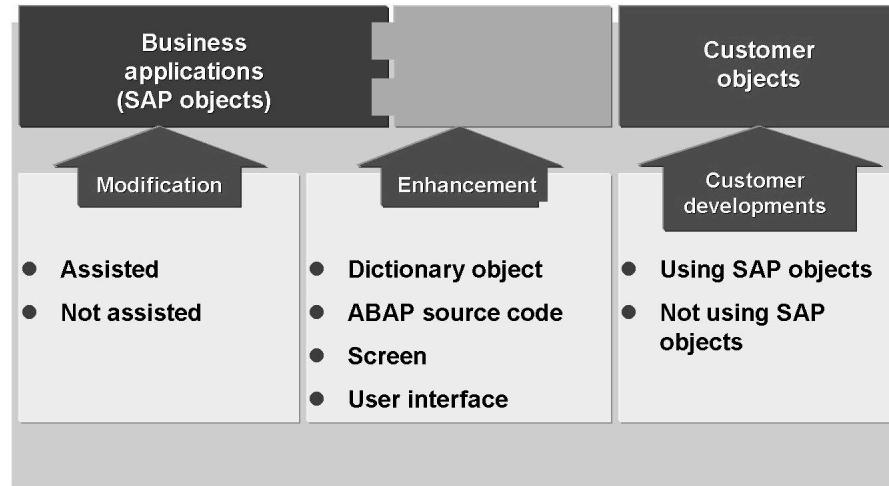


Figure 224: Overview: Software Adjustment Possibilities

There are four different ways of changing the SAP System to meet customer requirements:

- **Customizing:**

Allows you to change system parameters using your own maintenance transaction. Therefore, all possible changes have been thought through and organized. Customizing is a mandatory part of setting up an SAP System.

- **Enhancement concept:**

Allows you to change SAP Repository objects without modifications.

- **Customer development:**

Repository objects are created in accordance with customer namespace conventions.

- **Assisted modification:**

SAP Repository objects can be changed using the Modification Assistant. Modifications can lead to a considerably increased workload for an upgrade. If, in the new Release, there are changes to the same Repository object that was previously modified, you have to adjust the versions manually. This is supported by the Modification Assistant semi-automatically. For more details, refer to the online documentation:

Basis → ABAP Workbench → Changing the SAP Standard → Modification Assistant

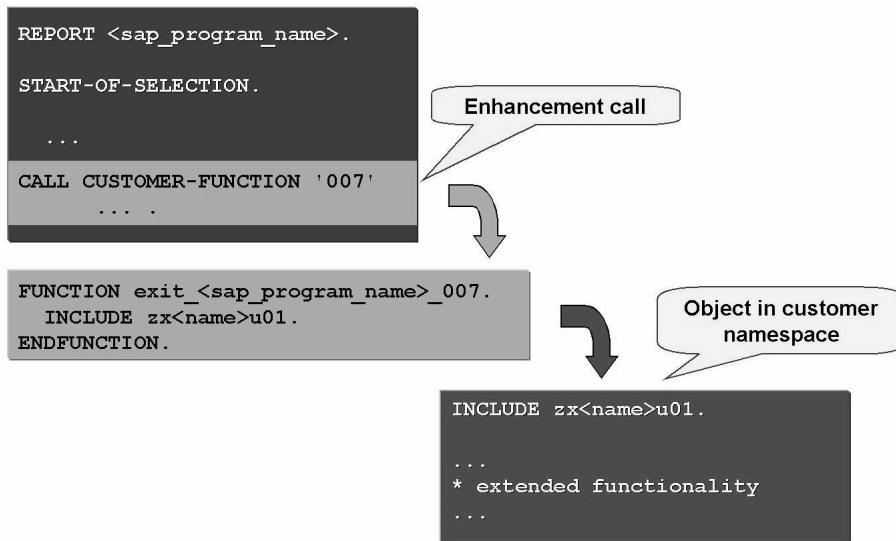


Figure 225: Enhancement Concept Using the Function Module Exit as an Example

SAP objects are usually enhanced as shown above.

The SAP object provides a starting point for a customer object. Since this object usually does not exist, you have to implement it in a specified manner.

Different enhancement techniques are implemented differently. Depending on the enhancement technique, you have to use a different maintenance transaction to create the corresponding objects and activate the enhancement.

The relevant maintenance transaction has a search function for finding a suitable enhancement. Documentation about the corresponding enhancement is also available here.

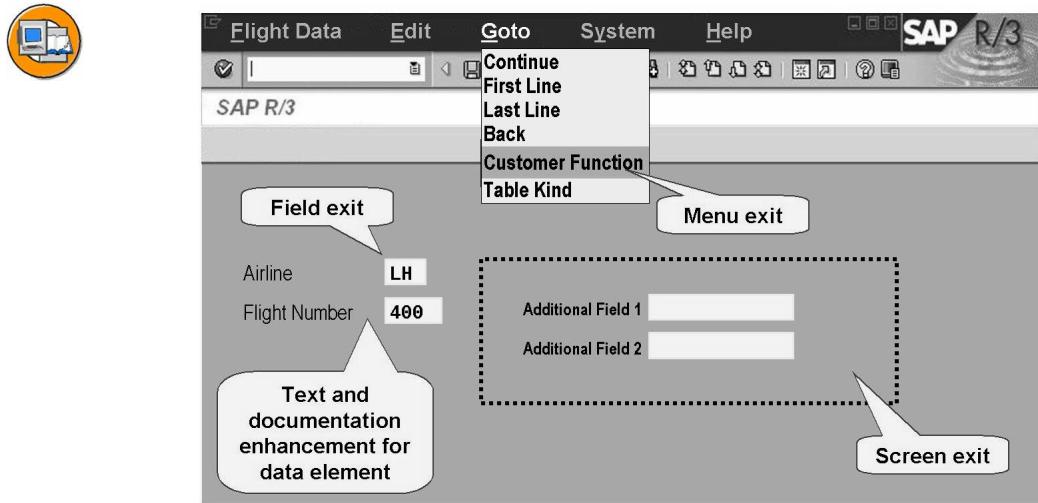


Figure 226: Enhancing User Dialogs

Enhancement of user dialogs is possible at the following levels:

Menu exit

There are various points provided to allow you to call customer menu items from the standard menus.

Screen exit

Some screens contain special areas (subscreens) in which you can display your own screens.

Field exit

You can attach additional checks or conversions to fields on screens.

In the *ABAP Dictionary*, you can also enhance attributes of data elements:

Field documentation:

You can replace the field documentation that is displayed when the user presses F1 with your own texts.

Field label

You can replace the short, medium, and long text field labels with your own texts.

As well as the enhancement techniques already mentioned, there are also other possibilities, such as:

Further Enhancement Examples**Append structure**

You can create any number of append structures for every standard table. With this, you can append fields to a table without changing the table itself.

Predefined but not yet effective source code

In some standard programs, special source code areas are commented out. If the comment symbols (asterisks) are removed from these lines of source code, they will also be executed. This is technically a modification.

Business Add-Ins (Add-Ins)

Business Add-Ins represent a new alternative technology that allows you to implement the full scope of program enhancements - that is, function exit, menu exit, and screen exit. Add-Ins are object-oriented and therefore offer further possibilities.

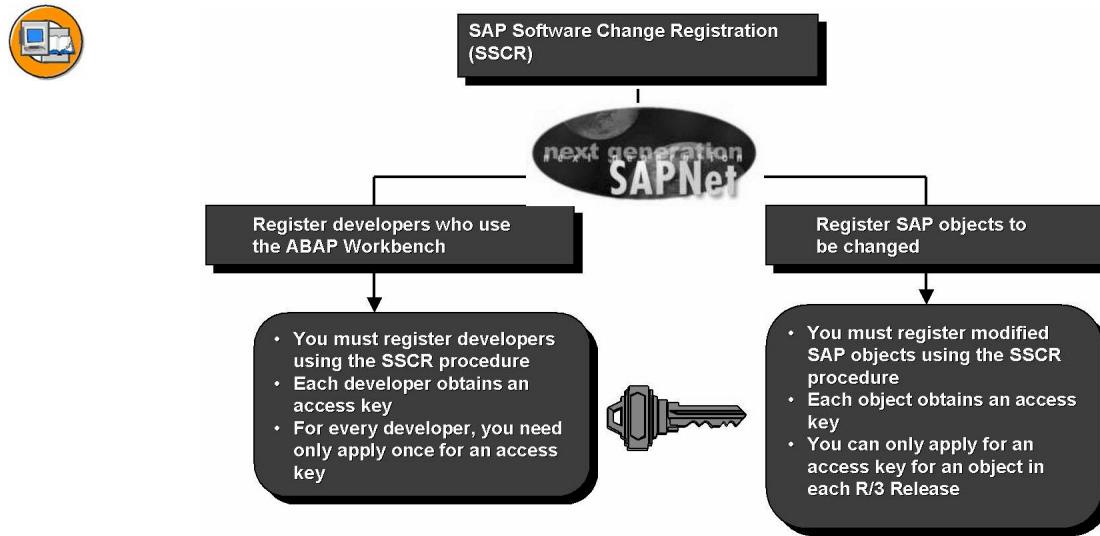


Figure 227: Registering Developers Using SSCR

All users who wish to use the *ABAP Workbench* to create or change a Repository object in the SAP System (including customer developments), must make a **single** request for a key using the *SAP Software Change Registration (SSCR)* procedure.

After the registration process all developers receive a key. The key is linked to the developer's user ID and the license number of the SAP System. The system prompts development users for their key the first time they attempt to create or change a Repository object.

Additionally, **every** SAP Repository object that is **modified** must be registered. When registering, you must specify the Repository object name, the object type, and the SAP System license number and Release. The registration is valid until the next upgrade.



Lesson Summary

You should now be able to:

- Explain the terms original, copy, correction, repair, and modification
- Describe how to avoid modifications to SAP objects



Unit Summary

You should now be able to:

- Explain the concepts for orderly development
- Use the tools for orderly development
- Explain the terms original, copy, correction, repair, and modification
- Describe how to avoid modifications to SAP objects

Related Information

... Refer to the articles "Transport Organizer " and "Changing the SAP Standard" in the online documentation.



Course Summary

You should now be able to:

- Create an ABAP Objects program containing user dialogs and database accesses
- Describe the different type of development objects and their typical intended purposes
- Create basic examples for all those Repository objects introduced in the course using the appropriate tools

Related Information

... Refer to the online documentation for the concepts, keywords, and tools addressed.

Index

A

ABAP documentation, 39
ABAP join, 146
ABAP syntax, 34
activate, 40
active, 40
actual parameter, 177
APPEND, 117
Application Component, 19, 24
application hierarchy, 20
application server level, 3
Application Toolbar, 296
AT LINE-SELECTION, 207
AT SELECTION-SCREEN, 235
Attributes, 350
AUTHORITY-CHECK, 162
authorization, 162
Authorization Check, 162
authorization object, 163

B

BAPI, 369
 Explorer, 371
 standard BAPIs, 370
branches
 conditional, 84
breakpoint, 88, 91

C

calculation, 83
CALL FUNCTION, 339
CALL METHOD, 351
Call-by-reference, 177
Call-by-value, 177

Call-by-value-and-result, 177
CASE, 84
chained statement, 36
change request, 29, 379
 release, 381
CHECK, 85-86
CL_GUI_ALV_GRID, 355
CL_GUI_CUSTOM_CONTAINER, 355
class, 350
Class Builder, 356
CLEAR, 82, 123
COLLECT, 117
Column Header, 204
comment, 36
COMPUTE, 83
constant, 81
CONSTANTS, 81
 VALUE addition, 81
constructor, 351
CONSTRUCTOR, 351
container control, 354
context menu, 23
Control Framework, 352
conversion rule, 82
CREATE OBJECT, 351
Customizing, 386

D

DATA, 78, 101, 115
 BEGIN-OF-addition, 101
 LIKE addition, 79
 TYPE REF TO, 351
 VALUE addition, 80
data element, 61

- Data Model, 57
D
 data object
 elementary, 79
 fix, 81
 global, 181
 local, 181
 structured, 100
 data type
 global, 77
 local, 181
 Data Type, 77
 Database, 57
 database interface, 135
 database level, 3
 database management system
 relational, 3
 database table, 61
 debugging mode, 88, 103
 DELETE, 117-118
 detail list, 207
 dialog message, 86
 DO, 85
 documentation
 ABAP, 39
 keyword, 39
 online, 39
 domain, 61
E
 enhancement
 append structure, 389
 field documentation, 389
 field exit, 388
 field label, 389
 menu exit, 388
 screen exit, 388
 enhancement concept, 387
 EnjoySAP controls, 352
 entity, 57
 Entity Relationship Model,
 57
 ERM, 57
F
 Event block, 193
 event keyword, 195
 EXIT, 86
 expression
 arithmetical, 83
G
 flight data model, 58
 foreign key field, 60
 foreign key relationship, 60
 FORM
 CHANGING- addition, 178
 USING addition, 178
 FORM...ENDFORM, 177
 formal parameter, 177
 type assignment, 180
 FREE, 123
 function
 predefined, 83
 STRLEN, 83
 Function code, 262
 Function group, 333
 function key setting, 296
 Function Module, 333
H
 Hashed table, 114
 header, 126
 HIDE, 211
 HIDE area, 211
I
 IF, 84
 inactive, 40
 INITIALIZATION, 195
 INSERT, 117, 119
 instance, 350

- interface
 - changing parameter, 333
 - exception, 333
 - export parameter, 333
 - import parameter, 333
 - of a subroutine, 177
- internal table
 - access type, 113
 - attributes, 112
 - deleting contents, 123
 - global data type, 114
 - header, 126
 - index, 113
 - index access, 113, 121
 - inserting rows, 119
 - key, 113
 - key access, 113, 122
 - kind, 113
 - line type, 112
 - line types, 112
 - local data type, 115
 - loop through rows, 120
 - sorting entries, 123–124
 - table kind, 113
- Internet service
 - publish, 326
- Internet Service, 324
- Internet Transaction Server, 320
 - programming models, 320
- ITS, 320
- ITS Flow Logic, 321
- K**
 - key field, 60
 - keyword documentation, 39
- L**
 - list buffer, 206
 - List Header, 204
 - literal, 81
 - LOAD-OF-PROGRAM, 195
- locally obscured, 182
- logical database, 147
- Loop, 85
- LOOP, 118, 120
 - INDEX addition, 121
 - INTO clause, 118
 - WHERE addition, 122
- M**
 - menu bar, 296
 - message, 86
 - MESSAGE, 86
 - TYPE addition, 86
 - WITH addition, 86
 - method, 350
 - modification, 383
 - adjustment, 384
 - assisted, 387
 - MODIFY, 117, 120
 - Module, 265
 - MOVE, 82
 - MOVE-CORRESPONDING, 102
- O**
 - object component selector, 351
 - Object Navigator, 23
 - object orientation, 350
 - online documentation, 39
 - Open SQL, 135
 - original, 382
- P**
 - package, 19, 24, 31
 - PARAMETERS, 230
 - PERFORM, 183
 - presentation server level, 3
 - Pretty Printer, 35
 - primary key field, 60
 - processing block, 192
 - program
 - activate, 40
 - generate, 41

test, 41

R

RDBMS, 3

READ, 117

READ TABLE

- INDEX addition, 121
- WITH addition, 122

record, 100

REFRESH, 123

registration, 390

Repository, 19

Repository Information System, 20

repository object, 19, 24
create, 26

runtime object, 41

S

SAP Grid Control, 353

SAPGUI for HTML, 321

scalability, 3

screen

- call, 247
- command field, 262
- component, 247
- creating, 251
- data transport, 254
- layout, 252
- module, 265
- OK-CODE field, 262

PROCESS AFTER INPUT, 265

PROCESS BEFORE OUTPUT, 265

pushbutton, 261

TABLES, 255

Screen, 65, 246

Secondary index, 144
SELECT, 137
SELECT-OPTIONS, 232
selection screen

- variant, 229

Selection Screen, 65, 224

service file, 324

SET PF-STATUS, 298, 306

SET TITLEBAR, 306

SET_TA-BLE_FOR_FIRST_DISPLAY, 357

SORT, 123-124

AS TEXT addition, 124

STABLE addition, 124

sorted table, 114

SSCR, 390

registration, 390

Standard table, 114

standard toolbar, 296

START-OF-SELECTION, 195

Status, 296

structure

complex, 63

data object, 100

global data type, 63

structure component, 63, 101

subroutine, 177

system field

SY-DATUM, 193

SY-DBCNT, 139-140

SY-INDEX, 85

SY-LSIND, 207

SY-SUBRC, 37

T

table

internal, 112

table key, 60

Table kind, 113

Table type, 114

TABLES, 100, 255

task, 31

release, 380

text symbol, 205

title, 296

title bar, 296

Transparent Table, 60

- transport
 - copy, 382
 - correction, 383
 - original, 382
 - repair, 383
- transport layer, 32
- Transport Organizer, 31, 379
- transport route, 379
- type conversion, 82
- TYPES, 78, 100, 115
 - BEGIN-OF addition, 100
 - LIKE addition, 79
- U**
- User interface, 296
- V**
- variable
- reference, 351
- Variable, 79
- version
 - active, 40
 - inactive, 40
- version management, 381
- view, 146
- W**
- watchpoint, 92
- Web Application Builder, 326
- Web Transaction, 321
- WHILE, 85
- worklist, 40

Feedback

SAP AG has made every effort in the preparation of this course to ensure the accuracy and completeness of the materials. If you have any corrections or suggestions for improvement, please record them in the appropriate place in the course evaluation.