

BC400

Introduction to the ABAP Workbench

SAP NetWeaver

Date _____
Training Center _____
Instructors _____
Education Website _____

Participant Handbook

Course Version: 63
Course Duration: 5 Days
Material Number: 50087597



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

Copyright

Copyright © 2008 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

American English is the standard used in this handbook. The following typographic conventions are also used.

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.
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	vii
Unit 1: Flow of an ABAP Program	1
System Architecture and ABAP Programs	2
Unit 2: Introduction to the ABAP Workbench.....	19
Repository and Object Navigator	20
Developing Programs and Organizing Developments	31
Unit 3: Basic ABAP Language Elements.....	67
Working with Elementary Data Objects.....	68
Unit 4: Modularization	111
Modularization - Basics and Overview	113
Modularization with Subroutines.....	120
Modularization with Function Modules.....	142
Modularization with Methods of Global Classes.....	178
Modularization with Methods of Local Classes (Preview)	208
Unit 5: Complex Data Objects	219
Working with Structures.....	220
Working with Internal Tables	232
Unit 6: Data Modeling and Data Retrieval.....	265
Data Modeling and Transparent Tables in the ABAP Dictionary	266
Reading Database Tables	282
Authorization Check.....	324
Unit 7: User dialogs.....	345
Screen.....	347
ABAP Web Dynpro	398
Classic ABAP Reports	436
Displaying Tables with the SAP List Viewer.....	469

Unit 8: Tools for Program Analysis.....	499
The Code Inspector	500
Unit 9: Adjusting the SAP Standard Software (Overview)	511
Adjusting the SAP Standard Software (Overview)	512
Index.....	527

Course Overview

This course introduces you to the ABAP programming language of SAP as well as its 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 follow-on documentation.

We always handle these topics using practical application examples so that you can immediately implement what you have learned here. Above all, it should become clear how little effort is required to create high-performance business applications in a short period of time in the *ABAP Workbench*.

We are convinced that this course will enable you to start developing immediately and provide you with the knowledge needed to focus on the essentials in subsequent courses.

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

Target Audience

This course is intended for the following audiences:

- Project team members
- ABAP programmers

Course Prerequisites

Required Knowledge

- Programming knowledge
- SAPTEC (SAP NetWeaver – Basics of the Application Platform)

Course Goals



This course will prepare you to:

- Understand and use basic ABAP 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 program containing user dialogs and database accesses
- Describe the different types of development objects and their typical intended purposes
- Use appropriate tools to create simple examples of the development objects presented

Unit 1

Flow of an ABAP Program

Unit Overview

The unit overview lists the individual lessons that make up this unit.



Unit Objectives

After completing this unit, you will be able to:

- Describe the *SAP NetWeaver Application Server* architecture
- Describe how a simple dialog program is executed by the ABAP runtime system

Unit Contents

Lesson: System Architecture and ABAP Programs	2
---	---

Lesson: System Architecture and ABAP Programs

Lesson Overview

In this lesson you will learn how a simple dialog program is executed by the *SAP NetWeaver Application Server*.



Lesson Objectives

After completing this lesson, you will be able to:

- Describe the *SAP NetWeaver Application Server* architecture
- Describe how a simple dialog program is executed by the ABAP runtime system

Business Example

Your task here is to explain the *SAP NetWeaver Application Server* architecture as well as the execution of ABAP programs.

System Architecture and ABAP Program

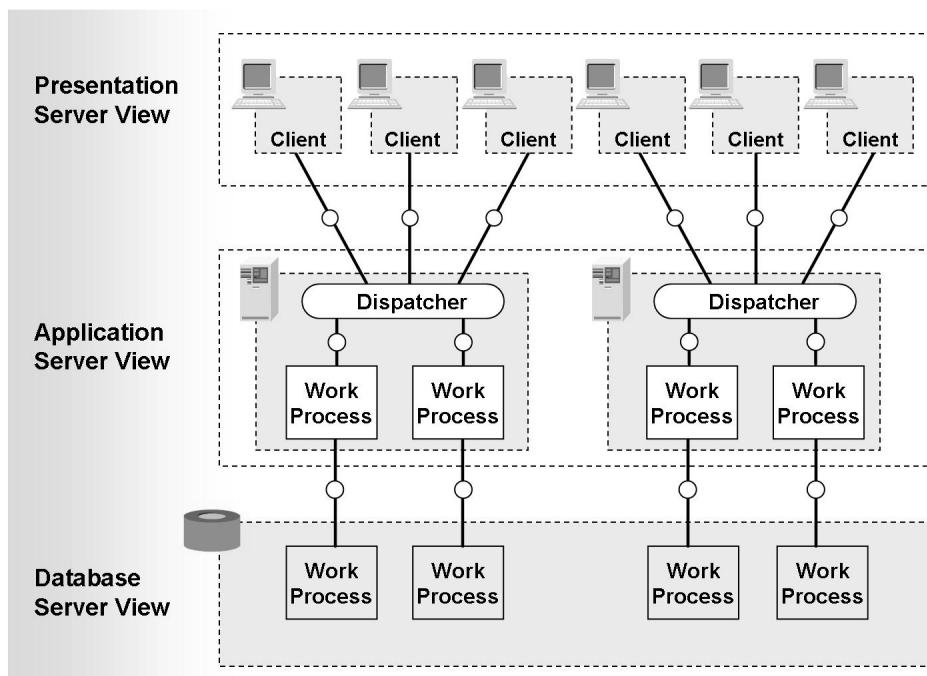


Figure 1: Client/Server Architecture of the SAP NetWeaver Application Server

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

In the *SAP NetWeaver 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 SAP 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, process it, and store new data there if necessary.

The third level is the **presentation server level**. This level contains the user interface where 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.

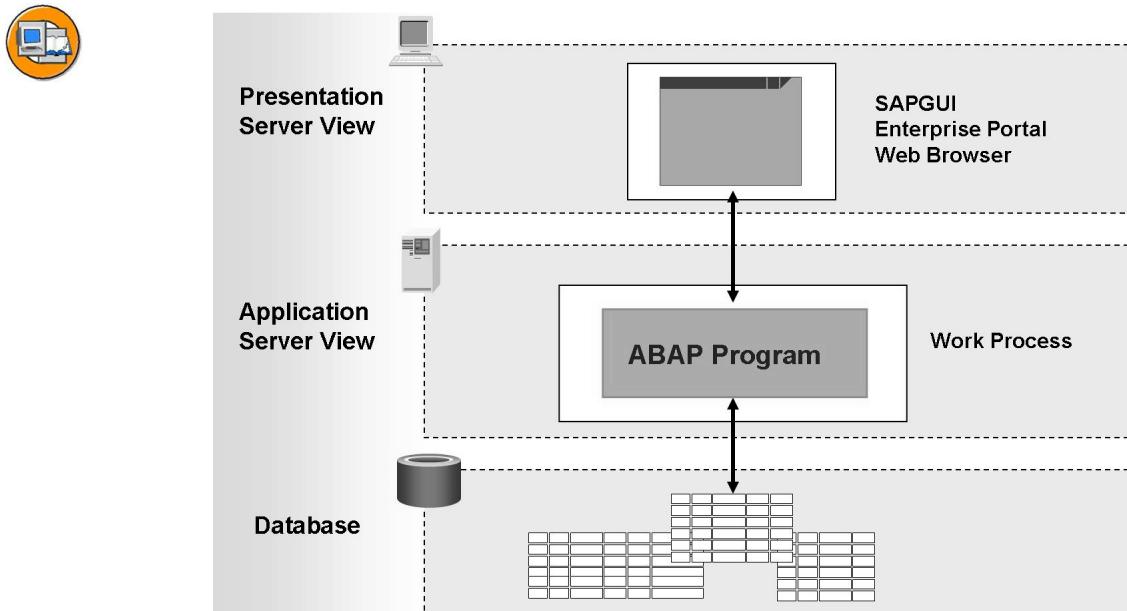


Figure 2: Excerpt for an ABAP Program

We can simplify this graphic for most of the topics that will be 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 slides 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 of particular importance when writing application programs.

In the following section we will discuss the basic process that takes place when an ABAP program is executed. To do this, we will use a program with which the user enters an airline ID (such as "LH") and sees details of the airline displayed as a result.

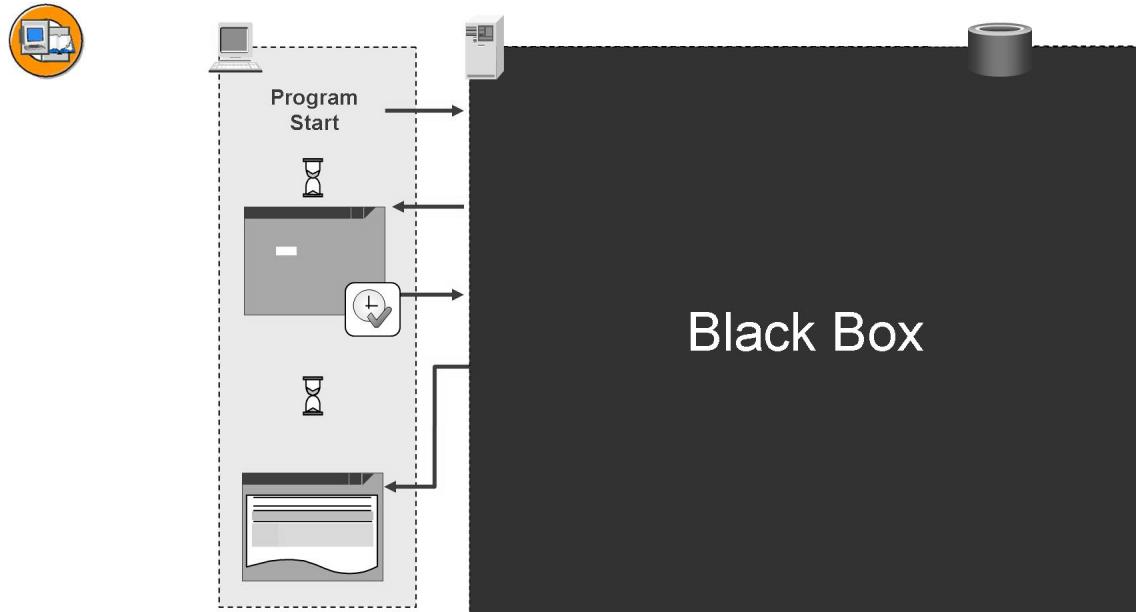


Figure 3: View for the User

The average user is only interested in the business process, and how data can be entered or displayed. The technical aspects of the program are of minor interest here. Users does not need to know the precise process flow of an ABAP program. The SAP system is like a "black box" to them.

By contrast, developers need to understand both the interplay between the server levels and the process flow when programs are executed in order to develop their own programs.

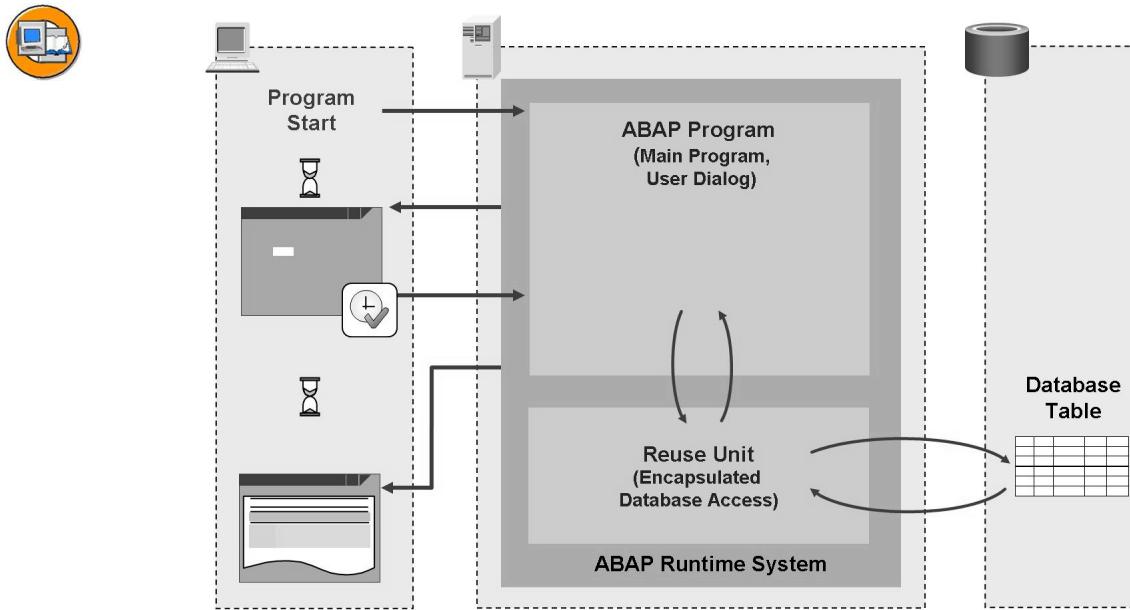


Figure 4: Interplay Between Server Levels and Program Flow

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

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.

Normally, a program is not made up of a single block, but of several units. This is known as modularization. Many of these modularization units can be used in more than one program, which is why they are often termed reuse units. A good program should have at least the database accesses encapsulated in such reuse units. This creates a division between the design of the user dialog and the database accesses. It is then possible to use the same database accesses for different user dialogs.

Process Flow of a Program with Selection Screen and List

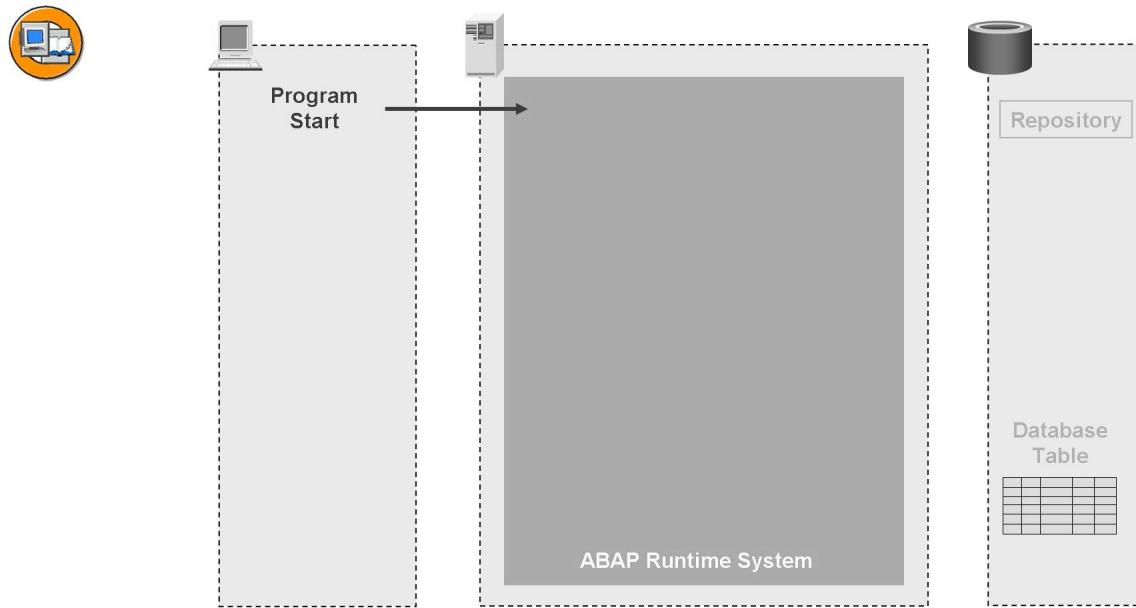


Figure 5: Program Start

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.

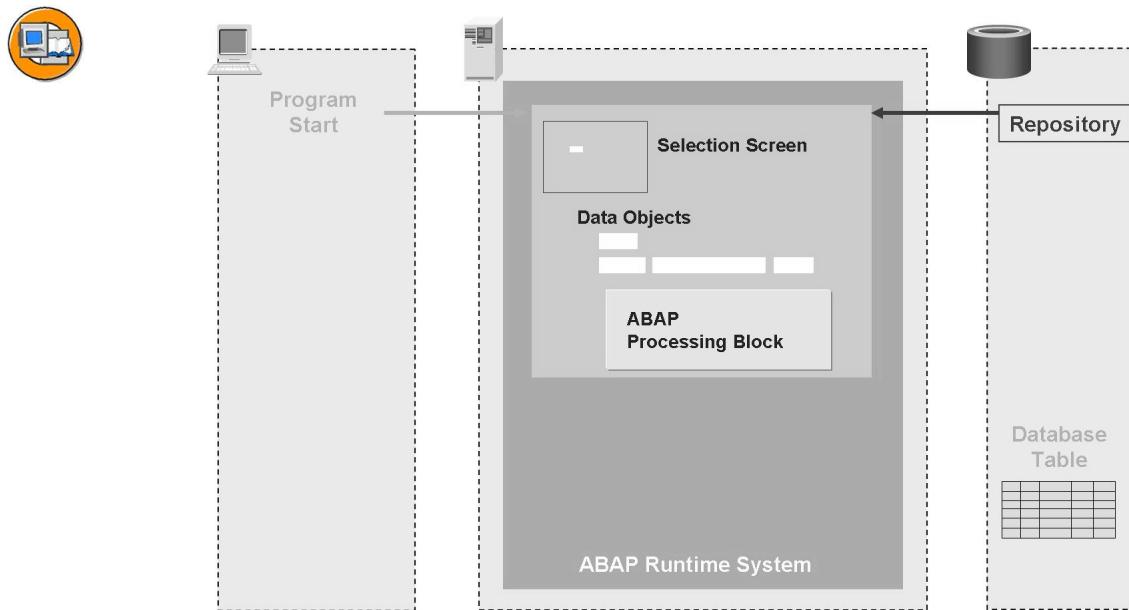


Figure 6: System Loads Program Context

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 used to display the data is created dynamically at runtime.

The ABAP runtime system controls the subsequent program flow.

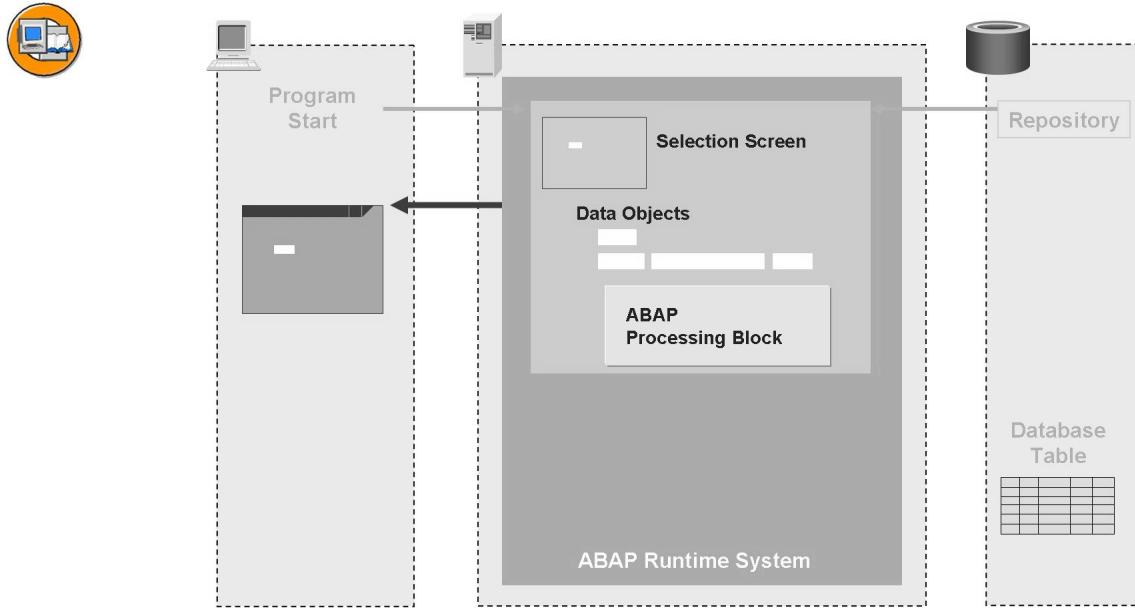


Figure 7: Runtime System Sends Selection Screen

Since the program contains a selection screen, the ABAP runtime system sends it to the presentation server. The presentation server controls the program flow for as long as the user has not finished entering data in the input fields.

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

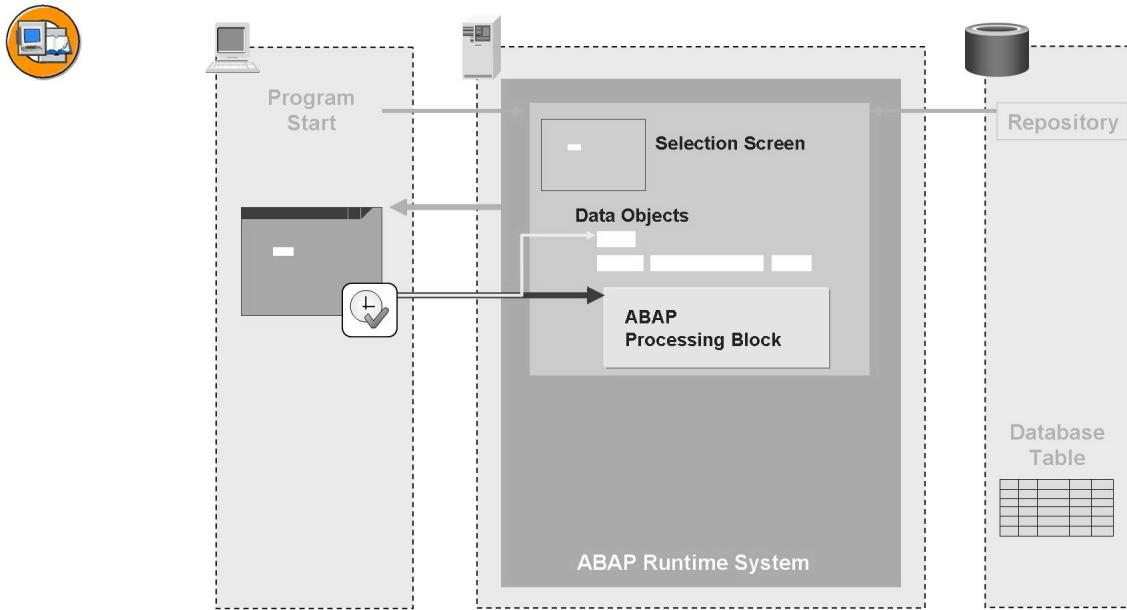


Figure 8: Input Values Are Inserted into Data Objects

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 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, an error message is **automatically** triggered. The user must correct his/her entries.

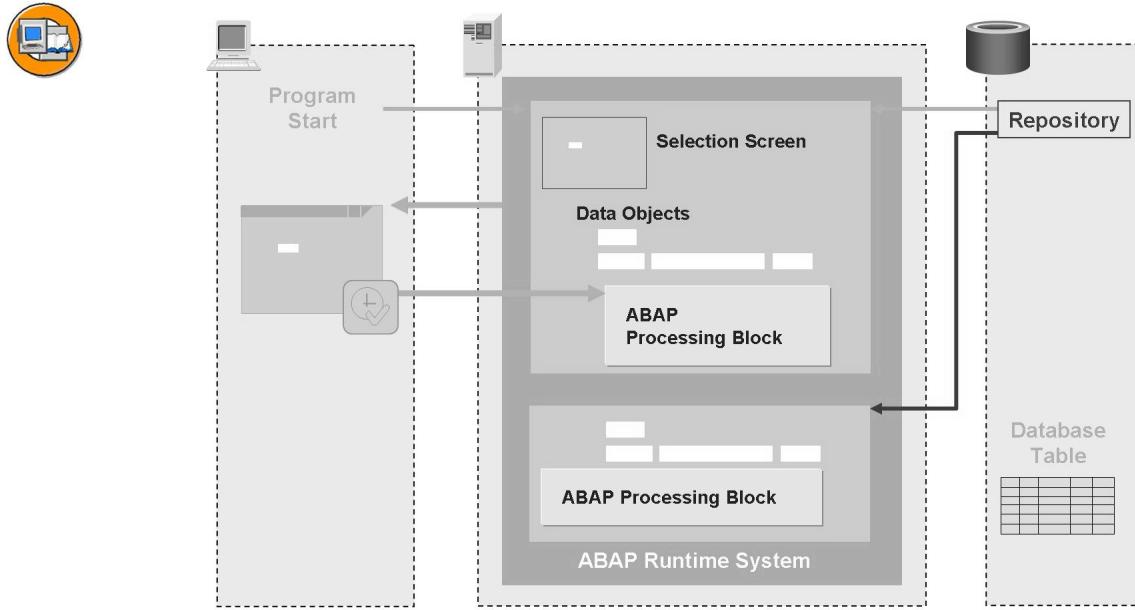


Figure 9: Reuse Unit Loading

A reusable unit is called in the processing block that encapsulates the database access. The reuse unit is a special processing block in an individual program. The actual example shows a method in a global class. When the reuse unit is called, the program in which it is contained is also read from the *Repository* and loaded to the application server.

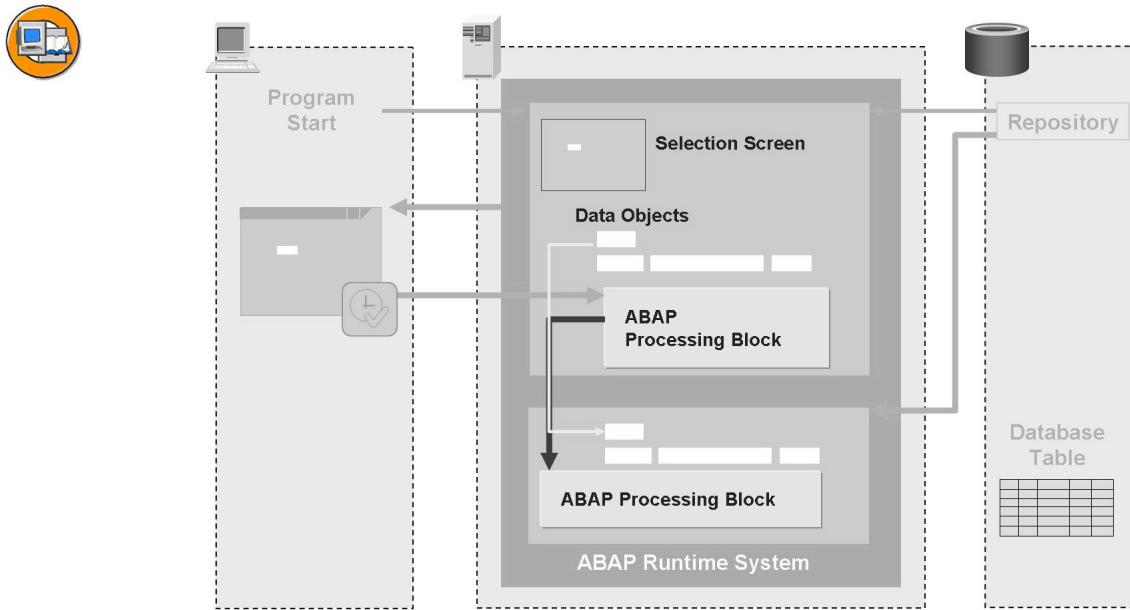


Figure 10: Reuse Unit Is Called

In the call, the required data is transferred to the called program, after which the reuse unit is executed. The execution is synchronous, which means that the calling program waits until the reuse unit has been processed completely.

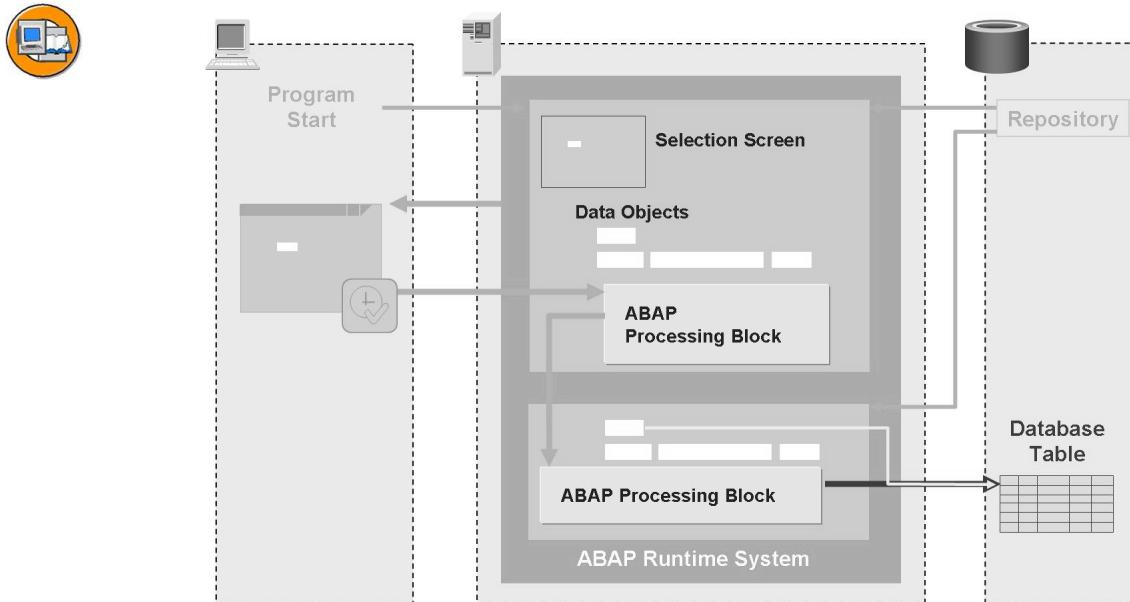


Figure 11: Program Requests Data Record from the Database

In the example program, read access to the database is programmed in the reuse unit. Correspondingly, information about the database table to be accessed and the row in the table to be read is passed on to the database.

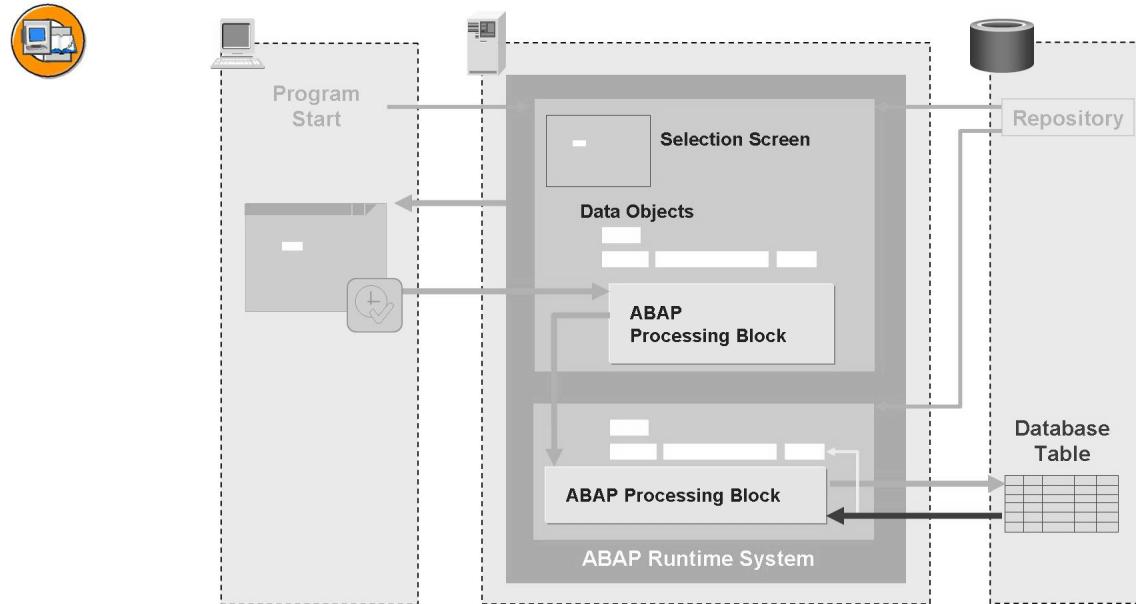


Figure 12: Database Supplies the Data Record

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 relevant components for all the required database fields.

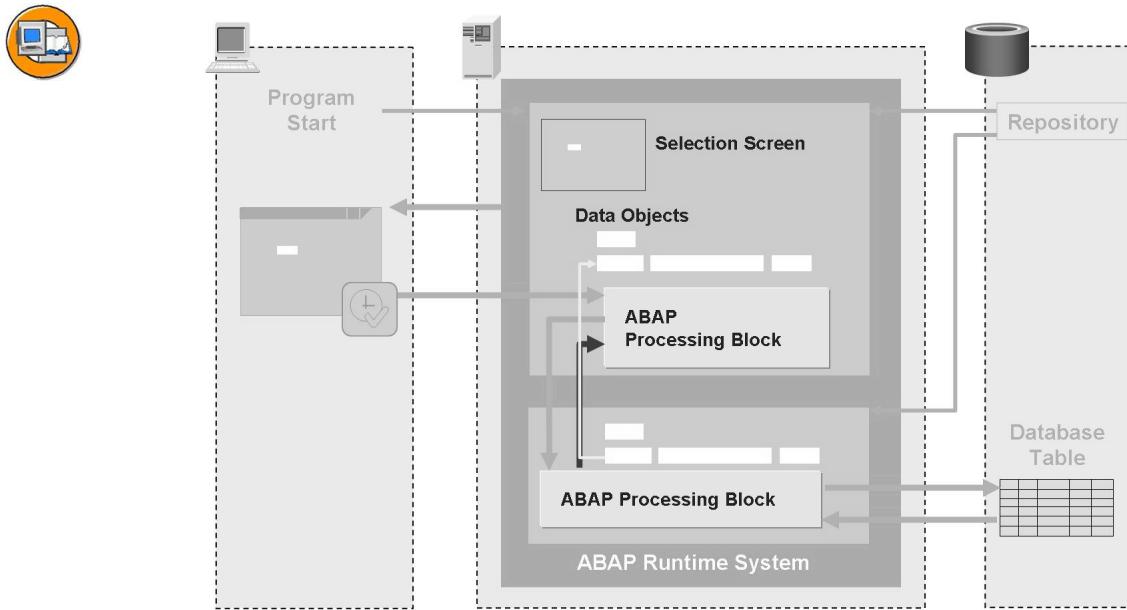


Figure 13: Reuse Unit Returns the Data

This concludes the processing of the reuse unit and control is returned to the calling program, which resumes immediately after the call. When the system exits the reuse unit, the data that was read from the database is written to a corresponding data object for the calling program, from where it can be processed further.

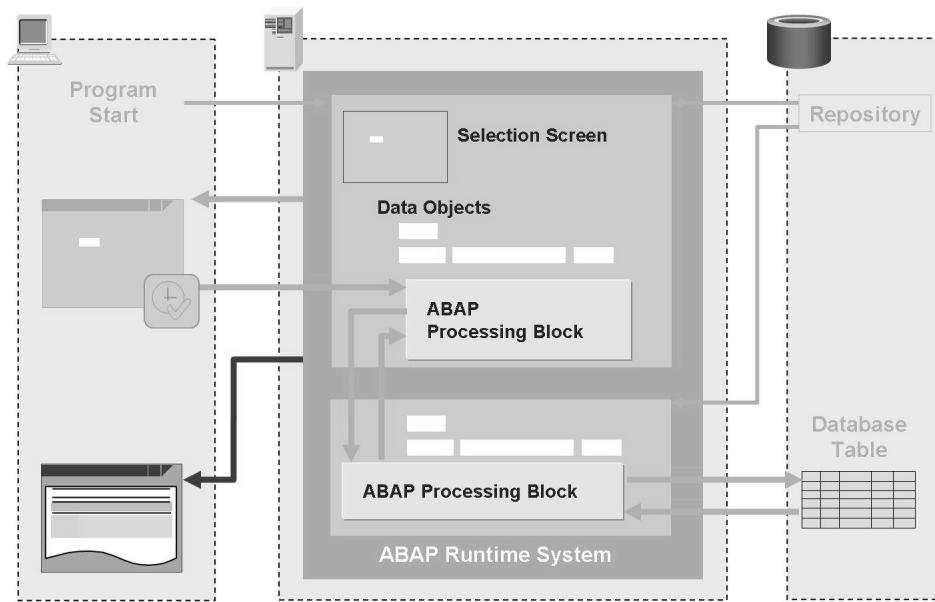


Figure 14: Runtime System Sends the List

After the reuse unit has been called, the ABAP processing block receives statements for structuring the list with which the result is to be displayed. After the processing block finishes, the runtime system sends the list as a screen to the presentation server.

Topic Overview

The above-described example program already demonstrates numerous concepts that make up the content of this course. The following table of contents, which shows all topics dealt with and their corresponding units, serves as an orientation aid for the remaining sections of this course.



Unit 1

Flow of an ABAP Program

Unit 2

Introduction to ABAP Workbench

Unit 3

Basic ABAP Language Elements

Unit 4

Modularization (Reuse Units)

Unit 5

Complex Data Types (Structures, Internal Tables)

Unit 6

Data Modeling and Data Retrieval (Database Tables, Database Accesses)

Unit 7

User Dialogs (Dynpro, Web Dynpro, Selection Screen, List)

Unit 8

Tools for Program Analysis

Unit 9

Adjusting the SAP Standard Software (Overview)



Lesson Summary

You should now be able to:

- Describe the *SAP NetWeaver Application Server* architecture
- Describe how a simple dialog program is executed by the ABAP runtime system



Unit Summary

You should now be able to:

- Describe the *SAP NetWeaver Application Server* architecture
- 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

The unit overview lists the individual lessons that make up 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 displaying Repository objects
- Name and use the utilities for orderly software development
- Create packages
- Create programs
- Create transactions

Unit Contents

Lesson: Repository and Object Navigator	20
Lesson: Developing Programs and Organizing Developments	31
Procedure: Creating Packages	34
Procedure: Creating an ABAP Program	38
Procedure: Creating Transactions	47
Procedure: Adding Transactions to your Personal Favorites	48
Exercise 1: Organizing Developments	51
Exercise 2: Developing a Simple ABAP Program	55
Exercise 3: Create Transactions	59

Lesson: Repository and Object Navigator

Lesson Overview

This lesson contains a short description of the *Repository* and a brief overview of the most important components of the *ABAP Workbench*. 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 displaying Repository objects

Business Example

You are to describe the structure of the *Repository*, use suitable tools to search for Repository objects and analyze their structure.

Introduction to the Repository

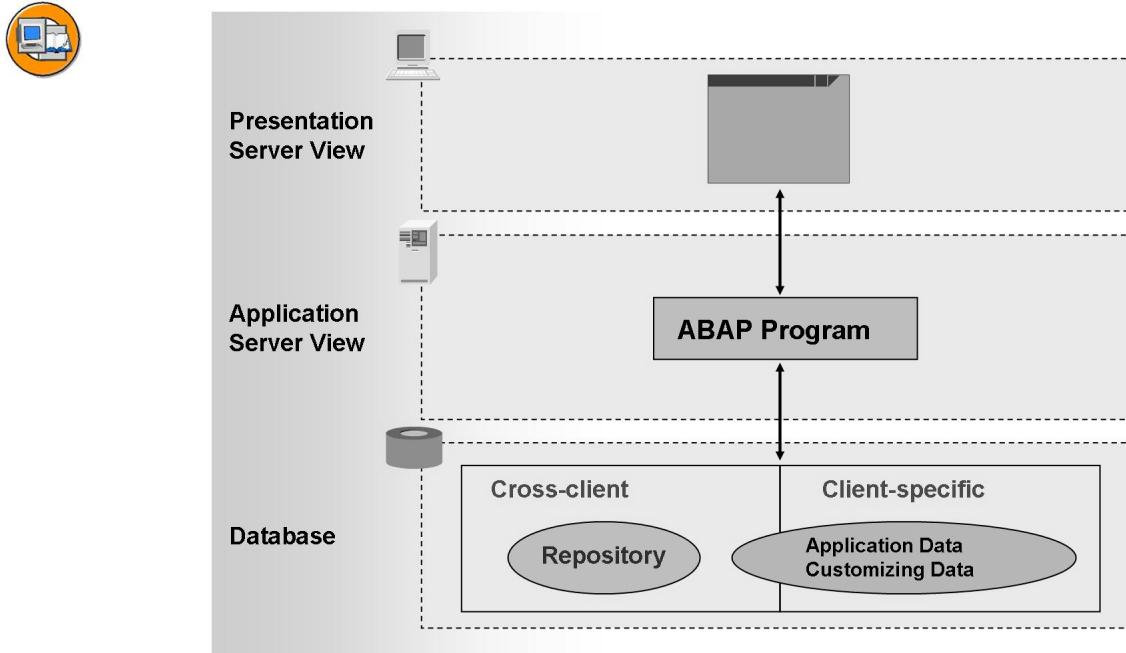


Figure 15: The Cross-Client Capability of the *Repository*

The *Repository* consists of all system development objects - programs, function modules, definitions of database tables, and so on. In the *Repository*, you have objects delivered by SAP as well as objects defined by the customer. The *Repository* is in the database and is always **independent** of the client, which means that a *Repository* object can be accessed from any client and looks the same in each client in the system.

As well as the *Repository*, the database also contains application and Customizing data which is normally **client-dependent**. This means that every data record is assigned to a particular client and can normally only be read and changed by users who have logged on to that particular client.

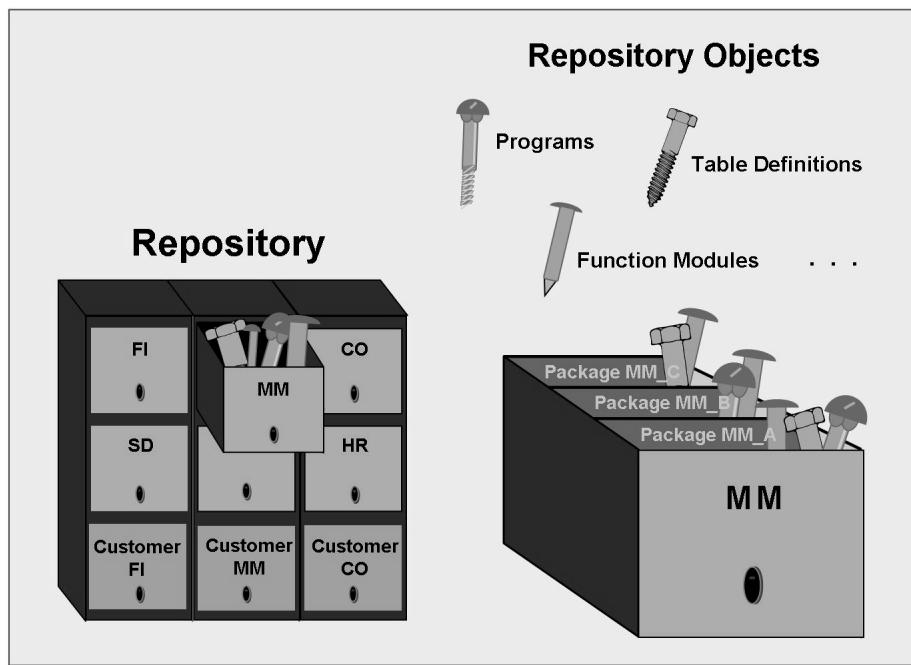


Figure 16: Structure of the Repository

The *Repository* is subdivided according to application components.

Within an application component (e.g., MM) there are several **packages** containing relevant objects for a more detailed logical subdivision.

Whenever a Repository object is created, it must be assigned to a package.

Some Repository objects can have subobjects that are themselves Repository objects. Furthermore, Repository objects can reference other Repository objects.

Search Tools in the Repository

The *Repository Information System* is suitable for the **random** (that is, not application-specific) **search** for Repository objects - such as all programs by a certain developer or all function modules that were changed after a certain date.

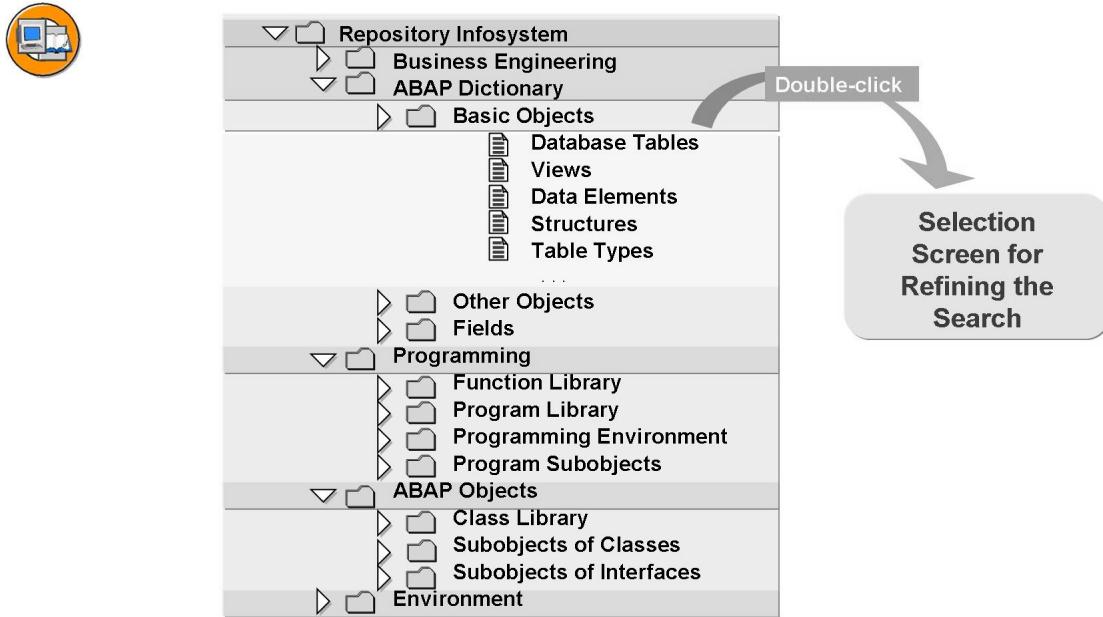


Figure 17: The *Repository Information System*

You get to the *Repository Information System* from the *SAP Easy Access Menu* by choosing *Tools* → *ABAP Workbench* → *Overview* → *Information System*. When you double-click a certain object type, a selection screen appears allowing you to limit your search.

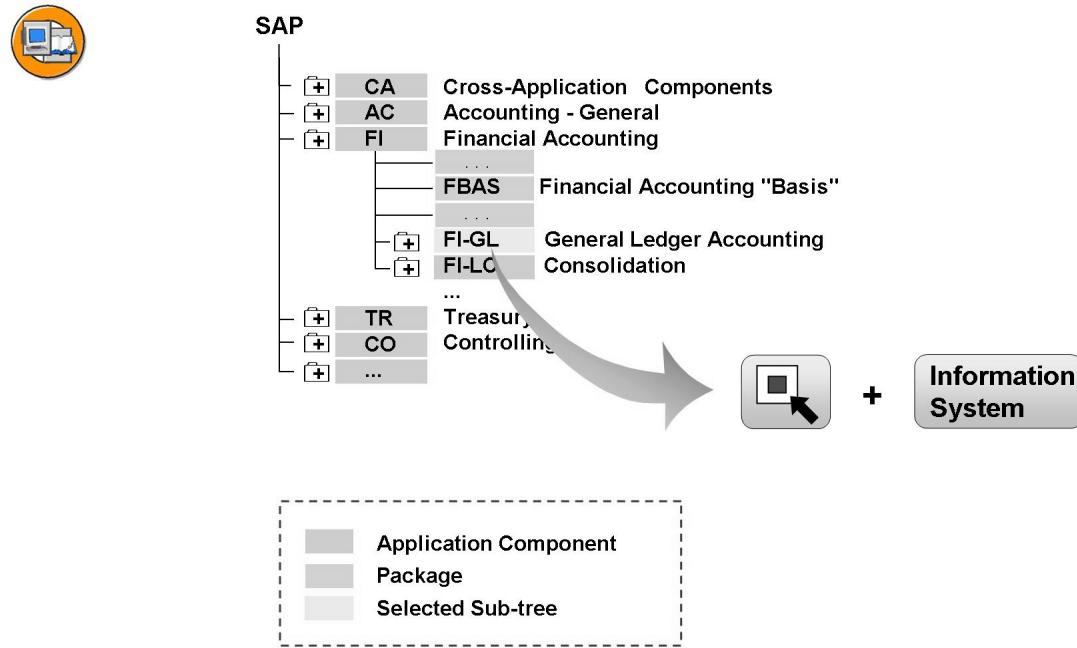


Figure 18: SAP Application Hierarchy

You use the **application hierarchy** for the **application-specific search** for Repository objects. There, the hierarchy of the application components is depicted in the form of a tree structure. From the *SAP Easy Access Menu*, you can easily branch to the SAP application hierarchy by choosing *Tools*→*ABAP Workbench*→*Overview*→*Application Hierarchy*→*SAP*. Expanding a component node displays all the packages that are assigned to the corresponding component. You can now select any subtree using the *Select* button and then navigate directly to the *Repository Information System* using the *Information System* button. The system remembers all relevant packages of the selected subtrees and automatically enters them in the selection screen of the information system. This allows you to carry out a search in the previously selected applications.

Working With the Object Navigator

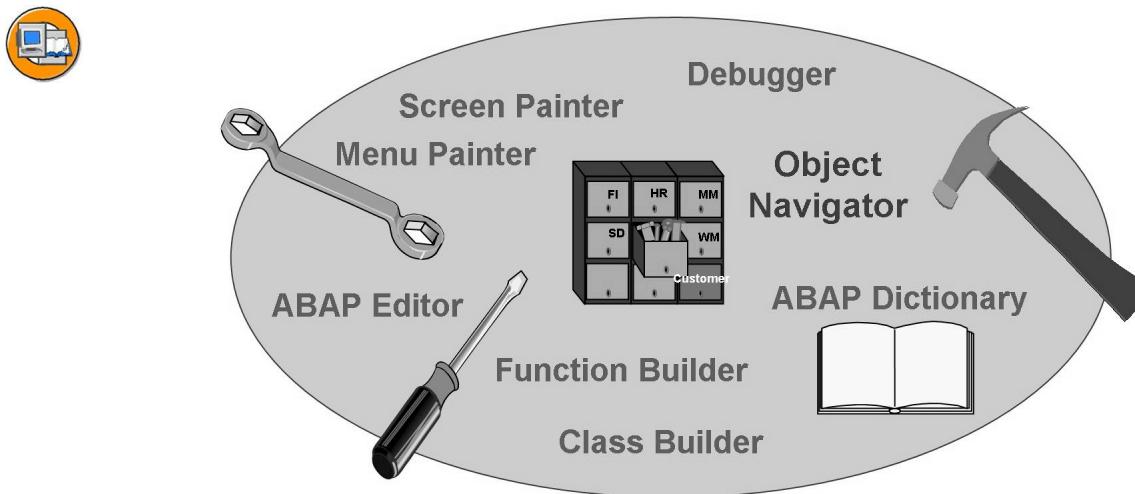


Figure 19: ABAP Workbench Tools

The *ABAP Workbench* includes all tools required for creating and editing Repository objects. These tools cover the entire software development cycle. The most important tools 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 maintaining function modules
- The **Class Builder** for maintaining global classes and interfaces

You can call each of these tools explicitly and then load a Repository object for processing. But it is much more elegant and convenient to access them using the *Object Navigator*. You can have the requested Repository objects listed in this central development tool. All you have to do to edit one of them is to double-click to select it. The corresponding tool is called automatically and includes the selected Repository for displaying or editing.

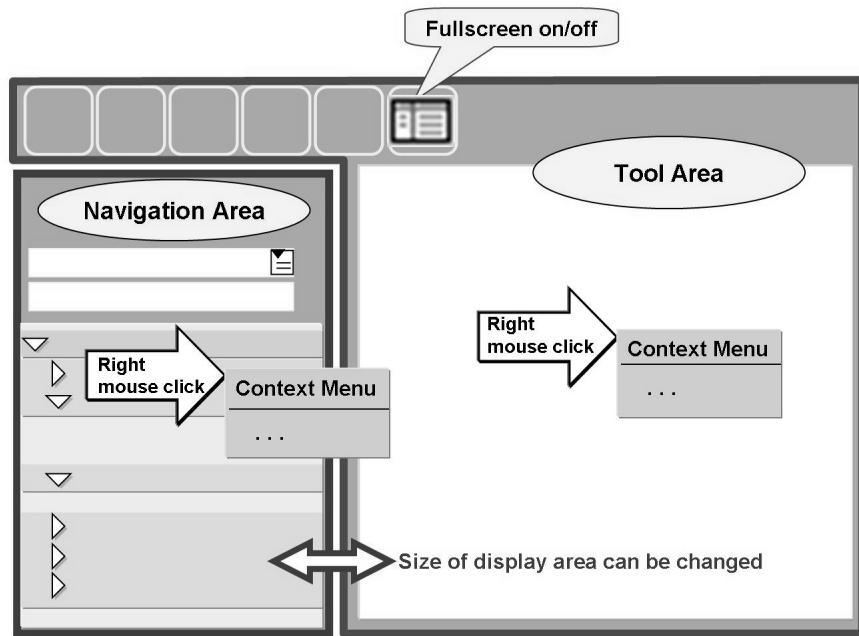


Figure 20: Screen Layout in the *Object Navigator*

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

- The **navigation area** for displaying a hierarchical object list
- A **tool area** for displaying and editing a development object using the appropriate tool

You can display or hide the navigation area (“*Fullscreen on/off*”).

In both areas, you can choose the functions using a context menu, which you access using the right mouse button (or left mouse button if you have set your mouse for left-handed users). The context menu offers only those functions which have been designed for the respective object.

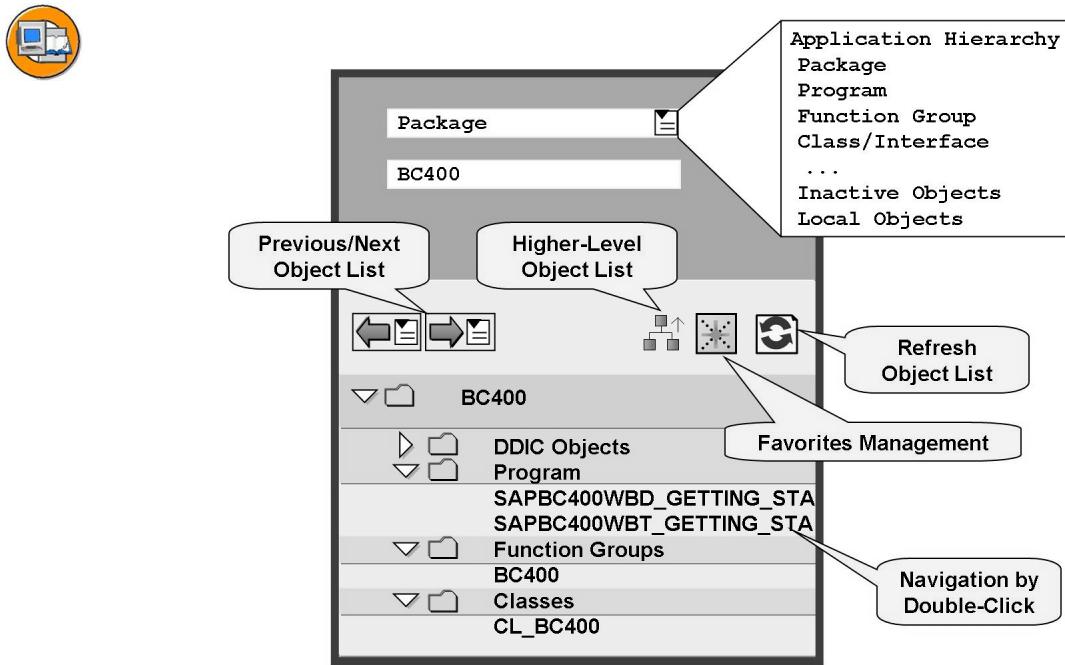


Figure 21: Working with the Navigation Area

Object lists are displayed in the navigation area. For example, if you choose to display a package there, all of the Repository objects belonging to the specified package are listed.

Double-clicking a listed object lets you display or edit it.

You can navigate between the object lists that have been previously displayed in the current *Object Navigator* session (*blue arrows*).

You can add frequently used object lists to your favorites.

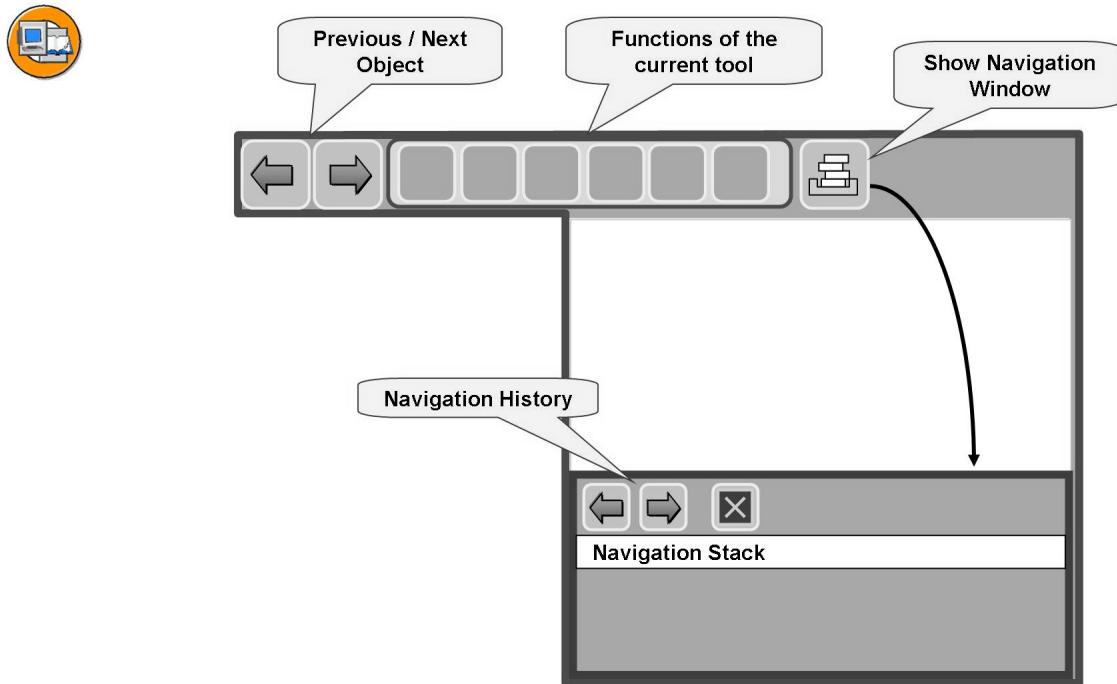


Figure 22: Functions in the Tool Area

In the tool area, a Repository object is displayed in the corresponding tool.

You can navigate between the objects that have been previously displayed in the current *Object Navigator* session (blue arrows).

Furthermore, you can also display a subwindow with the previous navigation history. When you double-click an object in the navigation history, it is displayed in the tool area.

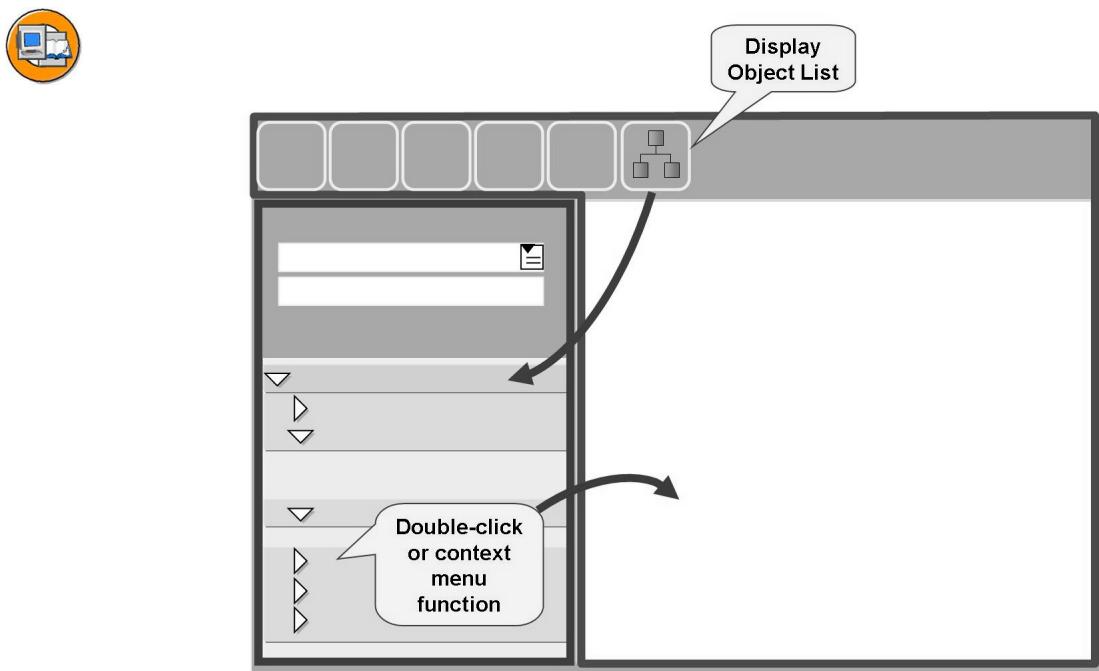


Figure 23: Synchronizing the Navigation and Tool Areas

Navigation in the navigation area is **independent** of navigation in the tools area. This allows both areas to be used in a flexible manner.

If necessary, you can synchronize both areas as follows:

- You can display an object in the tool area by double-clicking in the navigation area or by using the corresponding context menu function of the object.
- You can display the object list of an object that you are currently editing in the tool area by choosing *Display Object List* in the navigation area.

To **create a new object**, you can use the context menu for an object type in the corresponding object list. Alternatively, you can use the *Edit Object* or *Other Object ...* buttons to create any objects you want.



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 displaying Repository objects

Lesson: Developing Programs and Organizing Developments

Lesson Overview

In this lesson you will learn how to create new Repository objects, special packages, programs and transaction codes. You will also learn how change requests in the *ABAP Workbench* are used in the SAP environment to organize development projects and ensure consistent transport of changes to the productive systems.



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 transactions

Business Example

As part of a development project, you have to create a new package and a new ABAP program and make this program available to users by way of a transaction code.

Organizing Developments

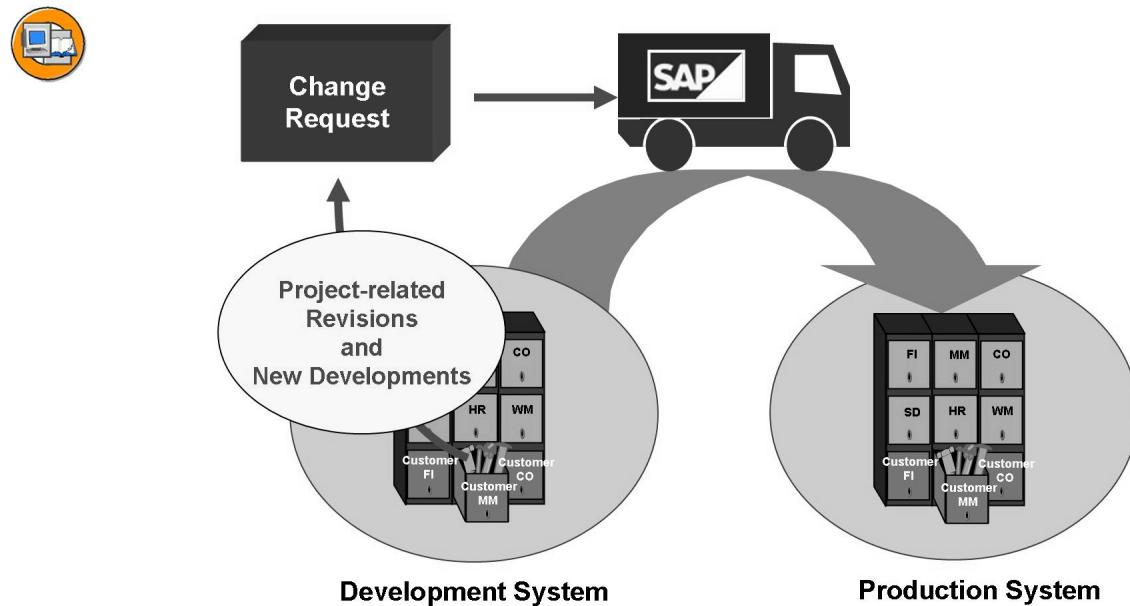


Figure 24: Transporting Development Objects

Development projects are carried out in a development system. The development objects edited or created in a project are transported to subsequent systems (test and/or production system) on project completion. At the start of a development project the **project manager** creates a *change request*, in which he or she names the employees of this project, in the *Transport Organizer* or directly in the *ABAP Workbench*. The *Transport Organizer* then creates a **task** for each project employee in the change request.

When a development object is edited or created, the relevant employee assigns this to the change request. The object is entered into the **task** of the employee. Thus, all repository objects that an employee works on during a development project are collected within his or her task.

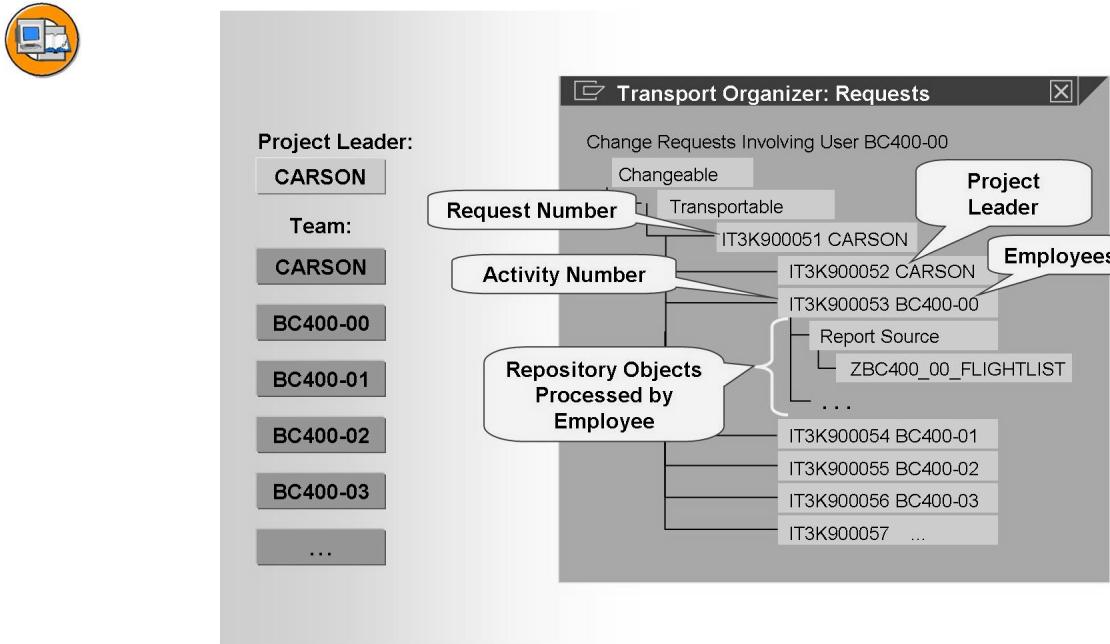


Figure 25: Organization of a Development Project in the Change Request

Organizing a development project using a change request offers the following advantages:

- Each employee can track and check his or her project-specific activities.
- The respective development objects can be processed by **all** employees involved in the project. For those developers who do not belong to the project team, the repository objects remain locked until the project is completed (=change request is released).
- **The joint transport** of the development objects processed in the project at the time of **project completion** (= **release of the change request**) is automatically ensured by assigning the objects to the change request. The **transport route** of the involved packages (in which development took place) specifies to which subsequent system they are transported.

Unlike packages that distinguish between Repository objects in a logical and functional way, change requests are project-related and therefore delimit the objects **over a period of time**. 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:
 - a) In the navigation area, choose the object type *Package* and enter the name of the package in the input field below. (Make sure you comply with the customer namespace conventions.) Choose **Enter**. If the specified package does not already exist, the system branches to a dialog for creating a package.
 - b) Choose *Edit Object* on the initial screen of the *Object Navigator*. In the dialog box, search for the option of specifying a package and enter the name of the package. Afterwards, click the button for creating the object (*F5*).



Hint: In SAP R/3 Release 4.6, packages are still called development classes. However, they are handled in an almost identical manner to the one described here. There is a minor limitation regarding Release 4.6a/b: There you only have the option given under 1b) for creating a development class.

2. Create the attributes of the package to be created.



Create a Package	
Package	ZBC400_00
Short Description	Exercises for group 00
Application Component	CA
Software Component	HOME
Transport Layer	ZDEV
Package Type	No main package

Comply with customer namespace!

To which application component does the package belong?

Where do you want to transport the development objects?

Figure 26: Setting Package Attributes (Example)

Continued on next page

The attributes have the following meaning (detailed information can be obtained by using the field help, **F1** pushbutton):

Application Component

Determine the location of the package within the application hierarchy by specifying the corresponding application component.

Software Component

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

Transport layer

If you are executing your own developments, you must set up a transport layer for customer developments, which must be specified here. The transport layer determines if the objects of this package are to be transported to a subsequent system and, if so, to which system.

Package Type

You can choose between three package types:

- **Standard package** (can contain Repository objects and other packages)
- **Main package** (can **only** contain other packages)
- **Structure package** (can **only** contain main packages)

Continued on next page

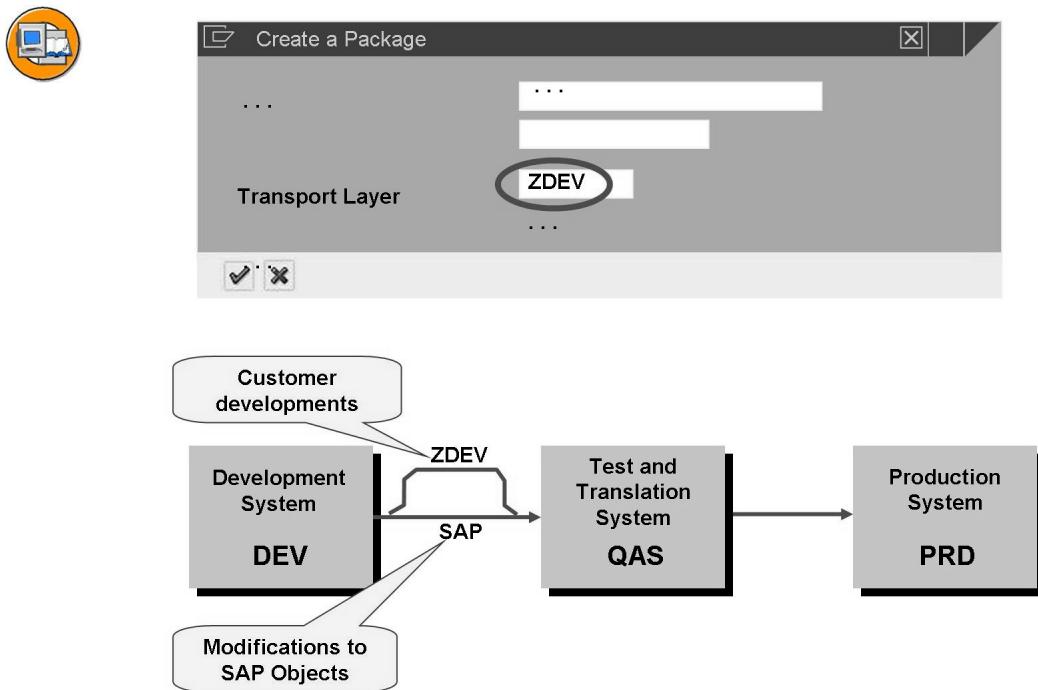


Figure 27: Specifying a Transport Route

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. Simply select the relevant request by double-clicking.

Continued on next page

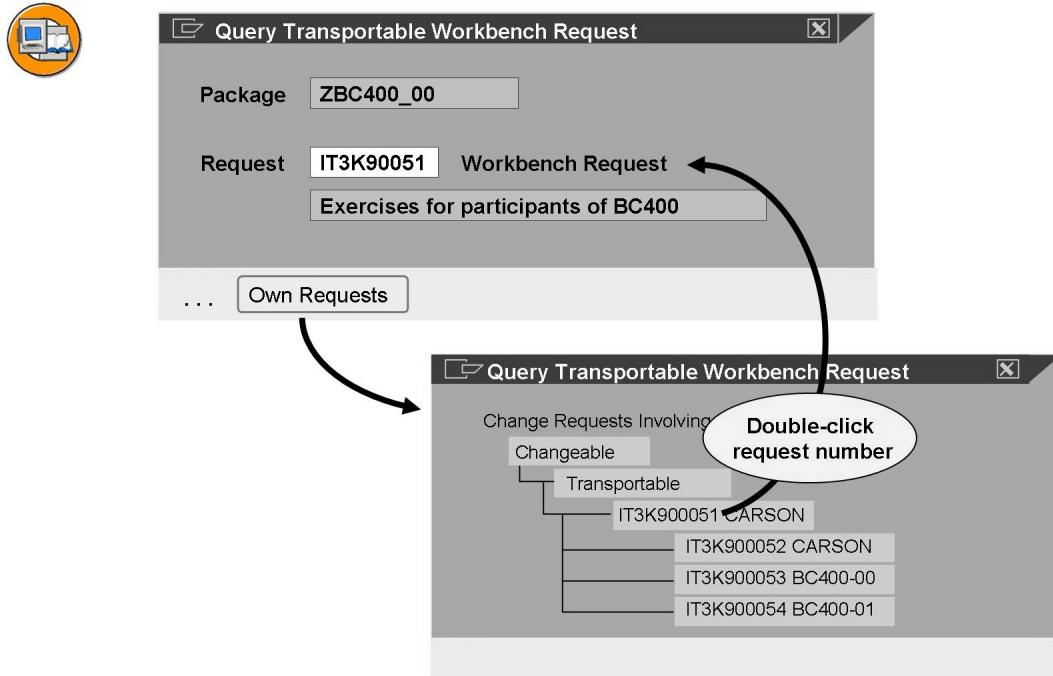


Figure 28: Assignment to a Change Request

→ **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 of the course has a task within this change request.

Developing an ABAP Program

In this section you will learn how to create programs and transactions in the *ABAP Workbench*. You will also learn about the purpose of program activation.



Creating an ABAP Program

1. Navigate to the *Object Navigator*. Three options are available:
 - a) In the navigation area, choose the object type *Program* and enter the name of the program in the input field below. (Comply with the customer namespace conventions.) Confirm with **Enter**. If the program does not exist, the system goes to the dialog that lets you create a program.
 - b) Display the package where you want to create the program. You can branch to the dialog for creating a program via the context menu for the package or the *Programs* node.
 - c) Choose the *Edit Object* button on the initial screen of the *Object Navigator*. In the dialog box, look for the option for specifying a program and enter the name of the program. Afterwards, click the button for creating the object (*F5*).
2. In this training course, remove the flag for *With TOP Include*. (Otherwise, your source code would be distributed to several programs.)
3. Change the title to a self-explanatory short text and, within this course, always choose *Executable Program* as the program type. All other program attributes are optional. Refer to the **F1** help for details.

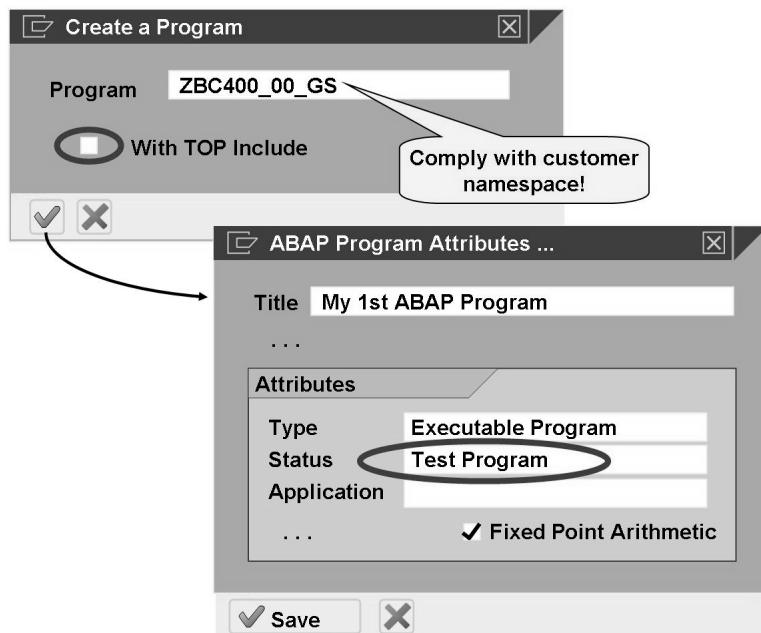


Figure 29: Creating an ABAP Program

Introduction to the ABAP Programming Language

The ABAP programming language ...



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

It is especially designed for dialog-based business applications.

To support the type-specific processing of data, type conversions and *type casting* are supported.

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

The Open SQL standard embedded in ABAP allows direct database accesses.

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

The ABAP syntax is platform-independent. This means that it always has the same meaning or function, irrespective of the relational database system and operating system for the application and presentation server.

Applications implemented in ABAP will also be able to run in future releases (upward compatibility of the language).



General Structure of an ABAP Statement

X X X	Y Y Y	.
ABAP keyword	Additions and operands (keyword-specific)	Period to close the statement

Example program

```

PARAMETERS pa_num TYPE i.

DATA gv_result TYPE i.

MOVE pa_num TO gv_result.

ADD 1 TO gv_result.

WRITE 'Your input:' .
WRITE pa_num.

NEW-LINE .

WRITE 'Result: ' .
WRITE gv_result.

```

Figure 30: General ABAP Syntax I

For ABAP syntax, the following applies in general:

- ABAP programs are comprised of individual **sentences** (statements).
- The first word in a statement is called an **ABAP keyword**.
- Each statement ends with a **period**.
- Words must always be separated by at least one **space**.
- Statements can be indented as you wish.
- With keywords, additions and operands, the ABAP runtime system **does not differentiate between upper and lowercase**.



Hint: Although the ABAP runtime system does not differentiate between upper and lowercase, it has become customary to write keywords and their additions in uppercase letters and operands in lowercase. This form of representation will also be used in this course.

For indentations and for converting uppercase/lowercase letters, you can use the *Pretty Printer* (the correspondingly labeled button in the Editor). You can use the following menu path in the *Object Navigator* to make a user-specific setting for the *Pretty Printer*: *Utilities → Settings → ABAP Editor → Pretty Printer*.

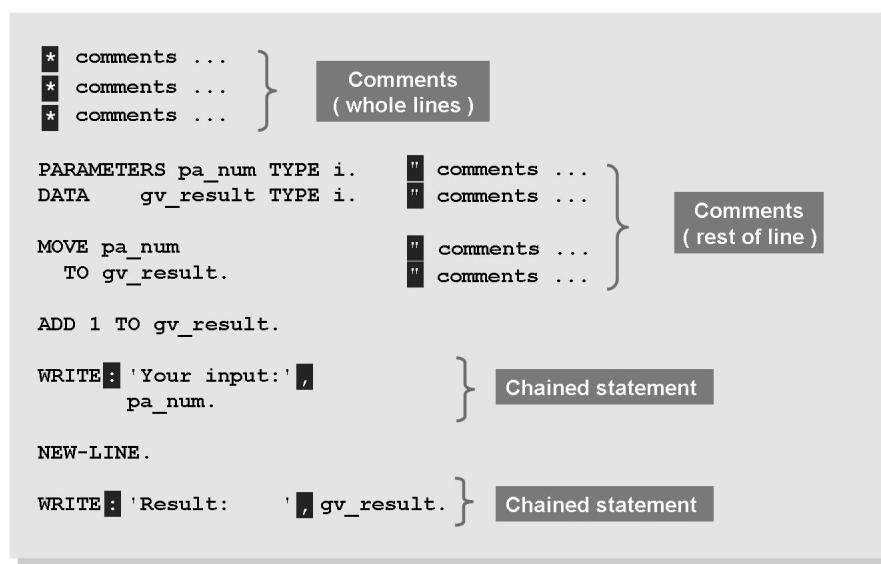


Figure 31: General ABAP Syntax II

- Statements can extend **beyond one line**.
- You can have several statements in a single line (though this is not recommended).
- Lines that begin with asterisk * in the first column are recognized as **comment lines** by the ABAP runtime system and are ignored.
- **Double quotations marks "**" indicate that the remainder of a line is a comment.

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

- Write the identical beginning part of the records followed by a colon.
- After the colon, list the end parts of the statements (separated by commas).
- Blank spaces and line breaks are allowed before and after the separators (colons, commas, periods).



Hint: Note that this short form merely represents a simplified form of syntax, and does not offer an improvement in performance. As is otherwise the case, the ABAP runtime system processes each of the individual statements.

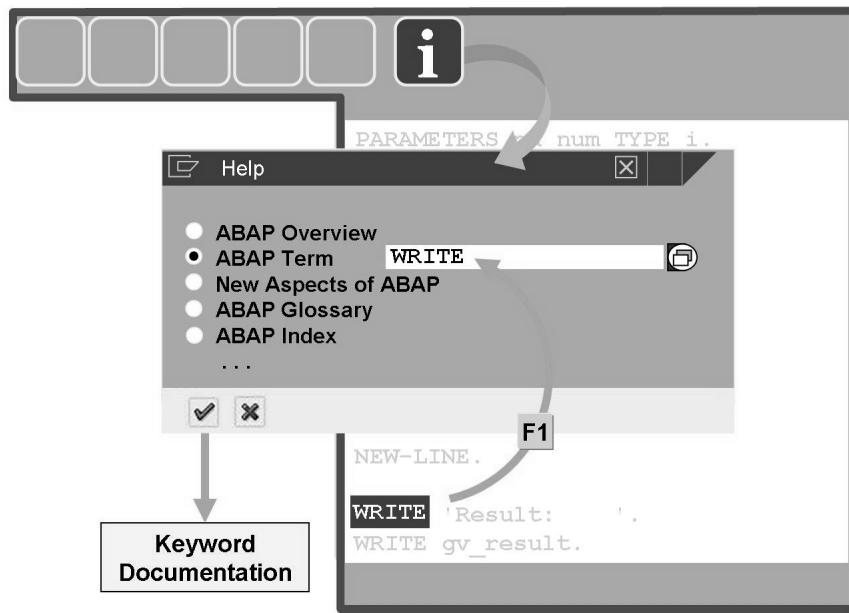


Figure 32: Keyword Documentation in the ABAP Editor

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

- The **F1** key takes you directly to the documentation for the statement on which the cursor is positioned.
- The **i** button with the description *Help on ...* takes you to a dialog box where you can enter the required ABAP keyword.

The ABAP Editor

There are several editors you can use to develop ABAP programs, depending on the release and support package level. The choice of editor is user-specific and is made in the *Object Navigator* by following the menu path *Utilities → Settings → ABAP Editor*.

The latest editor (*Front-End Editor (New)*) offers a number of new options and easy-to-use additional functions in comparison with the classic editor (*Front-End Editor (Old)*). It was developed for *SAP NetWeaver 7.0*. It can also be imported by support package for *SAP Web Application Server 6.40* and *SAP Web Application Server 6.20* (SP 18 or SP 59).



```

REPORT zbc400_00_hello.

* Data declarations
PARAMETERS pa_num TYPE i.          " input via selection screen
DATA gv_result TYPE i.             " variable for internal use

* Processing
MOVE pa_num                      " One statement
TO gv_result.                     " over more than one line

DO 10 TIMES.
  ADD 1 TO gv_result.
ENDDO.

* Output
WRITE: 'Your input:',           I
      pa_num.

NEW-LINE.

WRITE: 'Result: ', gv_result.


```

Ze 18, Sp 1 | Ze 8 - Ze 29 von 29 Zeilen

Figure 33: The previous ABAP Editor



The new ABAP Editor interface includes the following features highlighted in Figure 34:

- Line numbers
- Bookmarks for quick navigation
- Blocks that can be compressed
- Automatic selection of changed lines
- Different colors for keywords, variables, literals, ...
- Current line
- Current nesting
- User-specific settings

```

REPORT zbc400_00_hello.

* Data declarations
PARAMETERS pa_num TYPE i.          " input via selection screen
DATA gv_result TYPE i.             " variable for internal use

* Processing
MOVE pa_num                      " One statement
TO gv_result.                     " over more than one line

DO 10 TIMES.
  ADD 1 TO gv_result.
ENDDO.

* Output
WRITE: 'Your input:',           I
      pa_num.

NEW-LINE.

WRITE: 'Result: ', gv_result.


```

Scope: ID00 | ABAP | Ln 20 Col 22 Ch 22 | * |

Figure 34: The new ABAP Editor

Some of the important options provided by the new editor:

- You can choose different display colors for different objects in the source code.
- Fonts and font sizes can be set for each individual user.
- Blocks of source code (loops, conditional branches) can be compressed for display to provide a better overview.
- You can use bookmarks in order to find relevant points in the source code faster.
- Displaying line numbers and the current nesting improves orientation.
- Once you have typed a few characters, the editor automatically suggests complete words for ABAP keywords and data objects. This considerably reduces the actual amount of typing required.

Activating Programs

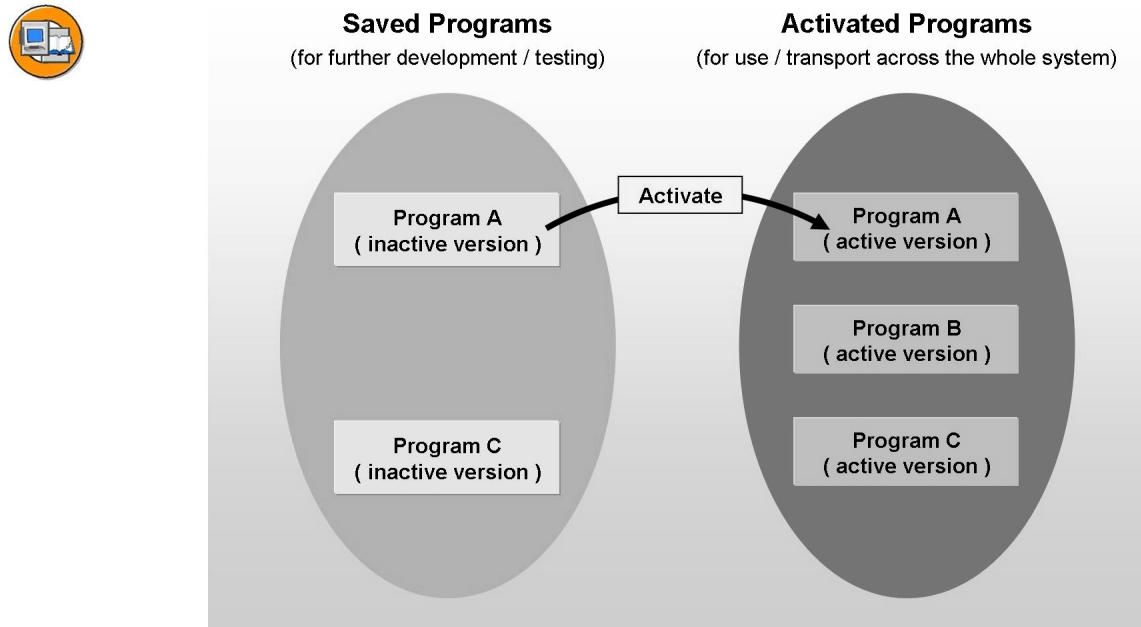


Figure 35: Inactive and Active Development Objects

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

After that, you have an active version and an inactive version of the object. At the end of your object development, you have to activate the inactive “editing version” of the object. This version becomes the new active version of the object.

Note that the request release and hence the transport of the developed objects are only possible if all objects in the request have been activated.

If your program is available in both versions (active and inactive), you can switch between the displays of these two versions by using the corresponding button in the editor.

Whenever you activate a program, the system displays a list of all inactive objects that you have processed: Your **worklist**. Choose those objects that you wish to activate with your current activation transaction.

The activation of an object includes the following functions:

- Saving the object as an inactive version
- Syntax or consistency check of the inactive version
- Overwriting the previously active version with the inactive version (only after a successful check)
- Generating the relevant runtime object for later executions, if the object is a program.

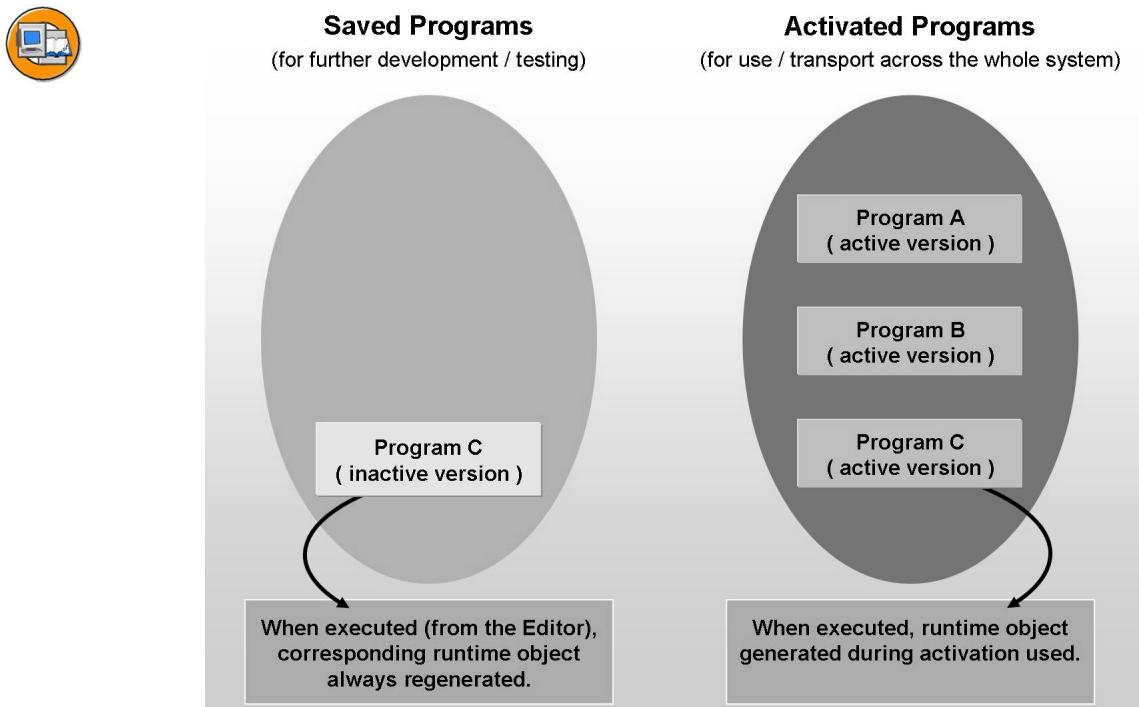


Figure 36: Generating Runtime Objects

When you generate a development object, the system creates a separate runtime object (LOAD compilation) and also stores it in the *Repository*. This generated version is the version that is executed (interpreted) at runtime.

If your program has an inactive version as well as an active version, you can get both versions to run as follows:

- If you start your program using the context menu of the navigation area or by means of a transaction, the active version is used. This means that the LOAD generated for the last activation is executed.
- By contrast, if you start the inactive version loaded to the editor using the *F8* button, a temporary runtime object is generated from it and executed.

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 become “visible” system-wide only when the object is activated.

Creating Transactions

Only transactions can be included in a role menu and in a user's favorites. Hence, if you want to place a program there, you must create a transaction that represents the program and integrate this in the menu. Alternatively, you can start the program by entering the transaction code in the command field.



Creating Transactions

1. In the *Object Navigator*, display the object list for your program.
2. In the navigation area, use the context menu of the program to choose *Create → More → Transaction*.
3. Enter the required transaction code. (Make sure you comply with the customer namespace conventions!)
Assign a short text and choose the label *Program and Selection Screen (Report Transaction)*.
4. On the next screen, enter the name of the program and choose *Professional User Transaction*.

Under *GUI enabled* set the indicator *SAP GUI for Windows*.

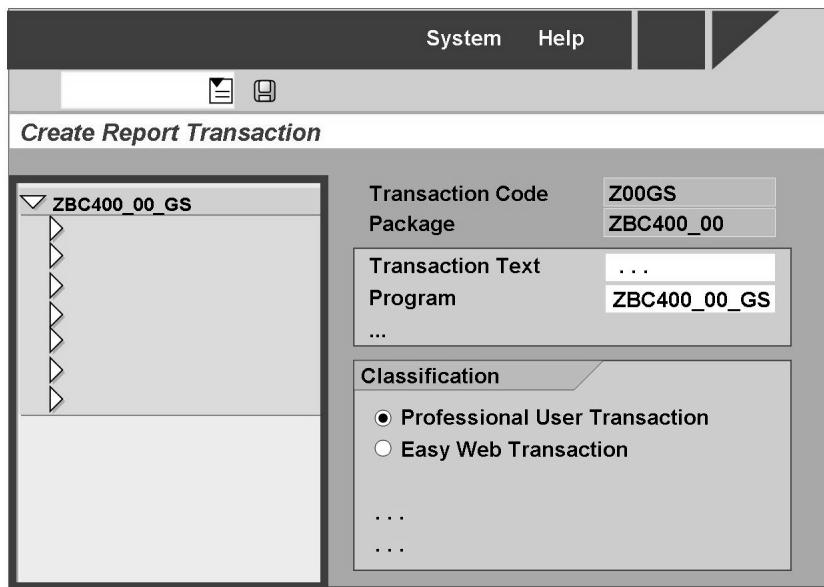


Figure 37: Creating a Transaction

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



Adding Transactions to your Personal Favorites



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.

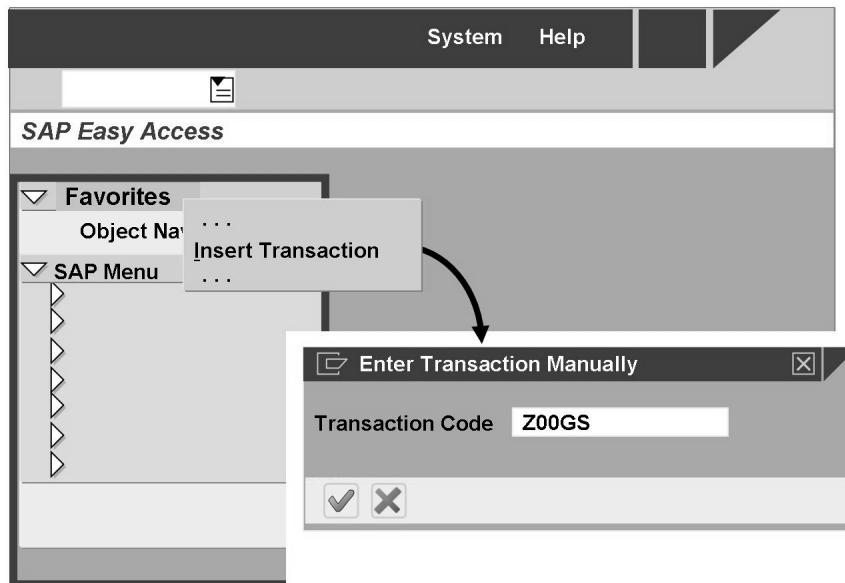


Figure 38: Inserting a Transaction Code into the Personal Favorites

Result

The short text of the transaction now appears under the *Favorites* and you can start the corresponding program by double-clicking.

Closing Development Projects

Once a project employee has completed the required development task, he or she carries out a quality check and releases the task within the change request.

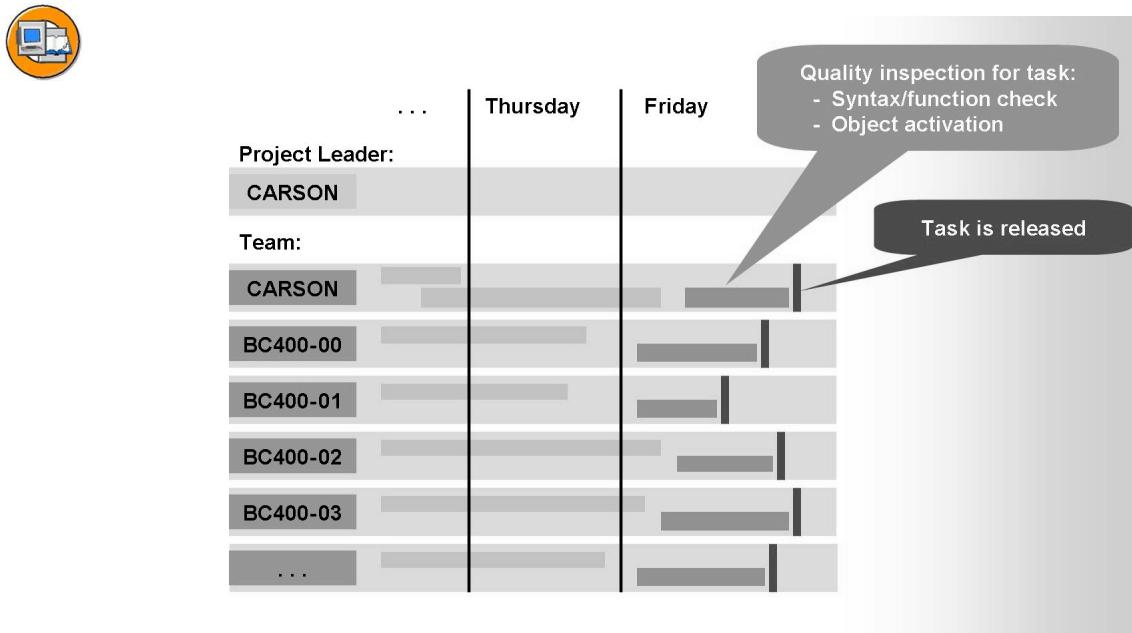


Figure 39: The Developer Releases his Task

The corresponding object entries as well as the change locks of the object for third parties that were automatically set at the start of project are transferred from the task to the request.

However, all project employees can still edit these objects.

Once all the tasks in a change request have been released, the project manager carries out the final check for the objects and releases the change request. This concludes the project.

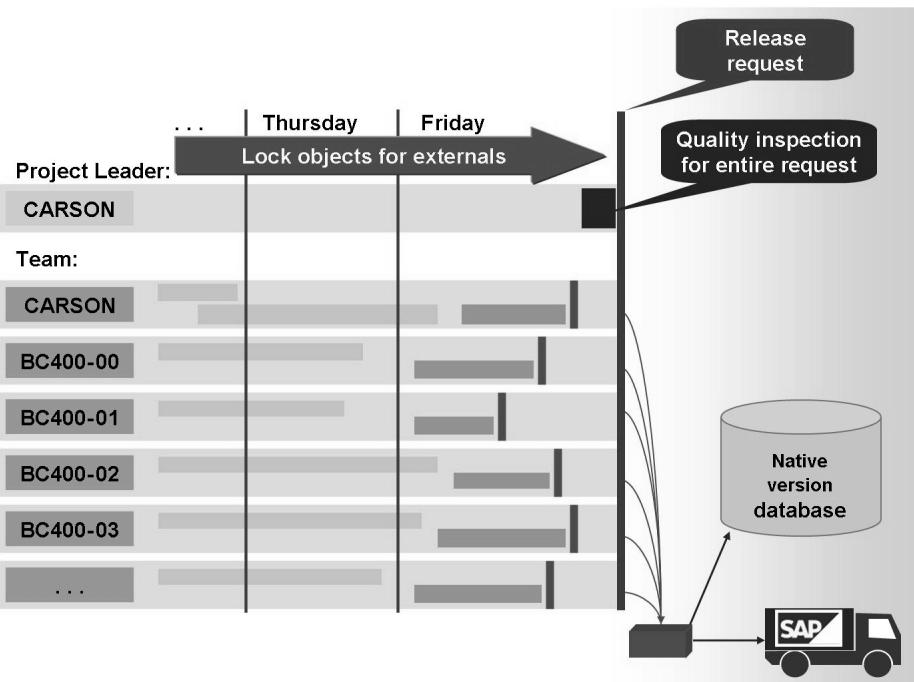


Figure 40: Closing the Project by Releasing the Request (Project Manager)

When the request is released, all object locks that belong to the request are removed.

Copies of the developed objects are exported to the system's own transport directory where they stay until the system administrator imports them to their intended target system.

Another copy of the exported development objects is stored in the system's own version database.

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 are given the task of developing evaluation programs for several airline companies.

To group the Repository objects together logically and transport them, you have to create a package.

Task 1:

Log on to the training system.

1. First log on to the operating system, then log on to the training system. To do so, use the client, user name and initial password given to you by your instructor.
2. Enter a new personal password.

Task 2:

Create a package named **ZBC400_##**, where ## stands for the group number given to you by your instructor.

1. Open the *Object Navigator* and create the package there.
2. On the dialog box that appears, enter a short description for your package. Assign the package to the application component **CA** and the software component **HOME**. Take the default value for all the other input fields in the dialog box.
3. In the following dialog box, assign the package to a change request. Use the change request in which the instructor has already created a task for your user.



Hint: To determine this request, choose the *Own Requests* pushbutton.

Continued on next page

Result

You have successfully created your own package. Assign all the Repository objects that you create in the following exercises to this package and use the same change request in each case.

Solution 1: Organizing Developments

Task 1:

Log on to the training system.

1. First log on to the operating system, then log on to the training system. To do so, use the client, user name and initial password given to you by your instructor.
 - a) Carry out this step in the usual way.
2. Enter a new personal password.
 - a) Carry out this step in the usual way.

Task 2:

Create a package named **ZBC400_##**, where ## stands for the group number given to you by your instructor.

1. Open the *Object Navigator* and create the package there.
 - a) Perform this step as described in the participants' handbook.
2. On the dialog box that appears, enter a short description for your package. Assign the package to the application component **CA** and the software component **HOME**. Take the default value for all the other input fields in the dialog box.
 - a) Perform this step as described in the participants' handbook.
3. In the following dialog box, assign the package to a change request. Use the change request in which the instructor has already created a task for your user.



Hint: To determine this request, choose the *Own Requests* pushbutton.

- a) Perform this step as described in the participants' handbook.

Result

You have successfully created your own package. Assign all the Repository objects that you create in the following exercises to this package and use the same change request in each case.

Exercise 2: Developing a Simple ABAP Program

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 are to create an ABAP program that allows simple user input and outputs information to a screen.

Task 1:

In your package, create an executable program named **ZBC400##_HELLO**.

1. Create the program **ZBC400##_HELLO** without a “TOP include”.
2. Make sure that the correct program type *Executable Program* is proposed in the next dialog box and set the status of the program to a meaningful value.

Task 2:

Edit the program. Enable the user to enter a name on the *selection screen* that will be output later in the *list*, together with the text “Hello”.

1. Define an input field (parameter) on the selection screen:

```
PARAMETERS pa_name TYPE string.
```

2. Implement the output of any text in the list, for example, the text “Hello World!”. To do this, use the ABAP WRITE statement.
3. Make sure that the following is output in a new line (NEW-LINE statement).
4. Then implement the output of the text “Hello” and the output of the name that the user has entered. Instead of using two separate statements for this, use a **chained statement**.
5. Check your program for syntax errors. Test and activate it.

Solution 2: Developing a Simple ABAP Program

Task 1:

In your package, create an executable program named **ZBC400_##_HELLO**.

1. Create the program **ZBC400_##_HELLO** without a “TOP include”.
 - a) Carry out this step as described in the training material.
2. Make sure that the correct program type *Executable Program* is proposed in the next dialog box and set the status of the program to a meaningful value.
 - a) Carry out this step as described in the participants' handbook. Set the program status to *Test Program*.

Task 2:

Edit the program. Enable the user to enter a name on the *selection screen* that will be output later in the *list*, together with the text “Hello”.

1. Define an input field (parameter) on the selection screen:

```
PARAMETERS pa_name TYPE string.
```

 - a) See source code excerpt from the model solution.
2. Implement the output of any text in the list, for example, the text “Hello World!”. To do this, use the ABAP WRITE statement.
 - a) See source code excerpt from the model solution.
3. Make sure that the following is output in a new line (NEW-LINE statement).
 - a) See source code excerpt from the model solution.
4. Then implement the output of the text “Hello” and the output of the name that the user has entered. Instead of using two separate statements for this, use a **chained statement**.
 - a) See source code excerpt from the model solution.

Continued on next page

5. Check your program for syntax errors. Test and activate it.
 - a) Perform this step as described in the participants' handbook. Use the pushbuttons *Check* (*Ctrl+F2*), *Direct* (*F8*) and *Activate* (*Ctrl+F3*).

Result

Source code extract from the model solution:

```
*&-----  
*& Report BC400_GSS_HELLO  
*&-----  
REPORT bc400_gss_hello.  
  
PARAMETERS: pa_name TYPE string.  
  
WRITE 'Hello World!' .  
  
NEW-LINE.  
  
WRITE: 'Hello',  
      pa_name.
```


Exercise 3: Create Transactions

Exercise Objectives

After completing this exercise, you will be able to:

- Create transactions
- Define transaction codes as personal favorites

Business Example

Users in your company should be able to start their ABAP programs as transactions or define them as personal favorites.

Task 1:

Create a **report transaction** (suggested name: **ZBC400_##_HELLO**) with which you can start your executable program **ZBC400_##_HELLO**.

1. Create the new transaction in your package. Enter a short description and select the appropriate transaction type (start object).
2. Refer to the program that is to be executed. Choose the correct transaction classification and make sure that it is GUI enabled.
3. Save and test the transaction.
4. Confirm that the program can now be started by entering the transaction code in the command field.

Task 2:

Include the new transaction in your personal favorites in the *SAP Easy Access* menu.

1. Open the *SAP Easy Access* menu, choose the function *Insert Transaction* under Favorites and enter the transaction code.
2. Execute your program by double-clicking the new favorite entry.

Solution 3: Create Transactions

Task 1:

Create a **report transaction** (suggested name: **ZBC400_##_HELLO**) with which you can start your executable program ZBC400_##_HELLO.

1. Create the new transaction in your package. Enter a short description and select the appropriate transaction type (start object).
 - a) Make sure that you select *Program and Selection Screen (Report Transaction)* as the start object.
2. Refer to the program that is to be executed. Choose the correct transaction classification and make sure that it is GUI enabled.
 - a) Carry out this step as described in the participants' handbook. Make sure that the transaction is classified as a *Professional User Transaction* and set it as GUI enabled for *SAP GUI for Windows* at least.
3. Save and test the transaction.
 - a) Use the and pushbuttons for this purpose.
4. Confirm that the program can now be started by entering the transaction code in the command field.
 - a) Carry out this step in the usual way.

Task 2:

Include the new transaction in your personal favorites in the *SAP Easy Access* menu.

1. Open the *SAP Easy Access* menu, choose the function *Insert Transaction* under Favorites and enter the transaction code.
 - a) Carry out this step as described in the participants' handbook.
2. Execute your program by double-clicking the new favorite entry.
 - a) Carry out this step in the usual way.



Lesson Summary

You should now be able to:

- Name and use the utilities for orderly software development
- Create packages
- Create programs
- Create transactions



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 displaying Repository objects
- Name and use the utilities for orderly software development
- Create packages
- Create programs
- Create transactions

Related Information

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



Test Your Knowledge

1. Repository objects are not client-specific.

Determine whether this statement is true or false.

- True
- False

2. Customer objects can be included in SAP packages.

Determine whether this statement is true or false.

- True
- False

3. Which two tools are used for searching for Repository objects?

When should each search tool be used?

4. You must create a new application program for each language used in the system.

Determine whether this statement is true or false.

- True
- False

5. What is a chained statement?

6. When is the transport of development objects for a development request triggered?

Choose the correct answer(s).

- A When an object is saved
- B When an object is activated
- C When a task is released
- D When a request is released



Answers

1. Repository objects are not client-specific.

Answer: False

Repository objects are cross-client objects and can therefore be used system-wide.

2. Customer objects can be included in SAP packages.

Answer: False

Customer objects must be assigned to customer packages.

3. Which two tools are used for searching for Repository objects?

When should each search tool be used?

Answer: Repository Information System (for application-independent searches) and the SAP Application Hierarchy (for application-related searches)

4. You must create a new application program for each language used in the system.

Answer: False

Only the texts belonging to the program need to be translated. This topic will be discussed later on in the training session.

5. What is a chained statement?

Answer: An abbreviated form of notation for several statements that have the same beginning.

6. When is the transport of development objects for a development request triggered?

Answer: D

Unit 3

Basic ABAP Language Elements

Unit Overview

The unit overview lists the individual lessons that make up this unit.



Unit Objectives

After completing this unit, you will be able to:

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

Unit Contents

Lesson: Working with Elementary Data Objects.....	68
Exercise 4: Basic ABAP Statements.....	95
Exercise 5: Debugging Statements on Elementary Data Objects	101

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 and use these in a program. You will also learn some basic ABAP statements. Furthermore, you will find out how to use the *ABAP Debugger* for analyzing a program flow.



Lesson Objectives

After completing this lesson, you will be able to:

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

Business Example

You are supposed to use simple variables in your programs and edit these with simple statements.

Furthermore, you want to use the *ABAP Debugger* to find the semantic errors in your programs.

Data Types and Data Objects

A formal variable description is called a **data type**. By contrast, a variable or constant that is defined concretely by means of a data type is called a **data object**.

Using Data Types

The following graphic shows how data types can be used:

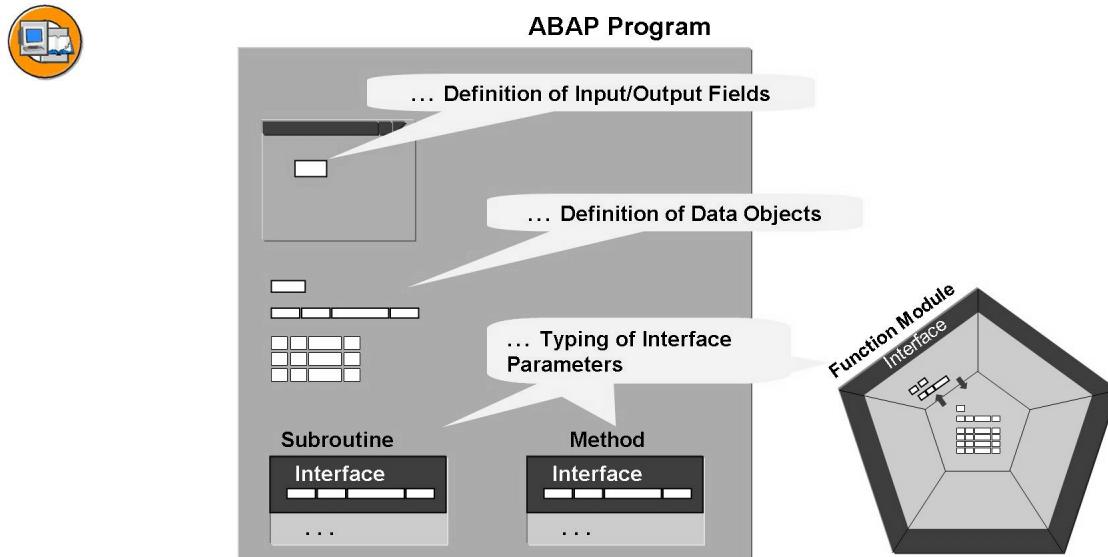


Figure 41: Using Data Types

Definition of data objects

The type of a data object defines its technical (and possibly also semantic) properties.

Definition of interface parameters

The type of an interface parameter determines the type of the actual parameter that is transferred when the modularization unit is called.

Definition of input/output fields

The type of an input/output field can provide further information in addition to the technical characteristics, such as the field and value input help.

In this lesson we will focus on how to use types for defining program-internal variables.

ABAP Standard Types

Let's have a look at the ABAP standard types predefined by SAP (implemented types) first. They are divided into two groups: **Complete** and **incomplete** types.

The following implemented ABAP standard types are complete. This means that they already contain a type-specific, fixed length specification:

Complete ABAP standard types

**D**

Type for date (**D**), format: **YYYYMMDD**, length 8 (fixed)

T

Type for time (**T**), format: **HHMMSS**, length 6 (fixed)

I

Type for integer (**I**), length 4 (fixed)

F

Type for floating point number (**F**), length 8 (fixed)

STRING

Type for dynamic length character string

XSTRING

Type for dynamic length byte sequence (HeXadecimal string)

The following standard types do not contain a fixed length (incomplete). When they are used to define data objects, you therefore still need to specify the length of the variable.

Incomplete ABAP standard types**C**

Type for character string (Character) for which the fixed length is to be specified

N

Type for numerical character string (Numerical character) for which the fixed length is to be specified

X

Type for byte sequence (HeXadecimal string) for which the fixed length is to be specified

P

Type for packed number (Packed number) for which the fixed length is to be specified.

(In the definition of a packed number, the number of decimal points may also be specified.)

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

Local Data Types

Using standard types, you can declare **local data types** in the program that can then be more complete or complex than the underlying standard types. Local data types only exist in the program in question and hence can only be used there. The declaration is made using the TYPES statement.



ABAP Program

```
REPORT ...  
  
Declaration of local data types  
  
TYPES gty_c_type TYPE c LENGTH 8.  
  
TYPES gty_n_type TYPE n LENGTH 5.  
  
TYPES gty_p_type TYPE p LENGTH 3 DECIMALS 2.
```

Figure 42: Declaring Local Types



Hint: There is an alternative syntax for specifying the length with the LENGTH addition that you will often find in older programs. Here, the length is specified in parentheses directly after the name of the type. For example:

```
TYPES gty_c_type(3) TYPE c.  
TYPES gty_p_type(3) TYPE p DECIMALS 2.
```

To improve the readability of your program, you should no longer use this obsolete syntax.

Global Data Types

A data type defined in the *ABAP Dictionary* is called **global** as it can be used throughout the entire system (that means in the entire SAP system in question).

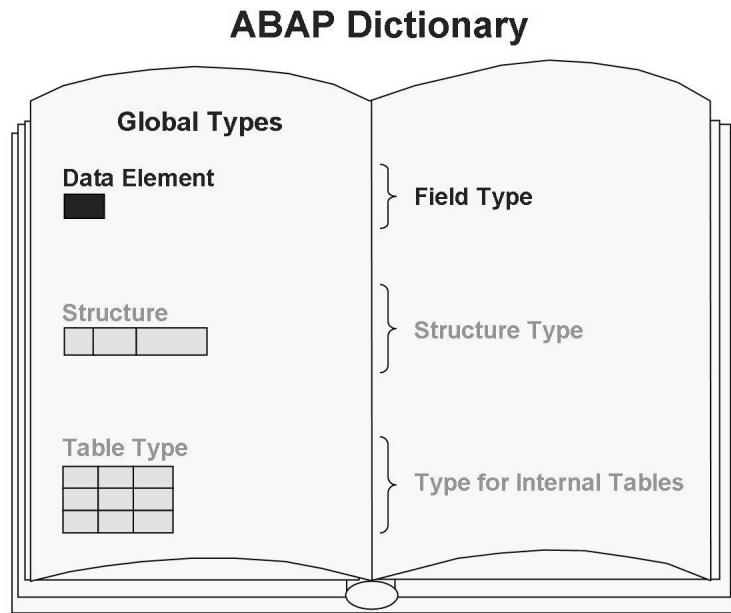


Figure 43: Global Data Types in the Dictionary

In this lesson, we will only describe the data elements and use them as field types. The other global data types are used later in this course.

Definition of Variable Data Objects

In the previous section we saw that there are three categories of data types in total: Standard, local and global.

We now want to use these types to define variables (data objects).

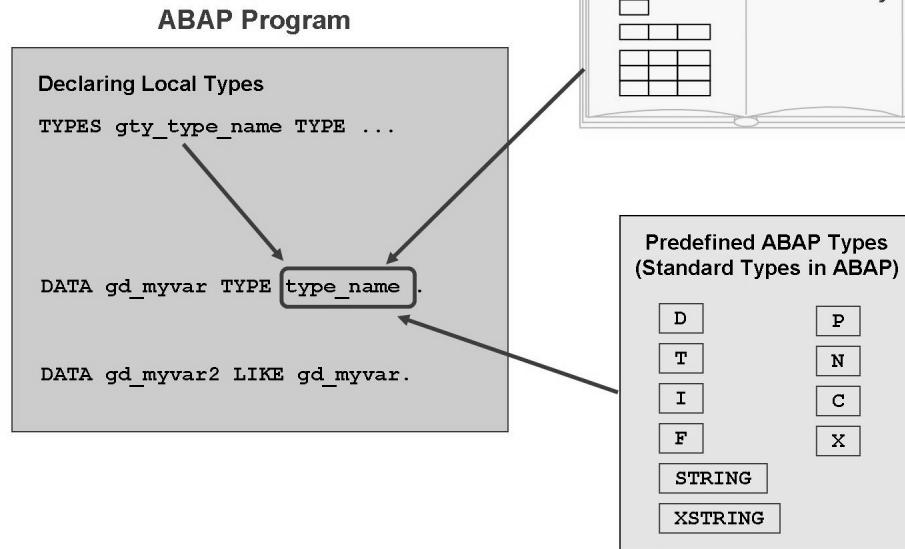


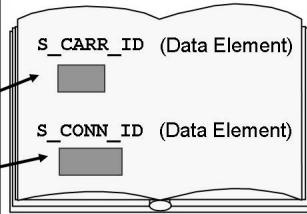
Figure 44: Defining Data Objects

Data objects are always defined with the DATA key word. You can use an ABAP standard type, a local type, or a global type to type a data object.

You can refer to an already defined data object when defining additional variables (LIKE addition).



```
...
TYPES gty_percentage TYPE p LENGTH 3 DECIMALS 2.
DATA: percentage1  TYPE gty_percentage,
      percentage2  TYPE gty_percentage,
      number1       TYPE i VALUE 17,
      number2       LIKE number1,
      city          TYPE c LENGTH 15,
      carrid        TYPE s_carr_id,
      connid        TYPE s_conn_id.
```



percentage1	0 0 0 0 0 +
percentage2	0 0 0 0 0 +
number1	17
number2	0
city	
carrid	
connid	0 0 0 0

Figure 45: Examples of the Definition of Elementary Data Objects

You can use the VALUE addition to preassign the value of an elementary data object.



Hint: In the DATA statement, you also have the option of specifying the length in parentheses after the name of the variable, for example:

```
DATA gd_myvar(15) TYPE c.
DATA gd_myvar_p(4) TYPE p DECIMALS 2.
```

Here too, it is recommended that for the sake of readability, you use the LENGTH addition.

If the length specification is missing from a variable definition, a default length for the (incomplete) standard type is drawn (length 1 for types C, N and X, and length 8 for type P). If the length specification is missing altogether, the standard type C is used. The "DATA gd_myvar ." statement without a type and length specification therefore defines a type C variable with length 1.



Hint: To improve readability, you should avoid this option of an implicit type or length specification.

For more details, refer to the keyword documentation for the TYPES or DATA statement.

Literals, Constants and Text Symbols



Fixed Data Objects Without Label

Literals

Numeric Literals

Positive Integer : 123
Negative Integer : -123

Text Literals

String : 'Hello'
Decimal Number : '123.45'
Floating Point Number : '123.45E01'

Fixed Data Objects with Label

Constants

```
CONSTANTS gc_myconst TYPE type_name VALUE {literal | IS INITIAL}.
```

Figure 46: Literals and Constants (Fixed Data Objects)

Fixed data objects have a fixed value, which is already fixed when the source text is written and cannot be changed at runtime. **Literals** and **constants** belong to the fixed data objects.

You can use **literals** to specify fixed values in your programs. There are **numeric literals** (specified without quotation marks) and **text literals** (specified with quotation marks). The figure above shows some examples of literals.

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



Hint: If possible, avoid literals completely when using statements. Use constants and text symbols instead. This makes it considerably easier to maintain your program.

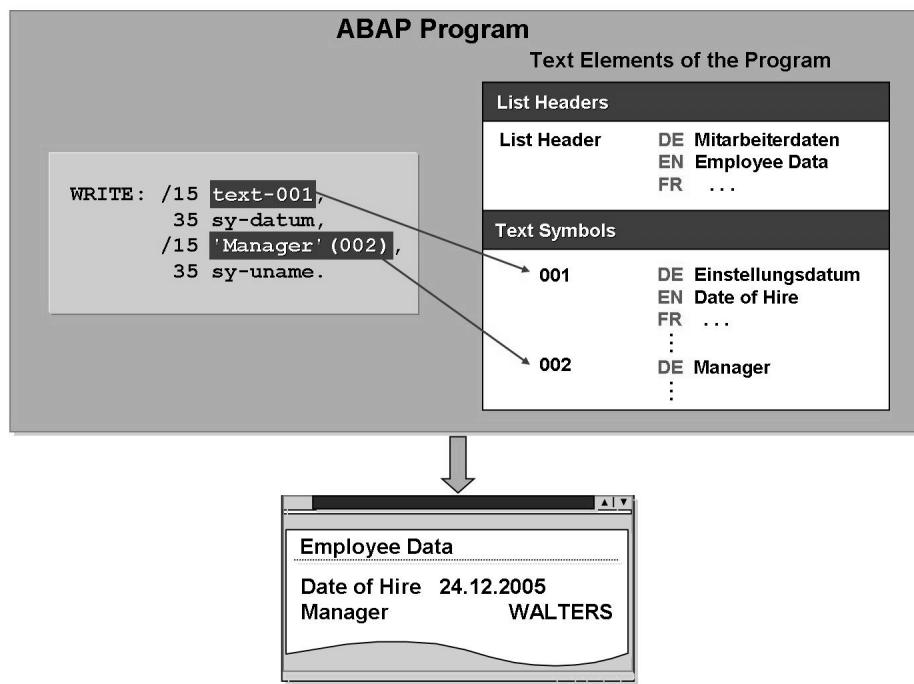


Figure 47: Text Symbols

A very important principle in ABAP development is **multilingual capability**. This means that when texts are displayed on the user interface, the **logon language** of the current user is taken into account. The use of text literals in this context is critical, since the literal only exists in one version in the source code, and is therefore language-independent.

You should therefore only create language-dependent texts as text literals for testing purposes. For productive programs that should be executable with different logon languages, the ABAP programming language provides so-called **text symbols**.

Text symbols each belong to a particular program and can be used in it directly. They are stored outside the source code in their own Repository object, the **text pool** for the program. Text symbols can be translated into different languages and each stored with a language indicator in the text pool (see the above graphic). If the program accesses a text symbol when it is executed, the system automatically takes account of the logon language of the user and supplies the text in this language.

A text symbol is identified by means of a **three character alphanumeric ID xxx**.

To use a text symbol in your program, you need to address it as **TEXT-xxx**, where **xxx** stands for the three character text symbol ID.

In order to make specifying a text symbol **more intuitive** you can also use the following syntax instead of TEXT-xxx: '...' (xxx). Here, '...' should be the text of the text symbol in the original language of the program.

There are two options for **defining text symbols** for your program:

- From the *ABAP Editor*, choose *Goto* → *Text Elements* → *Text Symbols*, or
- You address the text symbol in your source code using the syntax described above and double-click its ID. (Forward navigation).

To **translate the text symbols** of your program choose *Goto* → *Translation* from the menu in the *ABAP Editor*.

 **Hint:** Remember that text elements also have to be **activated**.

Comparison: Local and Global Data Types

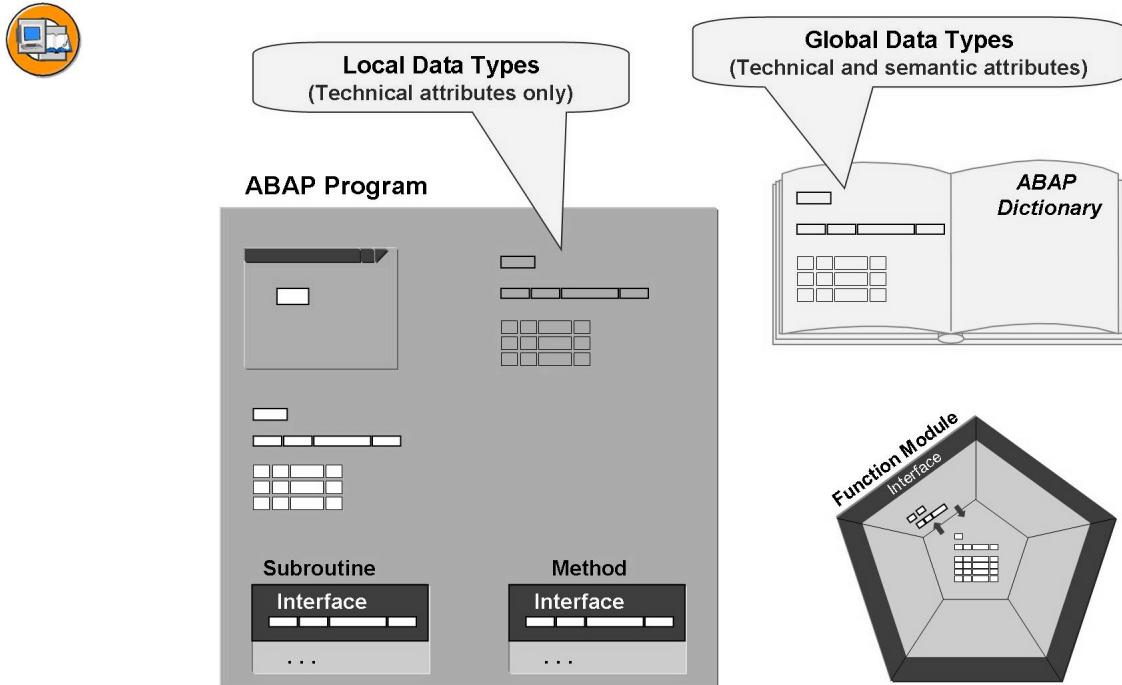


Figure 48: Local vs. Global Data Types

Local data types can only be used in the program where they are defined. Global data types, in contrast, can be used throughout the entire system. Apart from that, the latter also have the following advantages:

- The system-wide usability of global types increases the system's consistency and their reusability reduces the amount of maintenance required.
- In the *ABAP Dictionary* you have the option of generating a *where-used list* for a global data type. The where-used list lists the *Repository* objects that use the data type in question.
- In addition to the technical information, global data types can also contain semantic information that corresponds to the business descriptions of the objects to be defined. They can then also be used for designing screen displays (for example, the short description on the left of the input field).

Local data types should be used only if used exclusively in the program where they are defined and if semantic information does not matter for the definition of the corresponding data objects.

Basic ABAP Statements

In this section you will learn how to fill elementary data objects with values and perform calculations in ABAP. You will also be given an introduction to the constructions you can use to control the program flow dependent on the content of the data objects.

Value Assignments

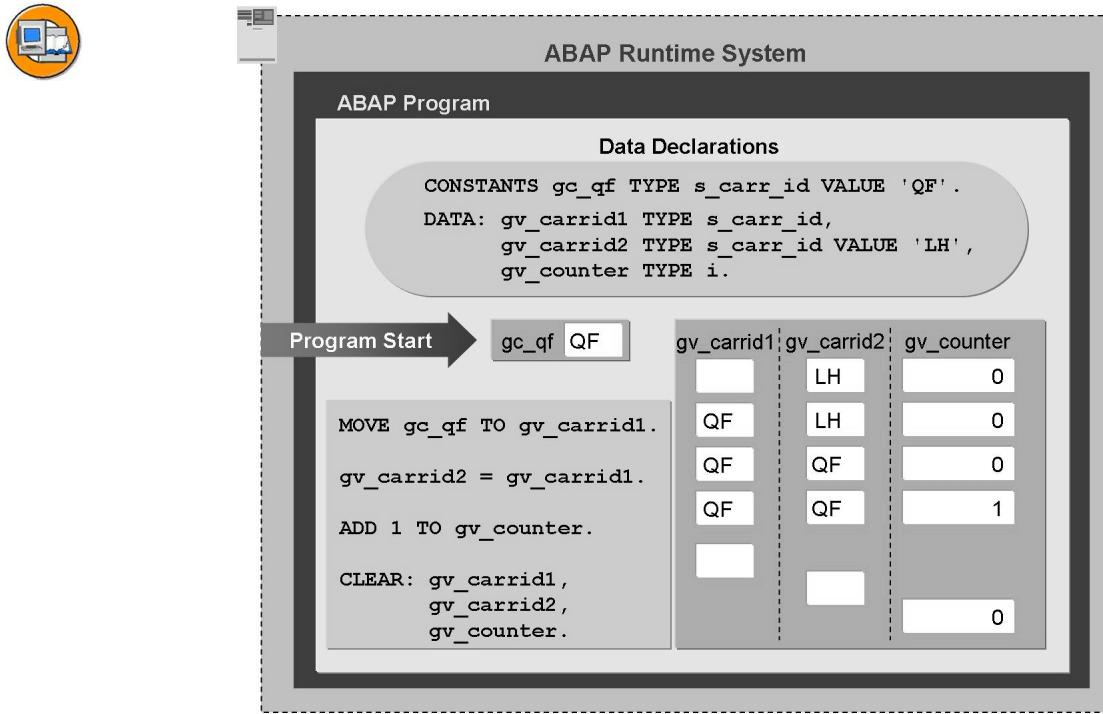


Figure 49: Value Assignments

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

You can use the **MOVE** statement to transfer the contents of a data object to another data object. The following two syntax variants have the same effect:

- **MOVE gd_var1 TO gd_var2.**
- **gd_var2 = gd_var1.**

If both data objects **gd_var1** and **gd_var2** are of different types, then there is a **type conflict**. In this case, a **type conversion** is carried out automatically, if a conversion rule exists. For detailed information on 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 on the initial values for a particular type, refer to the keyword documentation for the **CLEAR** statement.

Calculations and Arithmetic Expressions



ABAP Program

Data Declarations

```
DATA: gv_max          TYPE sbc400focc-seatsmax,
      gv_occ          TYPE sbc400focc-seatsocc,
      gv_percentage   TYPE sbc400focc-percentage.
```

```
COMPUTE gv_percentage = gv_occ * 100 / gv_max.
```

COMPUTE keyword is optional :

```
gv_percentage = gv_occ * 100 / gv_max.
```

Figure 50: Calculations

In ABAP you can program **arithmetic expressions** nested to any depth. Valid operations include:

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



Caution: Parentheses and operators are ABAP keywords and must therefore be separated from other words by at least one space.

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

```
gd_length = STRLEN( gd_cityfrom ).
```

In the case of functions, the opening parenthesis is part of the function name. 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 functions, then powers, then multiplication/division, and finally addition/subtraction.

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

Conditional Branches and Logical Expressions

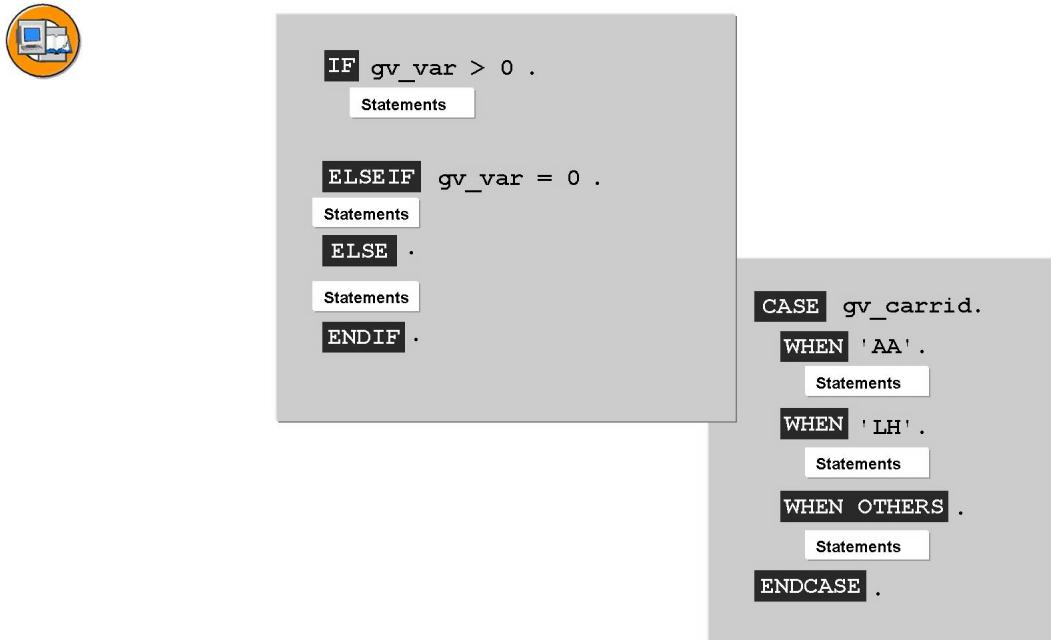


Figure 51: Conditional Branches

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

- In the **IF construct** you can define **any logical expressions** as check conditions. If the condition is met, the system executes the relevant statement block. Otherwise, the condition specified in the next ELSEIF branch (several branches are possible) is checked. If none of the specified conditions are fulfilled, the ELSE branch is executed - provided it exists. ELSEIF and ELSE branches are optional. For detailed information on formulating a logical expression refer to the keyword documentation on the IF statement.
- You can use the **CASE construct** to **clearly distinguish cases**. The content of the field specified in the CASE part is checked against the data objects listed in the WHEN branches to see whether they **match**. If the field contents match, the respective statement block is processed. 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.

In both constructs the condition or match check happens sequentially from the top down. As soon as the statement block of a branch has been executed, the system immediately jumps to ENDIF or ENDCASE.



Hint: If you want to implement similarity checks between a field and different values, you should use the CASEconstruct in preference to the IF statement, as it is **more transparent** and **performs better**.

The following shows several simple examples of using the IF statement.



```

Negation
IF gv_carrid IS NOT INITIAL.
    Statements
ELSE.
    Statements
ENDIF.

AND and OR Links with Parentheses
IF ( gv_carrid = 'AA' OR gv_carrid = 'LH' )
    AND gv_fldate = sy-datum.
    Statements
ELSEIF ( gv_carrid = 'UA' OR gv_carrid = 'DL' )
    AND gv_fldate > sy-datum.
    Statements
ENDIF.

Negation Before Logical Conditions
IF NOT ( gv_carrid = 'AA' OR gv_carrid = 'UA' )
    AND gv_fldate > sy-datum.
    Statements
ENDIF.

```

Figure 52: Examples: IF Statement

Negations are usually formulated by placing the NOT operator before the logical expression. When negating the IS INITIAL query, you can use the special IS NOT INITIAL query.



Hint: The formulation IF NOT var IS INITIAL is permitted too, but is not as readable and should therefore be avoided.

IF and CASE structures can, of course, be nested in any way you wish. You have to make sure that the logic of every structure is correct here. The inner structures in particular must be closed neatly before the system returns to the external logic, for example:

```

IF <log. condition1>.
...
    IF <log. condition2>.
    ...
        ENDIF.
    ENDIF.

```

Loops

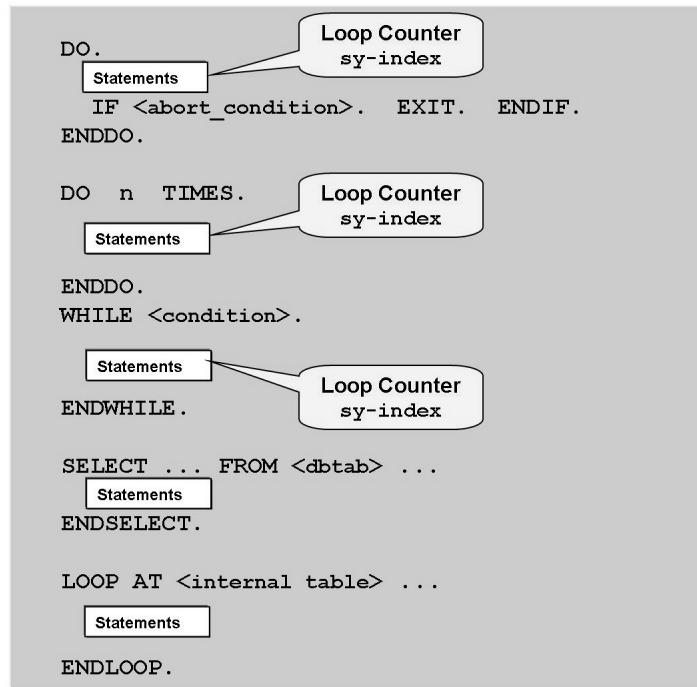


Figure 53: Loops

There are four loop constructs in ABAP. In the DO and WHILE loops, the system field **SY-INDEX** contains the number of the current loop pass. Hence, querying this system field only makes sense within a loop. In nested loops, SY-INDEX always contains the loop pass number of the loop in which it is located.

Unconditional/index-controlled loops

The statement block between DO and ENDDO is executed continuously until the loop is left using termination statements such as EXIT. You also have the option of specifying the maximum number of loop passes; otherwise you may get an **endless loop**.

Header-Controlled Loops

The statement block between WHILE and ENDWHILE is continuously executed until the specified condition is no longer met. The condition is always checked before executing the statement block.

Read Loops

You can use the SELECT loop to read several entries of a database table in succession. In an internal table (table variable in the program), the same read function is implemented with the LOOP.

System Fields

In the previous sections, several data objects were presented that you can use in ABAP source code without declaring them explicitly beforehand (for example, sy-datum, sy-index). The runtime system uses these **system fields** to provide the application program with information about the actual system status. Several interesting system fields are illustrated in the following graphic:



System Field	Meaning
<code>sy-mandt</code>	Logon Client
<code>sy-uname</code>	Logon Name of the User
<code>sy-langu</code>	Logon Language of the User
<code>sy-datum</code>	Local Date of the ABAP System
<code>sy-uzeit</code>	Local Time of the ABAP System
<code>sy-tcode</code>	Current Transaction Code
<code>sy-repid</code>	Name of the Current ABAP Program
<code>sy-index</code>	Loop Counter at DO and WHILE Loops
...	...

Figure 54: Some Interesting System Fields

Other system fields will be discussed in due course in the following lessons. You will find a complete list of system fields in the keyword documentation under the term “System Fields”.



Caution: To access system fields in your programs, use read access only.

Write access can result in the loss of important information for program parts that follow on from this. Furthermore, you have no control over when the ABAP runtime system changes the content again and thus overwrites the values you have set.



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

Figure 55: Return Code of an ABAP Statement

One of the most important system fields is the field **sy-subrc**. With many statements, it is filled by the ABAP runtime system with the corresponding return code to indicate whether the statement could be executed successfully. The value zero means that the statement was executed successfully. Read the keyword documentation for the respective statements to find out if and how this return value is set in individual cases.

Dialog Messages



`MESSAGE @nnn(message_class) [WITH v1 [v2] [v3] [v4]] .`

Type	Meaning	Dialog behavior	Message appears in
i	Info Message	Program continues after breakpoint	Modal dialog box
s	Set Message	Program continues without breakpoint	Status bar *) of next screen
w	Warning	Context-dependent	Status bar *)
e	Error	Context-dependent	Status bar *)
a	Termination	Program terminated	Modal dialog box
x	Short Dump	Runtime error MESSAGE_TYPE_X is triggered	Short dump

*) Display in modal dialog box also possible through GUI settings

Figure 56: Dialog Messages

You use the MESSAGE statement to send dialog messages to the users of your program. When you do this, you must specify the three digit message number and the message class.

Message number and message class clearly identify the message to be displayed. You use the message type to specify where the message is to be displayed. You can test the display behavior for the different message types by means of the DEMO_MESSAGES demo program that is delivered with the SAP standard system.

If the specified message contains placeholders, you can supply them with values from your program by using the WITH addition. Instead of the placeholders, the transferred values then appear in the displayed message text.

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

Working with the ABAP Debugger

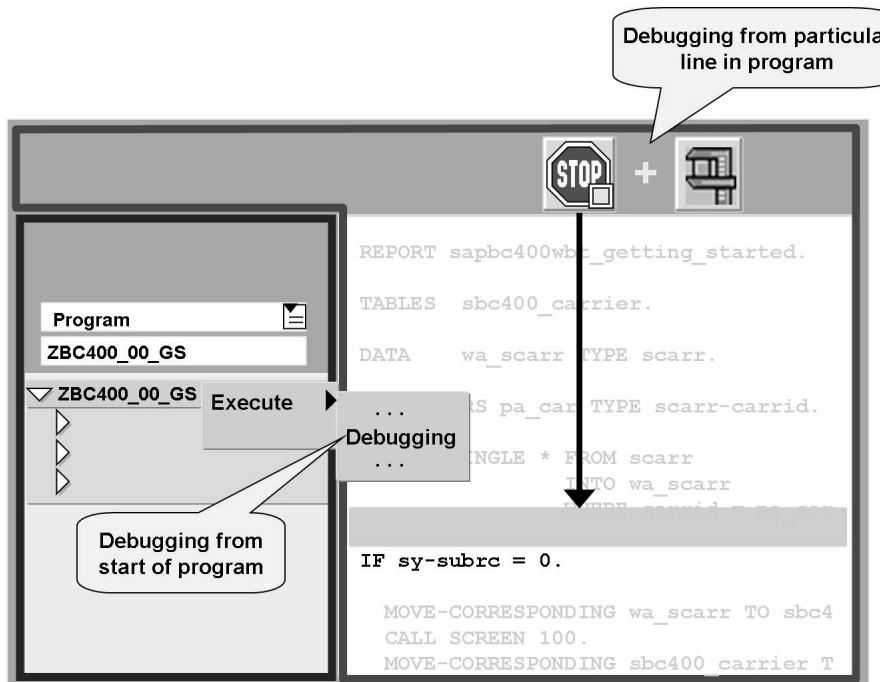


Figure 57: Executing a Program in Debugging Mode

There are several ways to start a program in debugging mode from the *Object Navigator*:

- In the navigation area for the selected program, choose the context menu *Execute* → *Debugging*.
- In the editor area, select the requested program line from which you wish to debug. Choose the *Set/Delete breakpoint* button. Afterwards, start the program by choosing *F8* or in the navigation area via the context menu *Execute* → *Direct*. (The above-described setting of a breakpoint in the editor is only possible for active source texts.)

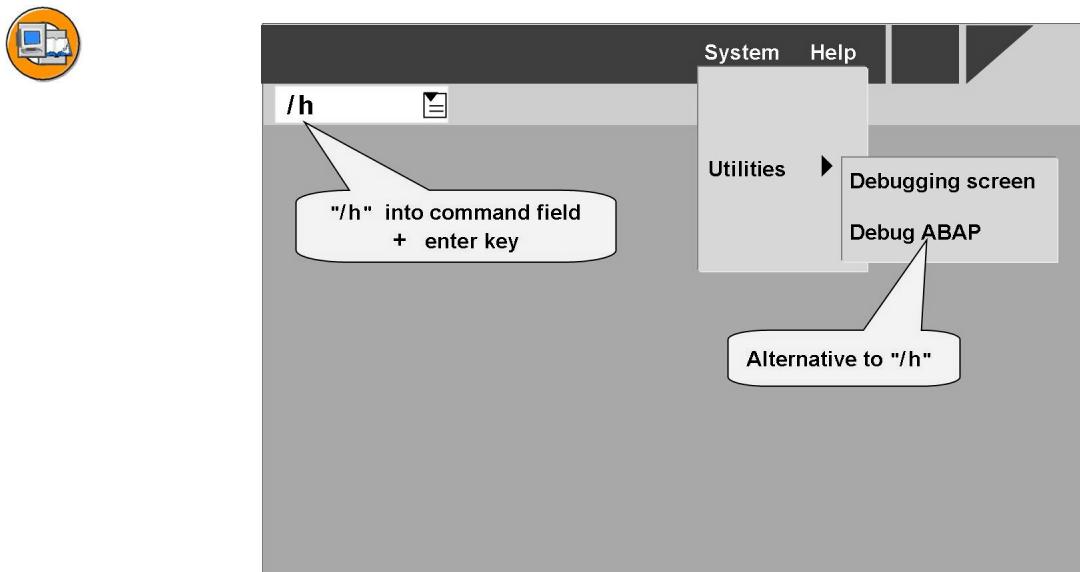
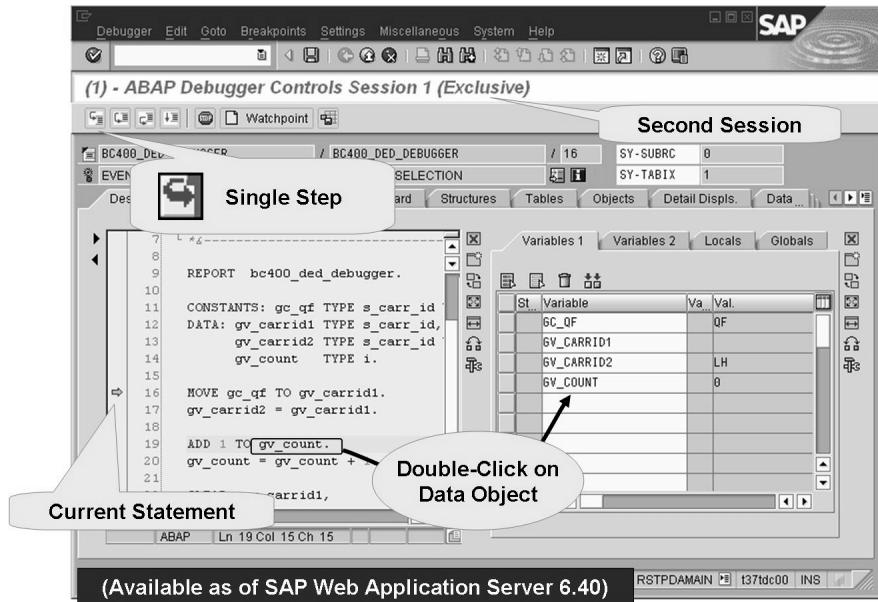


Figure 58: Switching to Debugging Mode at Runtime

If you want to “debug” **a certain function of a program**, first start the program without the Debugger and then switch to debug mode immediately before executing the function (e.g. pushbutton). There are 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**.



(Available as of SAP Web Application Server 6.40)

Figure 59: ABAP Debugger: Single Step and Field Contents

In the Debugger, you can choose *single step* processing to execute the program statement by statement. Furthermore, you can display data objects and their current content in the variable display. Simply enter the name of the data object there or copy them by double-clicking on corresponding data objects in the source code.

Only the classic debugger is available with *SAP Web Application Server 6.40* and below. With higher release levels you can use both the new and the classic debugger. You can switch easily from the new to the classic ABAP Debugger while debugging by choosing *Debugger → Switch to Classic ABAP Debugger* from the menu.

In the classic debugger you can display the contents of up to eight data objects. To do this, proceed as you would with the new debugger.

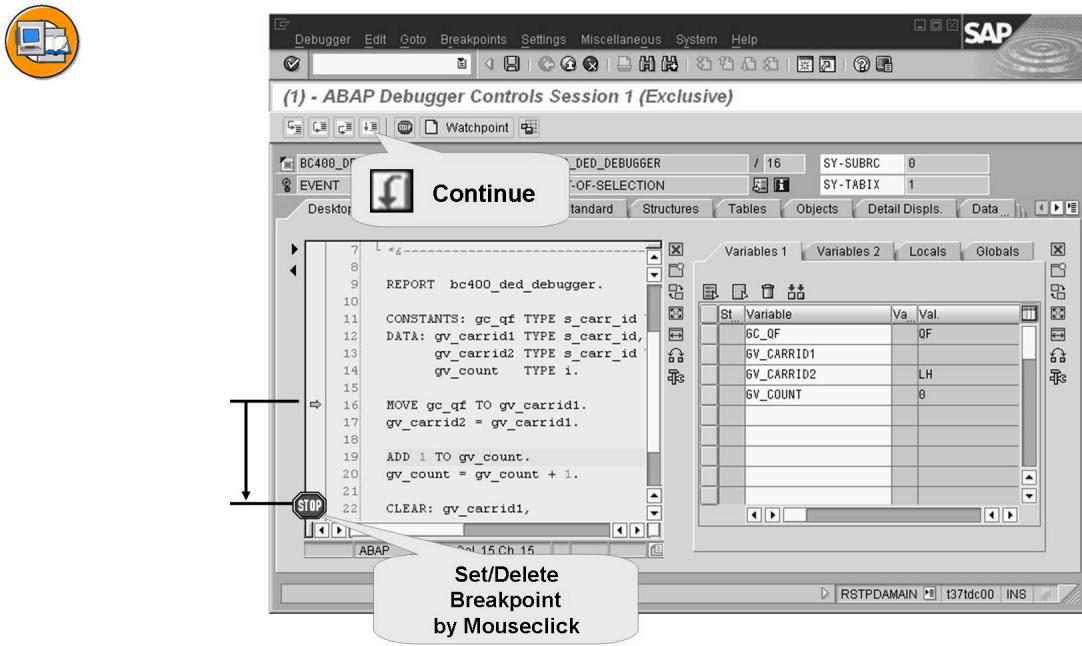


Figure 60: ABAP Debugger: Breakpoints

In the new debugger, you can set a **breakpoint** with a single click before the line in the source code (in the classic debugger you do this with a double-click). You can also set a breakpoint for specific ABAP statements: *Breakpoints* → *Breakpoint at* → *Statement*. If you choose *Continue*, the program is executed up to the next breakpoint.

The set breakpoints are only valid for the current debugger session. However, if you press *Save*, the breakpoints will stay in place for you for the duration of your current SAP session.

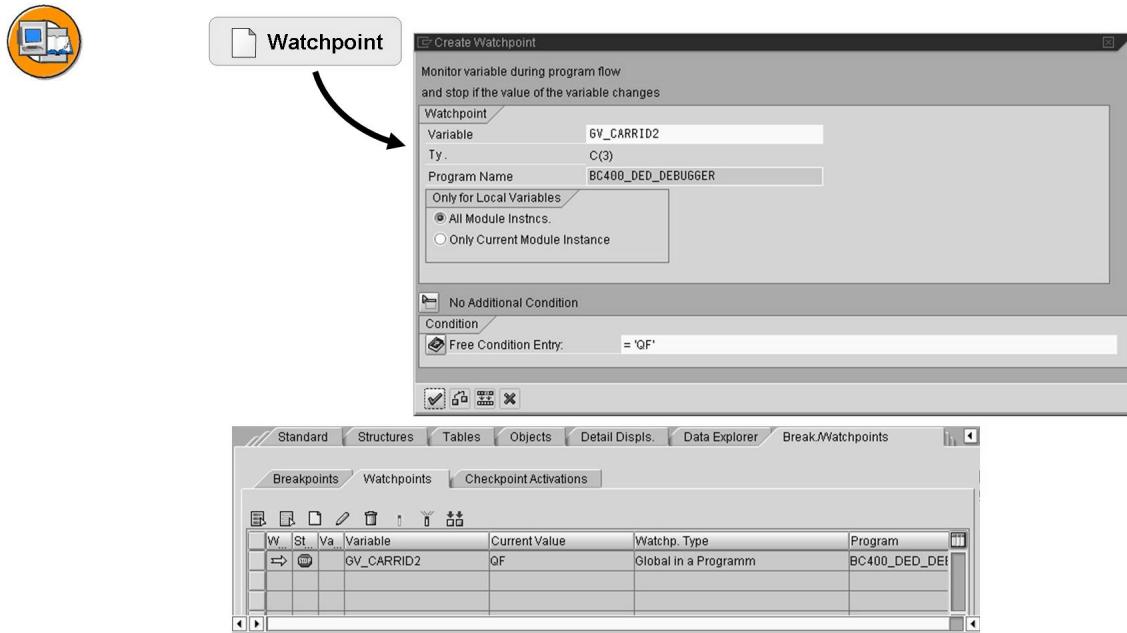


Figure 61: ABAP Debugger: Tracing Data Changes

Watchpoints are breakpoints that depend on the field content.

If you set a watchpoint **without specifying a relational operator/comparative value** on a field and choose *Continue*, then the program is executed until the content of the field changes.

However, if you **have specified the relational operator and comparative value**, then - once you choose *Continue* - the program will be executed until the specified condition is met.

In the classic debugger you can set a maximum of only 10 watchpoints. However, you can link them in the same way as in the new debugger using a logical operator (AND or OR).

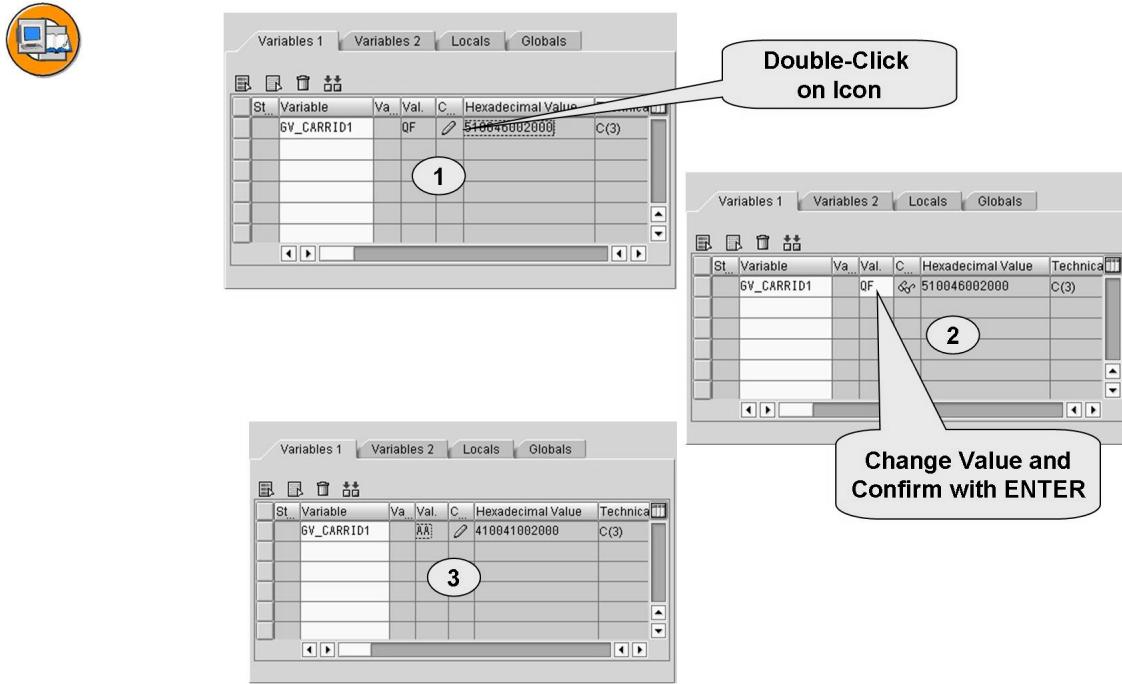


Figure 62: ABAP Debugger: Changing Field Contents

If you want to change the content of a field during debugging, double-click on the pencil icon in the variable display. The value is then ready for input. Now change the field value and confirm this by choosing **ENTER**. The value is now changed while the debugger is running.

In the classic debugger you can change the content directly in the field view. You then click on the pencil icon to accept the changed value.



- **Debugger in 2nd session**
→ Inputs for application program visible in parallel
- **Many parallel display options through freely configurable display areas**
- **Integration of New ABAP Editor**
(As of SAP NetWeaver 7.0)
- **Quick Info in source code area for data object value display**
(As of SAP NetWeaver 7.0)
- **Watchpoint in internal tables and object references**
(As of SAP NetWeaver 7.0)
- **Value comparison of strings, structures, and internal tables**
(As of SAP NetWeaver 7.0)

Figure 63: Additional Functions of the New Debugger

A number of useful functions have been added in the new debugger (most, however, were first incorporated with the *SAP NetWeaver 7.0* release). In the *Object Navigator* you can determine which debugger you want to use as standard by choosing *Utilities* → *Settings* → *ABAP Editor* → *Debugging*.

Exercise 4: 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. The result should be displayed in a list.

Template:

None

Solution:

BC400_DOS_COMPUTE

Task 1:

Create a program.

1. Create the executable program **ZBC400_##_COMPUTE** without a *TOP include* and assign it to your package.

Task 2:

Define a selection screen with three input parameters.

1. Define two input parameters for integers as operands (suggested names: **pa_int1**, **pa_int2**) and one character type arithmetic operator of one character in length (suggested name: **pa_op**). Use the ABAP keyword PARAMETERS to do this.

Continued on next page

Task 3:

Implement the basic calculation types and output the result in a list.

1. Create an elementary data object (suggested name: **gv_result**) using the DATA statement for the calculation result.. Define it as a packed number type of 16 digits in length and two decimal places.
2. Calculate the result dependent on the specified arithmetic operator. Use the CASE statement for a case distinction.
3. Output the result in a list using the WRITE statement. If you want to output texts in the process, make sure that these texts can be translated and are displayed as language-dependent.



Hint: Create text symbols and use these instead of literals.

Task 4:

Catch any errors that may occur.

1. Display an error message on the list if the user has specified an invalid arithmetic operator. Use the IF statement for checking. Use a translatable text.
2. Display an error message on the list if the user tries to divide by zero.

Task 5:

Activate and test your program.

1. Activate your program.
2. Test your program.

Solution 4: Basic ABAP Statements

Task 1:

Create a program.

1. Create the executable program **ZBC400_##_COMPUTE** without a *TOP include* and assign it to your package.
 - a) Perform this step in the same way as in the previous exercises.

Task 2:

Define a selection screen with three input parameters.

1. Define two input parameters for integers as operands (suggested names: **pa_int1**, **pa_int2**) and one character type arithmetic operator of one character in length (suggested name: **pa_op**). Use the ABAP keyword PARAMETERS to do this.
 - a) See source code excerpt from the model solution.

Task 3:

Implement the basic calculation types and output the result in a list.

1. Create an elementary data object (suggested name: **gv_result**) using the DATA statement for the calculation result.. Define it as a packed number type of 16 digits in length and two decimal places.
 - a) See source code excerpt from the model solution.
2. Calculate the result dependent on the specified arithmetic operator. Use the CASE statement for a case distinction.
 - a) See source code excerpt from the model solution.
3. Output the result in a list using the WRITE statement. If you want to output texts in the process, make sure that these texts can be translated and are displayed as language-dependent.



Hint: Create text symbols and use these instead of literals.

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

Continued on next page

Task 4:

Catch any errors that may occur.

1. Display an error message on the list if the user has specified an invalid arithmetic operator. Use the IF statement for checking. Use a translatable text.
 - a) See source code excerpt from the model solution.
2. Display an error message on the list if the user tries to divide by zero.
 - a) See source code excerpt from the model solution.

Task 5:

Activate and test your program.

1. Activate your program.
 - a) In the *ABAP Editor* toolbar, choose the  pushbutton.
2. Test your program.
 - a) In the *ABAP Editor* toolbar, choose the  pushbutton.
 - b) Supply the selection screen parameters with values and choose .
 - c) Check the result. Test the message output in the event of an error too.

Result

Source code extract from the model solution:

```
*-----*
*& Report  BC400_DOS_COMPUTE
*&
*-----*
*&
*& Simple calculator
*&
*-----*

REPORT  bc400_dos_compute.

PARAMETERS:
  pa_int1  TYPE i,
  pa_op    TYPE c LENGTH 1,
  pa_int2  TYPE i.

DATA gv_result TYPE p LENGTH 16 DECIMALS 2.
```

Continued on next page

```
IF ( pa_op = '+' OR
     pa_op = '-' OR
     pa_op = '*' OR
     pa_op = '/' AND pa_int2 <> 0 ).

CASE pa_op.
WHEN '+'.
  gv_result = pa_int1 + pa_int2.
WHEN '-'.
  gv_result = pa_int1 - pa_int2.
WHEN '*'.
  gv_result = pa_int1 * pa_int2.
WHEN '/'.
  gv_result = pa_int1 / pa_int2.
ENDCASE.

WRITE: 'Result'(res), gv_result.

ELSEIF pa_op = '/' AND pa_int2 = 0.

  WRITE 'No division by zero!'(dbz).

ELSE.

  WRITE 'Invalid operator! '(iop).

ENDIF.
```


Exercise 5: Debugging Statements on Elementary Data Objects

Exercise Objectives

After completing this exercise, you will be able to:

- Execute a program in the Debugger
- Set breakpoints in the Debugger
- Set content-dependent breakpoints (watchpoints)
- Change field contents during debugging

Business Example

You are given a program and have to use the Debugger to work out its semantic behavior.

Task 1:

Execute the program **BC400_DED_DEBUG_EXERCISE** in debugging mode and trace the content of **GV_COUNT**.

1. Start the program **BC400_DED_DEBUG_EXERCISE** in debugging mode by way of the object list in the *Object Navigator*.
2. First check the content of the data objects in the program at the start. To do this, include all the data objects defined in the program and the system field **SY-INDEX** in the variable view. Take a look at the typing of the fields too.

Task 2:

Analyze the WHILE loop by setting breakpoints and tracking the content of **GV_COUNT** and **SY-INDEX**.

1. Follow the program flow in single steps until you have run through the WHILE loop twice. In each case, track how the field contents change.
2. Set a breakpoint at the statement that assigns a new value to the data object **GV_COUNT**. Choose **Continue** to restart the program and observe the changes to the field contents. Repeat the final step until the program has run its course and the list is displayed.

Continued on next page

3. Execute the program again in debugging mode. Now analyze the change to the data object GV_COUNT by creating a watchpoint for the field. Choose *Continue* to restart the program and observe the change to the content of the data object and the system field SY-INDEX.

Task 3:

Analyze the nested DO loops by setting a breakpoint at the WRITE and EXIT statements.

1. Restart the program in debugging mode. Set breakpoints at **all** WRITE and EXIT statements in the program (choose *Breakpoints* → *Breakpoint at* → *Breakpoint at Statement*). Analyze how the contents of the loop counters GV_IDX_OUTER and GV_IDX_INNER change, the information that is output, and when it is output.
2. What effect does the EXIT statement have in a loop?

Task 4:

Use the Debugger option to change the program flow in such a way that the last statement in the program is executed too.

1. Execute the last statement in the program. Use the option of changing the content of data objects in debugging mode.

Solution 5: Debugging Statements on Elementary Data Objects

Task 1:

Execute the program **BC400_DED_DEBUG_EXERCISE** in debugging mode and trace the content of **GV_COUNT**.

1. Start the program **BC400_DED_DEBUG_EXERCISE** in debugging mode by way of the object list in the *Object Navigator*.
 - a) Display the program in the *Object Navigator*, right click on the program and choose *Execute → Debugging*.
 - b) Alternatively, in the editor, you can also set a breakpoint at the first executable statement.
2. First check the content of the data objects in the program at the start. To do this, include all the data objects defined in the program and the system field **SY-INDEX** in the variable view. Take a look at the typing of the fields too.
 - a) In the source code in the Debugger, double-click on each of the fields **GV_COUNT**, **GV_IDX_INNER**, **GV_IDX_OUTER** and **SY-INDEX**. Scroll to the right of the variable display in order to display the field types.

Task 2:

Analyze the WHILE loop by setting breakpoints and tracking the content of **GV_COUNT** and **SY-INDEX**.

1. Follow the program flow in single steps until you have run through the WHILE loop twice. In each case, track how the field contents change.
 - a) You can carry out the single step analysis by choosing *Debugger → Single Step*, the  pushbutton, or the **F5** key.
2. Set a breakpoint at the statement that assigns a new value to the data object **GV_COUNT**. Choose  *Continue* to restart the program and observe the changes to the field contents. Repeat the final step until the program has run its course and the list is displayed.
 - a) Click once in the first column in front of the value assignment to set a breakpoint there.
 - b) Choose *Debugger → Continue* from the menu, the  pushbutton, or **F8**.

Continued on next page

3. Execute the program again in debugging mode. Now analyze the change to the data object GV_COUNT by creating a watchpoint for the field. Choose *Continue* to restart the program and observe the change to the content of the data object and the system field SY-INDEX.
 - a) To access the Debugger, repeat the first step in this exercise.
 - b) Position the cursor on the data object GV_COUNT in the source code displayed in the Debugger. Click on the *Watchpoint* pushbutton and confirm the dialog box that appears with .
 - c) To restart the program, choose  *Continue*.

Task 3:

Analyze the nested DO loops by setting a breakpoint at the WRITE and EXIT statements.

1. Restart the program in debugging mode. Set breakpoints at **all** WRITE and EXIT statements in the program (choose *Breakpoints* → *Breakpoint at* → *Breakpoint at Statement*). Analyze how the contents of the loop counters GV_IDX_OUTER and GV_IDX_INNER change, the information that is output, and when it is output.
 - a) To access the Debugger, proceed as before.
 - b) In the Debugger, choose *Breakpoints* → *Breakpoint at* → *Breakpoint at Statement* and enter the WRITE statement in the dialog box that appears. Confirm the dialog box that appears with .
 - c) Choose  *Continue* and analyze the program flow.
2. What effect does the EXIT statement have in a loop?

Answer: The EXIT statement causes the system to cease processing the current DO loop.

Continued on next page

Task 4:

Use the Debugger option to change the program flow in such a way that the last statement in the program is executed too.

1. Execute the last statement in the program. Use the option of changing the content of data objects in debugging mode.
 - a) Set a breakpoint at the last IF statement and choose *Continue* to run the program to this point.
 - b) Include the field GV_IDX_INNER in the variable view. Double-click the  icon in the variable view. Change the field value from 6 to 5 and confirm the change by choosing **Enter**.
 - c) Restart the program by choosing either *Single Step* or *Continue*.



Note: Alternatively, you can also execute the last statement using the *Goto Statement* function. In debugging mode, place the cursor on this line at any event and choose *Debugger* → *Goto Statement*.



Lesson Summary

You should now be able to:

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



Unit Summary

You should now be able to:

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

Related Information

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



Test Your Knowledge

1. Which of the ABAP standard types are numeric?

2. Is it possible to define other types in addition to the standard types? If yes, where?

3. Which ABAP keyword do you use to define local types?

4. In which system field will you find the loop counter for the DO and WHILE loops, respectively?



Answers

1. Which of the ABAP standard types are numeric?

Answer: I, F, P

N is not a numeric type, but the type for character strings that may contain decimal numbers only.

2. Is it possible to define other types in addition to the standard types? If yes, where?

Answer: Yes, in the *ABAP Dictionary* (global types) and in the program (local types).

3. Which ABAP keyword do you use to define local types?

Answer: TYPES

4. In which system field will you find the loop counter for the DO and WHILE loops, respectively?

Answer: SY-INDEX

Unit 4

Modularization

Unit Overview

The unit overview lists the individual lessons that make up this unit.



Unit Objectives

After completing this unit, you will be able to:

- Name the basic modularization techniques
- Define subroutines
- Call subroutines
- Analyze the execution of subroutines in debugging mode
- Search for function modules
- Acquire information on the functionality and use of function modules
- Call function modules in your program
- Create a function group
- Create a function module
- Explain the role of BAPIs and identify their special properties
- Explain the basic terms of object-oriented programming
- Acquire information about the function and use of global classes and their methods
- Call methods of global classes in your programs
- Create global classes
- Create and implement simple methods in global classes
- Describe how local classes are defined, implemented and used

Unit Contents

Lesson: Modularization - Basics and Overview.....	113
Lesson: Modularization with Subroutines.....	120
Exercise 6: Subroutines	135

Lesson: Modularization with Function Modules.....	142
Exercise 7: Using a Function Module.....	161
Exercise 8: Creating a Function Group.....	169
Exercise 9: Creating and Using a Function Module	171
Lesson: Modularization with Methods of Global Classes.....	178
Exercise 10: Using Global Static Methods	195
Exercise 11: Creating Global Classes	199
Exercise 12: Creating and Using Global Static Methods.....	201
Lesson: Modularization with Methods of Local Classes (Preview).....	208

Lesson: Modularization - Basics and Overview

Lesson Overview

In this lesson you will learn why it makes sense to store parts of programs in modularization units. You will also gain an overview of the various modularization options in ABAP programs.



Lesson Objectives

After completing this lesson, you will be able to:

- Name the basic modularization techniques

Business Example

An employee in quality assurance has discovered than many program parts are frequently repeated. Your task is to find out which modularization techniques can be implemented.

Modularization Technique

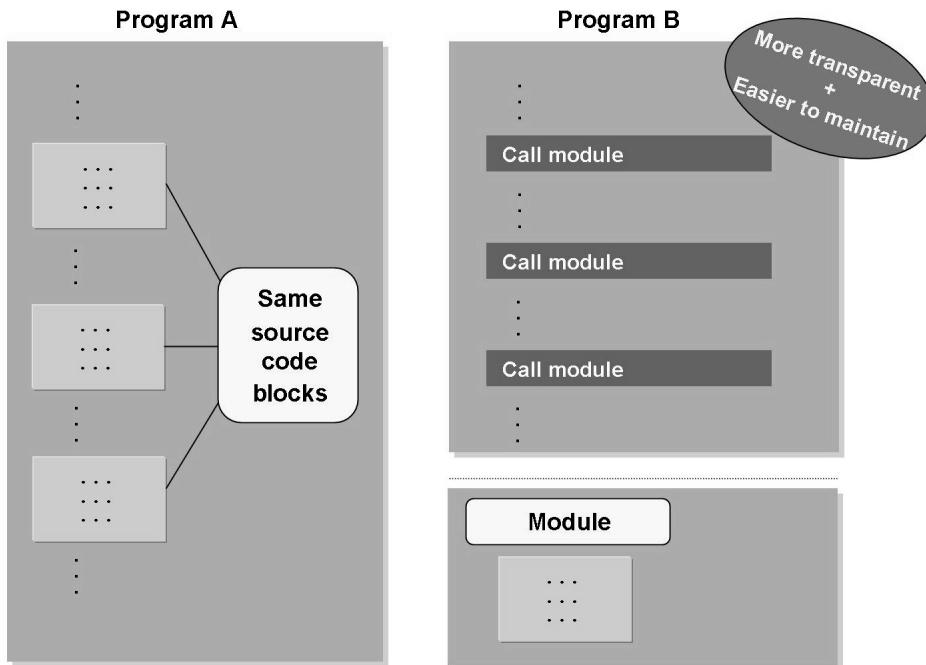


Figure 64: Options for Using Modularization Units

A **modularization unit** is a part of a program in which a particular function is encapsulated. You store part of the source code in a module to improve the transparency of the program as well as to use the corresponding function in the program **several times** without having to re-implement the entire source code on each occasion (see above graphic).

The improvement in transparency is a result of the program becoming more function-oriented: It divides the overall task into subfunctions, which are the responsibility of corresponding modularization units.

Modularization makes it **easier to maintain** programs, since you only need to make changes to the function or corrections in the respective modularization units and not at various points in the main program. Furthermore, you can process a call “as a unit” in the Debugger while executing your program and then look at the result. This usually makes it easier to find the source of the error.

Local Program Modularization

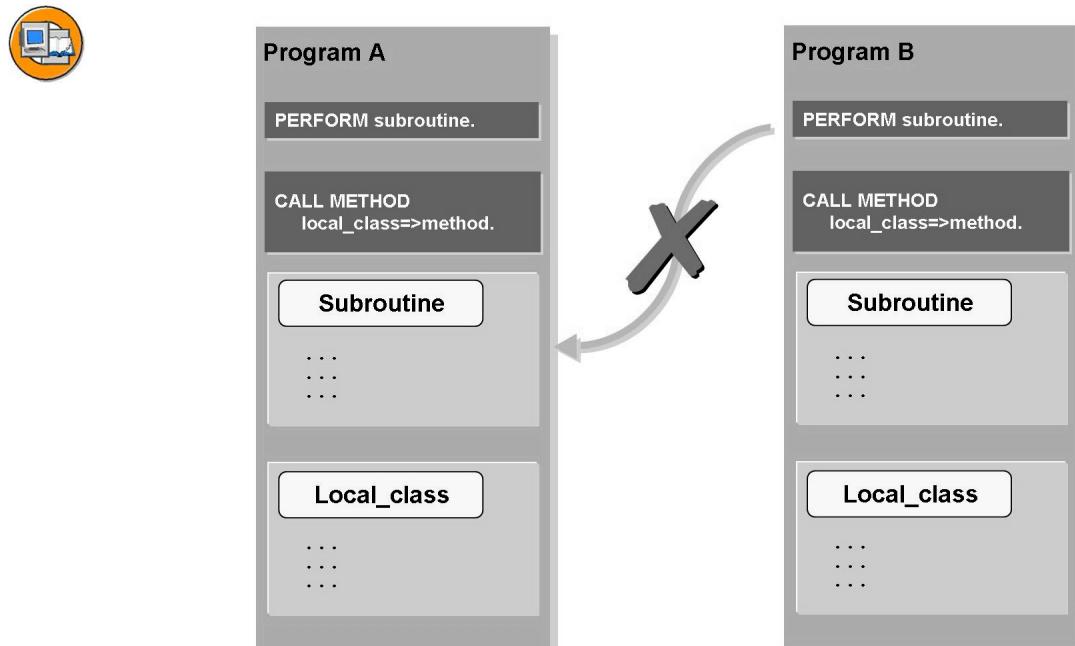


Figure 65: Local Program Modularization

There are two techniques for local program modularization in the ABAP programming language:

- **Subroutines**, also known as **form routines**
- **Methods in local classes**

With both local modularization techniques, modularization units are only available in the program in which they were implemented. To call the local module, no other program must be loaded to the user context at runtime. Local classes, methods and subroutines can have the same name in different programs without this resulting in conflicts. This is because the source code for the programs is handled separately in the main memory of the application server.



Hint: For historical reasons it is technically possible to call a subroutine from another program too. You should not use this option, however, since this technique contradicts the principle of encapsulation of data and functions.

Global Modularization

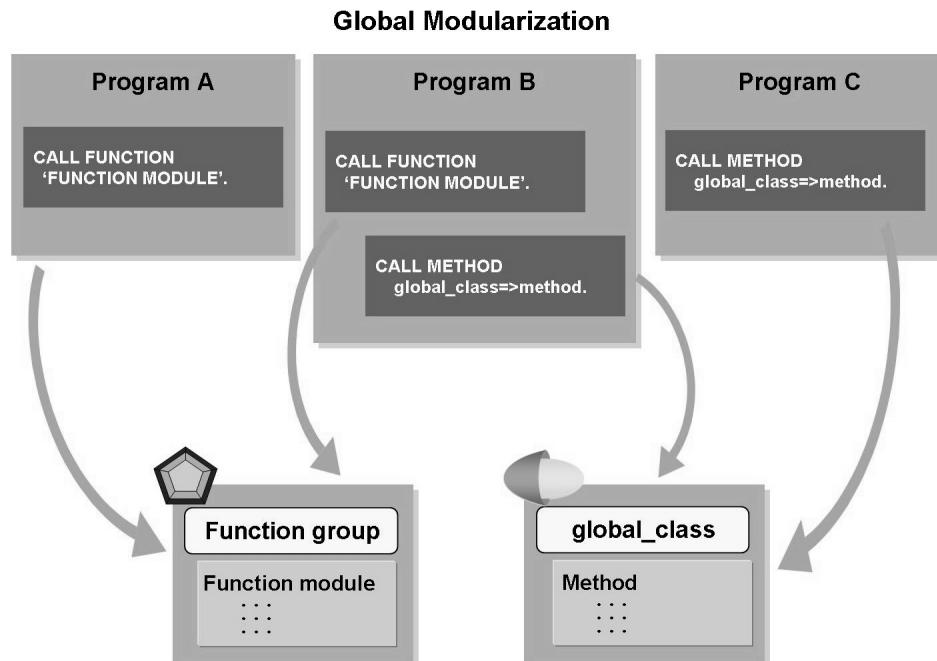


Figure 66: Global Modularization

There are also two techniques for global modularization in the ABAP programming language:

- **Function modules** that are organized in **function groups**
- **Methods in global classes**

Globally defined modularization units can be used by any number of programs at the same time. The globally defined modularization units are stored centrally **in the Repository** and loaded when required (in other words, when called) into the context of the calling program.

Encapsulating Data

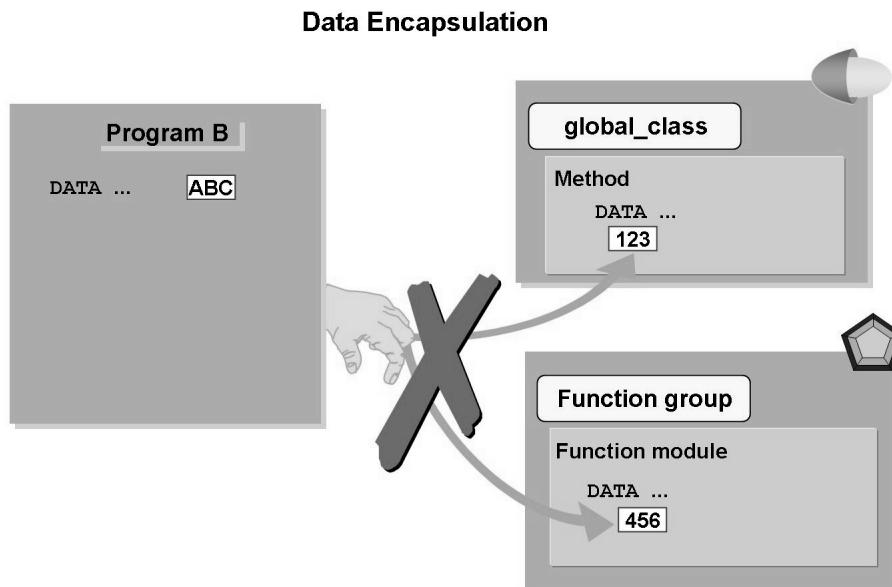


Figure 67: Separating Data

Ideally, the modularization units that are called do not use the data objects of the calling program directly. This applies the other way round too: Data in the modularization units should not be changed directly by the calling program. This principle is known as **data encapsulation**.

Data encapsulation is an important aid in developing transparent, maintainable source code. It makes it far easier to comprehend whereabouts in the program the contents of data objects were changed. In addition, it is easier to ensure that **data** within the modularization units is changed **consistently** if, for example, the contents of several data objects within a modularization unit are mutually dependent.

We will take as an example a modularization unit in which a series of invoice documents are processed. Serious consequences would arise if different invoice numbers were entered for invoice items that belonged to one another. If open external access to individual invoice items were possible, this is what could occur. It would then be difficult for those responsible for the modularization unit to determine the cause of the data inconsistency.

Data Transports, Parameters and Interface

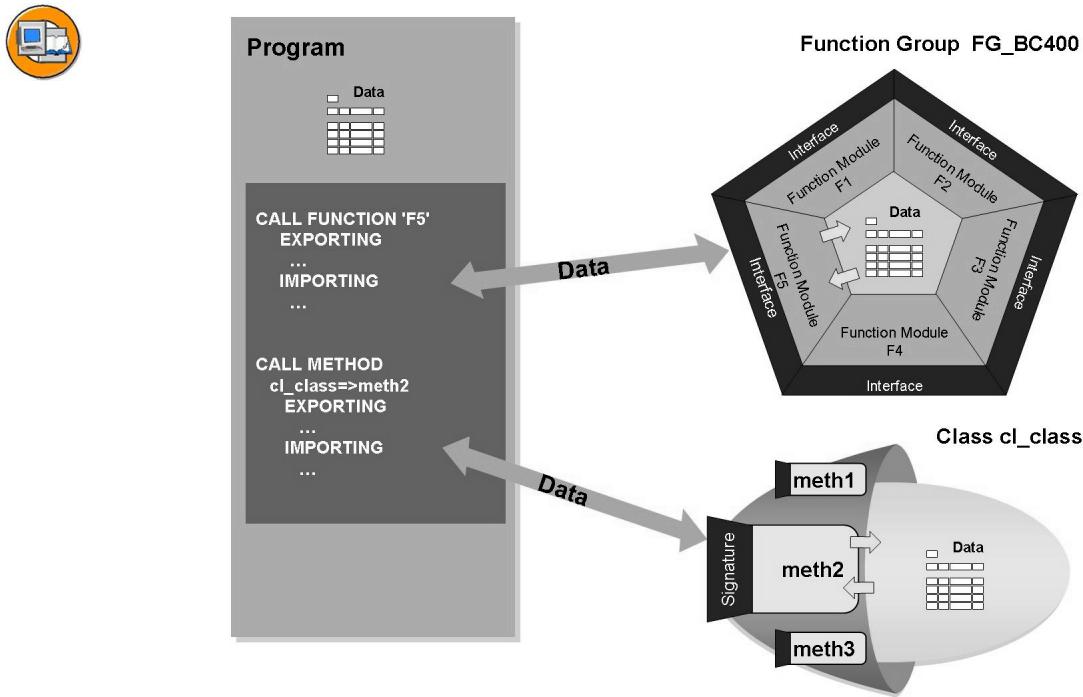


Figure 68: Data Transports Between the Program and the Modularization Unit

Parameters are used to **exchange data** between the program and the module. The total number of parameters in a modularization unit is termed the **interface** or **signature**. The parameters available are determined when you define the modularization unit. Parameters are differentiated on the basis of whether they are used to pass data to the

modularization unit (**importing parameters**), return data from the modularization unit to the caller (**exporting parameters**), or pass data to the modularization unit and return the data after it has been changed (**changing parameters**).



Hint: With subroutines (form routines), only changing parameters and the very specialized using parameters are available, which severely restricts your options for controlling the data transport. This is another good reason for using local classes for local program modularization where possible.



Lesson Summary

You should now be able to:

- Name the basic modularization techniques

Lesson: Modularization with Subroutines

Lesson Overview

In this lesson you will learn how to employ subroutines in ABAP programs. Furthermore, you will learn how the interface of a subroutine is used to pass parameters and how the different transfer types are used.



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 your program and encapsulate source code that is executed several times in a subroutine.

Internal Program Modularization with Subroutines

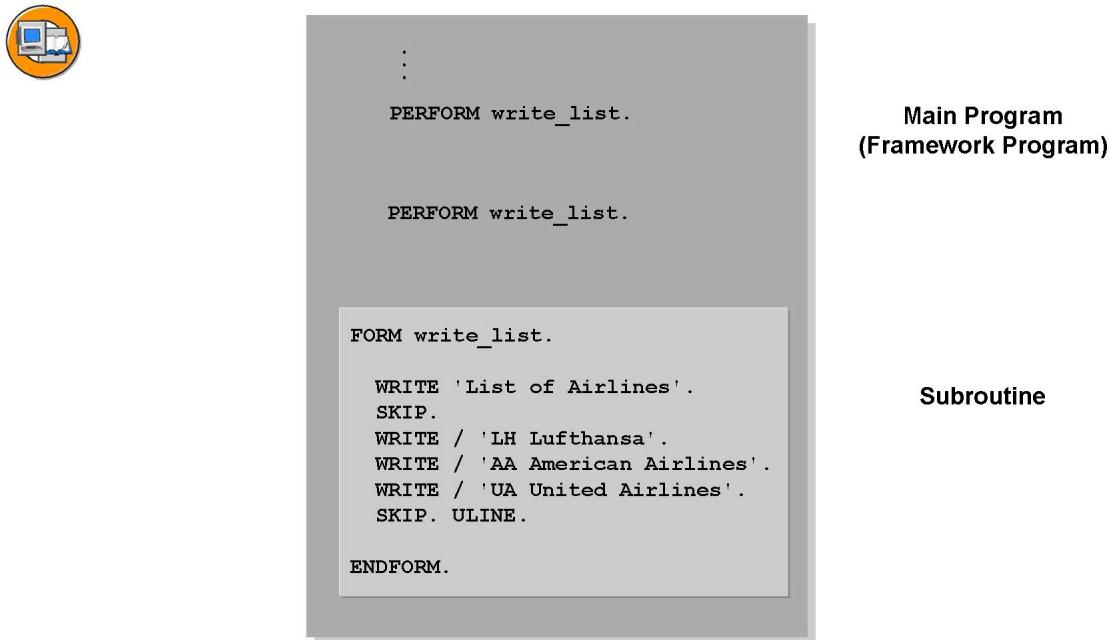


Figure 69: Simple Example of a Subroutine

A subroutine is a modularization unit within a program. For the ABAP interpreter, it is always part of the main program. No parameters are used in the above example, which makes the subroutine call very simple to structure. A subroutine normally uses values from data objects and returns values too. The next graphic demonstrates how these variables can be used in the subroutine.



Parameter Definition for Subroutines

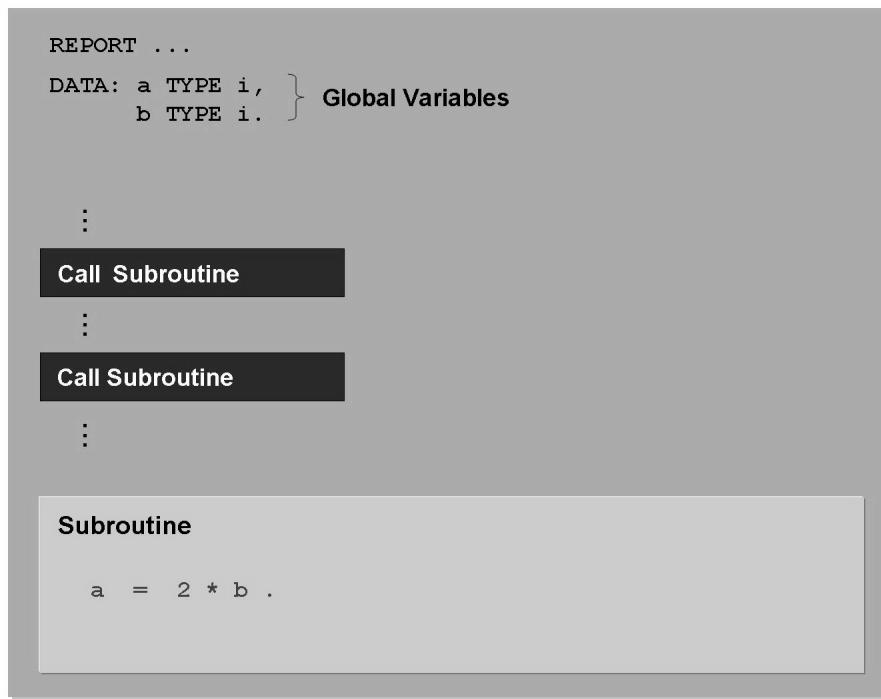


Figure 70: Passing Parameters - Visibility of Global Variables

If variables were defined in the main program, these are visible globally within the program and can be changed at any point on the entire program. This means that subroutines that are defined within the main program can change them too.

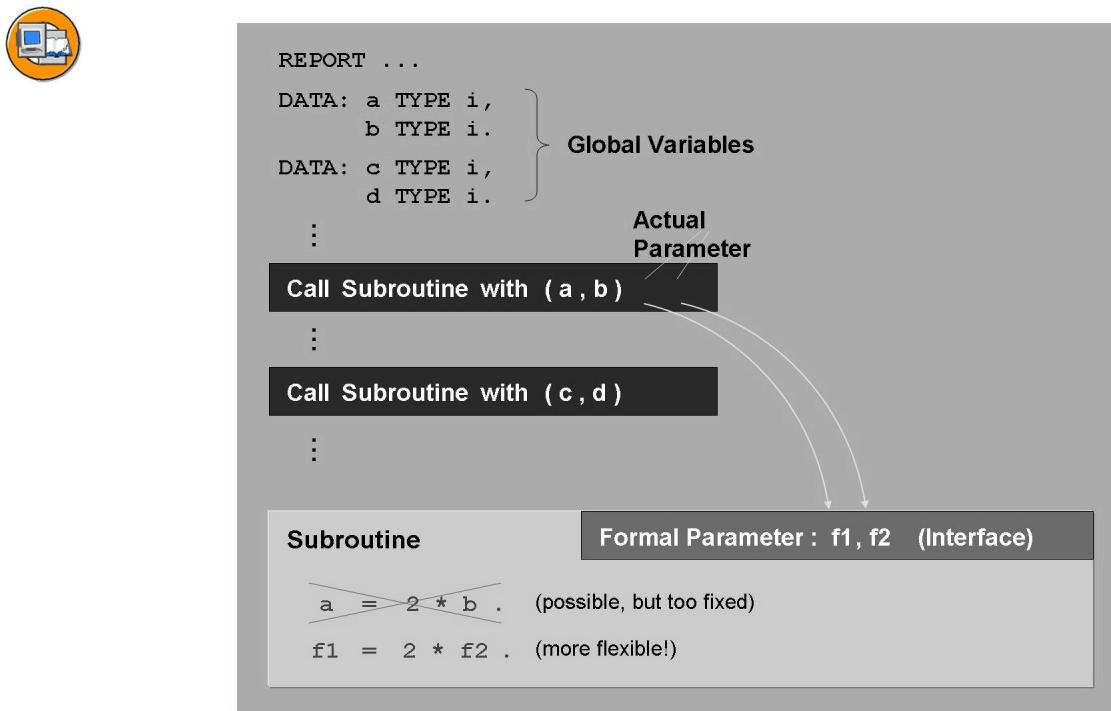


Figure 71: Passing Parameters - Defining an Interface

You can address all (**global**) variables defined in the main program from a subroutine. However, in order to call a subroutine with different data objects for each situation, you do not use global variables in the subroutine but placeholders. These are replaced with the required global variables when the subroutine is called. These placeholders are called **formal parameters** and together form the **subroutine interface**. You have to declare the interface when you define the subroutine.

When the subroutine is called, formal parameters must be specialized by means of corresponding global variables (**actual parameters**), in order to reference the subroutine processing to real variables. This assignment of actual parameters to formal parameters when calling a subroutine is called **parameter passing**.

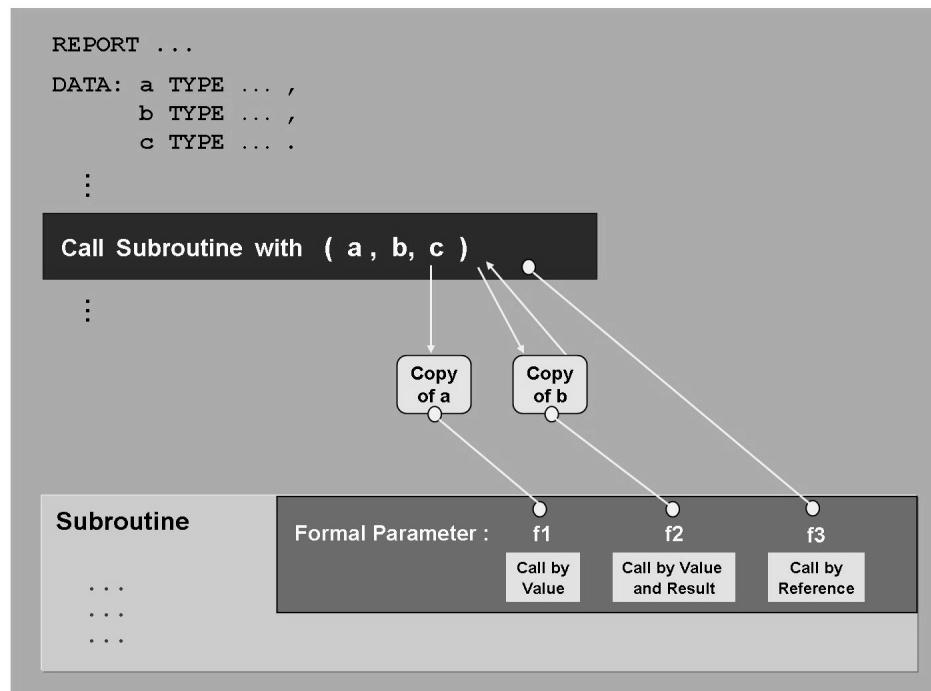


Figure 72: Ways of Passing Interface Parameters

The **way** these main program variables are passed to the formal parameters of the subroutine is called the **pass type** and is specified for each parameter in the interface of the subroutine.

There are three pass types for subroutines:

call by value

A **copy** is made of the actual parameter. This copy is assigned to the formal parameter. Any value assignments to the corresponding formal parameter in the subroutine therefore refer only to the copy of the actual parameter and not the original.

You use this pass type to make the value of a global variable available to the subroutine (in the form of a variable copy) without making it possible to change the respective global variable (protecting the original). Please note, however, that creating copies, especially for large internal tables, can be time-consuming.

call by value and result

The same applies for this pass type as for “call by value”. However, **at the regular end** of the subroutine, the value that was changed to this point in the copy is written back to the original. If the program is prematurely terminated by a STOP statement or a type E user message, the writing back of the value is suppressed.

You use this pass type to transfer the value of a global variable to the subroutine and to write the **fully processed final value** of the copy back to the original. Note, however, that it can be time-consuming to create copies and write back values, especially for large internal tables.

call by reference

The actual parameter is assigned **directly** to the formal parameter. This means that value assignments to the formal parameter are carried out **directly on the actual parameter**.

You use this pass type if you want to run subroutine processing directly on the specified actual parameter. It is a useful way of avoiding the time-consuming creation of copies for large internal tables.

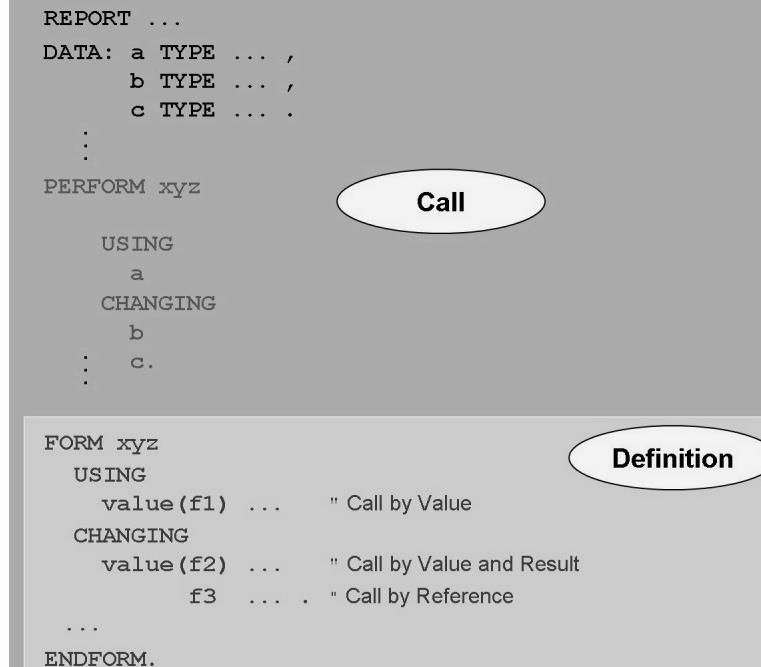


Figure 73: Defining and Calling Subroutines

Structure of a subroutine

- A subroutine is introduced with FORM.
- You specify the name and the interface of the subroutine after FORM.
- The statements of the subroutine then follow.
- The ENDFORM statement concludes the subroutine.

In the interface definition you list the formal parameters of the subroutine (here: *f1*, *f2*, *f3*) and type them if necessary. The required pass type has to be specified for each parameter:

call by value

You list each of the formal parameters that is supposed to have the pass type “call by value” (here: *f1*) with the **VALUE prefix** under **USING**. (Refer to the above graphic for the syntax.)

call by value and result

You list each of the formal parameters that is supposed to have the pass type “call by value and result” (here: *f1*) with the **VALUE prefix** under **CHANGING**. (Refer to the above graphic for the syntax.)

call by reference

You list each of the formal parameters that is supposed to have the pass type “call by reference” (here: *f3*) **without the** VALUE prefix under **CHANGING**. (Refer to the above graphic for the syntax.)



Hint: A parameter **without** the VALUE prefix, but placed under **USING**, also has the pass type “call by reference”. However, this declaration syntax only makes sense for formal parameters that are passed to larger internal tables, which are not to be changed in the subroutine (documentation via USING) but are to be passed using “call by reference” in order to avoid making time-consuming copies.

When the subroutine is called, the actual parameters to be transferred **without the** VALUE prefix are specified under USING or CHANGING. The **order** of specification determines their assignment to the formal parameters. In the example in the above graphic, *a* is passed to *f1*, *b* to *f2*, and *c* to *f3*.

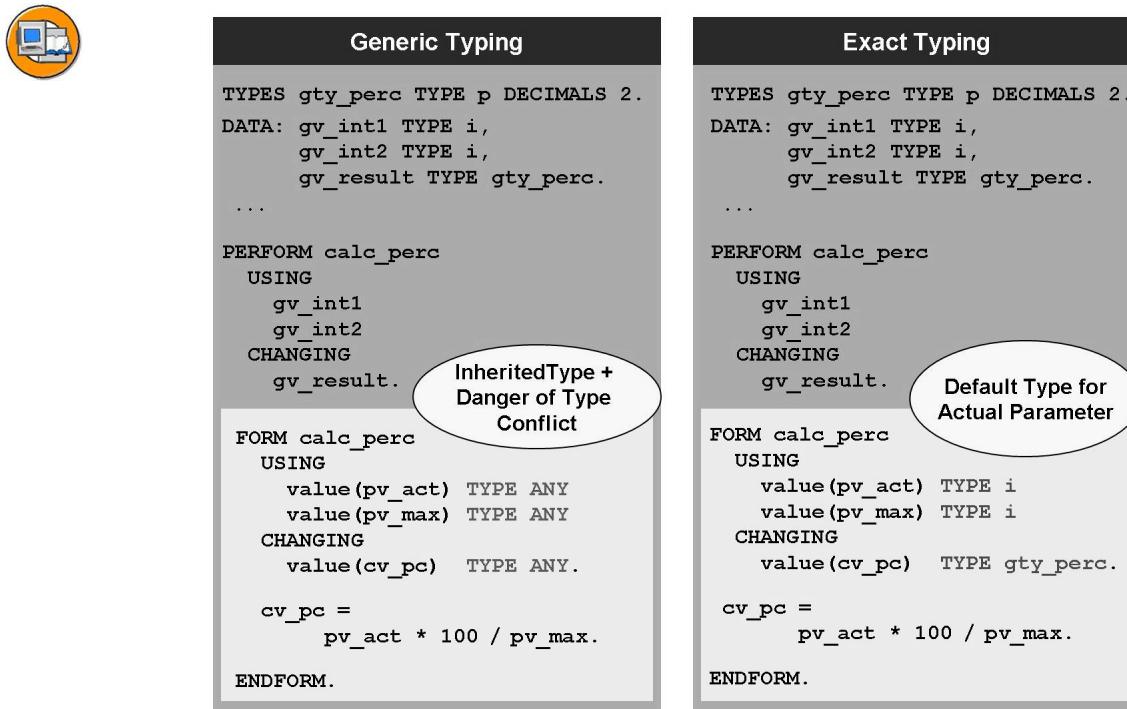


Figure 74: Typing the Interface Parameters

A formal parameter is **typed generically** if it is typed using **TYPE ANY**, or not typed at all. Actual parameters of **any type** can be transferred to such a parameter. At runtime, the type of the actual parameter is determined and assigned to the formal parameter (**type inheritance**) when the subroutine is called. However, if the statements in the subroutine are not suited to the inherited type, a **runtime error** may occur (**type conflict**). Hence, generic typing should only be used if the type of the actual parameter has yet to be determined when the program is created or if it can vary at runtime (dynamic programming).

You implement the **concrete typing** of a formal parameter by specifying a global or built-in type in the **TYPE** addition. In doing so, you determine that only actual parameters of the specified type can be passed to the subroutine. A violation of the type consistency between formal and actual parameters is already picked up in the syntax check. This increases the stability of your program as type conflicts in statements within the subroutine are prevented.

If you type with the standard types P, N, C or X, the missing characteristic “*field length*” is only passed on from the actual parameter to the formal parameter at runtime. You achieve complete typing with these types (that is, including the field length) by defining and specifying locally defined types.



Hint: You will be introduced to structures and internal tables in a subsequent lesson. Formal parameters for such data objects **must always be typed** in order for their components to be accessed within the subroutine.

<pre> DATA wa_flightinfo TYPE sbc400focc. ... PERFORM fill_wa CHANGING wa_flightinfo. ... FORM fill_wa CHANGING f_wa [TYPE sbc400focc]. f_wa-carrid = f_wa-connid = ENDFORM. </pre> <p style="text-align: center;">Structure type</p> <p style="text-align: center;">Structure components can be addressed</p>	<pre> DATA it_flightinfo TYPE sbc400_t_sbc400focc. ... PERFORM fill_itab CHANGING it_flightinfo. ... FORM fill_itab CHANGING f_itab [TYPE sbc400_t_sbc400focc]. LOOP AT f_itab ENDLOOP. ENDFORM. </pre> <p style="text-align: center;">Table type</p>
--	---

Figure 75: Typing the Interface Parameter for Structures and Internal Tables

Local and Global Data Objects



```

TYPES gty_perc TYPE p DECIMALS 2.

DATA: gv_int1    TYPE i,
      gv_int2    TYPE i,
      gv_result  TYPE gty_perc.
      ...
      } Global variables

      PERFORM calc_perc
          USING gv_int1
              gv_int2
          CHANGING gv_result.
      ...

      FORM calc_perc
          USING
              pv_act TYPE i
              pv_max TYPE i
          CHANGING
              cv_pc  TYPE gty_perc.
          ...
          } Formal parameters
          (only visible locally)

          DATA lv_pc TYPE p LENGTH 16 DECIMALS 1.  }
          (only visible locally)
          ...
          } Local variables

      ENDFORM.
  
```

Figure 76: Visibility of Global and Local Data Objects

Variables defined in the main program are **global** data objects. They are visible (can be addressed) in the entire main program as well as in every subroutine called.

Variables defined within a subroutine are called **local** since they only exist in the relevant subroutine - just like formal parameters. Memory space for formal parameters and local data objects is only allocated when the subroutine is called and is released again after execution.

The formal parameters and local data objects of a subroutine **cannot** have the same names. If there is a **global data object with the same name** as a formal parameter or a local data object, the formal parameter or local data object is addressed **within** the subroutine and the global data object is addressed **outside** the subroutine. This is the so-called **shadow rule**: Within a subroutine the local data object “*shadows*” the global one with the same name.

To identify your internal program objects uniquely, you should use the following prefixes for your subroutine objects: **g...** for “global data objects”, **p...** for “using parameters”, **c...** for “changing parameters”, **l...** for “local data objects”.



```
TYPES gty_perc TYPE p DECIMALS 2.  
  
DATA: gv_int1    TYPE i,  
      gv_int2    TYPE i,  
      gv_result  TYPE gty_perc.  
...  
PERFORM calc_perc  
      USING gv_int1  
           gv_int2  
      CHANGING gv_result.  
...  
  
FORM calc_perc  
  USING  
    value(pv_act)  TYPE i  
    value(pv_max)  TYPE i  
  CHANGING  
    value(cv_pc)   TYPE gty_perc.  
  
  DATA lv_pc TYPE p LENGTH 16 DECIMALS 1.  
  
  lv_pc = pv_act * 100 / pv_max.  
  cv_pc = lv_pc.  
  
ENDFORM.
```

Figure 77: Syntax Example: Local Auxiliary Variables for Rounding

In the syntax example above, the result of the percentage calculation should be rounded internally to the nearest whole number with one decimal place, but nonetheless returned with two decimal places.

To do this, you create a local variable with only one decimal place and store the result of the calculation there first. The runtime system automatically rounds the result to the nearest whole number in accordance with the available number of decimal places. When you then copy the result to the return parameter, a zero is merely added for the second decimal place.

Calling Subroutines

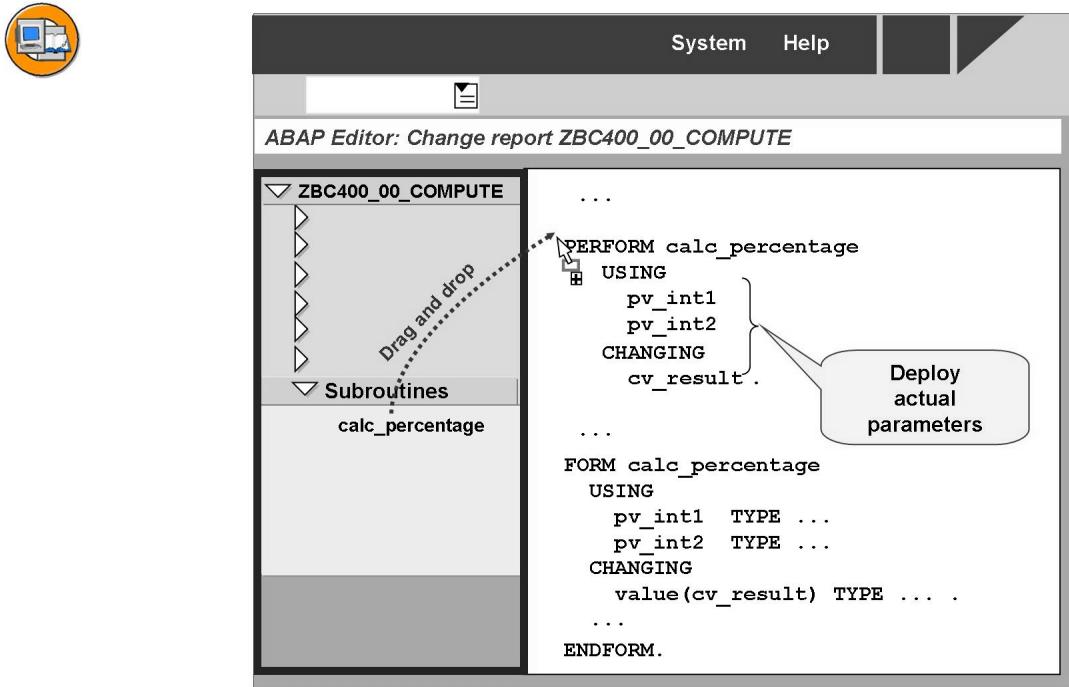


Figure 78: Implementing a Subroutine Call

You can have the **PERFORM statement** for calling a subroutine **generated** into your source code. First, define the subroutine and then save your main program. The newly-defined subroutine appears in the navigation area. Move it to the required call point in your program by drag and drop. All you have to do now is replace the formal parameters in the generated source code with corresponding actual parameters.

(Alternatively, you can also implement the generation of the call using the “*Pattern*” pushbutton in the *ABAP Editor*.)

The advantage of generating the call is that it is impossible to forget or mix parameters.

Modularization Units in the Debugger

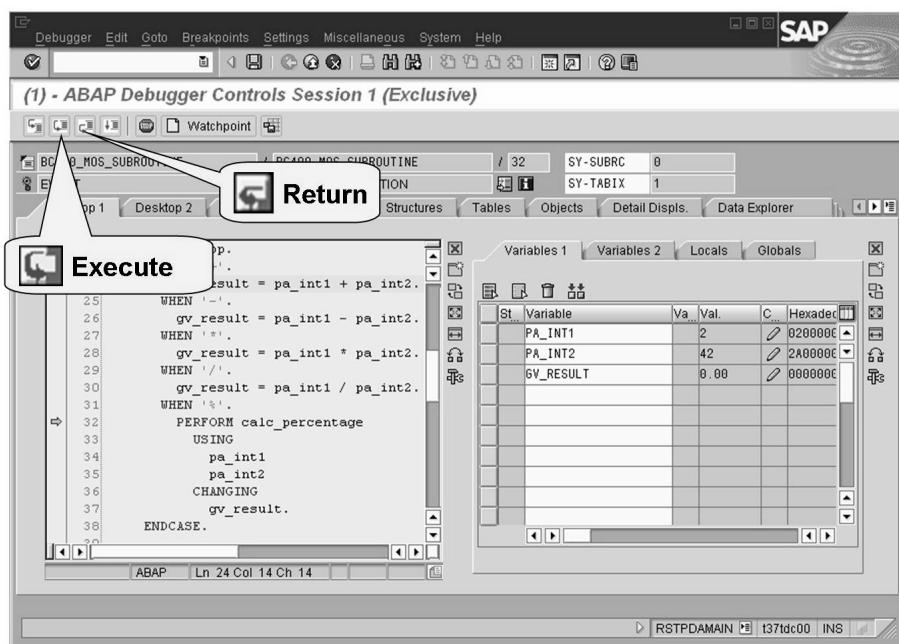


Figure 79: Subroutines in Debugging Mode

If the current statement is a subroutine call, you can execute the entire subroutine without stopping by choosing **Execute**. Processing only stops once the subroutine has been completed.

In contrast, you can use **Single Step** to stop at the first statement of the subroutine and trace its operations in more detail.

If the current statement is located in a subroutine, you can execute the rest of the subroutine without stopping by choosing **Return**. Processing only stops once the subroutine has been completed.

All the Debugger control functions (single step, execute, return, and continue) are also available in the classic *ABAP Debugger* with the same meaning.

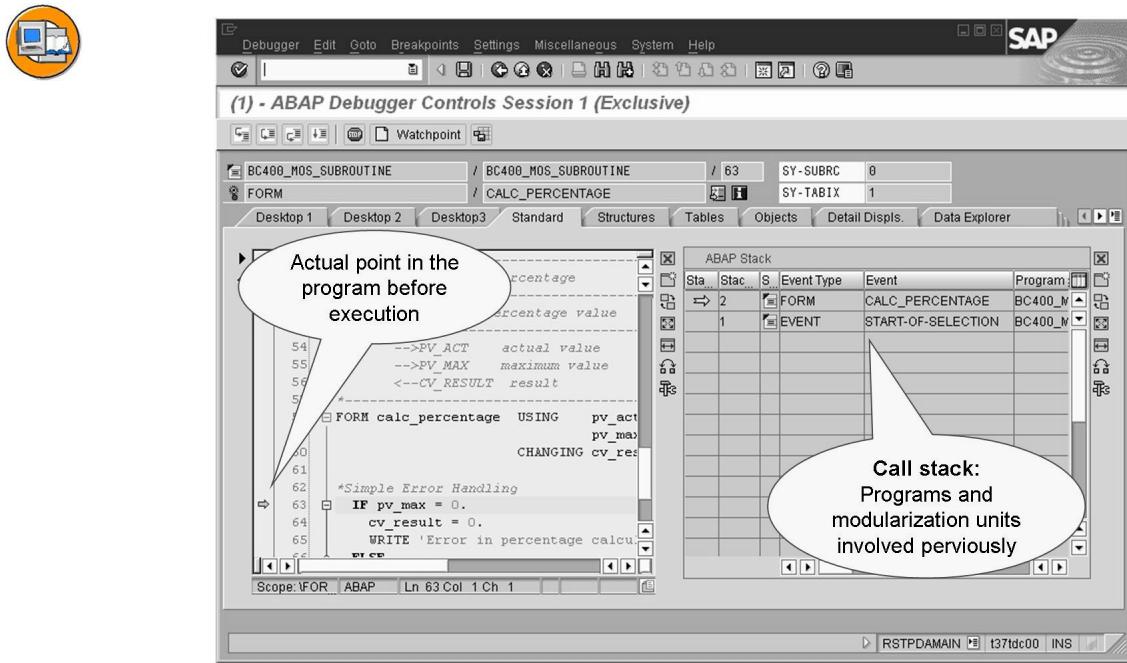


Figure 80: Debugging - Call Stack

Under the *Standard* tab page in the Debugger you can see from which programs the subroutine was called. The tool for this is the **Call Stack**.

Exercise 6: Subroutines

Exercise Objectives

After completing this exercise, you will be able to:

- Create subroutines
- Use the subroutine interface to pass data

Business Example

Modify your calculator program for the basic calculation types so that you can perform another form of calculation (percentage calculation) in a subroutine.

Template:

BC400_DOS_COMPUTE

Solution:

BC400_MOS_SUBROUTINE

Task 1:

Copy your program ZBC400_##_COMPUTE or the template and give it the name **ZBC400_##_SUBROUTINE**.

1. Copy your program or the template.

Task 2:

Create a subroutine (suggested name: **CALC_PERCENTAGE**) that carries out a percentage calculation using two input parameters and returns the result. The value of the first parameter should be determined as a percentage of the second parameter if the value of the second parameter represents 100%.

1. At the end of your program, create a subroutine for percentage calculation.

Continued on next page

2. Define two USING parameters for transferring the operands (suggested name: **PV_ACT** and **PV_MAX**) as well as a CHANGING parameter for returning the result (suggested name: **CV_RESULT**). Type the parameters appropriately to the corresponding global data objects in the main program.



Hint: You cannot use implicit typing for the return parameters (TYPE p LENGTH 16 DECIMALS 2). Declare a local program type at the start of your program with the TYPES statement instead. You can then use this for the actual parameter and the formal parameter.

3. Implement the percentage calculation in the subroutine. Make sure that you avoid the error of division by zero. In this case, output a warning text on the list.

Task 3:

Extend the main program so that you call a new subroutine if the user has entered the arithmetic operator “%”.

1. Extend the IF or CASE structure with a branch that is processed if the parameter contains the value “%”.
2. In this branch, call the subroutine and supply the parameters with suitable actual parameters.



Hint: Generate the subroutine call by using the drag and drop function in the navigation area or the *Pattern* pushbutton.

Task 4:

Test and debug your program.

1. Execute your program and check the result.
2. Follow the program flow with the Debugger. Use the function keys or pushbuttons to exit or skip the subroutine.

Solution 6: Subroutines

Task 1:

Copy your program ZBC400_##_COMPUTE or the template and give it the name **ZBC400_##_SUBROUTINE**.

1. Copy your program or the template.
 - a) Display your template in the navigation area of the *Object Navigator*.
 - b) Right click on the selected program to open the context menu and choose *Copy....*
 - c) In the dialog box that follows, enter the name of the target program and confirm your entry with *Copy*.
 - d) Select all parts of the program using the pushbutton and confirm this with *Copy*.
 - e) Assign this program to your package and your request in the usual way.

Task 2:

Create a subroutine (suggested name: **CALC_PERCENTAGE**) that carries out a percentage calculation using two input parameters and returns the result. The value of the first parameter should be determined as a percentage of the second parameter if the value of the second parameter represents 100%.

1. At the end of your program, create a subroutine for percentage calculation.
 - a) See source code excerpt from the model solution.
2. Define two USING parameters for transferring the operands (suggested name: **PV_ACT** and **PV_MAX**) as well as a CHANGING parameter for returning the result (suggested name: **CV_RESULT**). Type the parameters appropriately to the corresponding global data objects in the main program.



Hint: You cannot use implicit typing for the return parameters (TYPE p LENGTH 16 DECIMALS 2). Declare a local program type at the start of your program with the TYPES statement instead. You can then use this for the actual parameter and the formal parameter.

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

Continued on next page

3. Implement the percentage calculation in the subroutine. Make sure that you avoid the error of division by zero. In this case, output a warning text on the list.
 - a) See source code excerpt from the model solution.

Task 3:

Extend the main program so that you call a new subroutine if the user has entered the arithmetic operator “%”.

1. Extend the IF or CASE structure with a branch that is processed if the parameter contains the value “%”.
 - a) See source code excerpt from the model solution.
2. In this branch, call the subroutine and supply the parameters with suitable actual parameters.



Hint: Generate the subroutine call by using the drag and drop function in the navigation area or the *Pattern* pushbutton.

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

Task 4:

Test and debug your program.

1. Execute your program and check the result.
 - a) Carry out this step as usual.
2. Follow the program flow with the Debugger. Use the function keys or pushbuttons to exit or skip the subroutine.
 - a) Perform this step as described in the course materials.

Result

Source code extract:

```
*&-----*  
*& Report BC400_MOS_SUBROUTINE *  
*&-----*  
REPORT bc400_mos subroutine.
```

```
TYPES gty_result TYPE p LENGTH 16 DECIMALS 2.
```

Continued on next page

```

PARAMETERS:
  pa_int1  TYPE i,
  pa_op    TYPE c LENGTH 1,
  pa_int2  TYPE i.

DATA gv_result TYPE gty_result.

IF ( pa_op = '+' OR
     pa_op = '-' OR
     pa_op = '*' OR
     pa_op = '/' AND pa_int2 <> 0 OR
     pa_op = '%' ).

CASE pa_op.
  WHEN '+'.
    gv_result = pa_int1 + pa_int2.
  WHEN '-'.
    gv_result = pa_int1 - pa_int2.
  WHEN '*'.
    gv_result = pa_int1 * pa_int2.
  WHEN '/'.
    gv_result = pa_int1 / pa_int2.

  WHEN '%'.
    PERFORM calc_percentage
      USING
        pa_int1
        pa_int2
      CHANGING
        gv_result.

ENDCASE.

WRITE: 'Result:'(res), gv_result.

ELSEIF pa_op = '/' AND pa_int2 = 0.
  WRITE: 'No division by zero!'(dbz).
ELSE.
  WRITE: 'Invalid operator!''(iop).
ENDIF.

```

Continued on next page

```
*&-----*
*&      Form calc_percentage
*&-----*
*      calculate percentage value
*-----*
*      -->PV_ACT    actual value
*      -->PV_MAX    maximum value
*      <--CV_RESULT  result
*-----*
FORM calc_percentage USING      pv_act TYPE i
                           pv_max TYPE i
                           CHANGING cv_result TYPE gty_result.

*Simple Error Handling
IF pv_max = 0.
  cv_result = 0.
  WRITE 'Error in percentage calculation'(epc).
ELSE.
*Calculate result
  cv_result = pv_act / pv_max * 100.
ENDIF.

ENDFORM.                      " calc_percentage
```



Lesson Summary

You should now be able to:

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

Lesson: Modularization with Function Modules

Lesson Overview

In this lesson you will learn how to search for function modules, analyze their interfaces and documentation, and test their functions. You will learn how to use existing function modules in your programs and what you have to do to catch and handle errors. Finally, you will create function modules yourself and encapsulate functions in them that you can then reuse in other programs.



Lesson Objectives

After completing this lesson, you will be able to:

- Search for function modules
- Acquire information on the functionality and use of function modules
- Call function modules in your program
- Create a function group
- Create a function module
- Explain the role of BAPIs and identify their special properties

Business Example

You need a function for your program that might already be available in the form of a function module, and want to use this function module in your program, if possible.

If the function does not already exist, or does not meet your requirements, you want to create your own function group and a function module.

Working with Function Modules

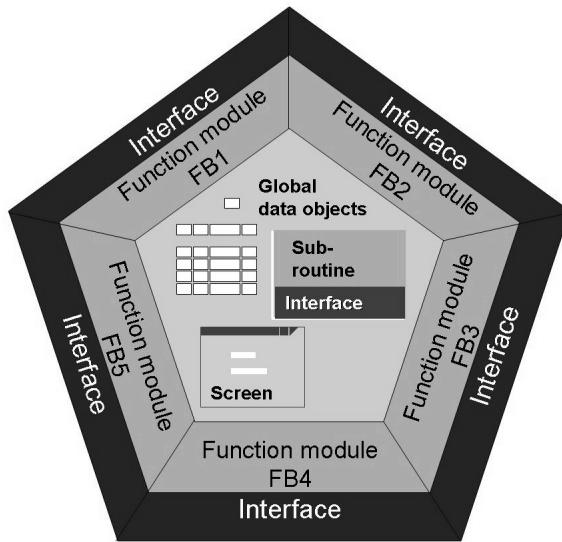


Figure 81: Function Group

A **function module** is a subroutine with the corresponding function that is stored centrally in the Function Library of the SAP system. Each function module has an interface with parameters for importing or exporting data. The main purpose of function modules is their reusability. Hence, they belong to the reuse components.

Function modules are assigned to **function groups**. Each function group is a collection of function modules that have similar functions and/or process the same data.

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 can be changed by all function modules within the group.

Subroutines

These can be called from all function modules in the group.

Screens

These can be called from all function modules in the group.

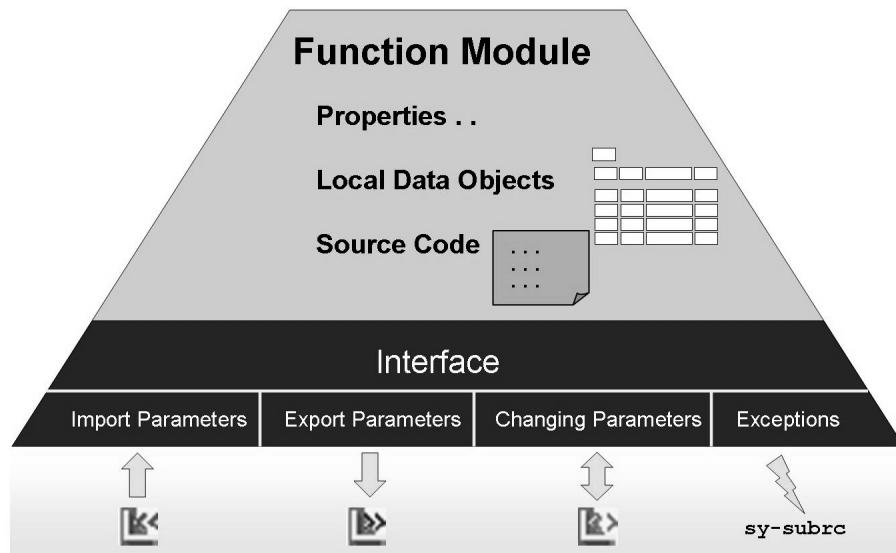


Figure 82: Function module

The **properties** of a function module include, among other things, a short description and the function group it belongs to.

As is the case with subroutines, a function module can contain its own **local type and data object definitions**. These can only be seen within the function module.

The **interface** of a function module can contain the following elements:

- **Import parameter:** Values or variables of the calling program can be transferred to them when the function module is called. The optional parameters do not have to be supplied with data during the call.
- **Export parameter:** The calling program accepts the output of the function module by way of the assignment of a “receiving variable” to an export parameter. Export parameters are always optional.
- **Changing parameter:** You can transfer the variables of the calling program that are changed by the function module to the changing parameters.
- **Exceptions:** They can be raised by the function module in certain error situations and provide information on the respective processing error in the function module. Exceptions should be handled by the calling program.

In general, the interface parameters are assigned to types from the *ABAP Dictionary*.

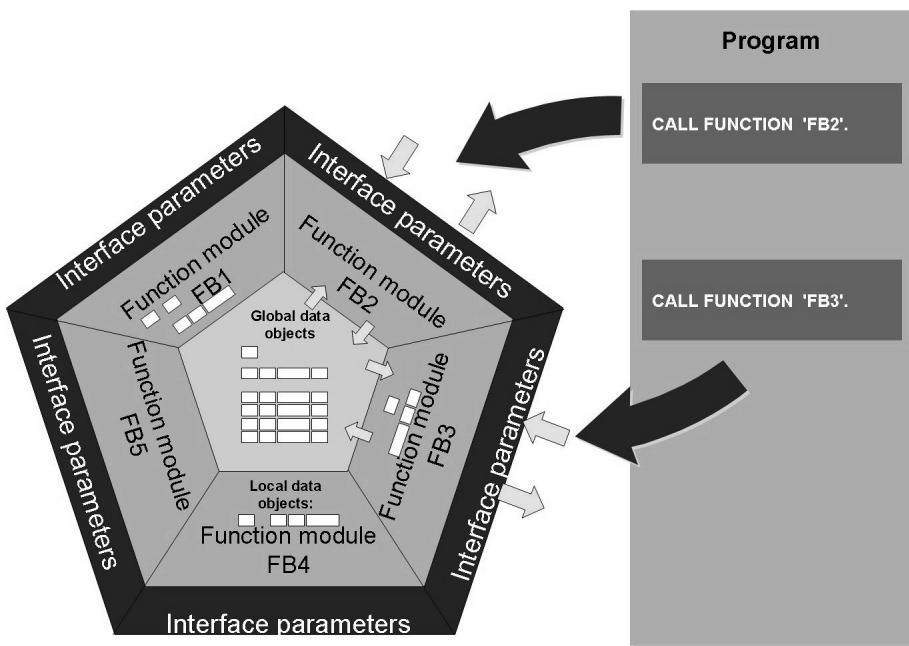


Figure 83: Data Objects Within a Function Group

If a program calls a function module, the entire corresponding function group is loaded and the function module is executed. The function group remains loaded in the working memory until the calling program is closed. When you call another function module of this group, it is therefore processed without needing to be reloaded and with the same global data of the function group.

Thus, if a function module that writes values to the global data of the function group is called, other function modules in the same function group can access this data when they are called in the same program run.

Apart from the global data of its function group, a function module can also access its own locally defined data objects as well as its interface parameters. The latter are used to accept data or return it to the calling program.

Searching for Function Modules



Application-related search:

Search within a particular application component:
(Application hierarchy)

Application-independent search:

Free search independent of application components
(Repository Information System)

Program-related search:

Search within a program that uses the function module being searched for. Search methods :

- Search for CALL FUNCTION statement within the program source code
- Before triggering a program function (e.g. pushbutton) via 'h' activate *Debugger* and set breakpoint at CALL FUNCTION statement
- If the function module searched for passes a screen, use "F1" and "*Technical Information*" to determine the screen number, navigate to it by double-clicking and execute "Where Used List in Programs".

Figure 84: Searching for Function Modules

The above graphic describes various search scenarios:

- The **application-related search** through the *Application Hierarchy* should be used whenever you want to search for function modules within one or more application components.
- You use the **free search** through the *Repository Information System* when searching for function modules independent of application components.
- You use the **program-related search** if the function module you are searching for is called within an existing program.

Once you have found a function module, you should first determine whether it has been released (attributes of the function module). Only with released function modules do you have the right to support and upward compatibility. We recommend you only use released function modules.

You can use the documentation of the function module to find out about its functionality and obtain further information.

Function Module Interface

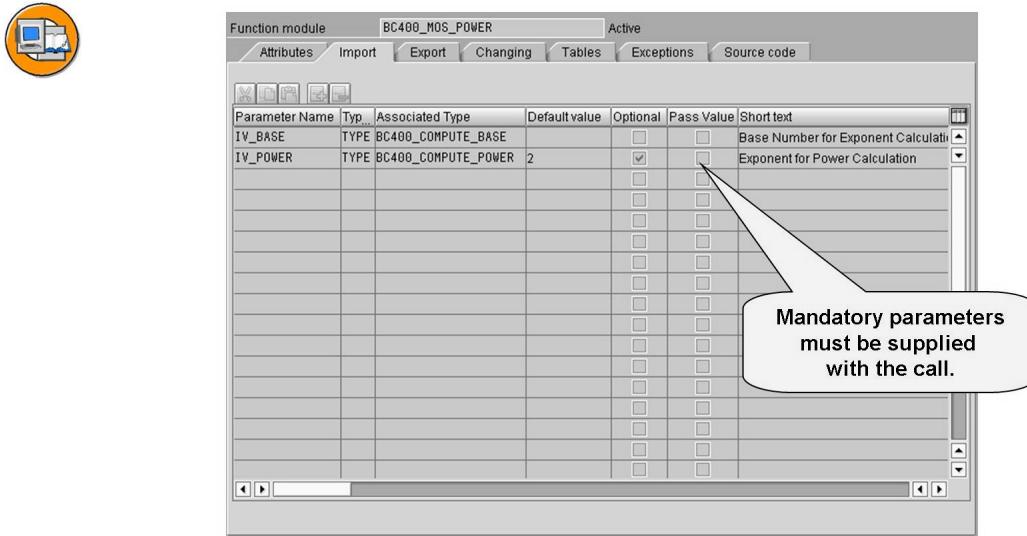


Figure 85: Function Module Interface

The interface of the function module consists of the import, export, changing parameters, and the exceptions.

As with subroutines you can distinguish between pass by value and pass by reference.

Non-optional parameters must be supplied with data when the function module is called. To find out which additional parameters are to be passed and when, refer to the function module documentation and the documentation for the interface parameters.

Documentation and Testing



Documentation

- Short Text
- Function Description
- Example of Use
- Notes
- Parameter Description
- Exception Description

Test Environment



Value Input for Import Parameters



Output of Export Parameters and Exceptions

Figure 86: Documentation and Test Environment

Function modules that are intended and released for use in other programs should be documented. A distinction is made between the **function module documentation** that describes the function of the module and the **parameter documentation** that tells you how individual parameters or exceptions are used.

You can access the function module documentation by choosing the *FModule Documentation* pushbutton. Links from here will take you to the parameter documentation (only in display mode). You can also access the parameter documentation by choosing the pushbutton in the **Long Text** column for the respective parameter. The “green light” indicates whether documentation has been created for the parameter.

If you are still unsure whether the function module does precisely what you want it to do, you can test it in the **test environment** prior to installing it in your program. An input template allows you to specify values for the import and changing parameters. You can also switch to debugging mode. The values in the export and changing parameters are listed after execution.

If the function module triggers an exception due to an error situation, it is displayed, where applicable, with its message text.

The runtime is also 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.

Furthermore, you can store test data in a test data directory and create test sequences.

Calling Function Modules

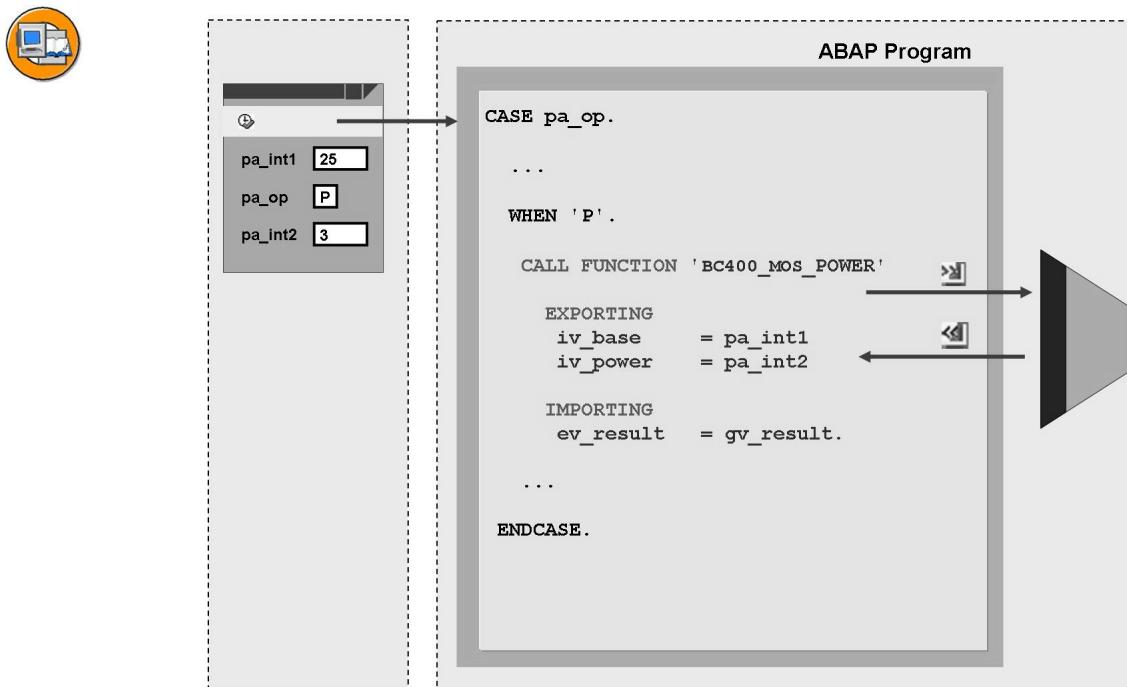


Figure 87: Calling a Function Module

Function modules are called using the `CALL FUNCTION` statement. The name of the function module follows in capital letters enclosed in single quotation marks.

In the **EXPORTING block**, the system passes values to the **import** parameters of the function module.

In the **IMPORTING block** of the call, **actual parameters** are assigned to the **export** parameters. You can use them to access the results of the call.

From the perspective of the calling program, values are exported to the import parameters of the function module and values are imported for the receiving variables that are assigned to the export parameters of the function module.

In the call syntax, you always have to specify the name of the interface parameter (formal parameter) on the left side of the parameter assignment and the data object or the value of the calling program (actual parameter) on the right side.

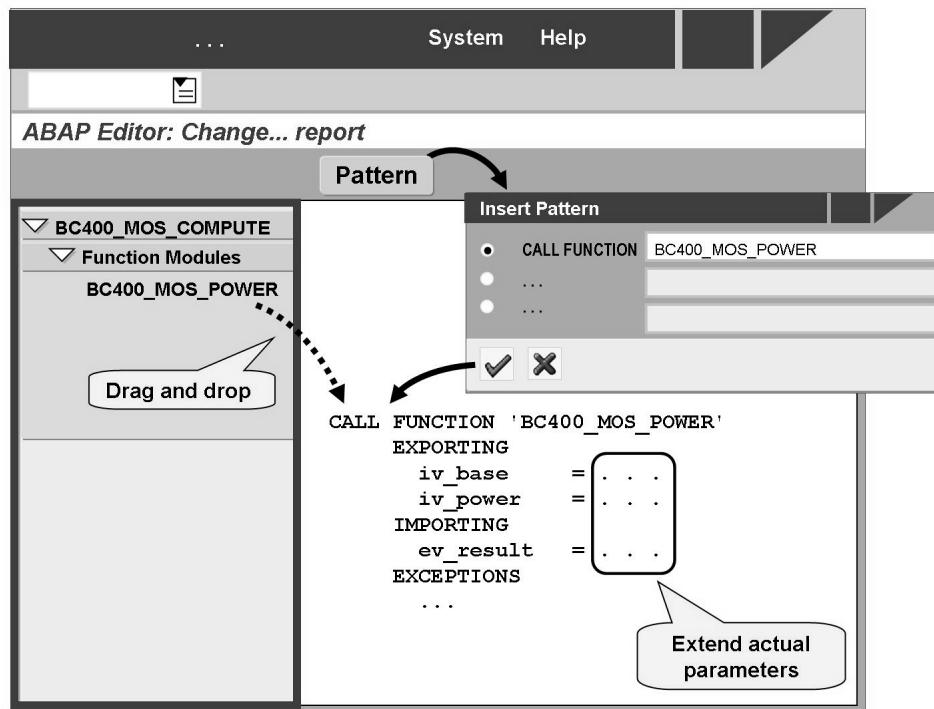


Figure 88: Creating Source Code for a Function Module Call

In order to avoid spelling errors, use the *Pattern* pushbutton to generate the call when you implement a call in the source code. Alternatively, you can display the function group in the navigation area and drag and drop the function module into the editor area to generate the call in the source code.

This generates the entire call syntax, which means that all interface parameters as well as exceptions are listed. Optional information is commented out in the call.

Afterwards, you must fill in the actual parameters and, if necessary, implement exception handling.

Exception Handling

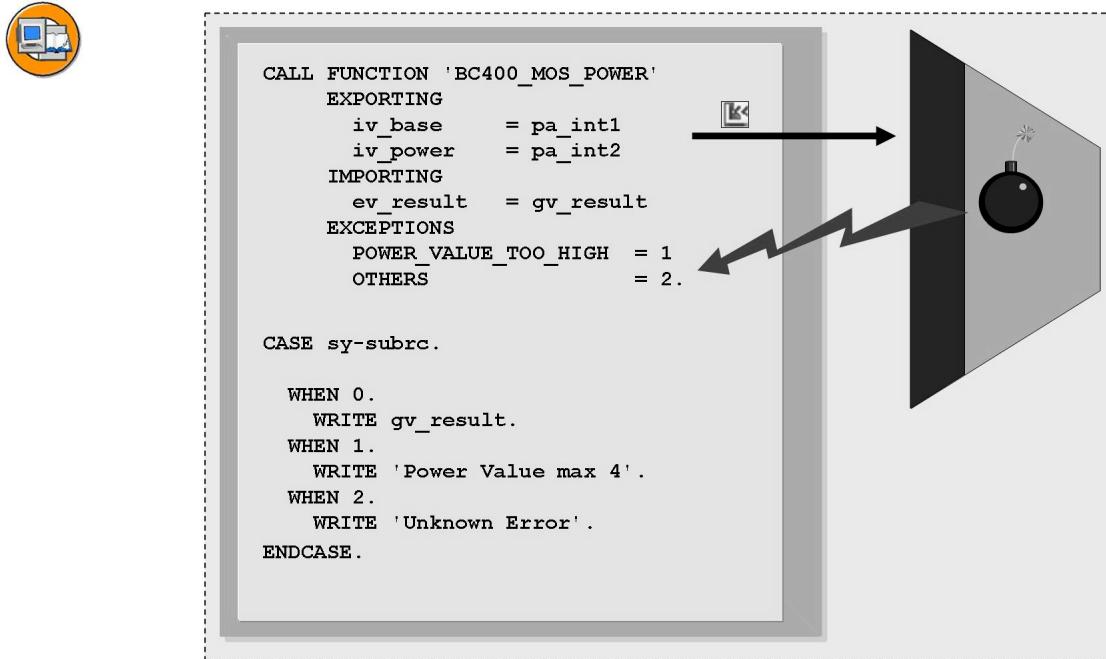


Figure 89: Handling Exceptions

If application errors occur when a function module is processed (for example, a value is not suitable for the calculation), the function module raises the corresponding **exception**. This causes the processing of the function module to be canceled and the system returns to the calling program. If the exception is listed (caught) in the EXCEPTIONS block of the call, the specified return code is entered in the sy-subrc system field of the calling program. By checking this field after the call, you can thus determine if and, where applicable, which exception was raised so you can react accordingly. If no exception was raised by the function module, the sy-subrc of the calling program is set to zero.

You also have the option of setting a return code explicitly for some of the possible exceptions and setting a different return code for all non-mentioned exceptions. To do so, list the *formal exception* OTHERS with the desired return code.

In your call, you should catch all exceptions and react to them in the program. If an exception is raised and is not caught, a runtime error occurs.

Creating Function Groups and Function Modules

In this section you will learn how to create a function module yourself, how to define its interface and implement the source code. Since each function module must be contained in a function group, you will first learn how to create a new function group, if necessary.

Creating Function Groups

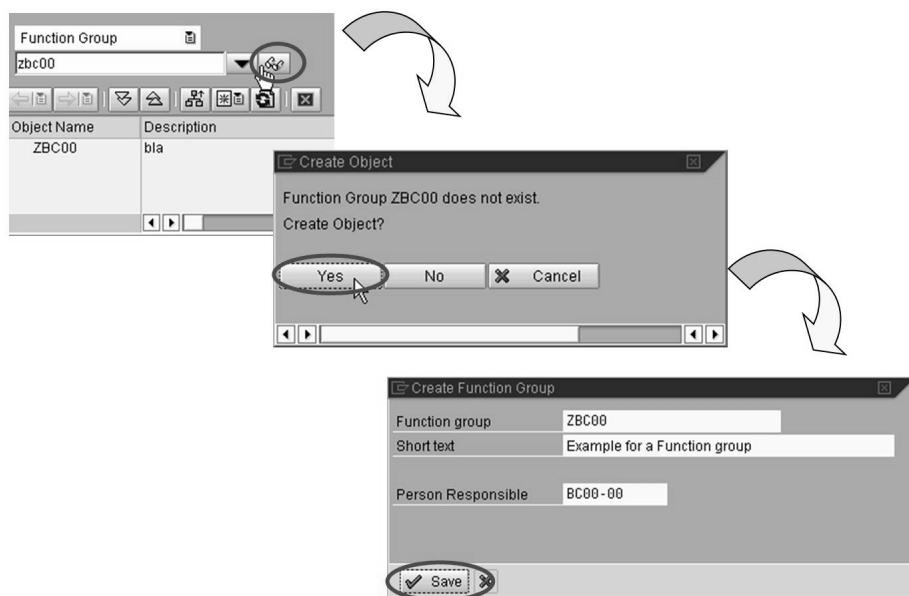


Figure 90: Creating a Function Group

To create a function group, proceed as follows:

1. Choose the object list for a function group in the *Object Navigator*.
2. Enter the name of the new function group and choose **ENTER** or the **Display** pushbutton Be sure to use the customer namespace in the process.
A dialog box appears, asking you if you want to create the new function group.
3. Click **Yes** to confirm.
A dialog box with the attributes of the function group appears.
4. Enter a short text and save it by choosing the **Save** pushbutton.
5. Assign the function group to a package and a correction in the usual manner in the dialog boxes that follow.

Creating Function Modules

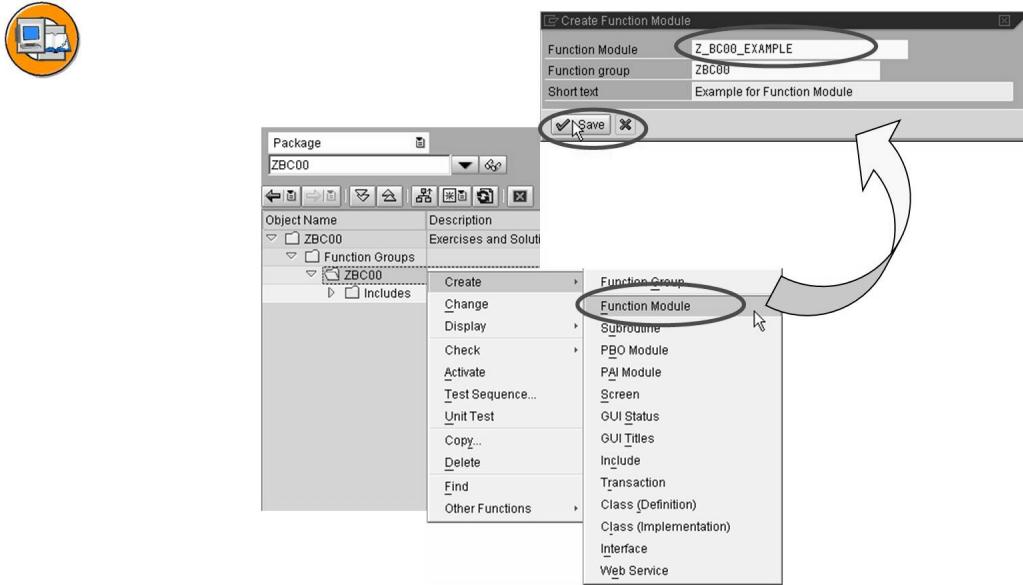


Figure 91: Creating a Function Module

To create the function module, proceed as follows:

1. Decide in which function group you want to create the new function module.
2. Display the object list for the function group in the navigation area of the *Object Navigator*.
3. In the tree structure, open the context menu for the function group and choose *Create → Function Module*.

A dialog box with the attributes of the function module appears.

4. Enter a name and a short text for the function module. Be sure to use the customer namespace for function modules (customer function modules start with "Z_" or "Y_").
5. Choose the *Save* pushbutton to save your entries.
6. If necessary, assign the function module to a package and a correction in the dialog boxes that follow.



The screenshot shows the SAP ABAP Function Module Editor. The title bar displays 'Function module Z_BC400_00_PERCENTAGE' and 'Inactive'. Below the title bar, there are tabs for 'Attributes', 'Import', 'Export', 'Changing', 'Tables', 'Exceptions', and 'Source code'. The 'Source code' tab is highlighted with a red oval. The main area contains the ABAP source code for the function module:

```
1 FUNCTION Z_BC400_00_PERCENTAGE.
2 /**
3  ** Local Interface:
4  ** IMPORTING
5  **   REFERENCE(IV_ACT) TYPE BC400_ACT
6  **   REFERENCE(IV_MAX) TYPE BC400_MAX
7  ** EXPORTING
8  **   REFERENCE(EV_PERCENTAGE) TYPE BC400_PERC
9  ** EXCEPTIONS
10 **   DIVISION_BY_ZERO
11 /**
12 *
13 *Simple error handling
14 IF iv_max = 0.
15   ev_percentage = 0.
16   RAISE division_by_zero.
17 ELSE.
18 *Calculate result
19   .
20 ENDIF.
21 *
22 ENDFUNCTION.
```

The status bar at the bottom indicates 'Scope:FUNCTION Z_BC400_00_PERCENTAGE|F' and 'ABAP Ln 19 Col 9 Ch 9'.

Figure 92: Editing the Source Code

After you have defined the corresponding IMPORTING and EXPORTING parameters you can switch to the *Source Code* tab page in order to implement the functions of the function module.



Hint: Note that the comment block directly under the key word FUNCTION is created automatically by the *Function Builder* and is changed when changes are made to the parameters.

Working with BAPIs

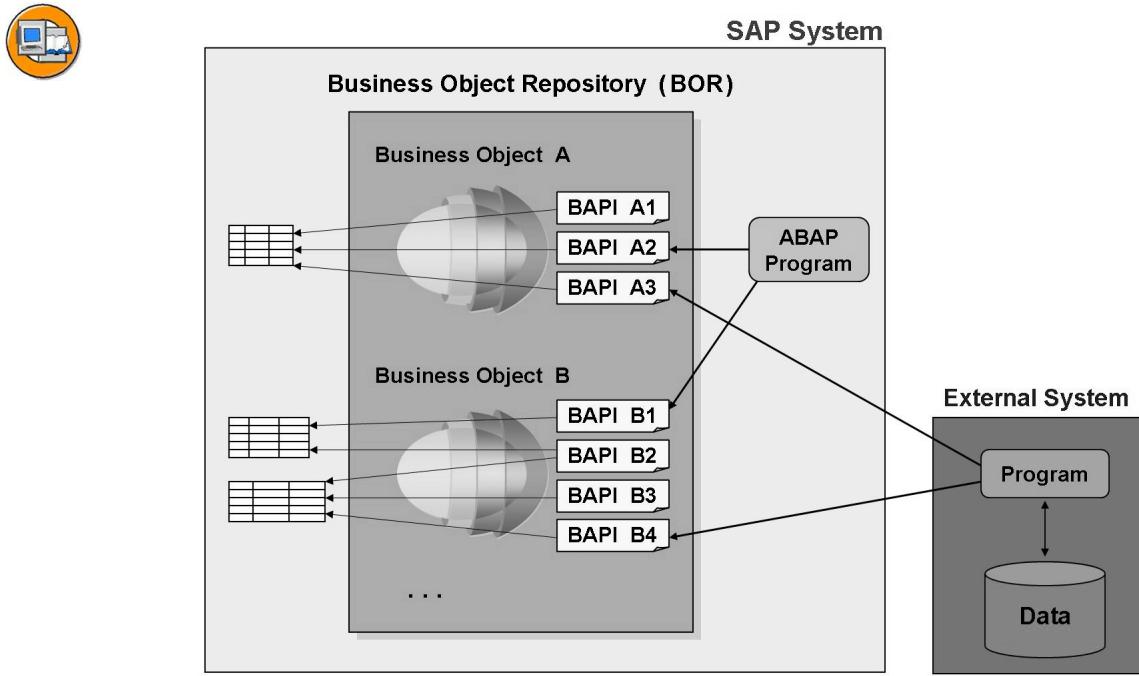


Figure 93: Business Objects and BAPIs

The **Business Object Repository (BOR)** in the SAP system contains **business objects**. Technically speaking, a business object is a class and corresponds to an SAP table or a table hierarchy. A business object has **BAPIs** (Business Application Programming Interfaces) as methods. You can call these BAPIs to access the corresponding table(s). Hence, a BAPI is a means of accessing data in the SAP system.

BAPIs usually exist for basic functions of a business object, such as:

- Creating an object
- Retrieving the attributes of an object
- Changing the attributes of an object
- Listing the objects

The functions of a BAPI are encapsulated in a function module that can be called remotely. BAPIs can therefore be called by ABAP programs of the same SAP system as well as by external programs.

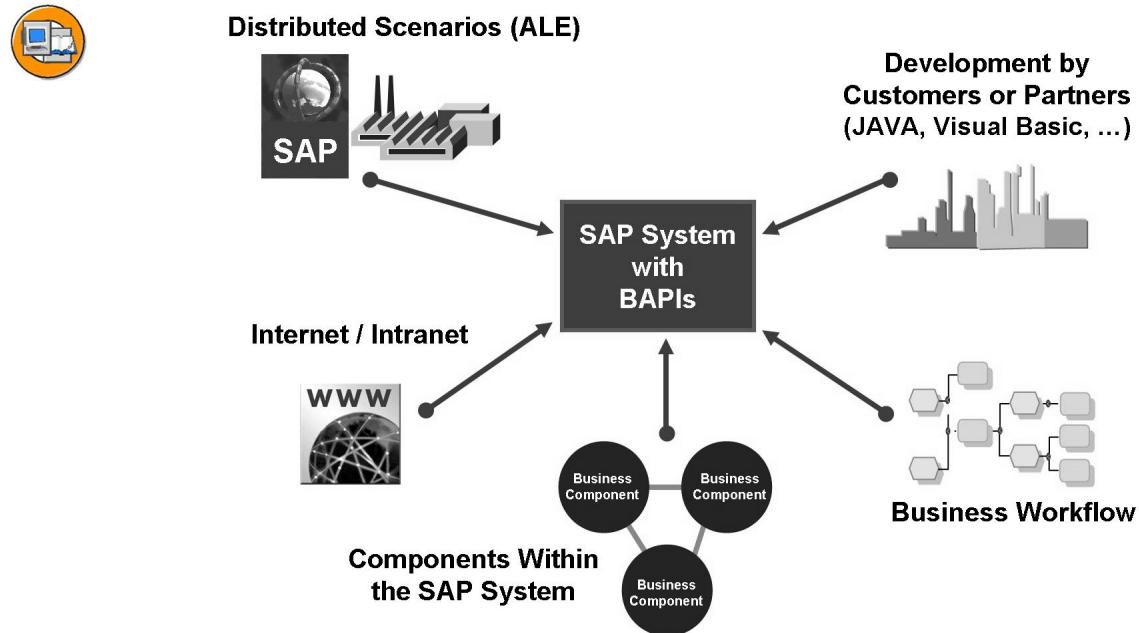


Figure 94: Using BAPIs

The graphic above shows how BAPIs can be used.

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 available objects that meet the specified selection criteria.

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

Adds and removes subobjects (for example, item for an order)

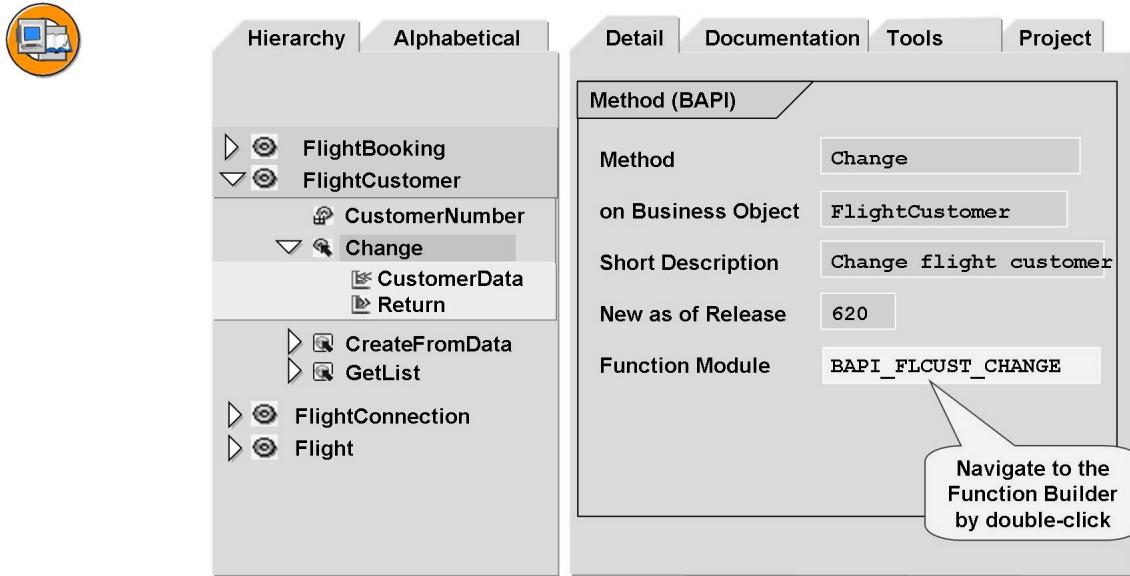


Figure 95: BAPI Explorer

You can use the **BAPI Explorer** to list business objects as well as the corresponding BAPIs with reference to the application. To call the **BAPI Explorer** use the following path in the *SAP Easy Access* menu: *Tools* → *Business Framework* → *BAPI Explorer* or the transaction BAPI.

Once you have found the required business object or BAPI, you can display the relevant details on the right of the screen by selecting the BAPI. You can navigate to its display using the *Function Builder* by double-clicking on the displayed function module.

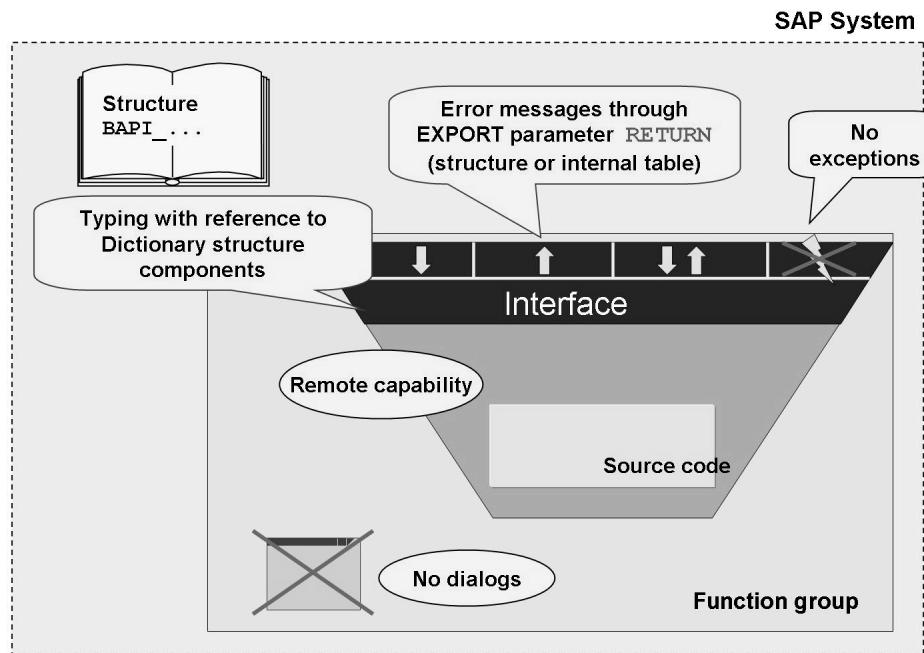


Figure 96: Properties of a BAPI Function Module

Function modules for BAPIs must fulfill the following technical requirements:

- Naming convention **BAPI_<business_object_name>_<method_name>**
- Remote enabled
- Contain neither user dialogs nor messages
- Interface parameters are typed with components of structures from the *ABAP Dictionary* that were created for this BAPI. (Name prefix for such structures: **BAPI_**)
- Must not contain any **changing** parameters until release 4.6
- Raise no exceptions: Errors are reported to the user through the special export parameter **RETURN**.

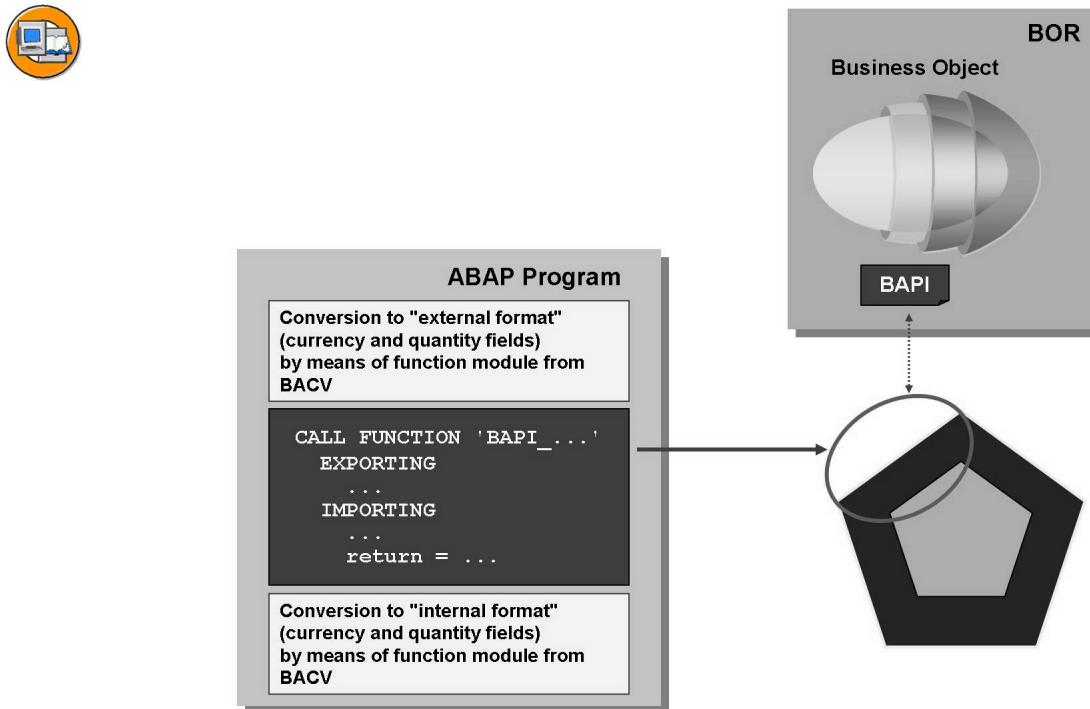


Figure 97: BAPI Call in an ABAP Program

If you want to use a BAPI within the same SAP system, you have to call the relevant function module directly. Note the restrictions already mentioned.



Hint: BAPI interfaces are created according to “external” call requirements, that is, a call from an external system. Amounts are therefore expected in an external format with 4 or 9 decimal places. During the call, 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 or re-conversion you can use function modules from the **BACV** function group (package **SBF_BAPI**).

Exercise 7: Using a Function Module

Exercise Objectives

After completing this exercise, you will be able to:

- Search for a function module in the Repository
- Use a function module

Business Example

You have to use a globally defined function module in your program to calculate powers.

Template:

BC400_MOS_SUBROUTINE

Solution:

BC400_MOS_FUNCTION_MODULE_1

Task 1:

Search for a function module to calculate powers and find out about its interface and function.

1. Search in the *Repository* for a function module that calculates a result from a base number and an exponent.
2. Analyze the function module interface and read the documentation.

Continued on next page

3. Which parameters do you have to pass with the call? How are they typed? What restrictions apply to the value range?

4. Which parameter contains the calculation result and how is it typed?

5. Which exceptions can the function module raise and to which errors do these exceptions refer?

Exception	Error

6. Test the function module.

Task 2:

Copy your program ZBC400_##_SUBROUTINE or the copy template to the new name **ZBC400_##_FUNCTION_MODULE** and implement a function module call in the event that the user enters the operator “P”.

1. Copy your program or the template.
2. Extend the corresponding IF or CASE structure with a branch that is processed if the corresponding parameter contains the value “P”.

Continued on next page

3. Call the function module in this processing branch. Transfer the two values entered by the user and receive the result.



Hint: Generate the function module call by using the drag and drop function in the navigation area or the *Pattern* pushbutton.

4. Evaluate any exceptions that may occur and output an explanatory text in the results list.

Solution 7: Using a Function Module

Task 1:

Search for a function module to calculate powers and find out about its interface and function.

1. Search in the *Repository* for a function module that calculates a result from a base number and an exponent.
 - a) Perform this step as described in the participants' handbook. Search in the *Repository Information System* for function modules that have the term "power" in the short text, for example.



Hint: If the options described do not yield a suitable function module (*Pattern* or *Repository Information System*), use the function module **BC400_MOS_POWER**.

2. Analyze the function module interface and read the documentation.
 - a) Perform this step as described in the participants' handbook.
3. Which parameters do you have to pass with the call? How are they typed? What restrictions apply to the value range?

Answer: Parameter iv_power is optional, and does not therefore have to be specified (the number is then squared).

Parameter iv_base and iv_power are whole numbers.

The parameter iv_power can have no value greater than 4.

4. Which parameter contains the calculation result and how is it typed?
Answer: Parameter ev_result is a packed number with 31 figures, two of which are decimal places.
5. Which exceptions can the function module raise and to which errors do these exceptions refer?

Exception	Error
POWER_VALUE_TOO_HIGH	Exponent too high (>4)
RESULT_VALUE_TOO_HIGH	Result too high

Continued on next page

6. Test the function module.
 - a) Perform this step as described in the participants' handbook.

Task 2:

Copy your program ZBC400_##_SUBROUTINE or the copy template to the new name **ZBC400_##_FUNCTION_MODULE** and implement a function module call in the event that the user enters the operator “P”.

1. Copy your program or the template.
 - a) Carry out this step in the usual way.
2. Extend the corresponding IF or CASE structure with a branch that is processed if the corresponding parameter contains the value “P”.
 - a) See the source code excerpt from the model solution.
3. Call the function module in this processing branch. Transfer the two values entered by the user and receive the result.



Hint: Generate the function module call by using the drag and drop function in the navigation area or the *Pattern* pushbutton.

- a) See the source code excerpt from the model solution.
4. Evaluate any exceptions that may occur and output an explanatory text in the results list.
 - a) See the source code excerpt from the model solution.

Result

Source code extract:

```
*&-----*
*& Report BC400_MOS_FUNCTION_MODULE_1 *
*&-----*
REPORT bc400_mos_function_module_1 .

TYPES gty_result TYPE p LENGTH 16 DECIMALS 2.

PARAMETERS:
  pa_int1  TYPE i,
  pa_op    TYPE c LENGTH 1,
```

Continued on next page

```

pa_int2  TYPE i.

DATA gv_result TYPE gty_result.

IF ( pa_op = '+' OR
     pa_op = '-' OR
     pa_op = '*' OR
     pa_op = '/' AND pa_int2 <> 0 OR
     pa_op = '%' OR
     pa_op = 'P' ).

CASE pa_op.
  WHEN '+'.
    gv_result = pa_int1 + pa_int2.
  WHEN '-'.
    gv_result = pa_int1 - pa_int2.
  WHEN '*'.
    gv_result = pa_int1 * pa_int2.
  WHEN '/'.
    gv_result = pa_int1 / pa_int2.
  WHEN '%'.
    PERFORM calc_percentage
    USING
      pa_int1
      pa_int2
    CHANGING
      gv_result.

  WHEN 'P'.
    CALL FUNCTION 'BC400_MOS_POWER'
      EXPORTING
        iv_base          = pa_int1
        iv_power         = pa_int2
      IMPORTING
        ev_result        = gv_result
      EXCEPTIONS
        power_value_too_high = 1
        result_value_too_high = 2
        OTHERS             = 3.

CASE sy-subrc.
  WHEN 0.

```

Continued on next page

```

*      no action needed
WHEN 1.
  WRITE 'Max value of power is 4' (mvp).
WHEN 2.
  WRITE 'Result value too high' (rvh).
WHEN 3.
  WRITE 'Unknown error' (uer).
ENDCASE.

ENDCASE.

WRITE: 'Result:' (res), gv_result.

ELSEIF pa_op = '/' AND pa_int2 = 0.
  WRITE: 'No division by zero!' (dbz).
ELSE.
  WRITE: 'Invalid operator!' (iop).
ENDIF.

&-----*
*&      Form calc_percentage
*&-----*
*      calculate percentage value
*-----*
*      -->PV_ACT    actual value
*      -->PV_MAX    maximum value
*      <--CV_RESULT  result
*-----*
FORM calc_percentage USING    pv_act TYPE i
                           pv_max TYPE i
                           CHANGING cv_result TYPE gty_result.

*Simple Error Handling
IF pv_max = 0.
  cv_result = 0.
  WRITE 'WARNING!!! Error in percentage calculation' (epc).
ELSE.
*Calculate result
  cv_result = pv_act / pv_max * 100.
ENDIF.

ENDFORM.          " calc_percentage

```


Exercise 8: Creating a Function Group

Exercise Objectives

After completing this exercise, you will be able to:

- Create function groups

Business Example

Your company wants to develop an application in which several functions are used in different programs.

You are in charge of implementing calculation functions. You decide to program it as a **function group** with **function modules**. You will implement this project successively in this exercise and the subsequent exercise in this unit.

Template:

None

Solution:

BC400_MOS (function group)

Task:

Create a function group.

1. Create the function group **ZBC400_##_COMP**.

Solution 8: Creating a Function Group

Task:

Create a function group.

1. Create the function group **ZBC400##COMP**.
 - a) Perform this step as described in the participants' handbook.

Exercise 9: Creating and Using a Function Module

Exercise Objectives

After completing this exercise, you will be able to:

- Create and use a function module

Business Example

You want to create a global function module to calculate percentage values. This is meant to replace the subroutine from a previous exercise.

Template:

BC400_MOS_FUNCTION_MODULE_1 (program)

Solution:

BC400_MOS_FUNCTION_MODULE_2 (program)

BC400_MOS_PERCENTAGE (function module)

Task 1:

In the function group you defined in the last exercise (ZBC400_##_COMP), create a function module named **Z_BC400_##_COMP_PERCENTAGE** that can be used to calculate percentage values.

1. Create the function module.
2. Maintain the interface for the function module.

To do this, create the following parameters and type them with the specified data elements:

Parameter Name	Parameter Type	Data element
iv_max	Import	BC400_MAX
iv_act	Import	BC400_ACT
ev_percentage	Export	BC400_PERC

3. Create an exception that is raised if the values that are transferred result in a division by zero (suggested name: DIVISION_BY_ZERO).

Continued on next page

4. Implement the function module. Use the subroutine in the template for orientation. If an error occurs, raise the exception instead of outputting an error text.



Hint: To raise the exception, use the RAISE <exception_name>. statement.

Task 2:

Copy your program ZBC400_##_FUNCTION_MODULE or the copy template to the new name **ZBC400_##_FUNCTION_MODULE_2** and replace the subroutine call with a call for the new function module.

1. Copy the program.
2. Remove the subroutine call and implement a call for the new function module in its place.



Hint: Generate the function module call by using the drag and drop function in the navigation area or the *Pattern* pushbutton.

3. Handle the exception and output a corresponding text if an error occurs.

Solution 9: Creating and Using a Function Module

Task 1:

In the function group you defined in the last exercise (ZBC400_##_COMP), create a function module named **Z_BC400_##_COMP_PERCENTAGE** that can be used to calculate percentage values.

1. Create the function module.
 - a) Perform this step as described in the participants' handbook.
2. Maintain the interface for the function module.

To do this, create the following parameters and type them with the specified data elements:

Parameter Name	Parameter Type	Data element
iv_max	Import	BC400_MAX
iv_act	Import	BC400_ACT
ev_percentage	Export	BC400_PERC

- a) Perform this step as described in the participants' handbook.
3. Create an exception that is raised if the values that are transferred result in a division by zero (suggested name: DIVISION_BY_ZERO).
 - a) Perform this step as described in the participants' handbook.
4. Implement the function module. Use the subroutine in the template for orientation. If an error occurs, raise the exception instead of outputting an error text.



Hint: To raise the exception, use the RAISE <exception_name>. statement.

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

Continued on next page

Task 2:

Copy your program ZBC400_##_FUNCTION_MODULE or the copy template to the new name **ZBC400_##_FUNCTION_MODULE_2** and replace the subroutine call with a call for the new function module.

1. Copy the program.
 - a) Carry out this step in the usual manner.
2. Remove the subroutine call and implement a call for the new function module in its place.



Hint: Generate the function module call by using the drag and drop function in the navigation area or the *Pattern* pushbutton.

- a) See the source code excerpt from the model solution.
3. Handle the exception and output a corresponding text if an error occurs.
 - a) See the source code excerpt from the model solution.

Result

Source code extract from program:

```
*&-----*
*& Report BC400_MOS_FUNCTION_MODULE_2 *
*&-----*
REPORT bc400_mos_function_module_2 .

PARAMETERS:
  pa_int1  TYPE i,
  pa_op    TYPE c LENGTH 1,
  pa_int2  TYPE i.

DATA gv_result TYPE p LENGTH 16 DECIMALS 2.

IF ( pa_op = '+' OR
     pa_op = '-' OR
     pa_op = '*' OR
     pa_op = '/' AND pa_int2 <> 0 OR
     pa_op = '%' OR
     pa_op = 'P' ).
```

Continued on next page

```

CASE pa_op.
  WHEN '+'.
    gv_result = pa_int1 + pa_int2.
  WHEN '-'.
    gv_result = pa_int1 - pa_int2.
  WHEN '*'.
    gv_result = pa_int1 * pa_int2.
  WHEN '/'.
    gv_result = pa_int1 / pa_int2.
  WHEN 'P'.
    CALL FUNCTION 'BC400_MOS_POWER'
      EXPORTING
        iv_base          = pa_int1
        iv_power         = pa_int2
      IMPORTING
        ev_result        = gv_result
      EXCEPTIONS
        power_value_too_high = 1
        result_value_too_high = 2
        OTHERS             = 3.
  CASE sy-subrc.
    WHEN 0.
    *      no action needed
    WHEN 1.
      WRITE 'Max value of power is 4' (mvp).
    WHEN 2.
      WRITE 'Result value too high' (rvh).
    WHEN 3.
      WRITE 'Unknown error' (uer).
  ENDCASE.
  WHEN '%'.
    CALL FUNCTION 'BC400_MOS_PERCENTAGE'
      EXPORTING
        iv_act           = pa_int1
        iv_max          = pa_int2
      IMPORTING
        ev_percentage   = gv_result
      EXCEPTIONS
        division_by_zero = 1
        OTHERS           = 2.

```

Continued on next page

```

        IF sy-subrc <> 0.
          WRITE 'Error in Function Module'(efm).
        ENDIF.

      ENDCASE.

      WRITE: 'Result:'(res), gv_result.

ELSEIF pa_op = '/' AND pa_int2 = 0.
  WRITE: 'No division by zero!'(dbz).
ELSE.
  WRITE: 'Invalid operator!''(iop).
ENDIF.

```

Source code extract from function module:

```

FUNCTION BC400_MOS_PERCENTAGE.

*-----*
*   Lokale Schnittstelle:
*   IMPORTING
*     REFERENCE(IV_ACT) TYPE BC400_ACT
*     REFERENCE(IV_MAX) TYPE BC400_MAX
*   EXPORTING
*     REFERENCE(EV_PERCENTAGE) TYPE BC400_PERC
*   EXCEPTIONS
*     DIVISION_BY_ZERO
*-----*
*Error handling
  IF iv_max = 0.
    ev_percentage = 0.
    RAISE division_by_zero.
  ELSE.
*Calculate result
    ev_percentage = iv_act / iv_max * 100.
  ENDIF.
ENDFUNCTION.

```



Lesson Summary

You should now be able to:

- Search for function modules
- Acquire information on the functionality and use of function modules
- Call function modules in your program
- Create a function group
- Create a function module
- Explain the role of BAPIs and identify their special properties

Lesson: Modularization with Methods of Global Classes

Lesson Overview

In this lesson you will be introduced to another technique for cross-program modularization: The call of methods of global classes. In a similar way to the function groups and function modules, we will first look for, analyze, test, and use existing classes and methods. In an additional step we will then create a global class with a simple method.

Note that this lesson is not intended to provide a comprehensive introduction to object-oriented programming with ABAP. You should merely familiarize yourself with the basic terms of object-oriented programming so that you can use the functions of existing global classes in your own programs.

The course BC401 - ABAP Objects offers a systematic introduction to object-oriented programming with ABAP objects.



Lesson Objectives

After completing this lesson, you will be able to:

- Explain the basic terms of object-oriented programming
- Acquire information about the function and use of global classes and their methods
- Call methods of global classes in your programs
- Create global classes
- Create and implement simple methods in global classes

Business Example

In your program you need a function that may already exist in the form of a method of a global class. You would like to implement this method in your program.

If the function does not already exist, or does not meet your requirements, you want to create your own global class and method.

Principles of Object-Oriented Programming

Before we can use global classes and methods, we need to know some of the basic terms of object-oriented programming.

Classes, Attributes, and Methods

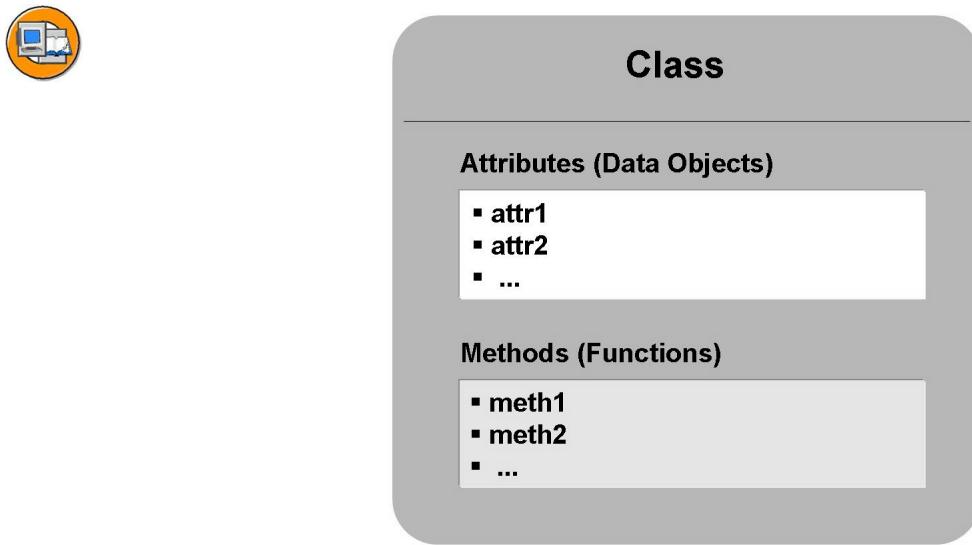


Figure 98: Representation of a Global Class

In one of the previous lessons you were introduced to function groups that make reusable units available in the form of function modules.

In the course of object-oriented enhancements to ABAP, global **classes** were introduced that make functions available in a similar way in the form of **methods**. Like function modules, methods also have an interface, known as a **signature**, that comprises import, export and changing parameters and exceptions.

In addition to methods, classes have other components too. Amongst other things, they also contain global data objects, known as **attributes**. In the same way that the global data objects of a function group can be accessed by all the function modules in the group, all the methods of a class can access the attributes of that class.

Encapsulating Data, Public and Private Components

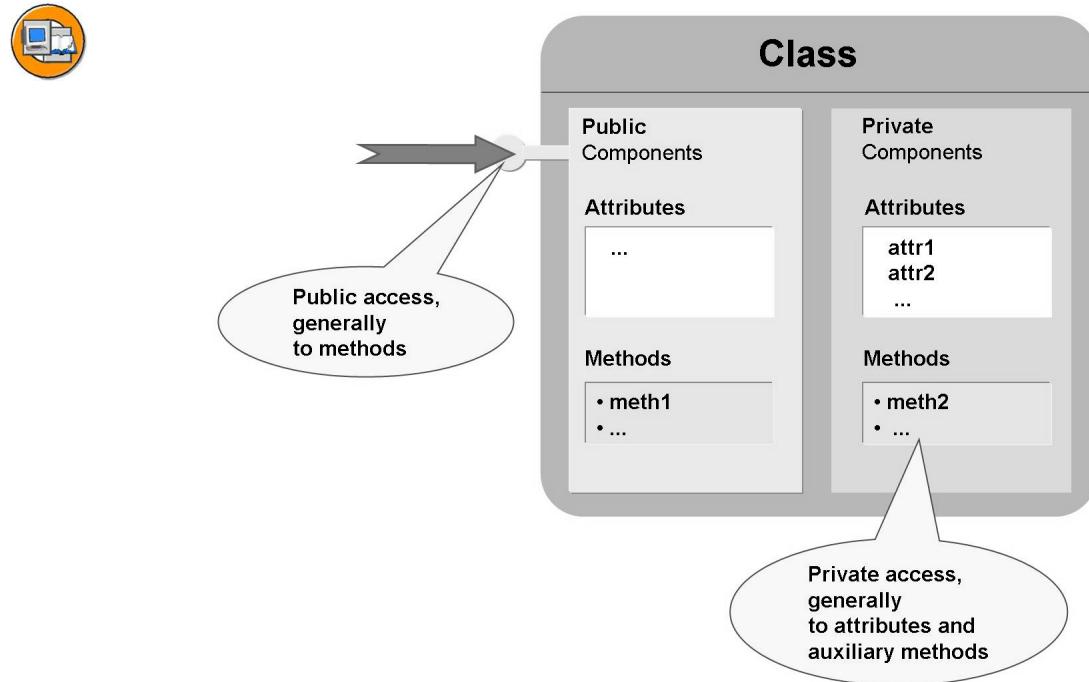


Figure 99: Example of the Access Options for a Global Class

With function groups, we know that the global data objects they contain are not visible outside the function group. This is termed the **encapsulation** of data in the function group.

Attributes too are normally encapsulated in the class, and can therefore only be read or changed using methods of the same class.

In contrast to function modules, classes also allow you to make specific attributes visible to users of the class. A distinction is therefore made between **public** and **private attributes**.

This distinction can be applied not only to attributes but also to methods. Whereas all function modules can be called from outside the function group, only **public methods** are available outside the class. **Private methods** can only be called by other methods of the same class, and are thus fairly similar in this respect to subroutines (for routines) within a function group.

Multiple Instantiation of Classes

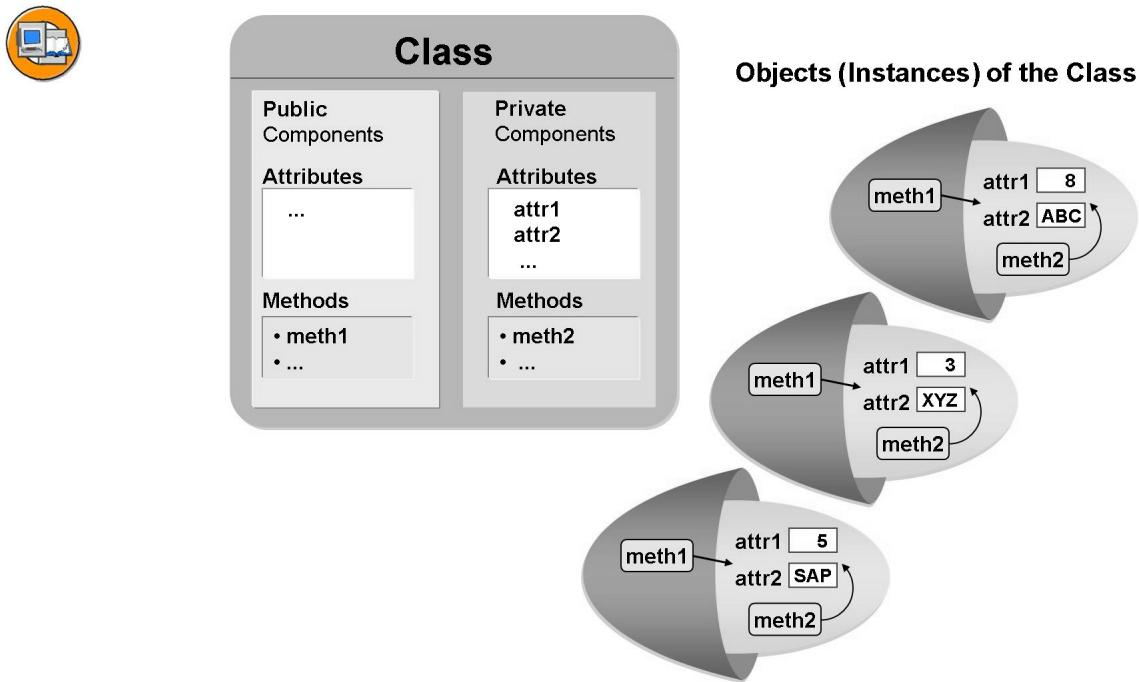


Figure 100: Classes and Objects

The major difference between global classes and function groups is that a function group with its global data objects can only be loaded once to the program context for each main program, whereas a global class can be loaded as many times as you like. This is known as **multiple instantiation** of the class. In practice, this means that the values in the global data objects of a function group are the same for all function module calls. By contrast, a class can have several instances (also known as **objects**), each of which is stored separately in the program context of the main program. Each instance can therefore have different attribute values. Consequently, a method sees different values in the attributes, depending on the instance for which it was called.

You have to generate a class instance explicitly in the ABAP source code using the **CREATE OBJECT** statement.

Instance Components and Static Components

Attributes that can have a different value for each instance are known as **instance attributes**, in order to distinguish them from **static attributes** (or class attributes). Static attributes exist only once for each program context, regardless of how many

class instances are generated when a program runs. If instance methods access a static attribute, all instances see the same value. Above all, an instance method can change the value and all the other instances will then see the new value.

With methods too, a distinction is made between **static methods** and **instance methods**. The main difference is that an instance method can only be called if a class instance was generated beforehand. By contrast, static methods can be called without previous instantiation of the class.



Hint: This course will focus predominantly on static methods. The course 401 - ABAP Objects offers a comprehensive and systematic introduction to working with instance methods.

Using Methods of Global Classes

As is the case with other Repository objects, the display and processing of global classes is incorporated in the *Object Navigator*. As usual, you have several options for opening a global class: In the navigation area, for example, choose *Class/Interface* and enter the name of the class in the input field directly under the name. Of course, you can also display the object list for a package first and then double-click on the name of the required class in the subnode *Class Library* → *Classes*.

Attributes of Global Classes

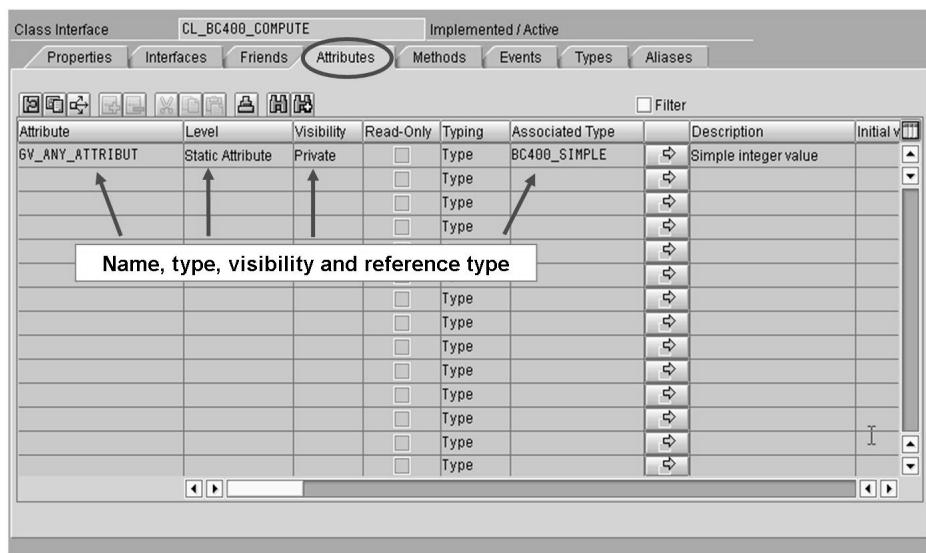


Figure 101: Definition of Attributes

Choose the *Attributes* tab to open the list of the attribute definitions in the class. If you only want to use the class, then only those attributes that are identified as “public” are of interest. You can address these directly outside the class.

Methods and Their Signature

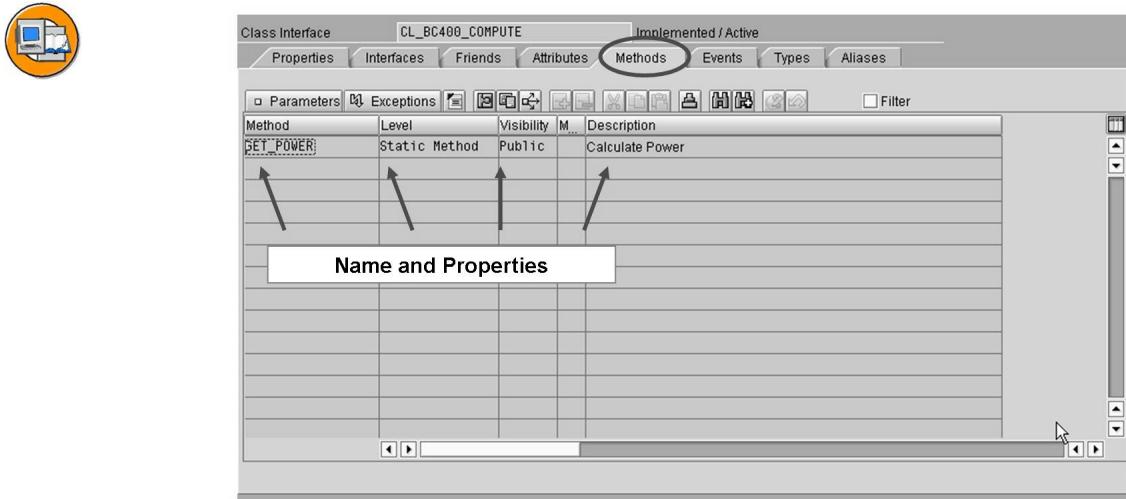


Figure 102: Definition of Methods

Choose the *Methods* tab to open the list of all method definitions in the class. With methods too, only entries that are identified as “public” are of interest to the user. Only these methods can be called externally. All other methods are used for modularization within the global class.

If a method is identified as a “static method”, it can be called directly without the need to generate an instance of the class first (see later). With “instance methods”, on the other hand, you first have to generate an instance and then call the method specifically for this instance. Under normal circumstances, the instance method then accesses instance attributes that contain different values for the various instances.

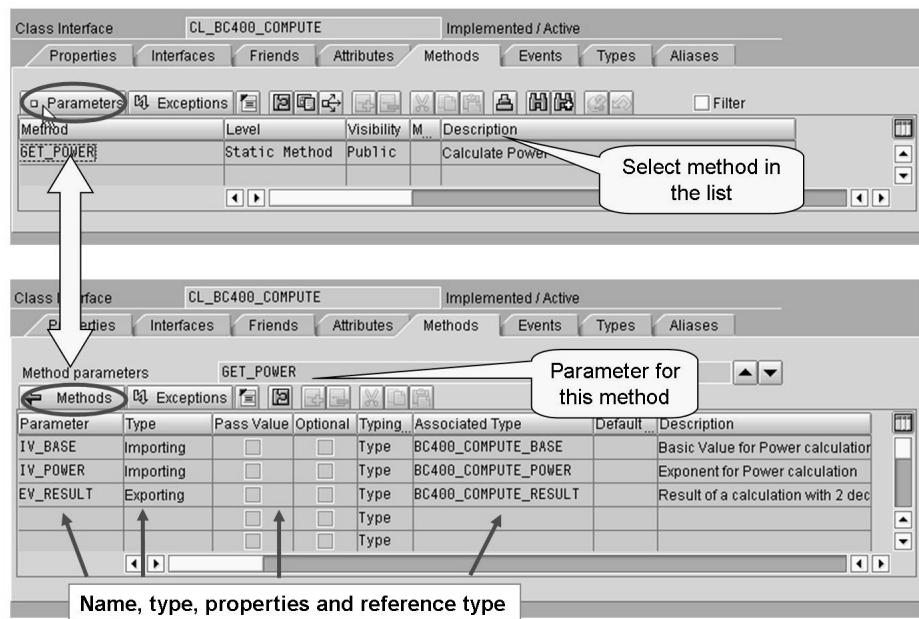


Figure 103: Method Parameters

There is a separate window for displaying the parameters for a method. You access it from the method list by placing the cursor on the required method and clicking on the *Parameter* pushbutton. In contrast to function modules there is no separate list for export, import, and changing parameters. The parameter type is entered in the *Type* column instead.

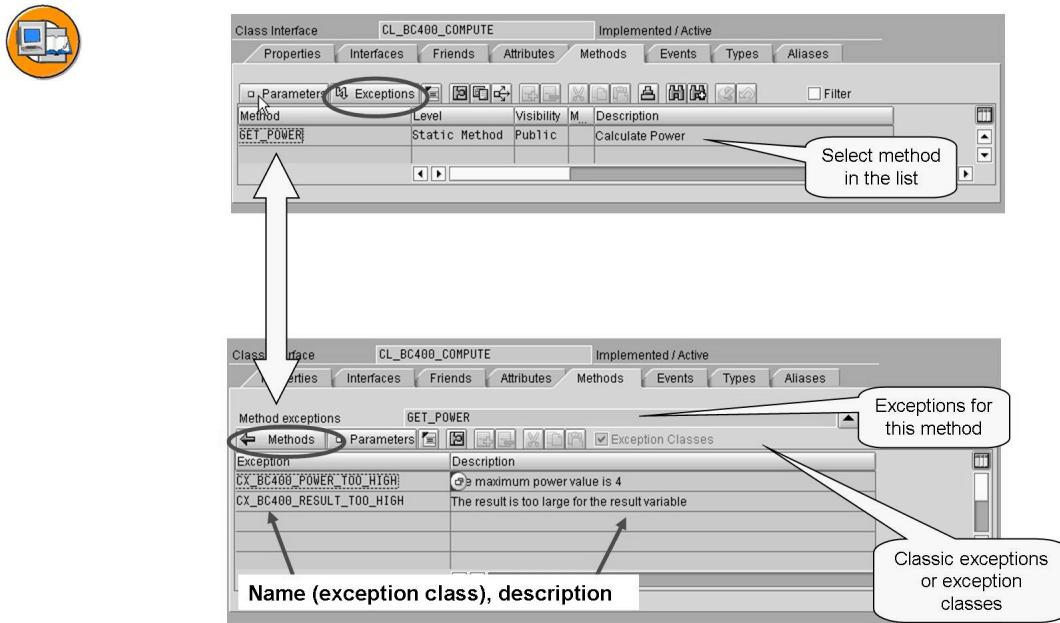


Figure 104: Method Exceptions

You also have to navigate to a separate display window to display the exceptions for a method. You access it from the method list by placing the cursor on the required method and clicking on the *Exceptions* pushbutton.

With Basis release 6.10, a new exceptions concept was introduced in ABAP, the so-called **class-based exceptions**. Handling these exceptions is very different to handling the classic exceptions from function modules. A method can either raise only new class-based exceptions or only classic exceptions. It is not possible to mix both concepts. A checkbox shows you which exception concept is used by the respective method.

While the classic exception concept allows you to choose any exception identifier, with the new exception concept you have to specify the names of special classes, the so-called exception classes.



Hint: The class-based exception concept is also available for function modules.

Documentation and Testing

With the global class documentation, a distinction is made between the documentation of the class as a whole and the documentation of individual components.

You can access the class documentation by choosing the *Class Documentation* pushbutton. To consult the documentation for an individual method or attribute, navigate to the corresponding list, select the required component with the cursor and then choose the *Documentation* pushbutton.

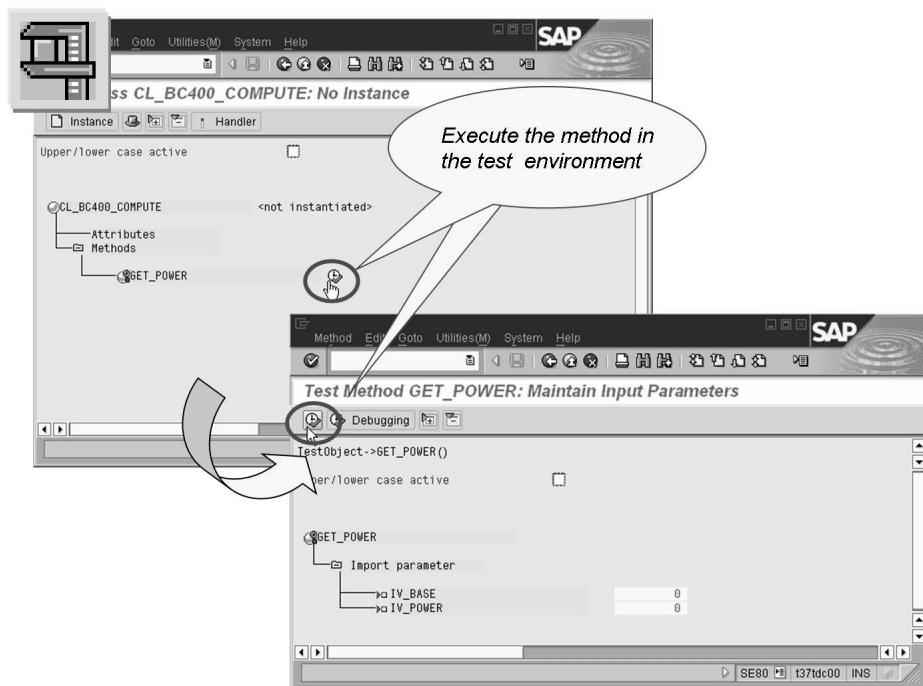


Figure 105: The Class Builder Testing Environment

You can test active global classes:

Temporary storage is allocated for the components of the class. For static components, this is allocated immediately, whereas for instance components it is allocated after you have chosen the *Instance* button.

The system only lists the public components. Methods can be tested using the *Execute Method* icon.

If you want to test a static method you do not need to generate an instance. It should be possible to execute the static method immediately.

If importing parameters exist, these will appear on the screen after you have chosen *Execute*.

After you have supplied the parameters with values you can test the method, and the result of the exporting parameter will be displayed.

Calling Static Methods

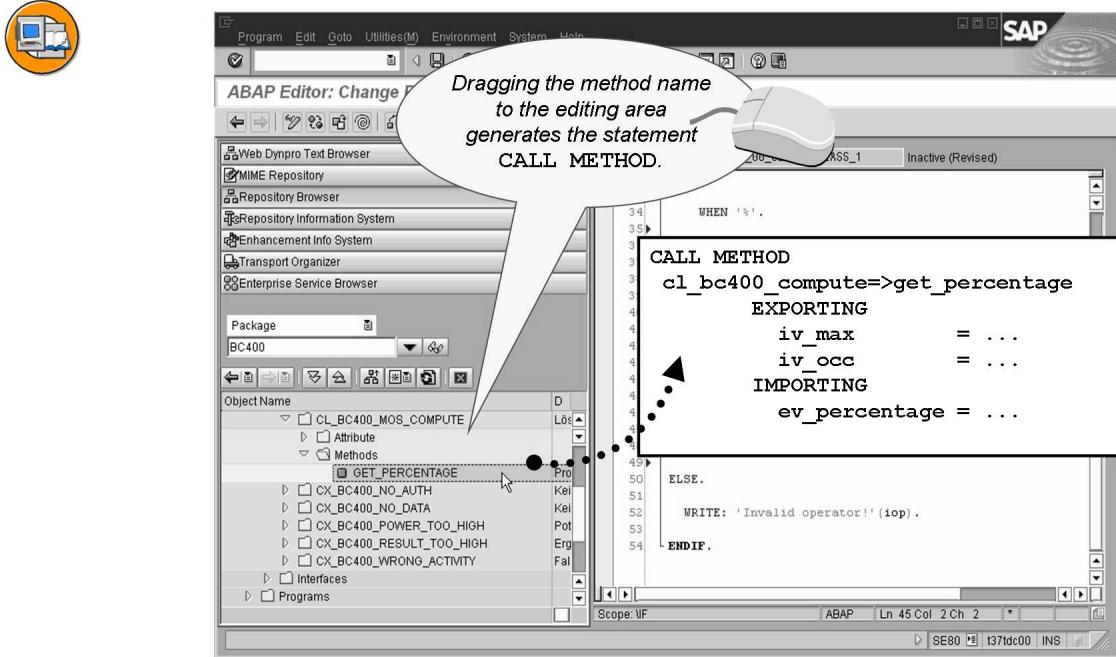


Figure 106: Method Calls Using Drag and Drop

Methods are called using the `CALL METHOD` statement. After this, you specify the method. You have to distinguish here between an instance method or a static method. With static methods, this specification comprises the name of the class and the name of the method, separated by the so-called **static component selector** “`=>`” (double-headed arrow).

The parameters are then passed in a very similar way to the function module call in both an **EXPORTING block** and an **IMPORTING block**.

As with function modules, you have several options for generating the method call. We recommend you use these options in order to avoid typing errors.

In the navigation area, select a **method name** and drag it to the editing area by holding down the left mouse button. In the statement that is generated this way, you only need to add the actual parameter.

Alternatively, you can also press the *Pattern* button. Under *ABAP Objects Pattern* you will then find the *Call Method* option. Here, you enter the class under *Class/Interface* and the name of the method under *Method*. With static methods the *Instance* field does not need to be filled.

Exception Handling

Methods can raise either classic exceptions or the more modern class-based exceptions. Classic exceptions are handled in the same manner that you encountered with function modules.



Handling Classic Exceptions

```

CALL METHOD cl_bc400_compute=>get_power
  EXPORTING
    iv_base    = pa_int1
    iv_power   = pa_int2
  IMPORTING
    ev_result = gv_result
  EXCEPTIONS
    POWER_VALUE_TOO_HIGH  = 1
    RESULT_VALUE_TOO_HIGH = 2.

CASE sy-subrc.
  WHEN 0.
    WRITE gv_result.
  WHEN 1.
    WRITE 'Max Value for Power is 4'.
  WHEN 2.
    WRITE 'Result value was too high'.
ENDCASE.

```

In the EXCEPTIONS block of the method call, a return code is assigned to the exception. When the method terminates with this classic exception, this return code is placed in the system field sy-subrc. By querying sy-subrc, the calling program can react to the exception.

By comparison, handling class-based exceptions is much more complex:



Handling Class-Based Exceptions

```

TRY.
  CALL MEHOD cl_bc400_compute=>get_power
    EXPORTING
      iv_base    = pa_int1
      iv_power   = pa_int2

```

```

IMPORTING
    ev_result = gv_result.
WRITE gv_result.
CATCH cx_bc400_power_too_high .
    WRITE 'Max Value for Power is 4'.
CATCH cx_bc400_result_too_high .
    WRITE 'Result value was too high'.
ENDTRY.

```

The call must be bound between the TRY. and ENDTRY. statements (see above). The actual exception is then handled in a processing block that begins with the CATCH <exception class>. statement, where <exception class> stands for the exception class that is to be handled. If the corresponding exception is raised somewhere within the TRY-ENDTRY block, processing is terminated and the system branches directly to the relevant CATCH block.

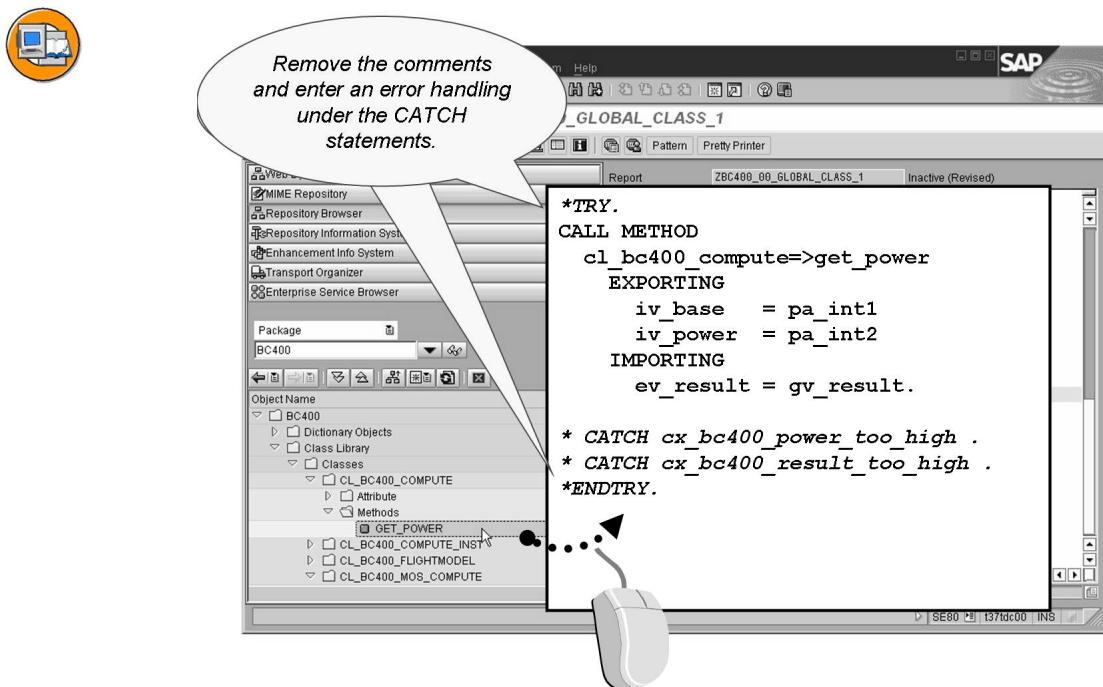


Figure 107: Handling Exceptions with Exception Classes

If you generate the method calls by drag and drop or by using *Pattern*, the statements for handling the class-based exceptions are generated as well. You then need only turn these statements into a comment and implement the CATCH blocks.

 **Note:** Class-based exceptions offer a variety of other options and are therefore much more powerful and versatile than classic exceptions. In this course, we limit ourselves to “minimal handling” of class-based exceptions. Other aspects are dealt with in the course BC401 - ABAP Objects, for example.

Generating Instances and Calling Instance Methods

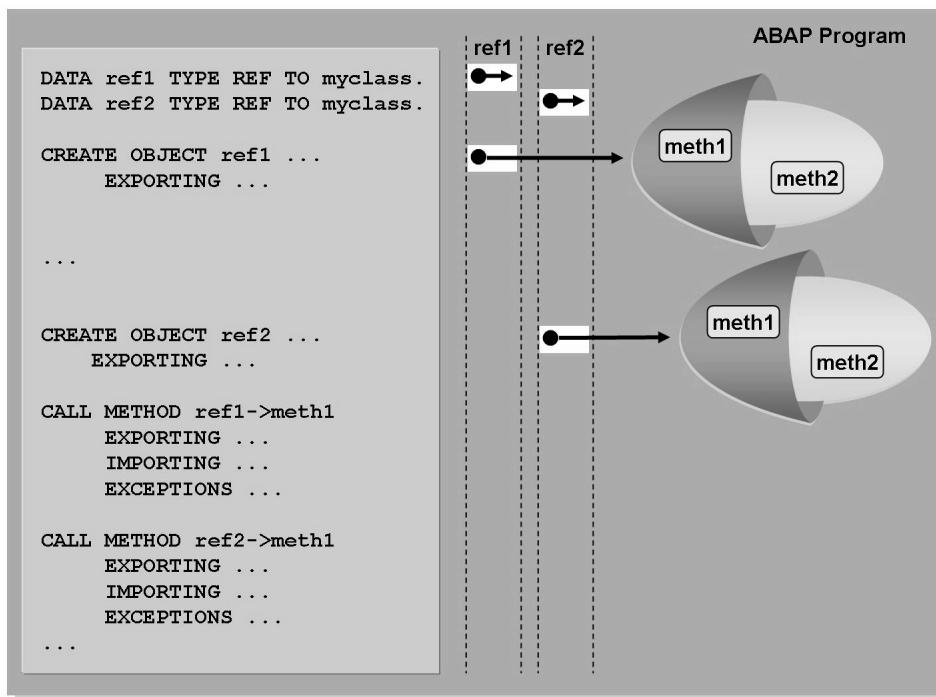


Figure 108: Creating Objects and Calling Methods

As instances do not have names, you have to define **reference variables** in order to be able to generate and address instances of classes. They are pointers that can be directed to corresponding instances. Reference variables each have a name that can be used to address the corresponding instance.

Reference variables are defined using:

```
DATA reference_name TYPE REF TO class_name.
```

When the program is started, a reference variable still has its initial value (“*does not point to an instance*”). Once it has been used to create an instance, it no longer has the initial value and points to that instance.

You can use the

CREATE OBJECT `reference_name`.

statement to generate an instance of the class that was specified in the definition of the reference variables. Afterwards, the reference variable points to the newly created instance.

When you use **CREATE OBJECT**, you might have to supply the import parameters of the special method **CONSTRUCTOR** with data. This special method is automatically executed directly after the creation of the instance. With its import parameters, it maintains the corresponding attributes of the new instance.

You call methods of an instance using the statement

CALL METHOD `reference_name->method_name`.

In contrast to calling a function module, the method name alone does not suffice here. You have to specify the relevant instance as well, as it is possible that the program has several instances of that class.

Creating Global Classes and Static Methods

In this section, based on a simple example, you will learn how to create a global class with a method in order to encapsulate a function for reuse. We will limit ourselves to creating a static method here.

Creating Global Classes

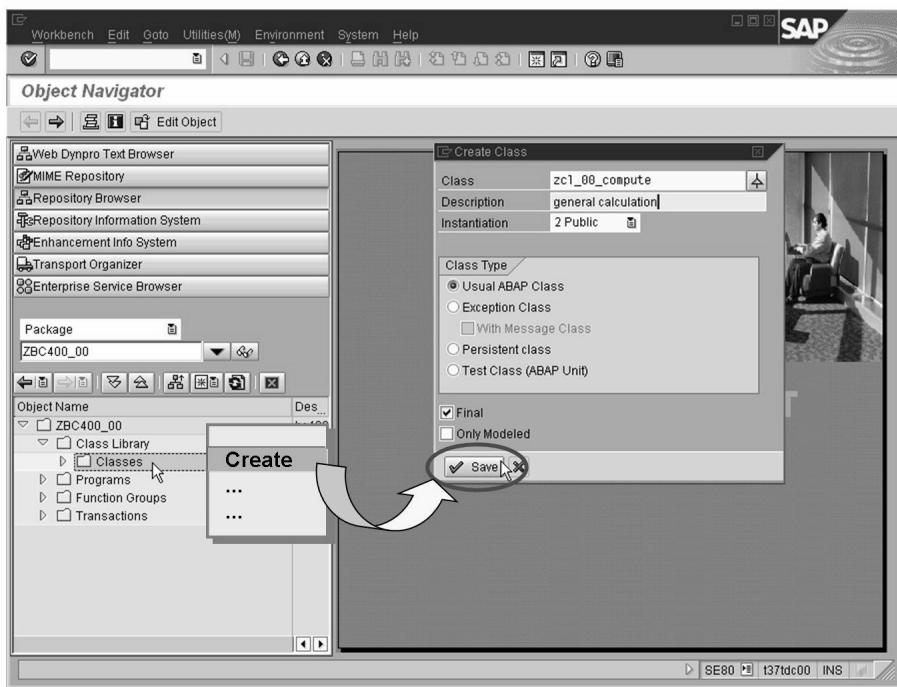


Figure 109: Creating Global Classes in the Object Navigator

To create a global class, open the context menu for your package in the navigation area and choose *Create* → *Class Library* → *Class*. In the dialog box that appears, enter the name of the class, a short description and several other properties. After you have saved your entries, assign it to a correction in the usual way.

Alternatively, in the navigation area you can choose *Class / Interface* from the dropdown list, enter the name of the new class in the field below it, and choose *Display*. Confirm that you want to create this new class.

Creating a Static Method

To create the static method, all you need to do is enter its name in the method list. You can maintain the visibility and type of method (static or instance) using the corresponding input help.

To create a parameter for a method, branch in the usual way to the parameter list for the corresponding method. Switch to change mode and enter the name of the parameter in the list. Use the input help to maintain the parameter type (import, export, ...) and specify an associated type.

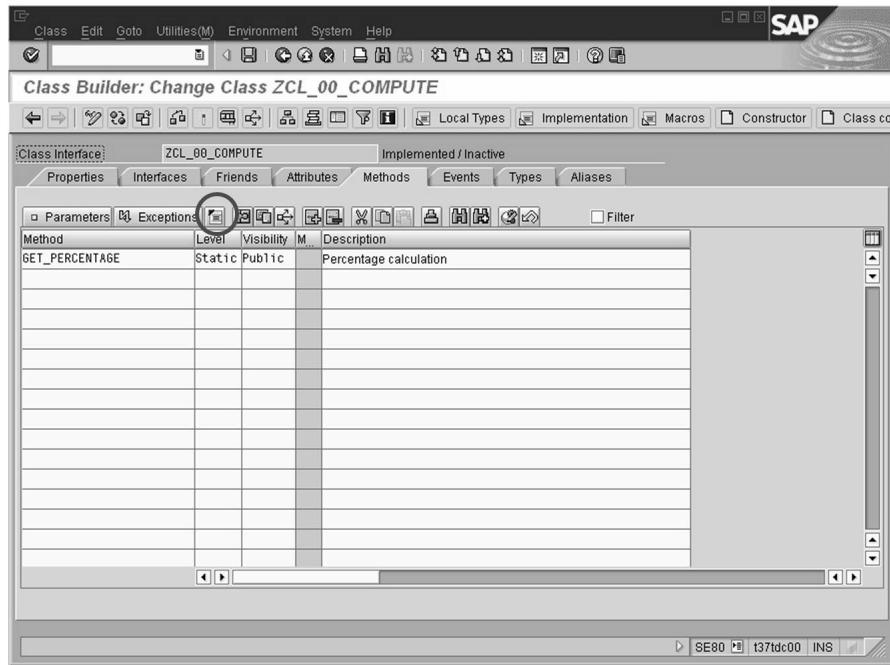


Figure 110: Calling the Source Code Editor

To implement the source code for a method, select the method in the method list and choose the *Source Code* pushbutton. Implement the source code in the usual way between the METHOD `<method_name>.` and ENDMETHOD. statements.

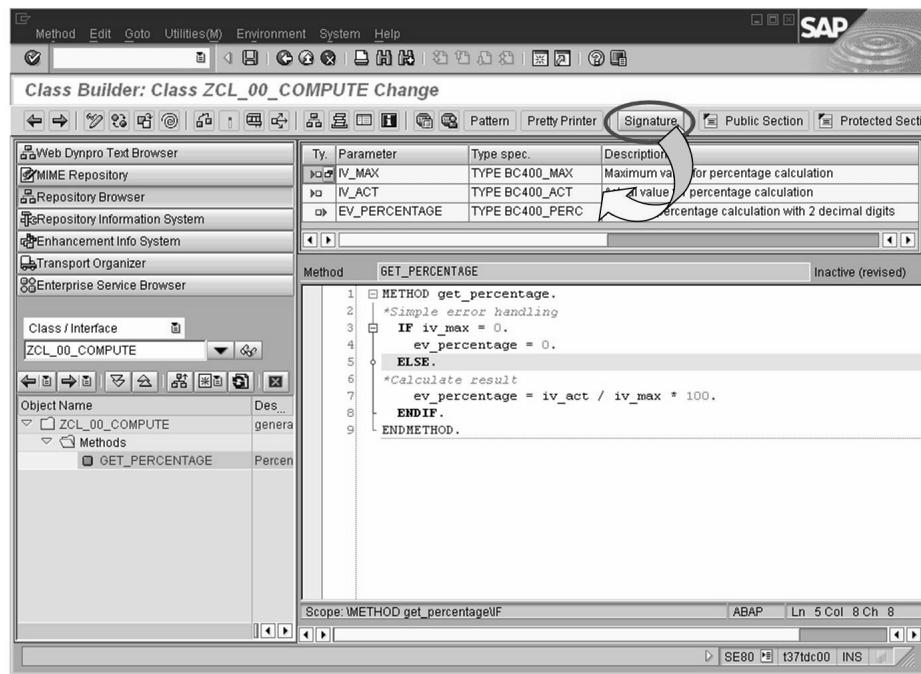


Figure 111: Showing the Signature in the Source Code Editor

While you are editing the source code, you can display the signature for the method. To do this, choose the *Signature* pushbutton (see graphic).

Exercise 10: Using Global Static Methods

Exercise Objectives

After completing this exercise, you will be able to:

- Call a static method of a global class

Business Example

You want to use a static method of a global class to calculate powers. This is meant to replace the function module call from a previous exercise.

Template:

BC400_MOS_FUNCTION_MODULE_2 (program)

Solution:

BC400_MOS_GLOBAL_CLASS_1 (program)

Task 1:

Open the global class CL_BC400_COMPUTE and familiarize yourself with the interface and the scope of functions of the method(s) it contains.

1. Analyze the interface of the method get_power and read the documentation about the class.
2. Test the static method get_power.

Task 2:

Copy your program ZBC400_##_FUNCTION_MODULE_2 or the copy template to the new name **ZBC400_##_GLOBAL_CLASS_1** and replace the function module call for the power calculation with a call for the method get_power.

1. Copy your program or the template.
2. Remove the call of the function module BC400_MOS_POWER and implement a call of the method cl_bc400_compute=>get_power in the same place.



Hint: Generate the method call by using the drag and drop function in the navigation area or the *Pattern* pushbutton.

3. Handle the exceptions and output a corresponding text if an error occurs.

Solution 10: Using Global Static Methods

Task 1:

Open the global class CL_BC400_COMPUTE and familiarize yourself with the interface and the scope of functions of the method(s) it contains.

1. Analyze the interface of the method get_power and read the documentation about the class.
 - a) Perform this step as described in the course materials.
2. Test the static method get_power.
 - a) Perform this step as described in the course materials.

Task 2:

Copy your program ZBC400_##_FUNCTION_MODULE_2 or the copy template to the new name **ZBC400_##_GLOBAL_CLASS_1** and replace the function module call for the power calculation with a call for the method get_power.

1. Copy your program or the template.
 - a) Carry out this step in the usual way.
2. Remove the call of the function module BC400_MOS_POWER and implement a call of the method cl_bc400_compute=>get_power in the same place.



Hint: Generate the method call by using the drag and drop function in the navigation area or the *Pattern* pushbutton.

- a) See source code extract from the model solution.
3. Handle the exceptions and output a corresponding text if an error occurs.
 - a) See source code extract from the model solution.

Result

Source code extract:

```
*-----*  
*& Report  BC400_MOS_GLOBAL_CLASS_1          *  
*-----*  
REPORT  bc400_mos_global_class_1 .
```

Continued on next page

```

PARAMETERS:
  pa_int1  TYPE i,
  pa_op    TYPE c LENGTH 1,
  pa_int2  TYPE i.

DATA gv_result  TYPE p LENGTH 16 DECIMALS 2.

IF ( pa_op = '+' OR
     pa_op = '-' OR
     pa_op = '*' OR
     pa_op = '/' AND pa_int2 <> 0 OR
     pa_op = '%' OR
     pa_op = 'P' ).

CASE pa_op.
  WHEN '+'.
    gv_result = pa_int1 + pa_int2.
  WHEN '-'.
    gv_result = pa_int1 - pa_int2.
  WHEN '*'.
    gv_result = pa_int1 * pa_int2.
  WHEN '/'.
    gv_result = pa_int1 / pa_int2.
  WHEN 'P'.

TRY.
  CALL METHOD cl_bc400_compute=>get_power
    EXPORTING
      iv_base    = pa_int1
      iv_power   = pa_int2
    IMPORTING
      ev_result = gv_result.
  CATCH cx_bc400_power_too_high .
    WRITE 'Max value of Power is 4'(mvp).
  CATCH cx_bc400_result_too_high .
    WRITE 'Result value was too high'(rvh).
ENDTRY.

WHEN '%'.
  CALL FUNCTION 'BC400_MOS_PERCENTAGE'
    EXPORTING

```

Continued on next page

```
        iv_act          = pa_int1
        iv_max          = pa_int2
IMPORTING
        ev_percentage   = gv_result
EXCEPTIONS
        division_by_zero = 1
        OTHERS           = 2.

IF sy-subrc <> 0.
    WRITE 'Error in Function Module' (efm).
ENDIF.
ENDCASE.

WRITE: 'Result:' (res), gv_result.

ELSEIF pa_op = '/' AND pa_int2 = 0.
    WRITE: 'No division by zero!' (dbz).

ELSE.
    WRITE: 'Invalid operator!' (iop).

ENDIF.
```

Exercise 11: Creating Global Classes

Exercise Objectives

After completing this exercise, you will be able to:

- Create a global class

Business Example

Your company wants to develop an application in which several functions are used in different programs.

You are in charge of implementing calculation functions. You decide to program it as a **global class** with **static methods**. You will implement this project successively in this exercise and the subsequent exercise in this unit.

Template:

None

Solution:

CL_BC400_MOS_COMPUTE (global class)

Task:

Create a global class.

1. Create the global class **ZCL_##_COMPUTE** and activate it.

Solution 11: Creating Global Classes

Task:

Create a global class.

1. Create the global class **ZCL_##_COMPUTE** and activate it.
 - a) Open the *Object Navigator* (transaction SE80).
 - b) In the navigation area, choose *Class / Interface* from the dropdown list and enter the name of the new class in the field below it.
 - c) Choose  *Display*.
 - d) In the modal dialog box that appears, confirm that you want to create a new class.
 - e) Then enter a short text to explain the function of this class in the *Description* field.
 - f) Make no other entries and confirm your description by choosing  *Save*.
 - g) In the dialog box that appears, assign your class to your package and the Workbench request allocated to you by your instructor in the usual way.
 - h) Activate the global class using the  pushbutton.

Exercise 12: Creating and Using Global Static Methods

Exercise Objectives

After completing this exercise, you will be able to:

- Create a global static method

Business Example

You want to use global static methods to calculate powers and percentages. These are meant to replace the function module calls from a previous exercise. A new percentage calculation method is to be created for this.

Template:

BC400_MOS_GLOBAL_CLASS_1 (program)

Solution:

BC400_MOS_GLOBAL_CLASS_2 (program)

CL_BC400_MOS_COMPUTE (global class)

Task 1:

Create a static method for calculating powers in the global class you defined in the previous exercise. Use the function module that has served as the copy template up to now for orientation.

1. Create the static method **GET_PERCENTAGE** in your class ZCL_##_COMPUTE.
2. Create parameters for the method. Two parameters are to be passed to the method for the calculation and one parameter returned as the result. Use the function module that is to be replaced for orientation.
3. Implement the source code for the static method GET_PERCENTAGE. Use the function module that is to be replaced for orientation here too.

Continued on next page

Task 2:

Copy your program ZBC400_##_GLOBAL_CLASS_1 or the copy template to the new name **ZBC400_##_GLOBAL_CLASS_2** and replace the function module call for the percentage calculation with a call for your new method.

1. Copy the program.
2. Remove the call of the function module and implement a call of the new method in the same place.



Hint: Generate the method call by using the drag and drop function in the navigation area or the *Pattern* pushbutton.

Solution 12: Creating and Using Global Static Methods

Task 1:

Create a static method for calculating powers in the global class you defined in the previous exercise. Use the function module that has served as the copy template up to now for orientation.

1. Create the static method **GET_PERCENTAGE** in your class **ZCL_##_COMPUTE**.
 - a) Display the global class in change mode.
 - b) Switch to the *Methods* tab page and enter the name of the new method in the *Methods* column.
 - c) In the *Type* column, select **Static Method** from the input help.
 - d) In the *Visibility* column, select **Public** from the input help.
 - e) Enter an explanatory text in the *Description* column.
2. Create parameters for the method. Two parameters are to be passed to the method for the calculation and one parameter returned as the result. Use the function module that is to be replaced for orientation.
 - a) Select the method **GET_PERCENTAGE** with one click of the mouse.
 - b) Choose *Parameters*.
 - c) Define the following parameters:

Parameter Name	Parameter Type	Data Element
iv_act	Import	BC400_ACT
iv_max	Import	BC400_MAX
ev_percentage	Export	BC400_PERC

Continued on next page

3. Implement the source code for the static method GET_PERCENTAGE. Use the function module that is to be replaced for orientation here too.
 - a) Switch to the method list and select the method by clicking on it.
 - b) Choose *Source Code*.
 - c) Implement the function as specified in the model solution for the method GET_PERCENTAGE.

Task 2:

Copy your program ZBC400 ##_GLOBAL_CLASS_1 or the copy template to the new name **ZBC400 ##_GLOBAL_CLASS_2** and replace the function module call for the percentage calculation with a call for your new method.

1. Copy the program.
 - a) Carry out this step in the usual manner.
2. Remove the call of the function module and implement a call of the new method in the same place.



Hint: Generate the method call by using the drag and drop function in the navigation area or the *Pattern* pushbutton.

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

Result

Source code extract:

```

REPORT BC400_MOS_GLOBAL_CLASS_2.

PARAMETERS:
  pa_int1  TYPE i,
  pa_op    TYPE c LENGTH 1,
  pa_int2  TYPE i.

DATA gv_result TYPE p LENGTH 16 DECIMALS 2.

IF ( pa_op = '+' OR
     pa_op = '-' OR
     pa_op = '*' OR
     pa_op = '/' AND pa_int2 <> 0 OR

```

Continued on next page

```

pa_op = '%' OR
pa_op = 'P' .

CASE pa_op.
WHEN '+'.
    gv_result = pa_int1 + pa_int2.
WHEN '-'.
    gv_result = pa_int1 - pa_int2.
WHEN '*'.
    gv_result = pa_int1 * pa_int2.
WHEN '/'.
    gv_result = pa_int1 / pa_int2.
WHEN 'P'.

TRY.
    CALL METHOD cl_bc400_compute=>get_power
        EXPORTING
            iv_base    = pa_int1
            iv_power   = pa_int2
        IMPORTING
            ev_result = gv_result.
    CATCH cx_bc400_power_too_high .
        WRITE 'Max value of Power is 4'(mvp).
    CATCH cx_bc400_result_too_high .
        WRITE 'Result value was too high'(rvh).
ENDTRY.

WHEN '%'.

    CALL METHOD cl_bc400_mos_compute=>get_percentage
        EXPORTING
            iv_act      = pa_int1
            iv_max     = pa_int2
        IMPORTING
            ev_percentage = gv_result.

ENDCASE.

WRITE: 'Result:'(res), gv_result.

ELSEIF pa_op = '/' AND pa_int2 = 0.
    WRITE: 'No division by zero!'(dbz).

```

Continued on next page

```
ELSE.  
    WRITE: 'Invalid operator!' (iop).  
ENDIF.
```

Source code extract for the method:

```
METHOD GET_PERCENTAGE.  
*Simple error handling  
    IF iv_max = 0.  
        ev_percentage = 0.  
    ELSE.  
        *Calculate result  
        ev_percentage = iv_act / iv_max * 100.  
    ENDIF.  
ENDMETHOD.
```



Lesson Summary

You should now be able to:

- Explain the basic terms of object-oriented programming
- Acquire information about the function and use of global classes and their methods
- Call methods of global classes in your programs
- Create global classes
- Create and implement simple methods in global classes

Lesson: Modularization with Methods of Local Classes (Preview)

Lesson Overview

In this lesson you will be given a brief preview of how local classes enable you to use object-oriented programming techniques for internal program modularization.



Lesson Objectives

After completing this lesson, you will be able to:

- Describe how local classes are defined, implemented and used

Business Example

You want to encapsulate a function in a static method of a local class.

Defining and Using Local Classes

In the previous lessons, you learned how to define and use local classes. This lesson now demonstrates the definition and usage of local classes. Local classes are named as such because they can only be used locally within the program in which they were defined.



Hint: The subject is dealt with very superficially here, since it is of no relevance for the rest of the course. For a detailed discussion of the definition of local classes, see the course “BC401 – ABAP Objects”.

The main difference between global classes and local classes lies in the way in which they are defined: Whilst global classes are maintained with a special tool (*Class Builder*), local classes are created directly in the source code of the respective main program.

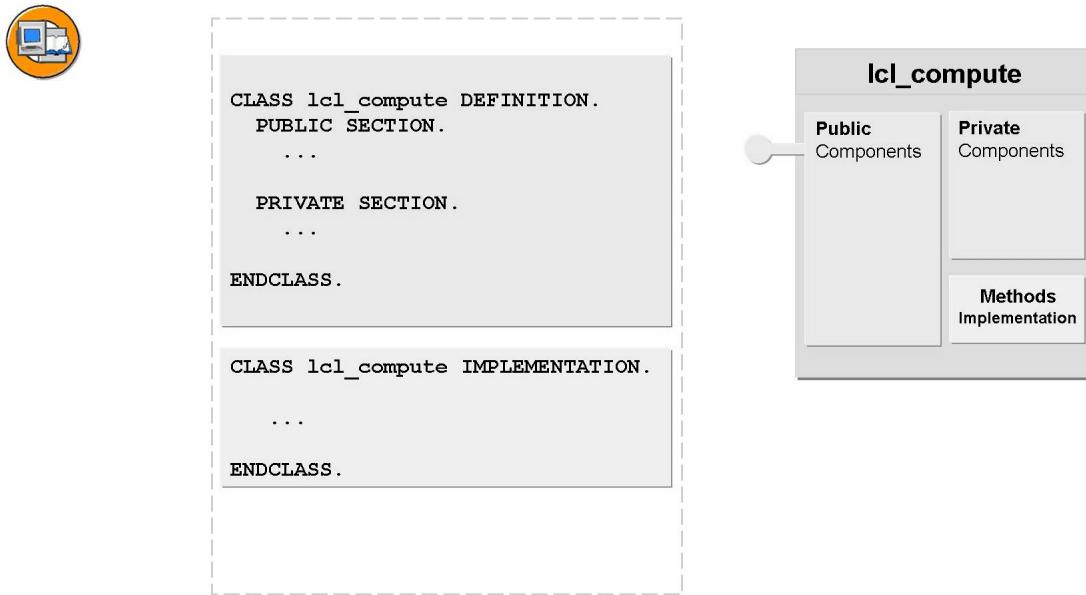


Figure 112: Defining a Local Class

The graphic shows how you create a local class in the source code of the program. A distinction is made between two blocks: The **definition** and the **implementation** of the class. The description of the attributes and the signatures of the methods are located in the definition block, whereas the implementation block merely contains the source code of the methods.

The **CLASS** and **ENDCLASS** statements are statements declared locally in the program. Just as the **TYPES** statement defines local data types, the **CLASS ... ENDCLASS** statement defines local object types.

The definition definition block is divided into several sections in which the public and private components are defined (**PUBLIC SECTION.** and **PRIVATE SECTION.**).

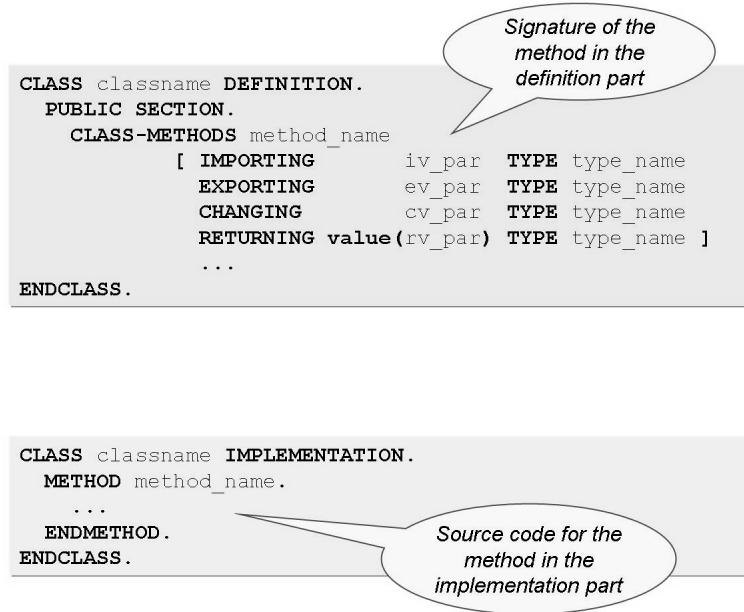


Figure 113: Syntax for Static Methods

The graphic provides a schematic representation of the definition and implementation of a public static method. The method is public because the definition is located in the PUBLIC SECTION. section of the class. To define an instance method as opposed to the static method, the METHODS statement would have to be used instead of CLASS-METHODS.

Methods have a **signature** (interface parameters and exceptions) that enables them to receive values when they are called and pass values back to the calling program. Methods can have any number of IMPORTING, EXPORTING, and CHANGING parameters. All parameters can be passed as values or as references.

 **Note:** One method return value can be defined using the RETURNING parameter. It must always be passed as a value. In this case, you cannot define any EXPORTING and CHANGING parameters. The RETURNING parameter can therefore be used to define **functional methods**. This option is explained in more detail in the course “BC401 – ABAP Objects”.

All input parameters (IMPORTING and CHANGING parameters) can be defined as optional parameters in the declaration using the OPTIONAL or DEFAULT additions. These parameters then do not necessarily have to be passed when the object is called. The DEFAULT addition enables you to specify a start value.

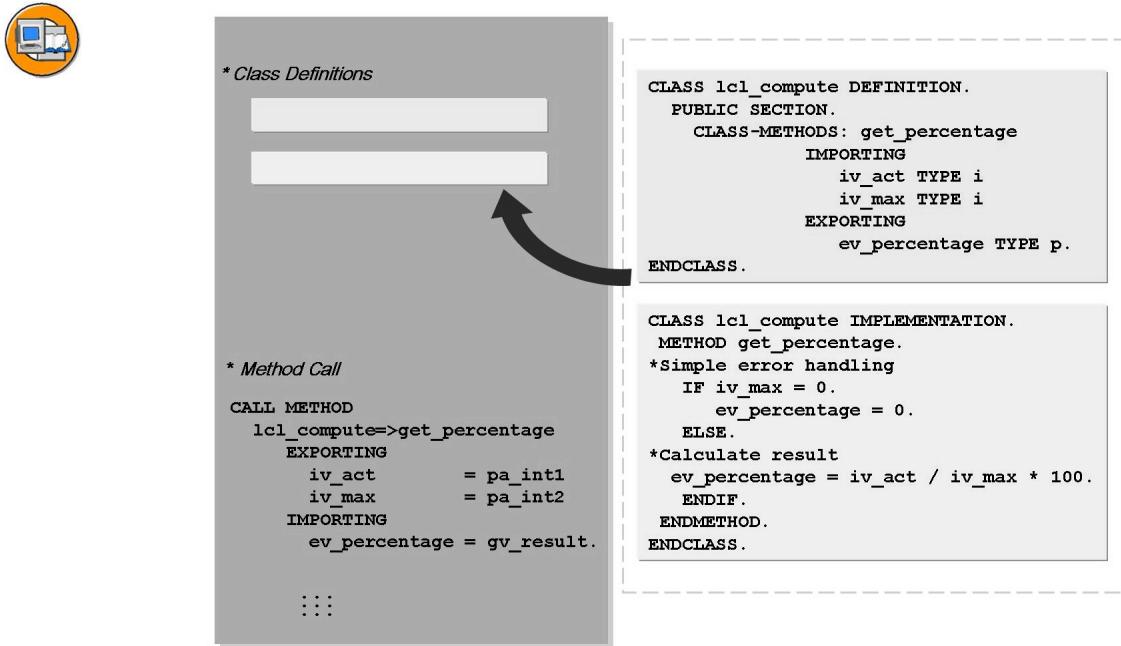


Figure 114: Implementing and Using a Static Method

The syntax example shows how a local class is defined with a static method (top right section) and how the method is implemented (bottom right section).

In the section on the left you can see the static method call from the main program. As you can see, the call is identical to a static method call in a global class.

You can define the local class either directly in the definition block of the main program or in an include.



Hint: You have to define the local class in the sequence before the call. The local class behaves in a different way to a subroutine here. The subroutine can also be defined after the call.



Lesson Summary

You should now be able to:

- Describe how local classes are defined, implemented and used



Unit Summary

You should now be able to:

- Name the basic modularization techniques
- Define subroutines
- Call subroutines
- Analyze the execution of subroutines in debugging mode
- Search for function modules
- Acquire information on the functionality and use of function modules
- Call function modules in your program
- Create a function group
- Create a function module
- Explain the role of BAPIs and identify their special properties
- Explain the basic terms of object-oriented programming
- Acquire information about the function and use of global classes and their methods
- Call methods of global classes in your programs
- Create global classes
- Create and implement simple methods in global classes
- Describe how local classes are defined, implemented and used

Related Information

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



Test Your Knowledge

1. For what should you use the subroutine technique?

Choose the correct answer(s).

- A For performance improvement
- B For a better overview of program layout
- C To encapsulate a function that is required many times within a program for multiple use
- D To implement the central maintainability of a function within a program
- E To make a function available across the system

2. Why should you pass parameters to subroutines?

Choose the correct answer(s).

- A Mainly to transfer user inputs to the subroutine
- B For performance improvement
- C For achieving more flexibility in the subroutine
- D For achieving more stability in the subroutine

3. Which ABAP statement is used to define or call a subroutine?

4. Which of the following statements concerning subroutines are correct?

Choose the correct answer(s).

- A When a subroutine is called, variables or literals are to be passed as actual parameters to the corresponding formal parameters.
- B When a subroutine is called, it is important to watch out for the same names of formal parameters and actual parameters.
- C The passing method for a formal parameter is defined in the subroutine definition.
- D There are two different passing methods for formal parameters.
- E USING parameters without the VALUE addition are call-by-reference parameters.
- F Large internal tables should be passed to the subroutine through a call-by-value parameter.

5. What is a function module?

6. Which statement is used to call a function module?

7. Which of the following statements concerning function modules are correct?

Choose the correct answer(s).

- A When you call a function module, all the import, export, and CHANGING parameters have to be filled with data.
- B Whenever a function module is called, all exceptions should be caught and handled. Otherwise, a runtime error occurs whenever an exception that is not caught is raised by the function module.
- C When a function module is called, the IMPORT parameters are listed with the IMPORTING addition and the EXPORT parameters are listed with the EXPORTING addition.
- D When a function module is called, the IMPORT parameters are listed with the EXPORTING addition and the EXPORT parameters are listed with the IMPORTING addition.



Answers

1. For what should you use the subroutine technique?

Answer: B, C, D

A function provided for system-wide use should not be implemented in the form of a local program subroutine, but should be in the form of a function module, since a search function is available for the latter.

2. Why should you pass parameters to subroutines?

Answer: C

3. Which ABAP statement is used to define or call a subroutine?

Answer: Definition: FORM . . . ENDFORM.

Call: PERFORM

4. Which of the following statements concerning subroutines are correct?

Answer: A, C, E

5. What is a function module?

Answer: A function that is stored centrally in the SAP system and can be used by all the programs in the system.

6. Which statement is used to call a function module?

Answer: CALL FUNCTION

7. Which of the following statements concerning function modules are correct?

Answer: B, D

Unit 5

Complex Data Objects

Unit Overview

The unit overview lists the individual lessons that make up this unit.



Unit Objectives

After completing this unit, you will be able to:

- Define structured data objects (structure variables)
- Use basic ABAP statements for 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 Structures	220
Exercise 13: Working with Structures.....	225
Lesson: Working with Internal Tables	232
Exercise 14: Working with Internal Tables	251

Lesson: Working with Structures

Lesson Overview

In this lesson, we will continue with the definition of structured data objects (structure variables) and their analysis using the *ABAP Debugger*. You will also learn how to use basic ABAP statements for structured data objects.



Lesson Objectives

After completing this lesson, you will be able to:

- Define structured data objects (structure variables)
- Use basic ABAP statements for structured data objects
- Analyze structured data objects in debugging mode

Business Example

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

Working with Structures

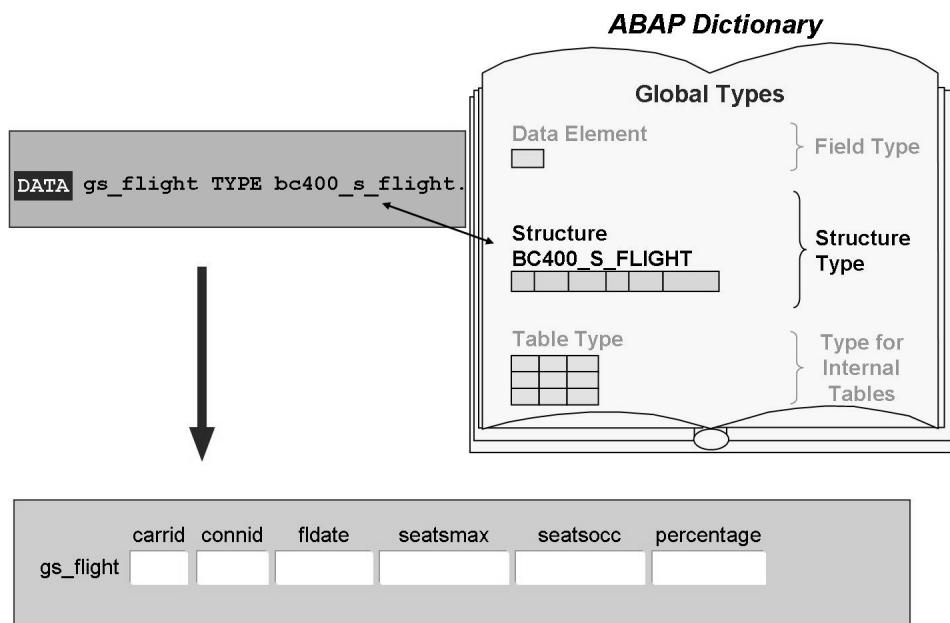


Figure 115: Definition of Structures with Global Types

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

Structure variables are defined in the program in the same way as elementary data objects with the DATA statement. When you set the types, you can refer to

- a structure in the *ABAP Dictionary* (global structure type) or
- a structure type that is declared locally in the program



Hint: In the context of data modeling and database access, you will be introduced at a later stage to other objects in the *ABAP Dictionary* that can be used like global structure types when typing structures. These objects are the “transparent table” and the “database view”.

The following graphic shows the definition of a structure variable using a locally declared structure type.



```
TYPES: BEGIN OF gty_s_flightinfo,
      carrid      TYPE s_carr_id,
      carrname    TYPE s_carrname,
      connid     TYPE s_conn_id,
      fldate      TYPE s_date,
      percentage  TYPE p LENGTH 3 DECIMALS 2,
   END OF gty_s_flightinfo.

DATA gs_flightinfo TYPE gty_s_flightinfo.
```

Declaration of a local structure type

Definition of a structure variable

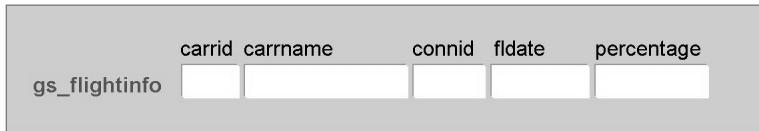


Figure 116: Defining Structures with Local Types

You can use the TYPES statement to define local structure types. Here the components are enclosed by

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

```
END OF structure_name.
```

You can assign any type you want to each component by using the TYPE addition. For more details, refer to the keyword documentation for the TYPES statement.

You define the data object itself in the usual way.

If necessary, you can also define a structured data object directly. To do so, all you have to do is replace the leading key word TYPES with DATA.

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



```
DATA gs_scarr TYPE scarr.
gs_scarr->carrid = 'LH'.
CALL METHOD cl_bc400_flightmodel=>get_carrier
      EXPORTING iv_carrid = gs_scarr->carrid
      IMPORTING es_carrier = gs_scarr.
WRITE: / gs_scarr->carrid,
        gs_scarr->carname,
        gs_scarr->url.
```

	mandt	carrid	carname	currcode	url
gs_scarr	400	LH	Lufthansa	EUR	http://www.lufthansa.com

Figure 117: Access to Structure Components

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



Hint: In principle, the ABAP syntax allows the names of data objects to contain hyphens (such as `DATA h-var TYPE c LENGTH 5.`). However, to avoid confusion with the addressing structure components, you should avoid using hyphens as part of the name.



```

DATA: gs_flight      TYPE bc400_s_flight,
      .
      gs_flightinfo  TYPE gty_s_flightinfo.
.
MOVE-CORRESPONDING      TO
MOVE-CORRESPONDING gs_flight TO gs_flightinfo.

```

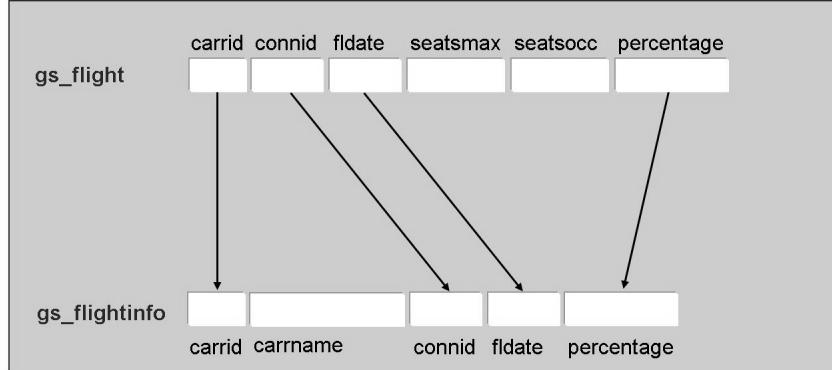


Figure 118: Copying Structure Components with the Same Name

The **MOVE - CORRESPONDING** statement copies the contents of the source structure to the target structure one component at a time. Here, only those components are considered that are available under the same name in both the source and the target structure. All other components of the structures remain unchanged.

The individual value assignments can each be executed using MOVE.

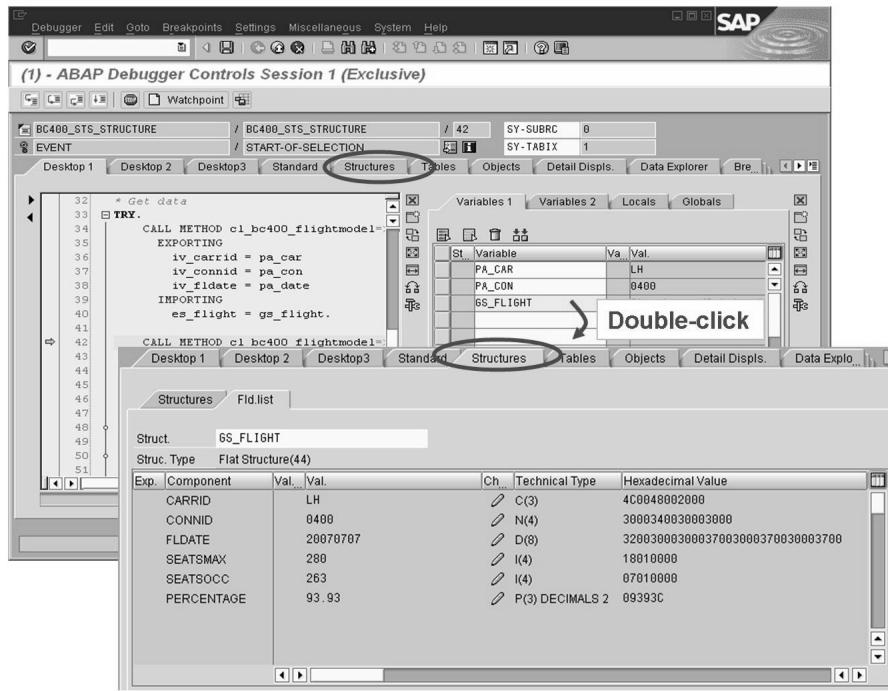


Figure 119: Structures in Debugging Mode

You can trace the field contents of a structure in the Debugger by first entering the structure name in the *Variable 1* area or copying it from the source code by double-clicking on it. You get the component display of the structure by double-clicking on the structure name in the *Variable* area. In addition, you can configure a display area for structure display on one of the desktops.

To display a structure variable in the **classic ABAP Debugger**, use the double-click method. By double-clicking on the structure in the source code, you copy this to the field view. Double-clicking on the field view takes you to the structure display, where you can read the name, content and type of the individual components.

Exercise 13: Working with Structures

Exercise Objectives

After completing this exercise, you will be able to:

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

Business Example

You want to manage a quantity of data from various structures in one large structure in the ABAP program.

Template:

None

Solution:

BC400_STS_STRUCTURE

Task 1:

Create a program and define a selection screen for entering an airline, a flight number and the flight date.

1. Create the executable program **ZBC400_##_STRUCTURE** without a TOP include. Assign the program to your package and your change request.
2. Define the input options for an airline (suggested name: **pa_car**), flight number (suggested name: **pa_con**) and flight date (suggested name: **pa_date**). When typing the input fields, refer to suitable fields of the *Dictionary* structure type BC400_S_FLIGHT.

Task 2:

Create two structures based on *Dictionary* types for data retrieval at a later stage and one for the structure intended for output that is defined by a local type.

1. Define two structures in the program with reference to the *Dictionary* structure type BC400_S_CARRIER (suggested name: **gs_carrier**) and BC400_S_FLIGHT (suggested name: **gs_flight**).

Continued on next page

2. Define a third structure that combines all the components of the two previous structures. The reference type should be implemented locally in the program (suggested name: `gtys_carrierflight`). Use the TYPES statement for the definition. Now use this local structure type to define a structure variable (suggested name: `gs_carrierflight`).



Hint: Alternatively, you can define the structure directly using DATA without using the TYPES statement.

Task 3:

Retrieve the flight data for the flight date that was entered, with information about the selected airline. Output this data in a simple list.

1. To retrieve the data, use the methods GET_CARRIER and GET_FLIGHT of the class CL_BC400_FLIGHTMODEL. Include the methods in your program by drag and drop and supply the signature of the methods with the correct type of data. The input parameters of the selection screen serve as import parameters and the first two structures serve as export parameters here.
2. Merge the received data from the first two structures into your output structure. To do this, use the statement MOVE-CORRESPONDING.
3. Use the WRITE statement to output the contents of the output structure in a simple list.
4. (OPTIONAL) Suppress the list output if no flight was found. Instead, output a short message in the error handling of the methods using the WRITE statement.



Hint: You can check with `IS INITIAL` to determine whether the output structure is filled with data.

Task 4:

Execute your program and check the result.

1. Check the filling of all the structures in the *ABAP Debugger*. Test the case when no flight data is found too.

Solution 13: Working with Structures

Task 1:

Create a program and define a selection screen for entering an airline, a flight number and the flight date.

1. Create the executable program **ZBC400_##_STRUCTURE** without a TOP include. Assign the program to your package and your change request.
 - a) Carry out this step in the usual way.
2. Define the input options for an airline (suggested name: **pa_car**), flight number (suggested name: **pa_con**) and flight date (suggested name: **pa_date**). When typing the input fields, refer to suitable fields of the *Dictionary* structure type **BC400_S_FLIGHT**.
 - a) See source code extract from the model solution.

Task 2:

Create two structures based on *Dictionary* types for data retrieval at a later stage and one for the structure intended for output that is defined by a local type.

1. Define two structures in the program with reference to the *Dictionary* structure type **BC400_S_CARRIER** (suggested name: **gs_carrier**) and **BC400_S_FLIGHT** (suggested name: **gs_flight**).
 - a) See source code extract from the model solution.
2. Define a third structure that combines all the components of the two previous structures. The reference type should be implemented locally in the program (suggested name: **gty_s_carrierflight**). Use the TYPES statement for the definition. Now use this local structure type to define a structure variable (suggested name: **gs_carrierflight**).



Hint: Alternatively, you can define the structure directly using DATA without using the TYPES statement.

- a) See source code extract from the model solution.

Continued on next page

Task 3:

Retrieve the flight data for the flight date that was entered, with information about the selected airline. Output this data in a simple list.

1. To retrieve the data, use the methods GET_CARRIER and GET_FLIGHT of the class CL_BC400_FLIGHTMODEL. Include the methods in your program by drag and drop and supply the signature of the methods with the correct type of data. The input parameters of the selection screen serve as import parameters and the first two structures serve as export parameters here.
 - a) See source code extract from the model solution.
2. Merge the received data from the first two structures into your output structure. To do this, use the statement MOVE - CORRESPONDING.
 - a) See source code extract from the model solution.
3. Use the WRITE statement to output the contents of the output structure in a simple list.
 - a) See source code extract from the model solution.
4. (OPTIONAL) Suppress the list output if no flight was found. Instead, output a short message in the error handling of the methods using the WRITE statement.



Hint: You can check with IS INITIAL to determine whether the output structure is filled with data.

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

Task 4:

Execute your program and check the result.

1. Check the filling of all the structures in the *ABAP Debugger*. Test the case when no flight data is found too.
 - a) Set a session breakpoint in the *ABAP Editor* and execute the program directly using the key.

Result

Source code extract: **BC400_STS_STRUCTURE**

```
REPORT  bc400_sts_structure.
```

Continued on next page

```

PARAMETERS: pa_car  TYPE bc400_s_flight-carrid,
            pa_con  TYPE bc400_s_flight-connid,
            pa_date TYPE bc400_s_flight-fldate.

DATA: gs_carrier TYPE bc400_s_carrier,
      gs_flight  TYPE bc400_s_flight.

TYPES: BEGIN OF gty_s_carrierflight,
         carrid      TYPE bc400_s_flight-carrid,
         connid      TYPE bc400_s_flight-connid,
         fldate      TYPE bc400_s_flight-fldate,
         seatsmax    TYPE bc400_s_flight-seatsmax,
         seatsocc    TYPE bc400_s_flight-seatsocc,
         percentage  TYPE bc400_s_flight-percentage,
         carrname    TYPE bc400_s_carrier-carrname,
         currcode    TYPE bc400_s_carrier-currcode,
         url         TYPE bc400_s_carrier-url,
      END OF gty_s_carrierflight.

DATA: gs_carrierflight TYPE gty_s_carrierflight.

* Get data
TRY.
  CALL METHOD cl_bc400_flightmodel=>get_flight
    EXPORTING
      iv_carrid = pa_car
      iv_connid = pa_con
      iv_fldate = pa_date
    IMPORTING
      es_flight = gs_flight.

  CALL METHOD cl_bc400_flightmodel=>get_carrier
    EXPORTING
      iv_carrid = pa_car
    IMPORTING
      es_carrier = gs_carrier.

CATCH cx_bc400_no_data .
  WRITE: 'No data found!' (ndf).
CATCH cx_bc400_no_auth .
  WRITE: 'No authority for this carrier!' (nau).
ENDTRY.

```

Continued on next page

```
* Fill gs_carrierflight
MOVE-CORRESPONDING gs_carrier TO gs_carrierflight.
MOVE-CORRESPONDING gs_flight TO gs_carrierflight.

IF gs_carrierflight IS NOT INITIAL.
  WRITE: / gs_carrierflight-carrid,
          gs_carrierflight-connid,
          gs_carrierflight-fldate,
          gs_carrierflight-carrname,
          gs_carrierflight-currcode,
          gs_carrierflight-seatsmax,
          gs_carrierflight-seatsocc,
          gs_carrierflight-percentage, '%',
          gs_carrierflight-url.
ENDIF.
```



Lesson Summary

You should now be able to:

- Define structured data objects (structure variables)
- Use basic ABAP statements for 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. Following this, you will analyze the internal tables in the *ABAP Debugger* at runtime.



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 are to use table variables as data stores in your programs and then search for semantic errors in such programs by means of the *ABAP Debugger*.

Working with Internal Tables

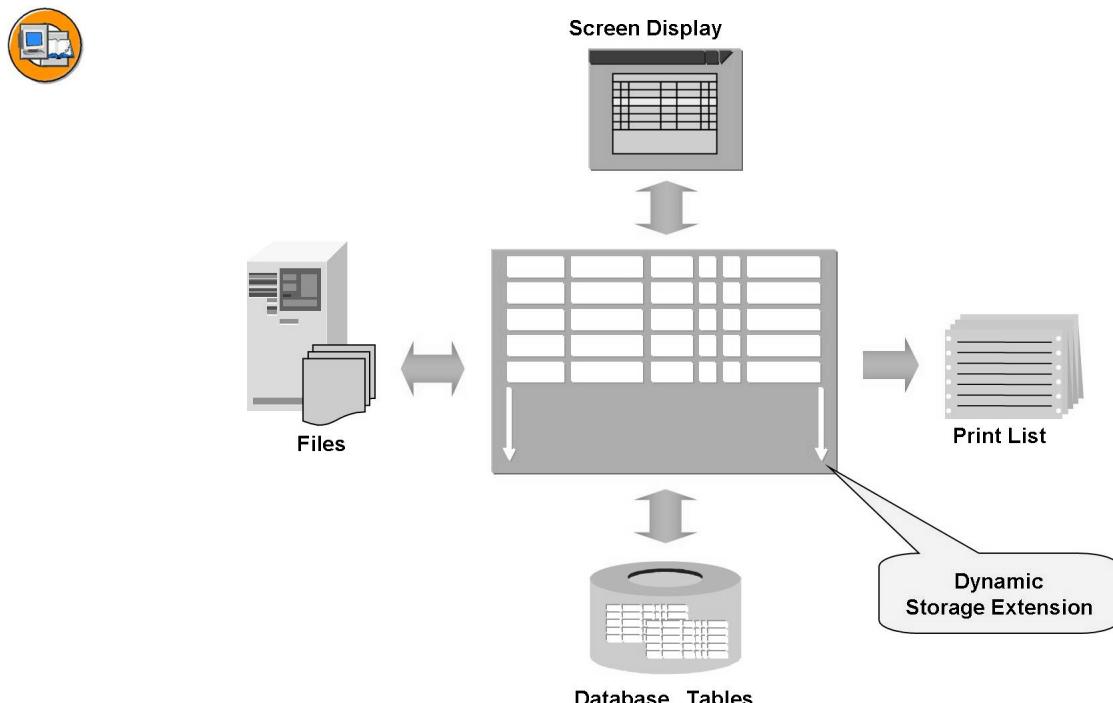


Figure 120: Internal Tables: Usage Options

An **internal table** is a **data object**, in which you can keep several identically structured data records at runtime (table variable). The number of data records is restricted only by the capacity of specific system installations.

The ABAP runtime system **dynamically** manages the size of the internal table. That means, you as the developer do not have to do any work concerning memory management.

The individual data sets in an internal table are known as a **table rows** or **table entries**.

For this reason, the individual components in a row are referred to as **fields** or **columns** of the internal table.

The **row type** of an internal table can be specified through **any data type** and describes the row structure of the table entries.

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

- Retaining data from database tables or sequential files for future processing
- Formatting data for screen or printer output (e.g., sort)
- Format data for using other services (for example, for method, function module or subroutine calls)



1 Line Type		CARRID	CONNID	DISTANCE	
Index		↓	↓		
1	AA	0017	2.572		
2	LH	0400	6.162		
3	LH	0402	7.273		
4	QF	0005	10.000		
5	SQ	0866	1.625		
6	UA	0007	2.572		

2 Key

- Components
- Uniqueness
- Sequence

3 Table Type

- Standard
- Sorted
- Hashed

Figure 121: Attributes of Internal Tables

The following properties specify an internal table completely:

Line type

The line type describes the structure of the table rows. You usually specify a structure type for that. But any data types are possible.

Key

The key of an internal table consists of the key fields including their order. The sequence of the key fields is used, amongst other things, for sorting according to keys. Depending on the access type, the key can be defined as **unique** or **non-unique**. Uniqueness means that a particular combination of key fields can only appear once within the table.

Table kind

You can choose from three different table types: **Standard**, **sorted**, and **hashed**. Depending on the **access type** used, you should use the appropriate table type for the definition in order to enable high performance access. The following graphic illustrates the selection of the appropriate table type.



	Index Tables		Hashed Table
Table Type	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"/>
Key Uniqueness	NON-UNIQUE	UNIQUE NON-UNIQUE	UNIQUE
Use in	Mainly Index Access	Mainly Key Access	Only Key Access

Figure 122: Attributes and Use of the Table Types

When a table entry is accessed by specifying the corresponding **row number**, this is called **index access**.

In contrast, when you access an entry by entering a key value, this is called key access.

Depending on the access type, you should always choose the most suitable of the following three table kinds in order to enable high performance accesses:

- With **standard tables**, the row numbering (index) is maintained internally. Both index and key accesses are possible.
You should choose this table type if you mostly use the index to access the internal table.
- With **sorted tables**, the data records are automatically sorted in ascending order of the keys. Here, too, the index is maintained internally. Both index and key accesses are possible.
You should choose this table kind if you normally access the internal table with the key or would like the table to be automatically sorted by key.
- With **hashed tables**, the data records are managed for fast key access using the hashing procedure. A unique key is required here. With hashed tables, only key accesses are possible.
You should choose this table type if the internal table is very large and you want to access it by key only.



Hint: Only standard tables are used for this course. However, with the exception of a few special cases, the syntax is identical for all three table types.

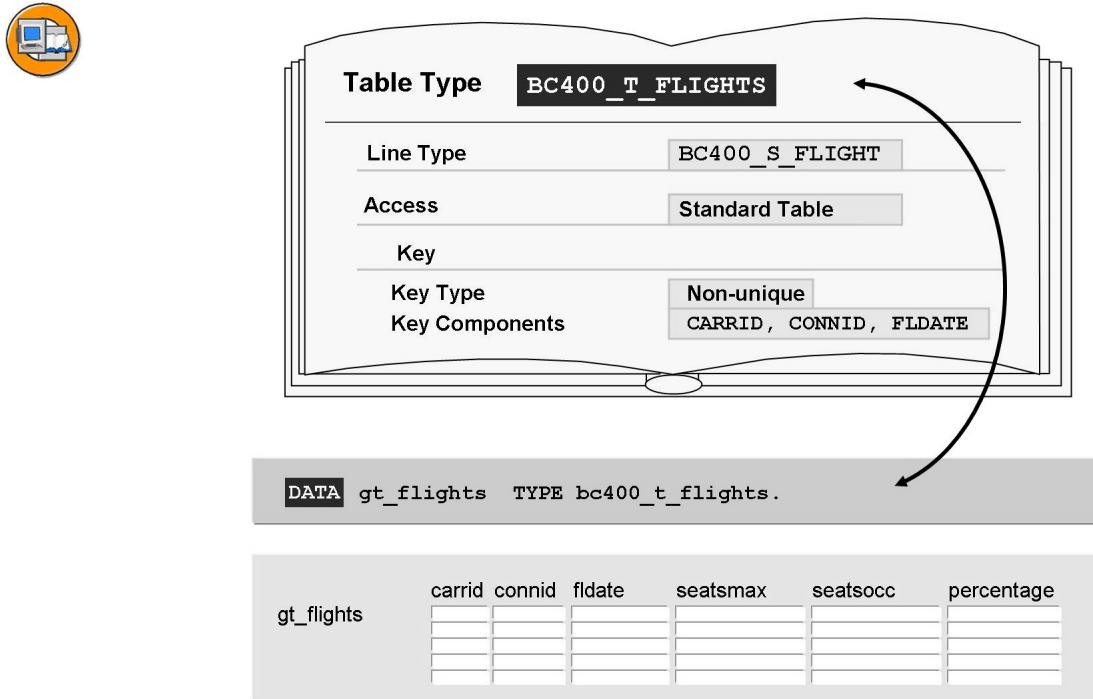


Figure 123: Defining Internal Tables with Global Types

The type of an internal table is called **table type**. Table types can be defined globally in the *ABAP Dictionary* or locally in a program. The figure above shows a table type declared in the *ABAP Dictionary* as well as the program-internal definition of a table variable with reference to the table type.

For detailed information on the declaration of global table types in the *ABAP Dictionary*, refer to the online documentation. You can access this using the **H** button in the display or in the maintenance section of the table type.



```
TYPES gty_t_flights
      TYPE STANDARD TABLE OF bc400_s_flight
      WITH NON-UNIQUE KEY carrid connid fldate.
```

Local Table Type

```
DATA  gt_flights  TYPE gty_t_flights.
```

Internal Table

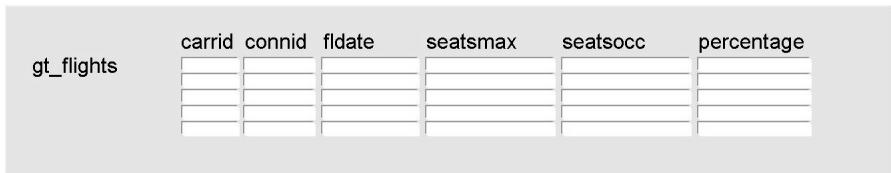


Figure 124: Defining Internal Tables with Local Types

The above graphic shows a table type declared locally in the program as well as the internal definition of a table variable in the program with reference to the locally declared table type.

When you list the key fields in the table type, note that the sequence matters for certain types of processing (such as “sort by key”). For detailed information on declaring local table types, refer to the keyword documentation for the TYPES statement.



Hint: Alternatively, you can also define an internal table **directly** to declare a local table type: All you have to do here is use **DATA** instead of **TYPES**.



Caution: A common beginner's error consists of the following syntax:

DATA gt_itab TYPE TABLE OF <Table type>.

This would result in the definition of an internal table with rows that are themselves internal tables (of the specified table type)!

In the previous definitions of internal tables, Dictionary objects were always used: Either a table type (BC400_T_FLIGHTS) or at least a structure type (BC400_S_FLIGHT). The following graphic shows an “independent” table definition:



```

TYPES: BEGIN OF gty_s_type,
        carrid TYPE s_carr_id,
        connid TYPE s_conn_id,
        ...
    END OF gty_s_type.                                Local Structure Type

STANDARD
DATA gt_itab TYPE SORTED TABLE OF gty_s_type
HASHED
WITH ... KEY ...                                Internal Table

```

Figure 125: Independent Definition of Internal Tables

This enables you to implement internal tables with any kind of structure without having to refer to existing Dictionary types.

The following graphic again shows an overview of possible definitions of internal tables:



- 1 DATA gt_itab TYPE <Table Type> .

- 2 DATA gt_itab TYPE SORTED TABLE OF <Structure Type>
 STANDARD
 HASHED
 WITH ... KEY ...

- 3 DATA gt_itab TYPE TABLE OF <Structure Type> .
 (Short form for definition of a standard table with
 non-unique default key)

Figure 126: Possible Definitions of Internal Tables

The short form of a table definition illustrated above implicitly uses the following default values:

- Table type: Standard (default)
- Uniqueness of key: Non-unique (only option for a standard table)
- Table key: **Default key** (All non-numeric table fields are key fields)



Hint: As the default key usually cannot be used in a meaningful way, you should only use it to define an internal table, if you do not need the key for processing your table.

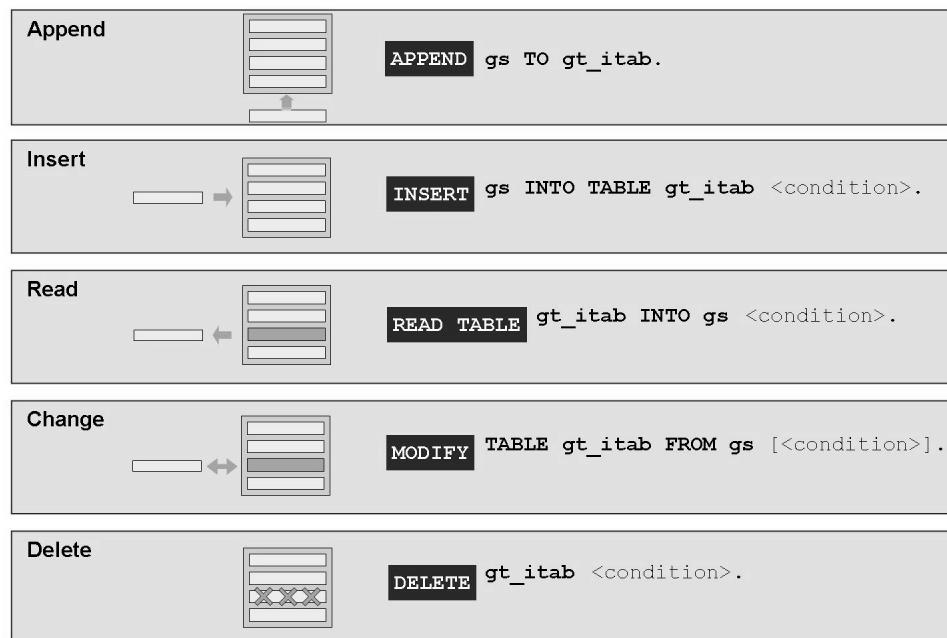


Figure 127: Accessing Single Records (Overview)

In most cases, you require a structure variable for single record processing of an internal table that must be defined as having the **same type** as the **line type** of the internal table. This structure variable is known as the **work area**. The graphic above illustrates the processing of an internal table via the corresponding work area.

APPEND

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

INSERT

Inserts the content of a structure into an internal table.

READ TABLE

Copies the content of a table row to 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 row of an internal table that has the same key. In doing so, only non-key fields are added. Hence, this statement can only be used for tables whose non-key fields are all numeric.

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

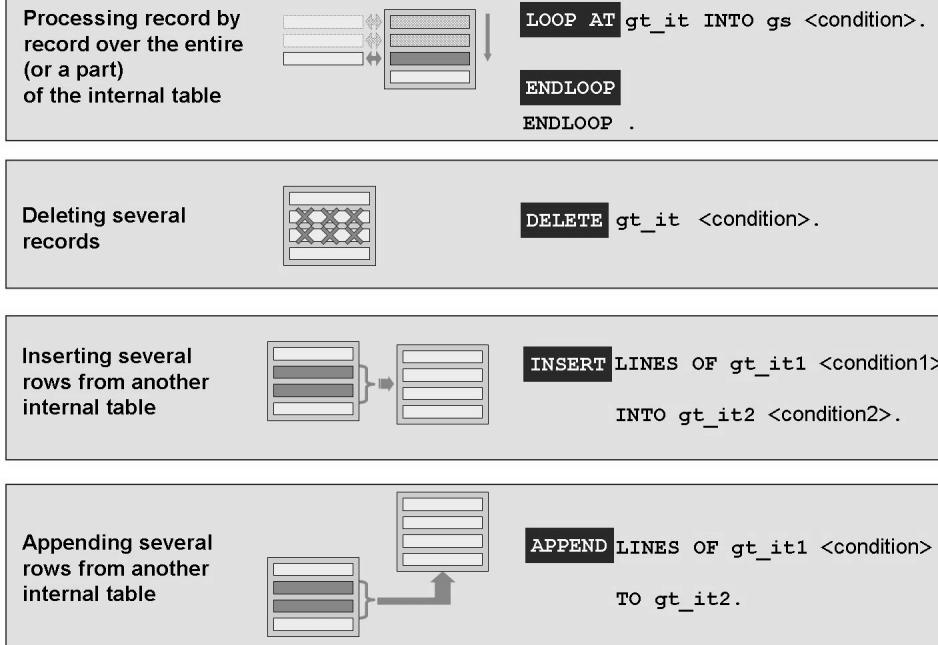


Figure 128: Processing Sets of Records (Overview)

LOOP AT . . . ENDLOOP

The LOOP places the rows of an internal table into the structure specified in the INTO clause one by one. Within the LOOP, the current content of the structure can be output or changed and written back to the table.

DELETE

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

INSERT LINES OF

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

APPEND LINES OF

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.

Some actual examples of syntax for the most common statements follow here.



```
* define internal table and workarea

DATA: gt_flightinfo TYPE bc400_t_flights,
      gs_flightinfo LIKE LINE OF gt_flightinfo.

gt_flightinfo
gs_flightinfo

* fill structure with values

gs_flightinfo-carrid      = .... .
gs_flightinfo-connid       = .... .
gs_flightinfo-fldate        = .... .
gs_flightinfo-seatsmax     = .... .
gs_flightinfo-seatsocc     = .... .
gs_flightinfo-percentage   = .... .

* insert structure into internal table
INSERT gs_flightinfo INTO TABLE gt_flightinfo.
```

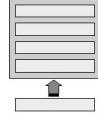


Figure 129: Syntax Example: Inserting a Row

You can insert a row into an internal table by writing the data for the required record into the prepared work area and then inserting it into the internal table with an `INSERT` statement.

With standard tables, this content is appended, in sorted tables, the row is inserted in the right place with reference to the key fields, and in hashed tables it is inserted according to a hash algorithm.

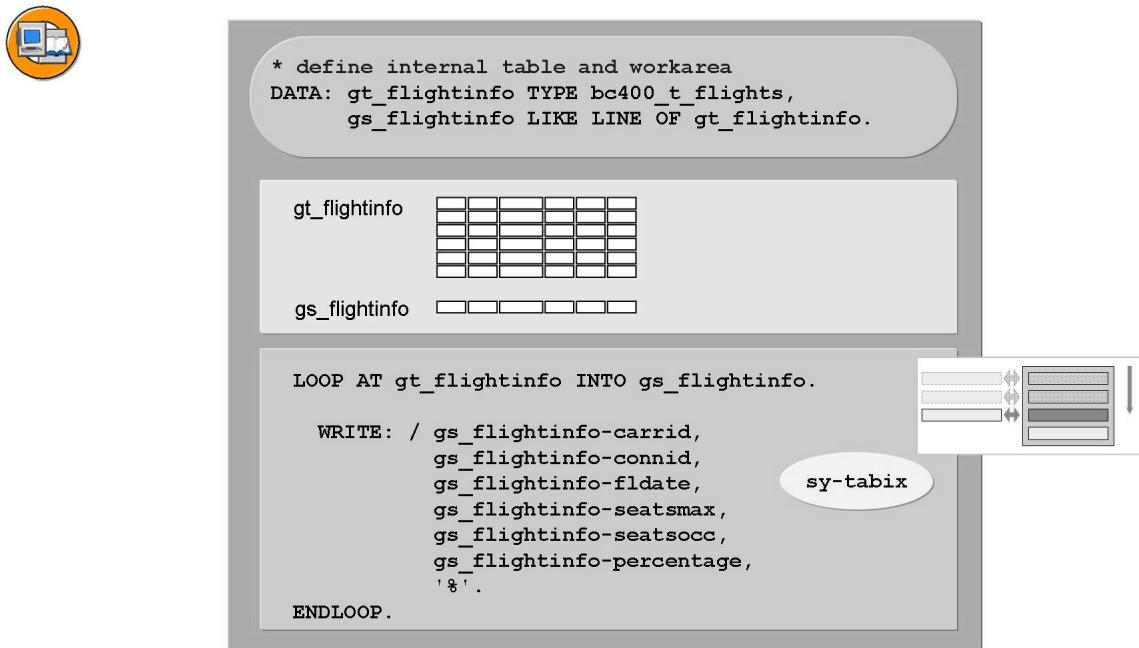


Figure 130: Syntax Example: Outputting an Internal Table Row-by-Row

You can read and edit the contents of an internal table row-by-row using a `LOOP`. Within a search run, the `sy-tabix` system field contains the row number of the current table entry.

In our example, all the rows of the internal table are processed consecutively and output using the `WRITE` statement.

If you want to change the content of the current table row within a loop pass, first change the copy of the row in the work area and then write it back to the current table row using the `MODIFY` statement. The syntax for this is:

```
MODIFY itab FROM wa.
```



```

LOOP AT gt_flightinfo INTO gs_flightinfo
      FROM 1 TO 5 .

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

READ TABLE gt_flightinfo INTO gs_flightinfo
      INDEX 3.

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

```

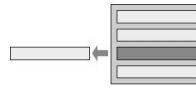
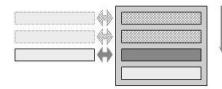


Figure 131: Syntax Example: Reading by Index

In the LOOP, you can restrict access to **specific rows** using the FROM-TO addition. In the above example, the system only processes the first five rows of the internal table consecutively.

You can use the READ TABLE statement to read a **single record**. You use the INDEX addition to specify the row number of the required record.

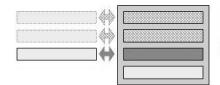
Note that in both examples in the above graphic, an index is used. This is only possible with index tables (that is, standard and sorted).



```

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

```



```

READ TABLE gt_flightinfo INTO gs_flightinfo
      WITH TABLE KEY carrid = 'LH',
            connid = '0400'
            fldate = sy-datum.

      IF sy-subrc = 0.
      WRITE: / gs_flightinfo-seatsmax,
              gs_flightinfo-seatsocc,
              gs_flightinfo-percentage,
              '%'.
ENDIF.

```

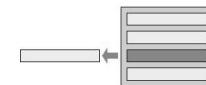


Figure 132: Syntax Example: Reading by Key

In the LOOP, you can restrict access to **specific rows** using the WHERE addition. In the above example, the system only processes those internal table rows in which the CARRID field has the value LH.



Hint: With regard to the runtime requirement, a sorted table with the CARRID field as the first key field is the most suitable type for this kind of processing.

You can use the READ TABLE syntax in the above graphic to read a **specific row** of the internal table. In this case, you have to specify the complete key value of the entry to be read after the WITH TABLE KEY addition. The return code **sy-subrc** is only set to zero if a corresponding row was found in the internal table.



Hint: With regard to the runtime requirement, a hash table is most suitable for this kind of access when you have a large dataset.

Note that with the WITH TABLE KEY addition **all** key fields have to be supplied with data. If you want to limit the fields you must use the WITH KEY addition.



```
SORT gt_flightinfo.  
  
SORT gt_flightinfo BY carrid.  
  
SORT gt_flightinfo BY percentage DESCENDING  
carrid ASCENDING
```



```
REFRESH gt_flightinfo.  
  
FREE gt_flightinfo.
```



Figure 133: Syntax Example: Sorting and Deleting Content

Standard and hash tables can be sorted in ascending or descending order by table key or columns by using the SORT statement. If neither ASCENDING nor DESCENDING is specified as the sorting order for a sort field, the field is sorted in ascending order by default.

Language-specific sort rules can even be taken into account, if necessary:

You can use the optional **AS TEXT** addition to implement lexicographical sorting. In a German-speaking environment, for example, this means that “ä” comes before “b”, and not behind “z”, as would be the case with non-lexicographical sorting.

If you use the **STABLE** addition, which is also optional, the relative order of data records which have identical sort keys will remain intact during sorting.

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

You can use the following statements for deleting the table contents:

REFRESH

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

CLEAR

With internal tables without a header line (all those that have been previously defined in the course), the CLEAR statement has the **same effect as REFRESH**.

For internal tables with a header line (see below) in contrast, it **only initializes the header line**.

FREE

This deletes the entire contents of the internal table and releases the previously used memory. You use the FREE statement for internal tables that have already been evaluated and are no longer required in the further course of the program. This has the effect that previously assigned but no longer required memory becomes available again.

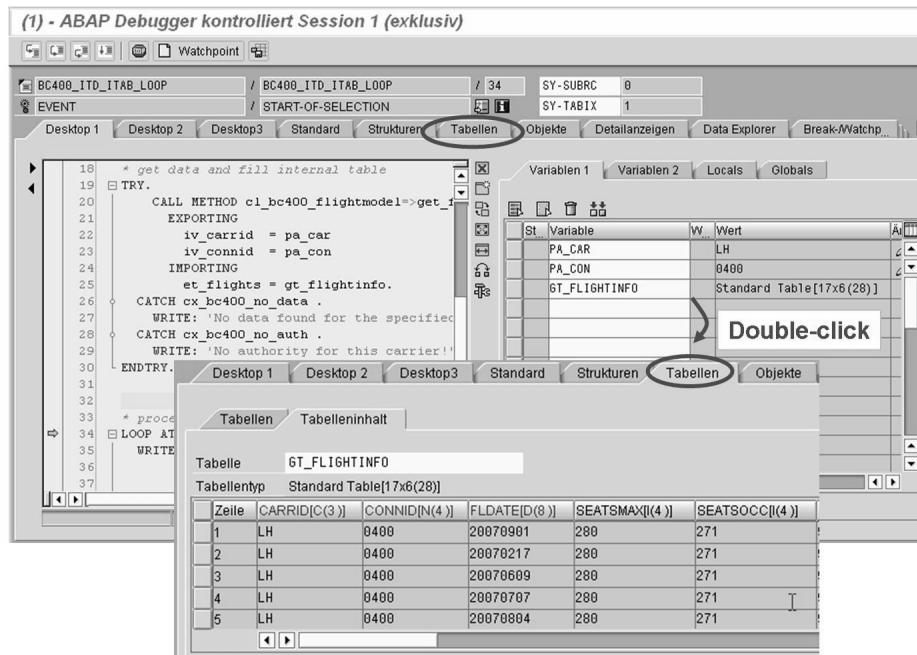


Figure 134: Internal Tables in Debugging Mode

In the Debugger, you can trace the content of an internal table by entering the table name in the *Variable 1* area and then branching to the table display by double-clicking.

Alternatively, you can also select the *Table* tab page and specify the table name in the *Internal Table* field. Following that, you can get to the display of the table contents by pressing the **Enter** key. In addition, you can configure an area for displaying an internal table on one of the desktops.



To display an internal table in the **classic ABAP Debugger**, a key with the label “Table” is provided. As with the new Debugger, however, you can also navigate to this display by double-click.

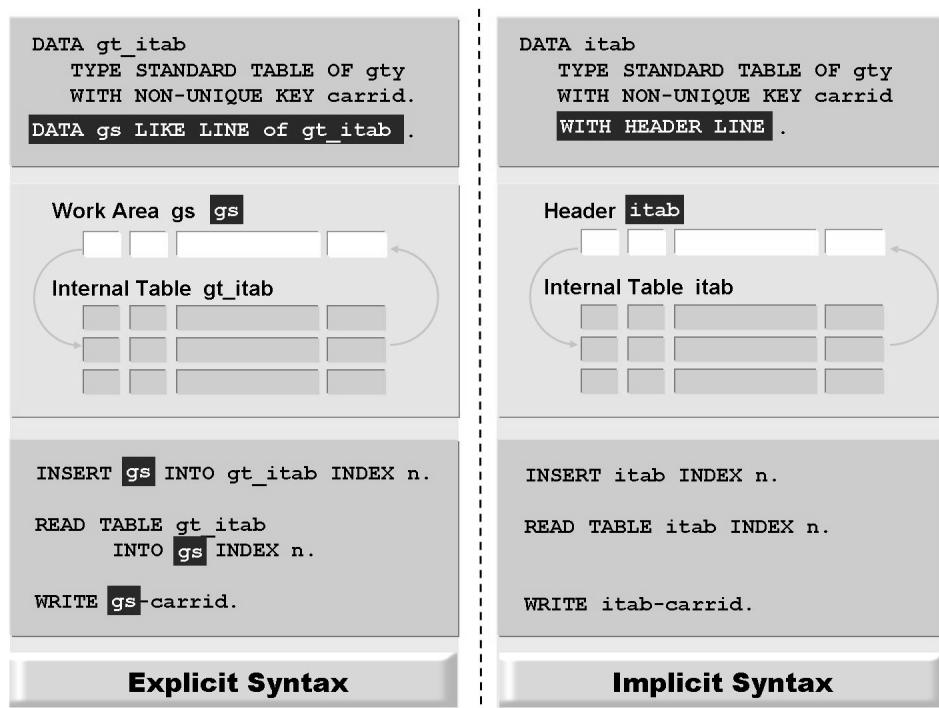


Figure 135: Comparison: Internal Tables with and Without Header Lines

With the `WITH HEADER LINE` addition in the definition of an internal table, you have the option of creating a table **with a header line**. When this is done, a work area (header row that suits the table is created automatically so that an additional definition of the same is not necessary. This also simplifies the syntax of the table commands, as the system always refers to the automatically generated work area, which therefore no longer has to be specified explicitly. In spite of the mentioned advantages we no longer want to use table with header lines because:

- The automatically generated work area has the same name as the internal table, which does not contribute to making the program readable.
- Tables with header lines are not allowed in the following areas:
 - Nested data objects (structures and internal tables that have internal tables as components)
 - ABAP objects (object-oriented extension of ABAP).

We only mention internal tables with header lines here because some older programs still use them and you have to work with such programs from time to time. It is for this purpose that further particularities of the header line are listed below:

If an internal table with a header line is called **itab**, then **itab - tablefield** is used to address the corresponding field in the **work area**.

You can address the **body of the table** with **itab []**.

The following **example** illustrates the above-mentioned situation:

```
DATA itab1 TYPE TABLE OF scarr WITH HEADER LINE.  
DATA itab2 LIKE itab1.
```

Only operations with header lines !

Operations with table bodies

The following very old syntax also defines an internal table **with header line**, even though that is not stated specifically.

```
DATA: BEGIN OF itab OCCURS n,  
      field1 TYPE ... ,  
      field2 TYPE ... ,  
      ... ,  
      END OF itab.
```


Exercise 14: 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
- Process the content of internal tables using a loop

Business Example

You want to transfer flight connection data from a database table to an internal table (as temporary storage) and output it in a list.

Template:

None

Solution:

BC400_ITS_ITAB_LOOP

Task 1:

Create a program and define an internal table for flight connections within it.

1. Create the executable program **ZBC400_##_ITAB_LOOP** without a “TOP include”. Assign the program to your package and your change request.
2. To buffer flight connection data in an internal table, we recommend you define your internal table in such a way that your line type is compatible with the structure type BC400_S_CONNECTION.

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



Hint: Display the structure BC400_S_CONNECTION in the *ABAP Dictionary* (the cursor should be placed on the structure name). There, use the appropriate pushbutton to list the *where-used list* for BC400_S_CONNECTION. (Pay attention to the correct selection when triggering the where-used list)

3. Define an internal table (suggested name: **gt_connections**) based on one of the global table types that were found.

Continued on next page

4. Define a suitable work area for the internal table (suggested name: `gs_connection`).

Task 2:

Fill your internal table and output the content in a list.

1. Get the flight connection data by calling the method `GET_CONNECTIONS` of the class `CL_BC400_FLIGHTMODEL`. Choose the internal table you defined as the passed parameter.
2. If no flight connection was found, output a short message to the list. Use the error handling of the method call to do this.
3. Use the `LOOP` statement to output the buffered data in the internal table in a list.
4. (OPTIONAL) Use the `SORT` statement to sort the internal table in ascending order according to departure time before the data is output in the list.

Task 3:

Execute your program and check the result.

1. Check the output of the internal table in the list. Use the *ABAP Debugger* to do this too.

Solution 14: Working with Internal Tables

Task 1:

Create a program and define an internal table for flight connections within it.

1. Create the executable program **ZBC400_##_ITAB_LOOP** without a “TOP include”. Assign the program to your package and your change request.
 - a) Carry out this step in the same way as before.
2. To buffer flight connection data in an internal table, we recommend you define your internal table in such a way that your line type is compatible with the structure type **BC400_S_CONNECTION**.

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



Hint: Display the structure **BC400_S_CONNECTION** in the *ABAP Dictionary* (the cursor should be placed on the structure name). There, use the appropriate pushbutton to list the *where-used list* for **BC400_S_CONNECTION**. (Pay attention to the correct selection when triggering the where-used list)

- a) Carry out this step as described.
3. Define an internal table (suggested name: **gt_connections**) based on one of the global table types that were found.
 - a) See source code excerpt in the model solution.
4. Define a suitable work area for the internal table (suggested name: **gs_connection**).
 - a) See the source code excerpt from the model solution.

Task 2:

Fill your internal table and output the content in a list.

1. Get the flight connection data by calling the method **GET_CONNECTIONS** of the class **CL_BC400_FLIGHTMODEL**. Choose the internal table you defined as the passed parameter.
 - a) See the source code excerpt from the model solution.

Continued on next page

2. If no flight connection was found, output a short message to the list. Use the error handling of the method call to do this.
 - a) See the source code excerpt from the model solution.
3. Use the LOOP statement to output the buffered data in the internal table in a list.
 - a) See the source code excerpt from the model solution.
4. (OPTIONAL) Use the SORT statement to sort the internal table in ascending order according to departure time before the data is output in the list.
 - a) See the source code excerpt from the model solution.

Continued on next page

Task 3:

Execute your program and check the result.

1. Check the output of the internal table in the list. Use the *ABAP Debugger* to do this too.
 - a) Set a session breakpoint in the *ABAP Editor* and execute the program directly using the  key.

Result

Source code extract: **BC400_ITS_ITAB_LOOP**

```

REPORT bc400_its_itab_loop.

DATA: gt_connections TYPE bc400_t_connections,
      gs_connection  TYPE bc400_s_connection.

* Get data
TRY.
  CALL METHOD cl_bc400_flightmodel=>get_connections
    IMPORTING
      et_connections = gt_connections.
  CATCH cx_bc400_no_data .
    WRITE: / 'No data found!' (ndf).
  ENDTRY.

  SORT gt_connections ASCENDING BY deptime.

* Output
LOOP AT gt_connections INTO gs_connection.
  WRITE: / gs_connection-carrid,
          gs_connection-connid,
          gs_connection-cityfrom,
          gs_connection-airpfrom,
          gs_connection-cityto,
          gs_connection-airpto,
          gs_connection-fltime,
          gs_connection-deptime,
          gs_connection-arrtime.

ENDLOOP.

```



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 structured data objects (structure variables)
- Use basic ABAP statements for 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 statements.



Test Your Knowledge

1. By which elements of the *ABAP Dictionary* can you set the type for a structure variable within the program?

Choose the correct answer(s).

- A Data element
- B Domain
- C Structure
- D Table type

2. Which fields are taken into account in the MOVE-CORRESPONDING statement, where two structures are to be specified?

3. What is an internal table?

Choose the correct answer(s).

- A Database table
- B Excel table
- C Variable in an ABAP program
- D Table that is embedded in another table

4. Using which keyword do you define an internal table?

5. Which three specifications are required in the definition of an internal table?

6. Which three kinds of internal tables are available so that you can influence system performance for table access?

7. What is a table type?

Choose the correct answer(s).

- A An internal table
- B Description of an internal table
- C Description of a database table
- D The specification "STANDARD"/"SORTED"/"HASHED" for an internal table

8. BC400_S_BOOKING is a Dictionary structure.

Which syntax do you use to create an internal table with the same structure as BC400_S_BOOKING?

Choose the correct answer(s).

- A DATA gt_itab TYPE bc400_s_booking.
- B DATA gt_itab TYPE TABLE OF bc400_s_booking.
- C DATA gt_itab TYPE LINE OF bc400_s_booking.
- D DATA gt_itab LIKE bc400_s_booking.

9. Which syntax do you use to create a work area for an internal table already defined with the name “gt_itab”?

10. Which statement retrieves a single record from an internal table?

Choose the correct answer(s).

- A SELECT ... ENDSELECT
- B SELECT SINGLE
- C READ TABLE
- D GET
- E FETCH

11. Should internal tables with header rows still be used in programming?



Answers

1. By which elements of the *ABAP Dictionary* can you set the type for a structure variable within the program?

Answer: C

2. Which fields are taken into account in the MOVE-CORRESPONDING statement, where two structures are to be specified?

Answer: Fields that are in both structures and have the same name.

3. What is an internal table?

Answer: C

4. Using which keyword do you define an internal table?

Answer: DATA

An internal table is a variable and variables are defined using DATA.

5. Which three specifications are required in the definition of an internal table?

Answer: Row type, key, and kind of internal table

6. Which three kinds of internal tables are available so that you can influence system performance for table access?

Answer: STANDARD, SORTED, and HASHED

7. What is a table type?

Answer: B

8. BC400_S_BOOKING is a Dictionary structure.

Which syntax do you use to create an internal table with the same structure as BC400_S_BOOKING?

Answer: B

9. Which syntax do you use to create a work area for an internal table already defined with the name “gt_itab”?

Answer: DATA gs LIKE LINE OF gt_itab.

10. Which statement retrieves a single record from an internal table?

Answer: C

11. Should internal tables with header rows still be used in programming?

Answer: No, the information provided on this is to be used solely for understanding older programs.

Unit 6

Data Modeling and Data Retrieval

Unit Overview

The unit overview lists the individual lessons that make up this unit.



Unit Objectives

After completing this unit, you will be able to:

- Explain the purpose and the benefits of using a data model in application development
- Describe the SAP flight data model
- Describe the meaning and the structure of a transparent table
- List different methods for searching relevant database tables
- Program read access to specific columns and rows within a particular database table
- List different methods for read accesses to several database tables
- Explain the SAP authorization concept
- Implement authorization checks

Unit Contents

Lesson: Data Modeling and Transparent Tables in the ABAP Dictionary	266
Exercise 15: Analyzing Transparent Tables in the ABAP Dictionary	275
Lesson: Reading Database Tables	282
Exercise 16: Data Retrieval by Single Record Access	303
Exercise 17: Data Retrieval Using a SELECT Loop	309
Exercise 18: (Optional) Data Retrieval by Mass Access (Array Fetch) ..	317
Lesson: Authorization Check	324
Exercise 19: Authorization Check	331

Lesson: Data Modeling and Transparent Tables in the ABAP Dictionary

Lesson Overview

In this lesson you will get an overview of data modeling. This needs to be performed before the application development. You will also learn how database tables are defined and described in the *ABAP Dictionary*.



Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose and the benefits of using a data model in application development
- Describe the SAP flight data model
- Describe the meaning and the structure of a transparent table

Business Example

You want to develop a program that accesses database tables. To do this, you need to know the underlying data model and the structure of the database tables.

Data Modeling

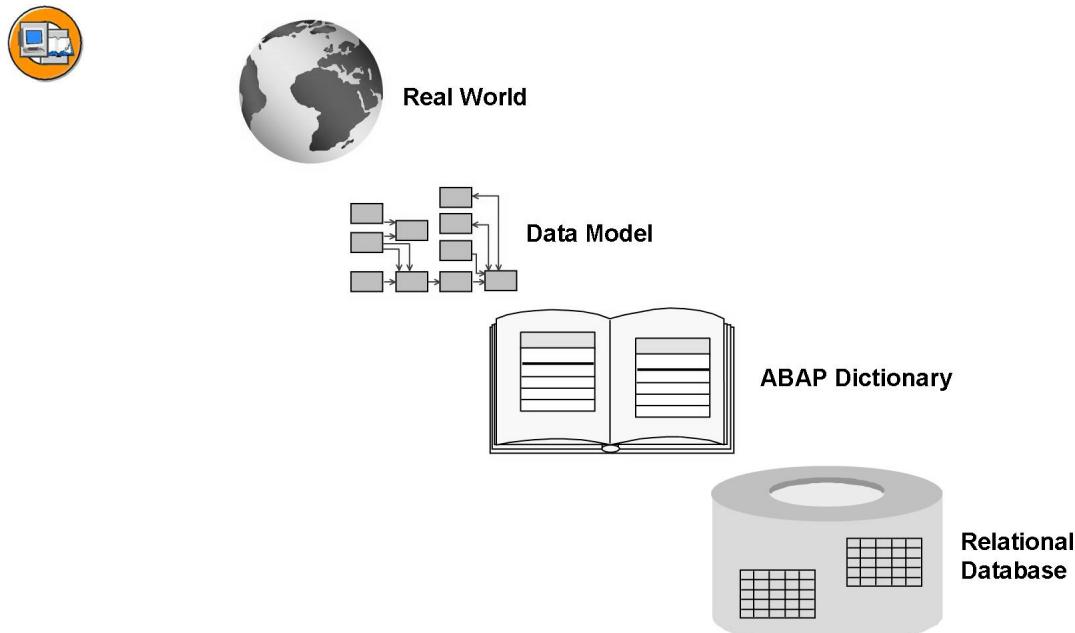


Figure 136: Data Modeling

In the development of business application software, parts of the real world must be represented in data form. A business unit represents an entity here. These entities exist in relationships with each other, which are fixed in the underlying data model. We also refer to an entity relationship model (ERM).

You use this data model as the basis for implementing appropriate table definitions (transparent tables) including their relationships with each other in the *ABAP Dictionary*.

By activating the table definitions, corresponding database tables are automatically created in the database. The actual application data is entered into those tables later on.

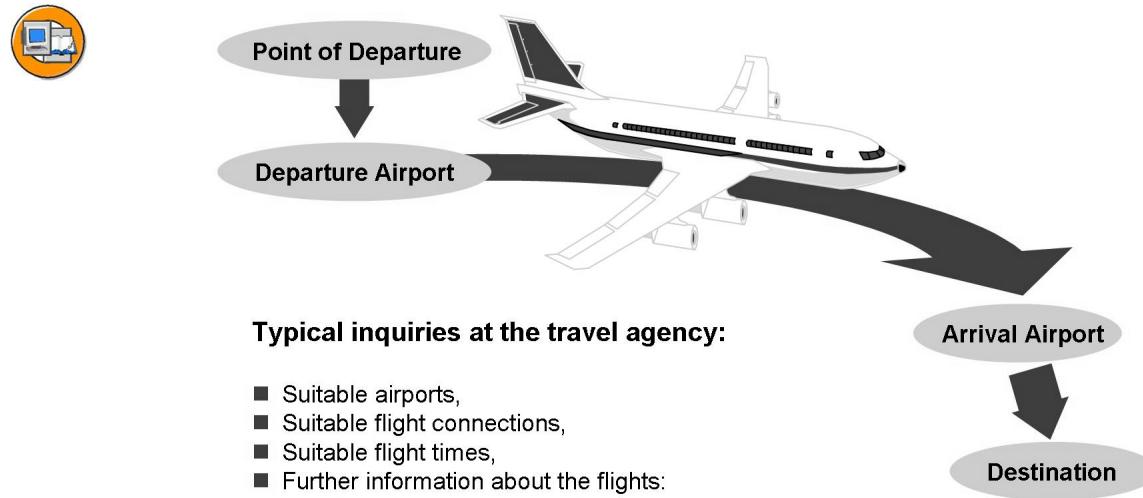


Figure 137: Flight Data Model for ABAP Training Courses

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

In the context of this course it is sufficient to show a simple excerpt from this data model: If **customers of a travel agency** want to travel from one place to another, they require the travel agency to find out:

- Which connection offers the best and most direct flight?
- What the acceptable flight times are on the proposed day of travel?
- Whether, dependent on individual conditions, there is an ideal solution, such as the cheapest flight, the fastest connection, a connection with a particular arrival time.

This perspective differs from that of a **travel agency**: In the data model that is designed for **managing** the required data, the data is stored according to **technical** criteria in tables in a central database. The amount of data stored far exceeds the requirements of a customer.

You therefore need to be able to compile the data to meet the individual requirements of the customer using application programs.

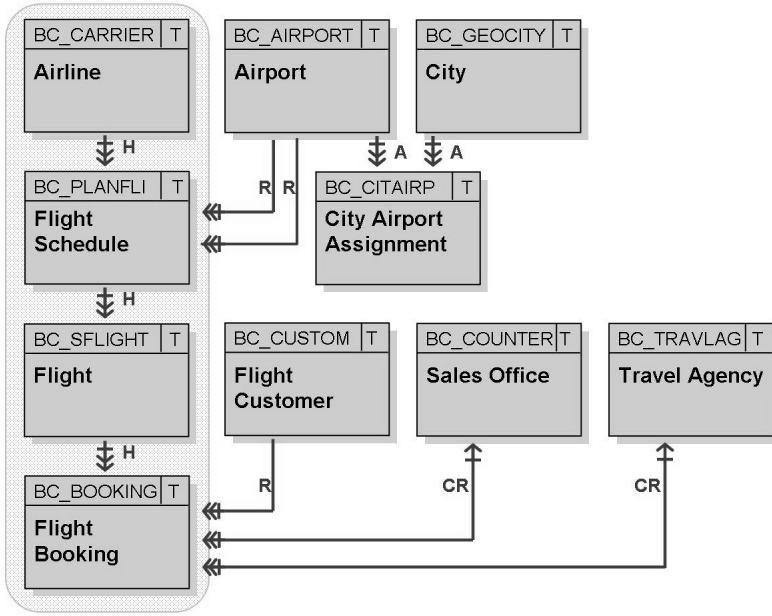


Figure 138: Relational Data Model

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 the data requested by the customer.

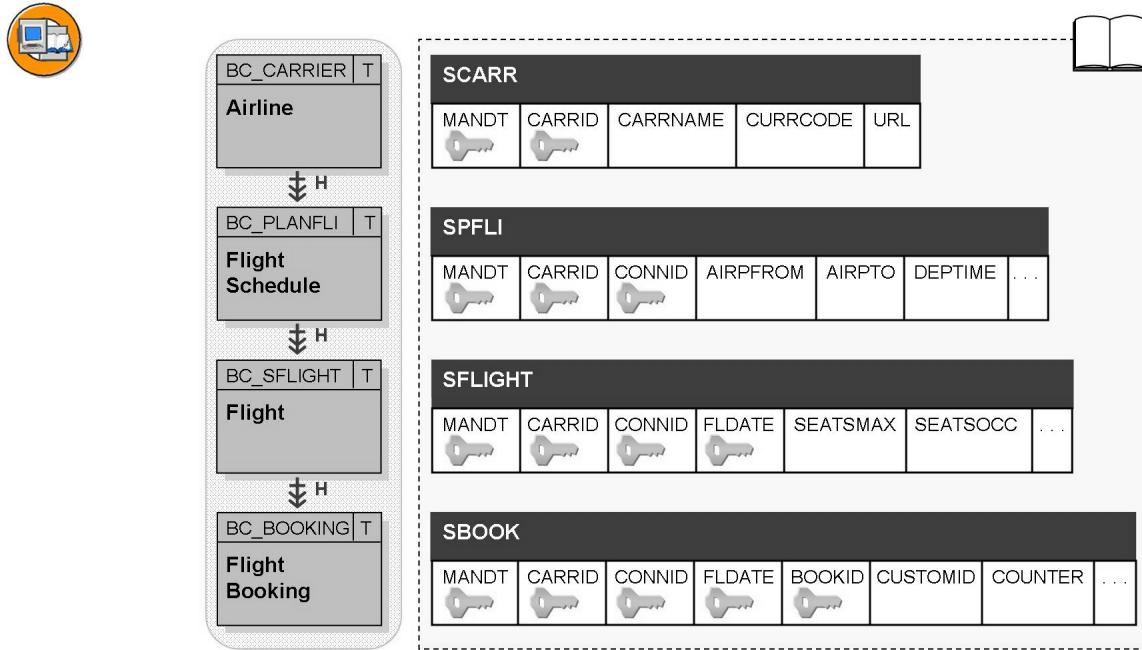


Figure 139: Implementation Using Transparent Tables

For each entity fixed in the data model, the developer creates a *transparent table* in the **ABAP Dictionary**. This is merely a **platform-independent description** of a database table, not the database table itself. However, when the transparent table is activated, a table of the same name is automatically created on the database.

A transparent table contains different fields (columns) to allow you to store and manage data records in a structured way. You have to declare table fields as **key fields** when their content is to be used for the unique identification of data records within the database table. The **key of a table (table key)** consists of key fields. This is also called a **primary key**. Data records in the same table have to be unique with regard to the primary key values. The key value of a data record is thus a unique ID within the table.

Transparent Tables

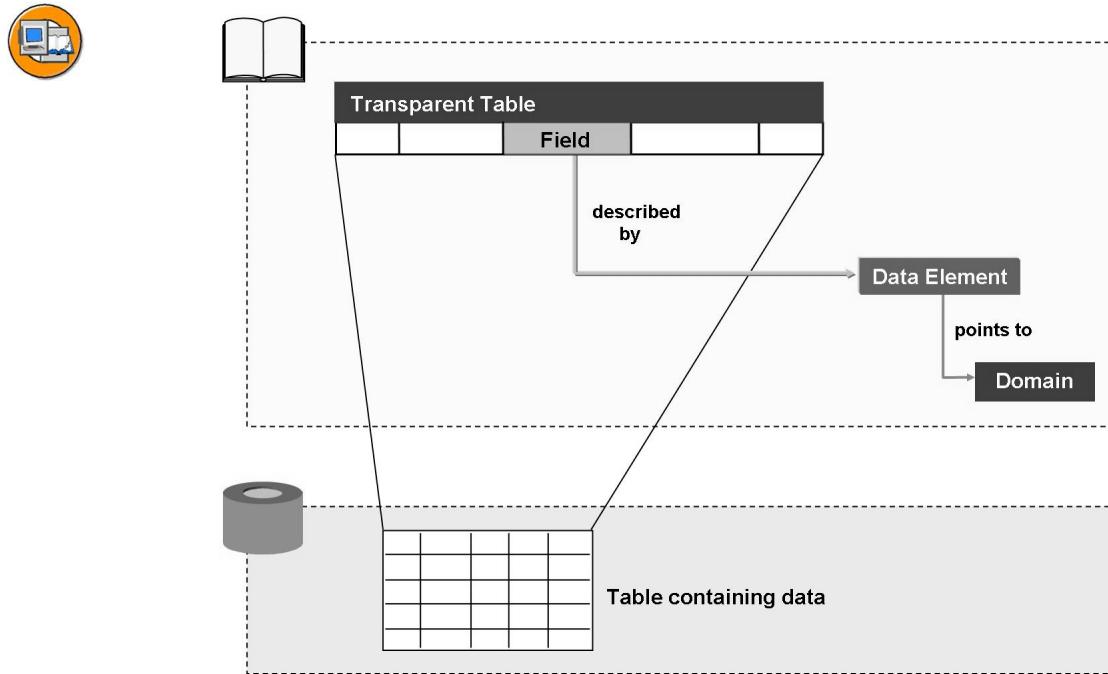


Figure 140: Technical Structure of a Transparent Table

In the *ABAP Dictionary*, a transparent table is an implemented description of the corresponding database table that contains the actual application data. The **fields** of the transparent table form the identically-named columns of the corresponding database table. Data elements, with which you are already familiar as global elementary data types, are normally used to describe the individual fields. The data elements themselves refer to domains for their technical properties.



The screenshot shows the ABAP Dictionary interface for the SPFLI table. The title bar reads "Dictionary: Display Table". Below it, the table name "SPFLI" is shown under "Transparent Table" and "Flight Schedule" under "Short Description". The status is "Active". The table has three tabs: "Properties", "Fields", and "Currency/Quantity Fields". The "Fields" tab is selected, displaying a list of fields with columns for "Fields", "Key", "Data Element", "Data Type", "Length", "Dec. Places", and "Short Description". Several fields are marked as keys (indicated by checked boxes in the "Key" column). The data includes:

Fields	Key	Data Element	Data Type	Length	Dec. Places	Short Description
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	Departure city
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 141: Transparent Tables in the ABAP Dictionary

As well as the list of fields, the transparent tables contain other information that is required in order to create a table of the same name on the database and describe its properties in full:

- The determination of the key for the database table (key fields)
- The technical properties that are required by the database in order to create the database table (expected size, expected frequency of access)
- Settings for technologies that can speed up access to the database table (secondary indexes, buffering)

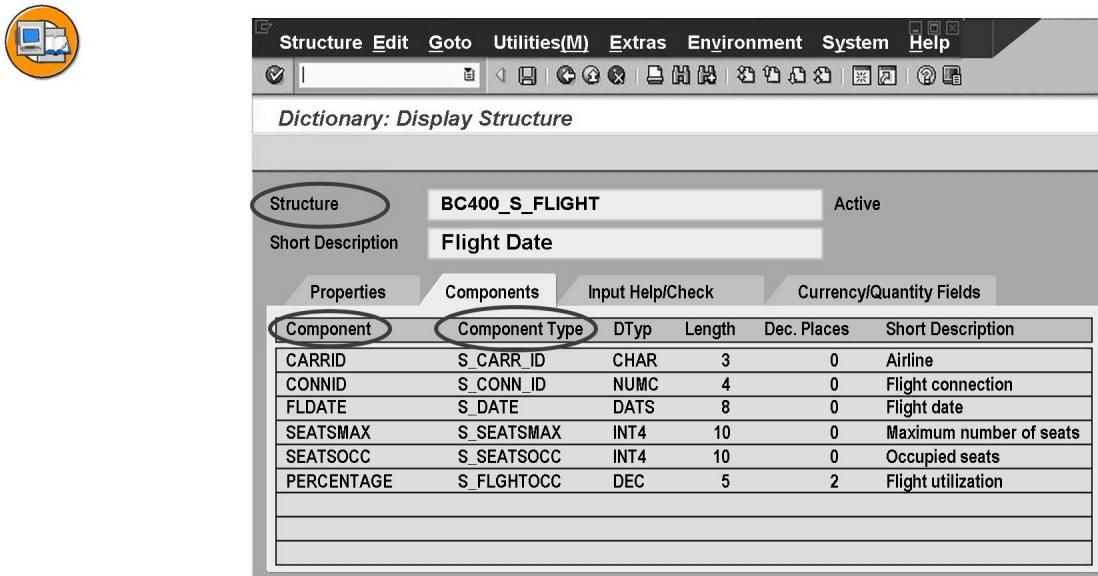


Figure 142: Structures in the ABAP Dictionary

At first glance, the definition of a transparent table appears very similar to the definition of a global structure type. In actual fact, transparent tables can be used in programming in the same way as structure types, for example, for defining a structured data object (structure variable), for typing an interface parameter, or as the line type of a global or local table type. Only the list of fields is important here, however. Other properties of the transparent table, such as the key definition or the technical properties, are irrelevant when using it as a data type.

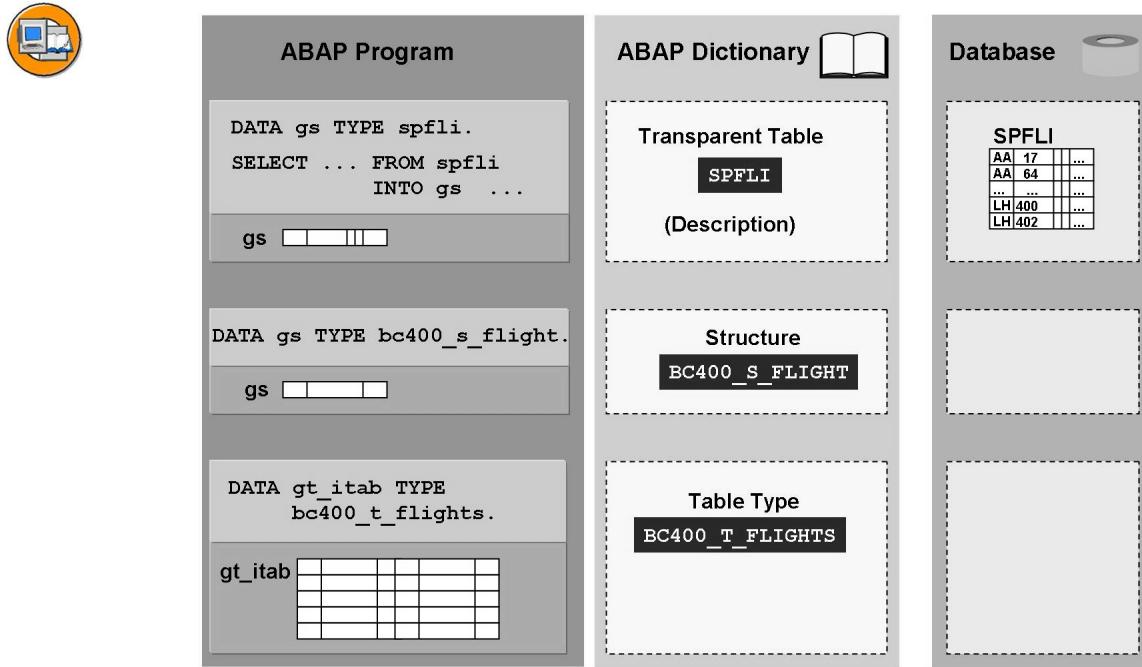


Figure 143: Transparent Table as an ABAP Data Type



Hint: In older programs, transparent tables were used as data types very freely. Nowadays, we recommend they are only used directly in connection with access to the database. Above all, transparent tables should not be used in the process of defining user interfaces. The problem here lies in the unwanted dependency between the definition of database objects or interfaces and the user interface.

As well as the additional properties relating to the database, there is another difference between transparent tables and structure types:

A transparent table is always a list of elementary fields, whereas the components of a structure type can themselves be structured again (nested structures). The component of a structure can even be typed with a table type (deep structures).

Exercise 15: Analyzing Transparent Tables in the ABAP Dictionary

Exercise Objectives

After completing this exercise, you will be able to:

- Analyze the properties of transparent tables in the *ABAP Dictionary*
- Use the *Data Browser* to acquire an overview of the table content

Business Example

You want to familiarize yourself with the flight data model. To do this, you analyze some of the relevant transparent tables in the *ABAP Dictionary* and display table contents with the *Data Browser*.

Task 1:

Examine the properties of the database table SPFLI and analyze the name and data types of the table fields.

1. Display the relevant transparent table.
2. The table has _____ fields, _____ of which are indicated as being key fields. The departure time and arrival time fields are called _____ and _____.

Fill in the blanks to complete the sentence.

Continued on next page

3. How long are the fields for the point of departure (CITYFROM) and the destination (CITYTO)? Why do they have the same technical properties?

Task 2:

Search for other database tables for the flight data model. Take advantage of the fact that the relevant transparent tables are in the same package.

1. In the navigation area of the *Object Navigator*, list all the transparent tables that belong to the same package as SPFLI.
2. In which database table is the flight customer data located? What is the name of the field for the name of a flight customer?

Task 3:

Find out the flight customer number of the customer “SAP AG”. Determine the number of flights for which the customer “SAP AG” has a booking. Find the most immediate flight and determine the point of departure and the destination of this flight and whether the flight is fully booked.

1. Use the *Data Browser* to display the content of the database table SCUSTOM for the customer “SAP AG”.

Continued on next page

2. What is the flight customer number for “SAP AG”?

3. Display the content of the database table SBOOK to determine the number of bookings for the customer “SAP AG” and to find the most immediate flight.
4. Which fields can be used to uniquely identify the corresponding flight?

5. Use the content of the database table SPFLI to determine the point of departure and the destination of the flight that was found.
6. Is the flight fully booked?

Solution 15: Analyzing Transparent Tables in the ABAP Dictionary

Task 1:

Examine the properties of the database table SPFLI and analyze the name and data types of the table fields.

1. Display the relevant transparent table.
 - a) Choose the  *Other Object ...* pushbutton.
 - b) Choose the *Dictionary* tab page.
 - c) Under *Database Table*, enter the table names and choose  *Display*.
2. The table has 16 fields, 3 of which are indicated as being key fields. The departure time and arrival time fields are called DEPTIME and ARRTIME.

Answer: 16, 3, DEPTIME, ARRTIME

3. How long are the fields for the point of departure (CITYFROM) and the destination (CITYTO)? Why do they have the same technical properties?

Answer: The fields are 20 characters in length. They have the same technical properties because they refer to the same domain, S_CITY.

Task 2:

Search for other database tables for the flight data model. Take advantage of the fact that the relevant transparent tables are in the same package.

1. In the navigation area of the *Object Navigator*, list all the transparent tables that belong to the same package as SPFLI.
 - a) You will find the name of the package on the *Properties* tab page for the transparent table (SAPBC_DATAMODEL).
 - b) In the navigation area, open the object list for this package in the usual way. You will find the transparent table under the node *Dictionary Objects* → *Database Tables*.



Hint: You can access the object list faster and without the name of the package by choosing the  *Display Object List* pushbutton from the SPFLI database table display.

Continued on next page

2. In which database table is the flight customer data located? What is the name of the field for the name of a flight customer?

Answer: The flight customer data is located in the table SCUSTOM; the name of the customer is in the table field NAME.

Task 3:

Find out the flight customer number of the customer “SAP AG”. Determine the number of flights for which the customer “SAP AG” has a booking. Find the most immediate flight and determine the point of departure and the destination of this flight and whether the flight is fully booked.

1. Use the *Data Browser* to display the content of the database table SCUSTOM for the customer “SAP AG”.
 - a) Open the transparent table SCUSTOM in the *Object Navigator* as you did before.
 - b) Choose the  *Content* pushbutton to branch to the *Data Browser*.
 - c) Enter the customer name as a selection and choose  *Execute*.
 2. What is the flight customer number for “SAP AG”?
- Answer:** The flight customer number for “SAP AG” is “00000001”.
3. Display the content of the database table SBOOK to determine the number of bookings for the customer “SAP AG” and to find the most immediate flight.
 - a) Proceed in a similar manner to the previous step in order to display the content of the database table SBOOK. Specify the flight customer number to limit the content.
 - b) The number of bookings found (“hits”) is specified in the title bar of the display.
 - c) Sort the display according to the flight date in order to determine the most immediate booking.



Hint: Since the data is regenerated before each training course, no general statement can be made regarding the most immediate flight at a particular point in time.

Continued on next page

4. Which fields can be used to uniquely identify the corresponding flight?
Answer: The necessary key fields are Client (MANDT), Airline ID (CARRID), Connection Number (CONNID) and Flight Date (FLDATE).
5. Use the content of the database table SPFLI to determine the point of departure and the destination of the flight that was found.
 - a) When displaying the data, restrict the airline ID and flight number to the flight that was found previously.
 - b) You will find the point of departure and the destination in the CITYFROM and CITYTO fields, and the departure and arrival airports in the AIRPFROM and AIRPTO fields.
6. Is the flight fully booked?
 - a) Display the content of the database table SFLIGHT.
 - b) Limit the display to the airline ID, flight number and date of the previously determined flight.
 - c) You will find the number of occupied seats and the maximum number of available seats in the SEATSOCC and SEATSMAX fields.



Lesson Summary

You should now be able to:

- Explain the purpose and the benefits of using a data model in application development
- Describe the SAP flight data model
- Describe the meaning and the structure of a transparent table

Lesson: Reading Database Tables

Lesson Overview

In this lesson you will learn how to retrieve information on database tables and how to read data from them. An overview of techniques that allow you to access multiple database tables will also be covered.

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



Lesson Objectives

After completing this lesson, you will be able to:

- List different methods for searching relevant database tables
- Program read access to specific columns and rows within a particular database table
- List different methods for read accesses to several database tables

Business Example

You need to evaluate data from database tables. Since there are no suitable reuse components for accessing the database tables you want to read, you implement them yourself.

Data Retrieval

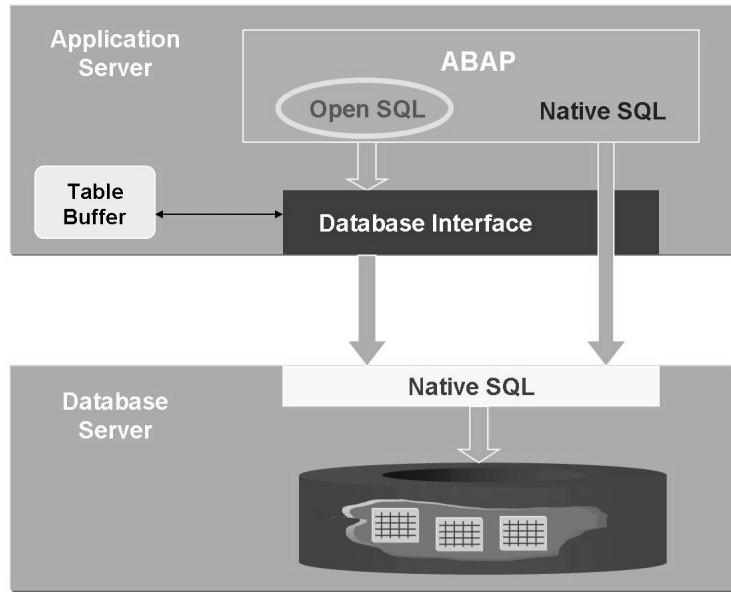


Figure 144: Database Access (Architecture)

SQL is the abbreviation of *Structured Query Language*, a language that enables define, change, and read access to database tables.

Every relational database system has a **native SQL**, which is unfortunately **database-specific**. Hence, an ABAP program with native SQL statements cannot be used without restrictions in all SAP systems (due to the different database systems of different SAP systems).

In contrast **Open SQL** is an SAP-defined, **database-independent** SQL standard for the ABAP language. The Open SQL statements are dynamically converted to the corresponding native SQL statements of the currently used database system and are thus independent of the database. They allow the ABAP programmer uniform access to data, regardless of the database system installed.



Application-Related Search

Search within a particular application component:
Application Hierarchy

Search by Program

Search within a program that accesses the table being searched for:

- Source code search : Search for the SELECT statement in the Editor
- Function debugging : Switch to debugging mode ("h") prior to executing a subfunction and set a breakpoint at the SELECT statement
- Using the screen field information : Display a corresponding structure field using F1 + "Technical Information", navigate to the relevant data element and query a "Where-Used List in Table Fields".

Figure 145: Searching for Database Tables

The above graphic illustrates the options for searching for the required database tables.

Of course, you can also execute a free search via the Repository Information System.

Before you program direct accesses to database tables, you should look for *reuse components* that take care of the read process. The following graphic provides an overview of those read routines supplied by SAP that you can use in your program.

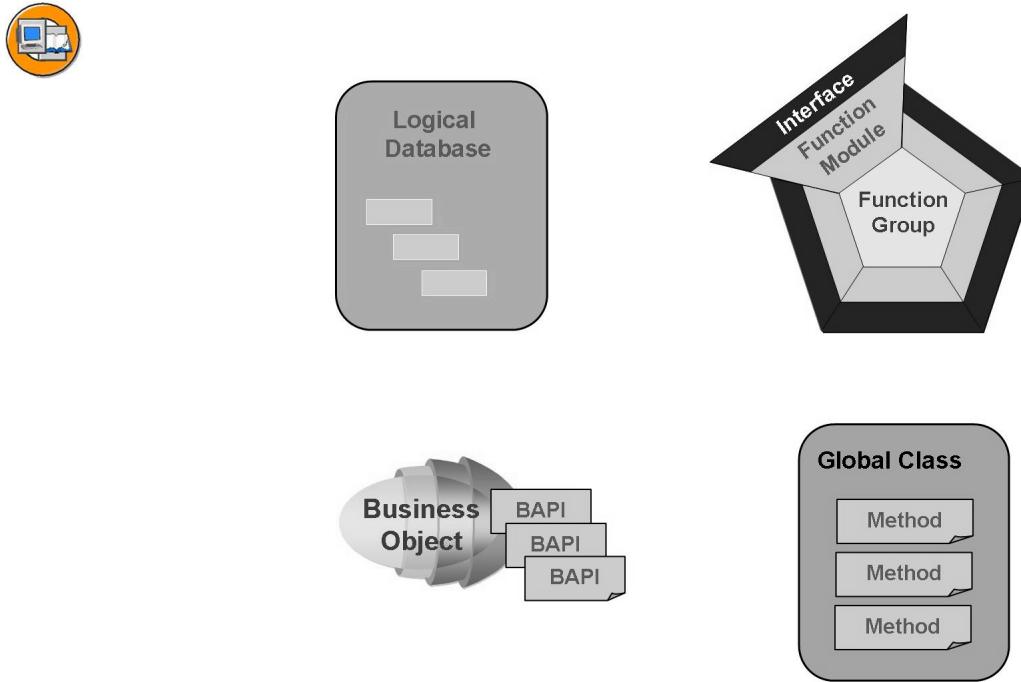


Figure 146: Reuse Components for Data Retrieval

There are four types of reuse components that encapsulate database accesses:

- Logical databases
(= data retrieval programs that read data from tables that belong together hierarchically)
- Function modules
(= subroutines stored in the function library of the SAP system with encapsulated functions, such as reading from hierarchically related tables)
- BAPIs
(= Methods of Business Objects with read function, for example)
- Methods of global classes

For detailed information on searching for and using logical databases, refer to the online documentation for “ABAP programming and runtime environment”, section on “ABAP database access” or course BC405.

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

If there are no useable reuse components available for your data selection, you have to implement the read access yourself. It is recommended that you encapsulate the access itself in a reuse component, in other words, create function modules or methods of global classes.

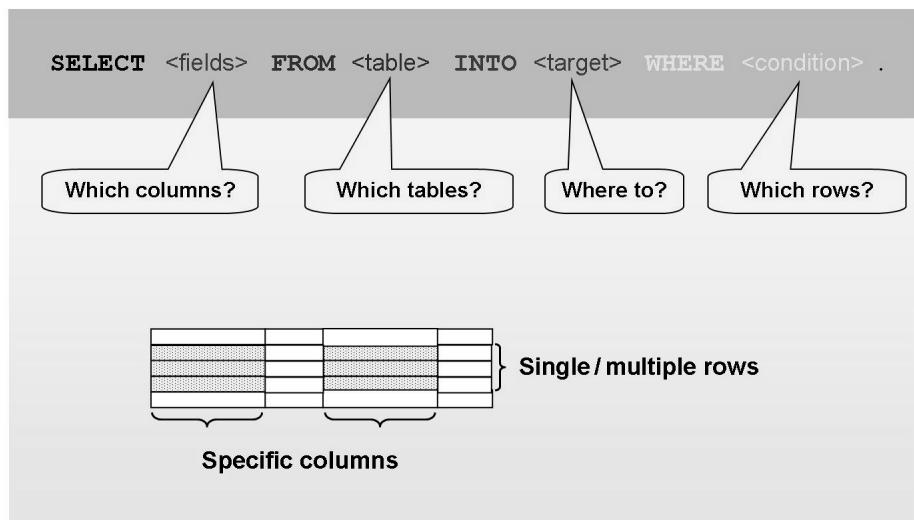


Figure 147: Database Read Access (Overview)

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:

- Amongst other things, the **SELECT** clause describes 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 target variable into which the selected data is to be placed.
- The **WHERE** clause specifies the columns of the table that are to be selected.

For information about other clauses, refer to the keyword documentation for the **SELECT** statement.

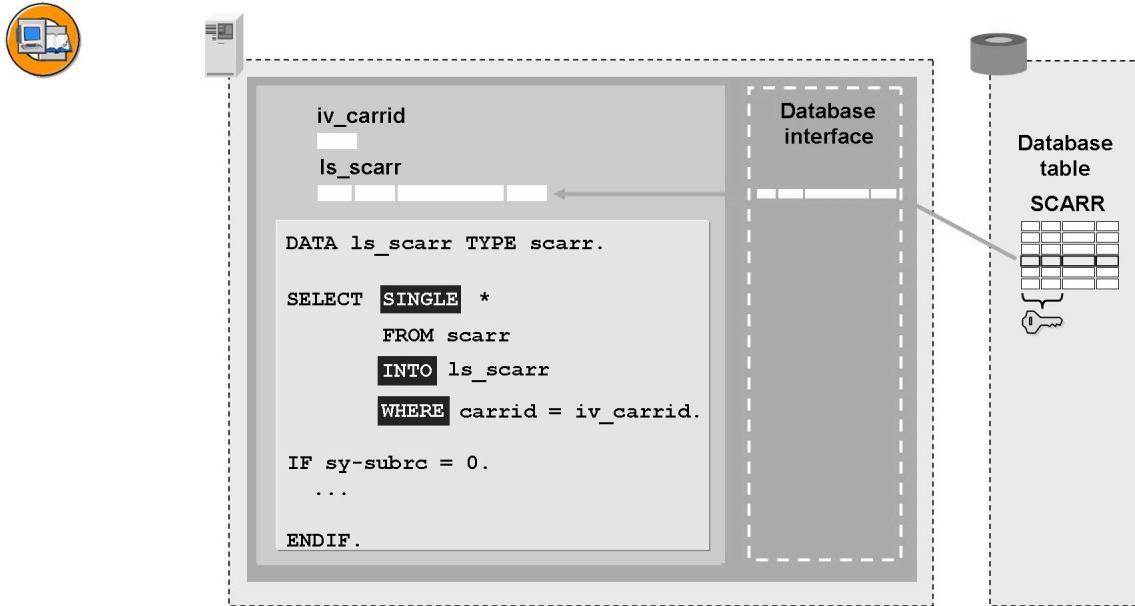


Figure 148: Reading Single Records

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 client field is an exception: If it is not specified the current client is assumed. (Remember that a client can only be specified in the SELECT select statement in combination with the **CLIENT SPECIFIED** addition. More details on this are provided in the course of this lesson.)

You use the asterisk ***** to specify that all fields of the table row to be selected are to be read. If you only want a specific selection of columns, you can list the required fields instead of the *****. The next graphic illustrates this.

You use the **INTO** clause to specify the target variable to which the data record is to be copied. Left-justified, the target area must be structured like the table row or the specified required fields of the row.

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

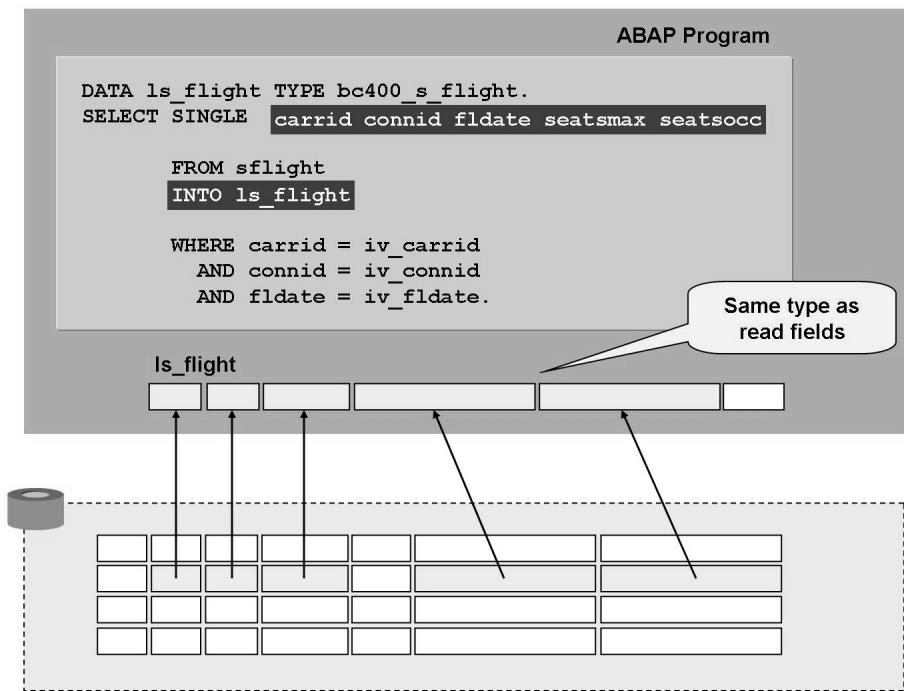


Figure 149: Corresponding Target Structure for the Field List

If you want only a certain selection of fields from the table row to be read, these can be specified as a field list within the `SELECT` statement (as described in the above graphic). You then have to name a **target structure variable** in the `INTO` clause that has the **same structure as the field list** (at least in the beginning), in other words, it contains the fields of the field list in the same order. Only the corresponding field types have to match. The names of the target structure fields are **not** taken into account.

An alternative to specifying the target structure is to list the corresponding **target fields** in the `INTO` clause.

```
INTO (field_1, ... , field_n)
```

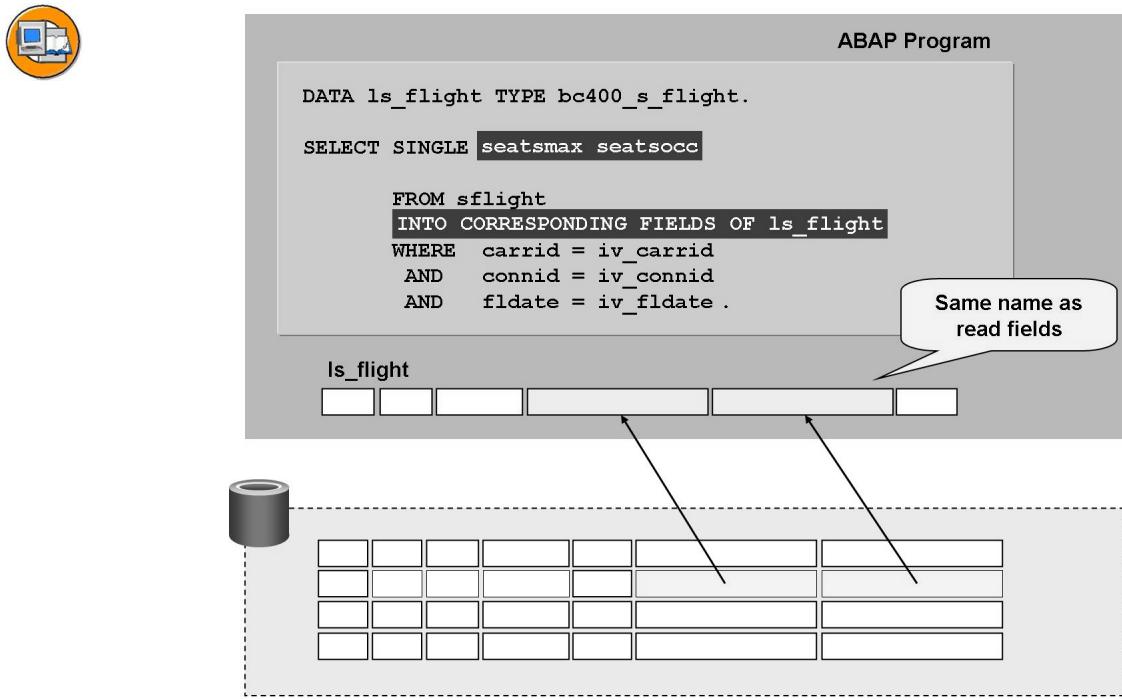


Figure 150: Fields in the Target Structure that have the Same Names as those in the Field List

If you want to use a structure variable for receiving the read record, which has fields with the same names as the fields in the target list but has a different structure (additional fields, different order of fields), it makes sense to use the `CORRESPONDING FIELDS OF` addition. This has the effect that only fields of the same name are filled in the target area. Make sure that the corresponding field types are also the same, otherwise (like in the `MOVE` statement), a (complicated) conversion takes place and it is possible that incomplete data (caused by cut-offs) can be transported to the target fields.

The advantages of this variant are:

- The target structure does not have to be left-justified in the same way as the field list.
- This construction is easy to maintain, since extending the field list or target structure does not require other changes to be made to the program, as long as there is a field in the structure that has the same name (and if possible, the same type as well).

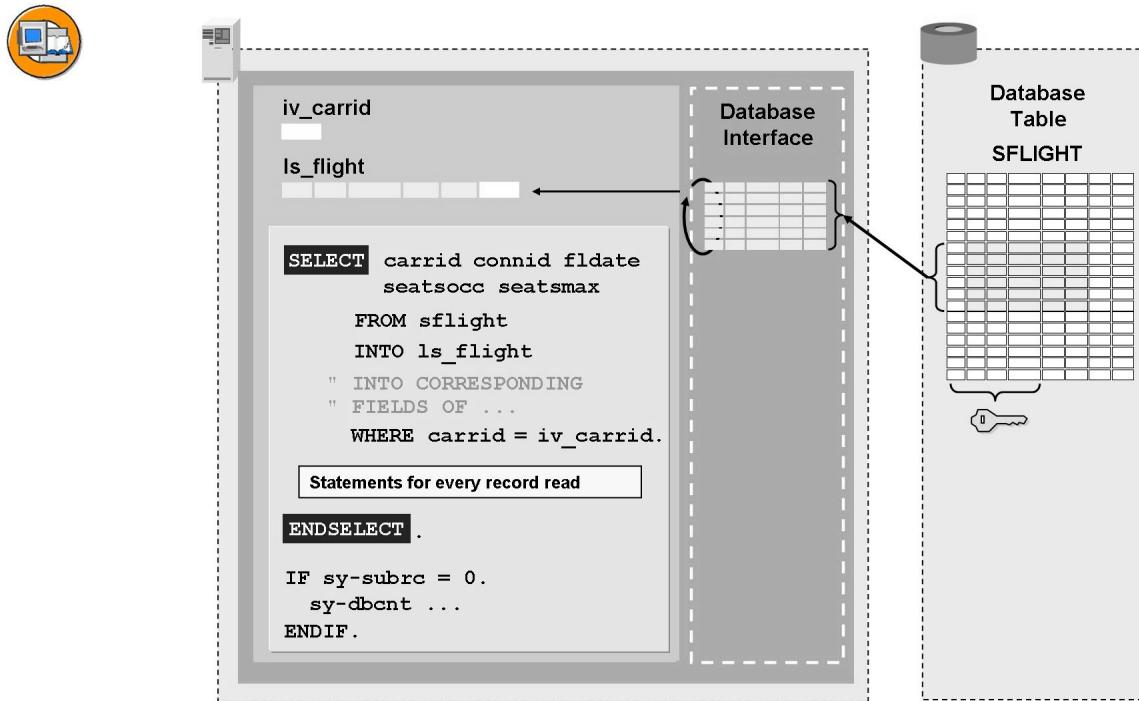


Figure 151: Reading several rows using a loop

You can use the above-illustrated **SELECT loop** to read several rows of a database table into the program in succession.

The WHERE clause determines which lines are read into the target structure and processed using the statement block specified in the loop body. Multiple logical conditions within the WHERE clause can be logically connected using AND or OR.

The database delivers the data to the *database interface* of the application server in packages. The specified processing block then copies the records to the target area row-by-row for processing.

The loop is left automatically once all required rows have been read and evaluated.

The return value should be queried **after** the **SELECT loop** (that is, after the **ENDSELECT** statement). SY-SUBRC has the value 0 if the system read at least one row. In this case, SY-DBCNT contains the number of records read.

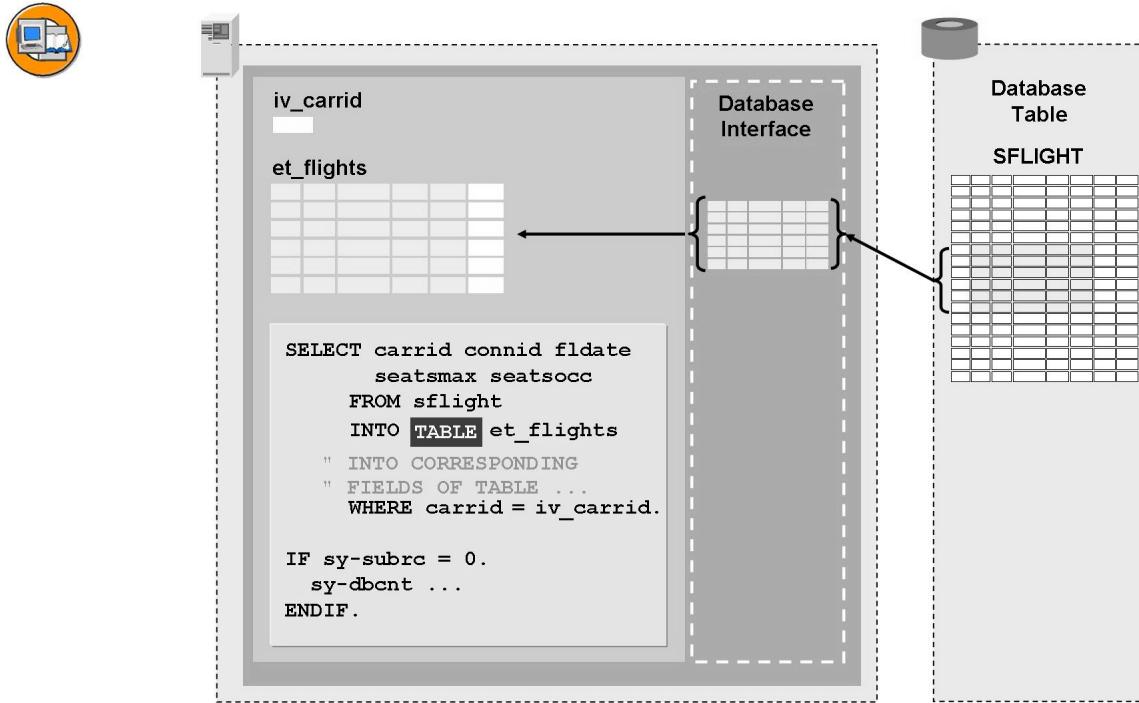


Figure 152: Reading Several Rows Using an Array Fetch

You can use the `INTO TABLE` addition to copy the selected part of the database into an internal table directly instead of doing so row-by-row. This technique is called an **array fetch**. It is a high performance technique for filling an internal table with entries from a database table, as the data transport is realized in blocks and not in rows.

Since an Array Fetch is not a type of loop processing, **no ENDSELECT statement** is required or allowed.

In the same way as the `SELECT` variants discussed before, the internal table that is specified as the target in the Array Fetch must be structured left-justified just like the field list. If an internal table does not meet this prerequisite, you have to use the `INTO CORRESPONDING FIELDS OF TABLE` addition instead of `INTO TABLE`. There, the database columns specified in the field list are copied to the columns of the internal table that have the same names. Here, you also have to make sure that the field types of the corresponding columns match in order to avoid complex conversions and possibly incomplete data in the target table.

With the array fetch, the content that might be in the internal table is overwritten. If you want to append rows instead, you can use the `APPENDING TABLE` addition.

The value of SY-SUBRC is zero if at least one record was copied to the internal table. SY-DBCNT then contains the number of rows read.

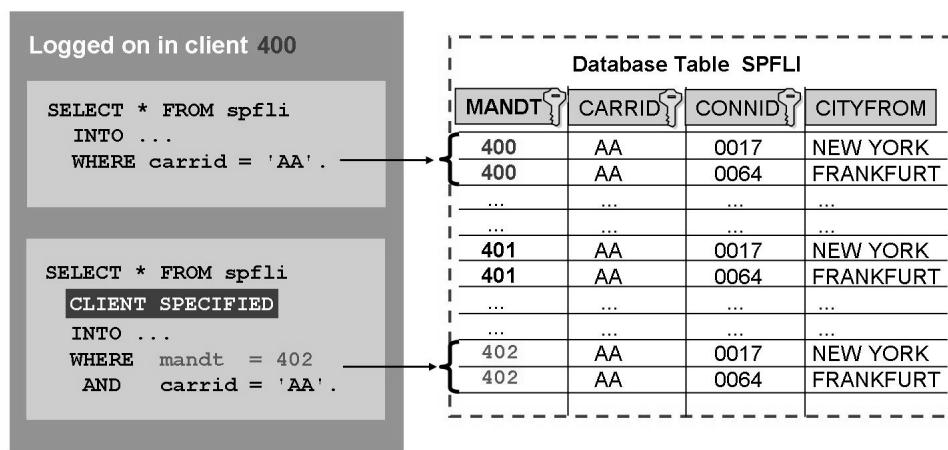


Figure 153: Reading Data From Client-Specific Tables

A database table is referred to as **client-specific** if it has a client field (data type CLNT) as the first key column and contains **client-specific** entries.

If you select data from client-specific tables **without specifying the client**, only data records from the **current client** are read. (The current client is transferred to the database system, where the data retrieval takes place.)

If you want to read data from an explicitly-specified client, you have to specify this client in the WHERE clause. Note, however, that this is only possible when you use the **CLIENT SPECIFIED** addition after the FROM clause.

As cross-client reading is rarely requested in practice and not relevant in the context of this course, the client field is usually omitted in the presentations.

Performance Aspects with Database Access

In most cases, database accesses make considerable demands on the runtime requirement of an ABAP application. In order not to overload the system unnecessarily and keep waiting time for the user to a minimum, you should pay special attention to runtime requirements with database access in particular. A range of technologies are available in Open SQL that enable you to optimize the runtime requirement. Some of them will be discussed in brief in the section that follows.

Secondary Index

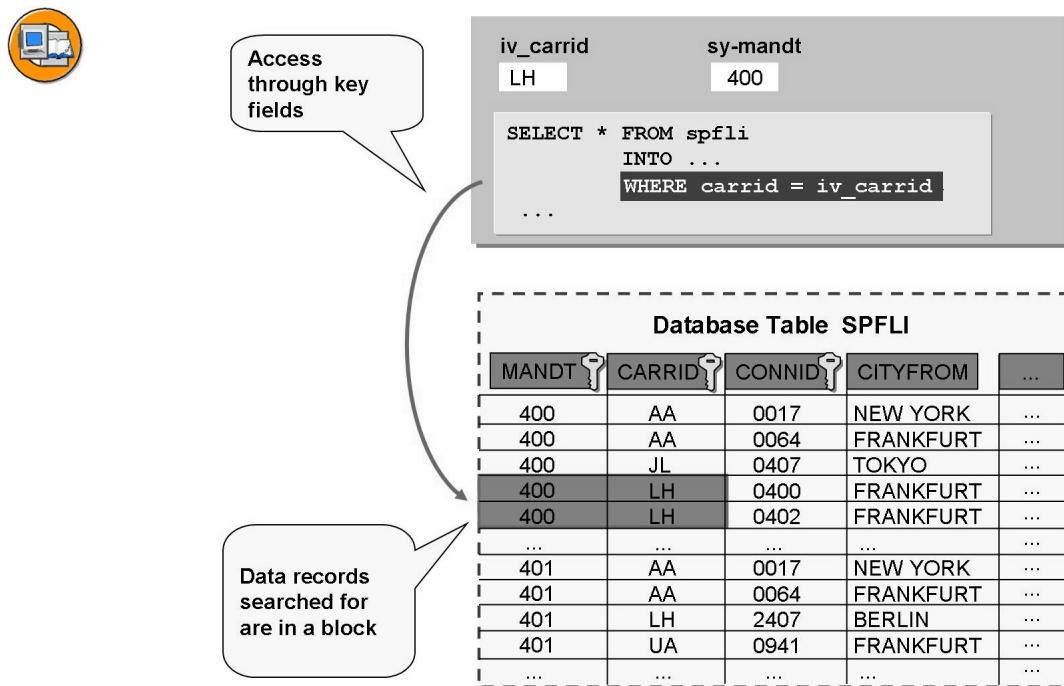


Figure 154: Access Through Key Fields

Every database manages data records within a database table based on the content of key fields. If access to the table is restricted to all, or at least the first few key fields, the database can get hold of the required data records very quickly and effectively.

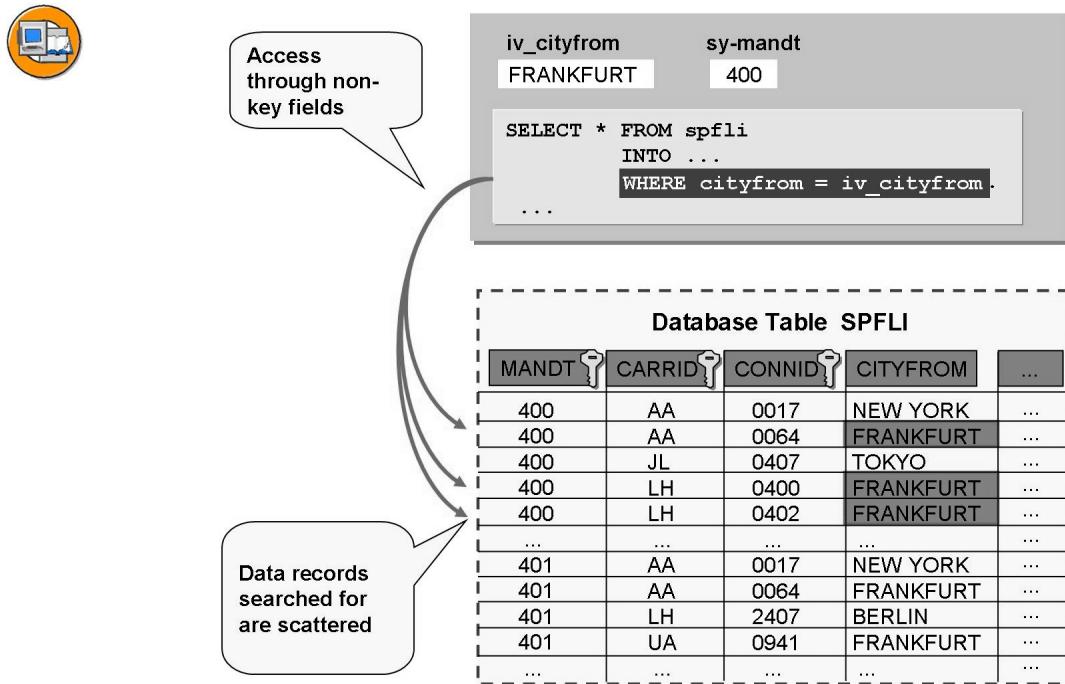


Figure 155: Access Through Non-Key Fields

If the database access is directed at fields that do not belong to the table key (non-key fields), the internal ordering principle cannot be used for rapid access. In the worst case scenario, the entire table, or at least a very large part thereof, has to be searched for the required entries. This is referred to as a **sequential search**. This can lead to very long waiting times for database access.

→ **Note:** In the illustration, the data records within the database table are sorted according to key field and stored on this basis. Some database systems actually work in this way. With others, the data records themselves are not sorted. Instead, an index is created using the key fields. This index is referred to as the **primary index**. This is of no significance for our considerations here.

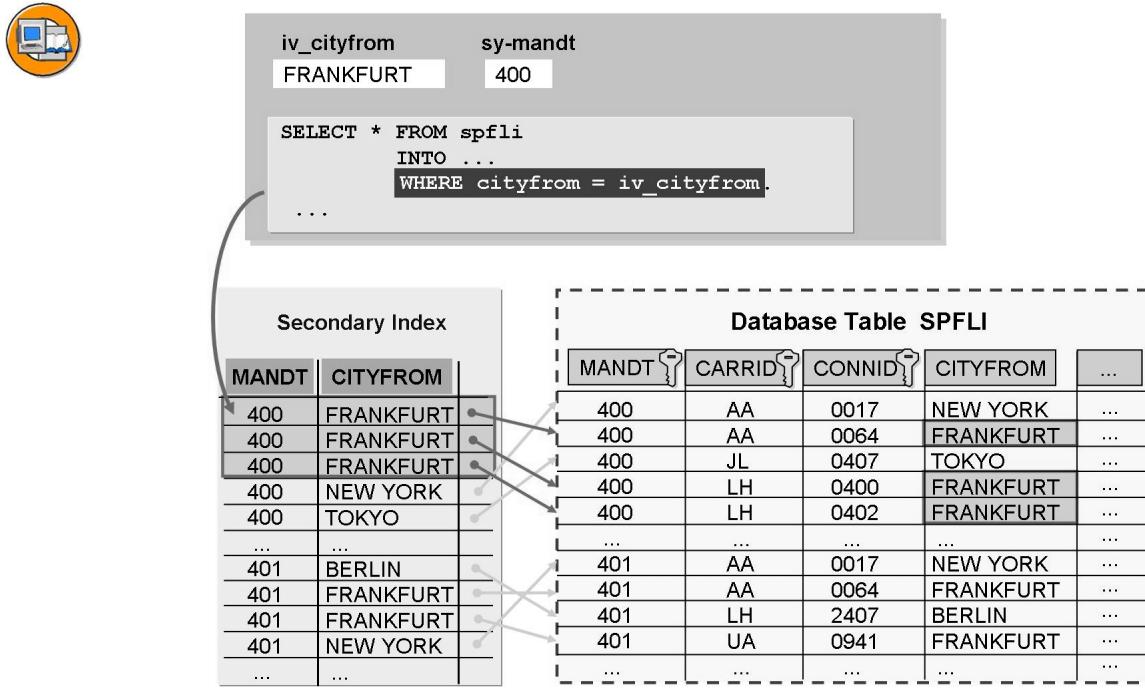


Figure 156: Access Through Secondary Index

If a database table is frequently accessed using a certain selection, you should define a **secondary index** for the fields contained in the selection in order to speed up corresponding accesses.

The secondary index is a separate, small table that contains the content of the index fields and a reference to the relevant data records for each data record in the actual database table. If the fields in the selection match the fields in the secondary index (at least left-aligned and gap free), the database first searches within the secondary index and then reads the corresponding data records through the reference.

You create a secondary index in display mode for the relevant transparent table in the *ABAP Dictionary* using the *Indexes* pushbutton. When you activate the index, the secondary index is then created on the database.

Whether and how an existing secondary index is used in a database access depends on a function in the database system known as the *Database Optimizer*. In Open SQL it is neither possible nor necessary to trigger the use of a secondary index by specifying this explicitly in the SELECT statement.



Hint: With selections from client-specific tables, the client is always transmitted to the database (that is, it is always part of the selection). Hence, it makes sense to include the client field when you define an index for such tables too. At runtime, this can be used to restrict the database search to the relevant client block by means of the secondary index.

SAP Table Buffer

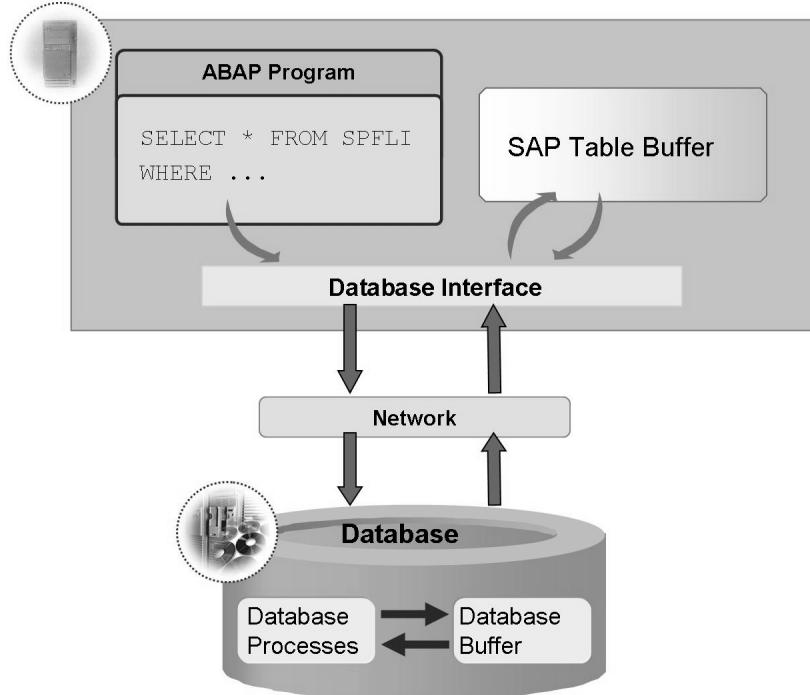


Figure 157: Access Through the SAP Table Buffer

With data retrieval, a major proportion of runtime is needed to transfer the data from the database server to the application server. If the data is read frequently but seldom changed, you can reduce runtime by buffering the data on the application server.

Whether and how data can be buffered is a decision that is made separately for each database table. It depends on how frequently the data is read or changed. The buffer settings are defined in the transparent tables in the *ABAP Dictionary (Technical Settings* pushbutton).



Caution: The decision as to whether a database table can be buffered is not that simple and should be made by experienced ABAP developers in consultation with the system administrator. The SAP table buffer exists separately for each application server. If a system comprises several applications, a special synchronization mechanism ensures that after changes have been made on the database, the corresponding buffer contents are invalidated. However, time lags in the synchronization process means that for a short period of time, old data could be read from the buffer. This has to be taken into account when a decision about buffering is made.

If an ABAP program reads a buffered table, the database interface first tries to get the required data from the *SAP table buffer*. This speeds up access by a factor of between 10 and 100 in contrast to reading data from the database. The precise increase in speed depends on the structure of the table and on the exact system configuration.

If the required data is not yet in the *SAP table buffer*, the database is accessed. The read data is then transferred from the database interface and stored in the SAP table buffer.



Hint: There are variants and additions for the SELECT statement that cause data to be read directly from the database, regardless of the buffer settings. Such access is said to **bypass the buffer**. This kind of access can pose performance problems with buffered tables and should be avoided.

Reading from Several Database Tables

There is often a requirement to read data from different tables and display it.

In general, the technique with the best performance is the implementation using a **table join**.

Let's read and output SPFLI records as an example. However, for each record the long name of the respective airline, which is stored in SCARR, is to be output as well. The following graphic shows the logical creation of the corresponding table join, from which you can select all the required data using the SELECT statement.

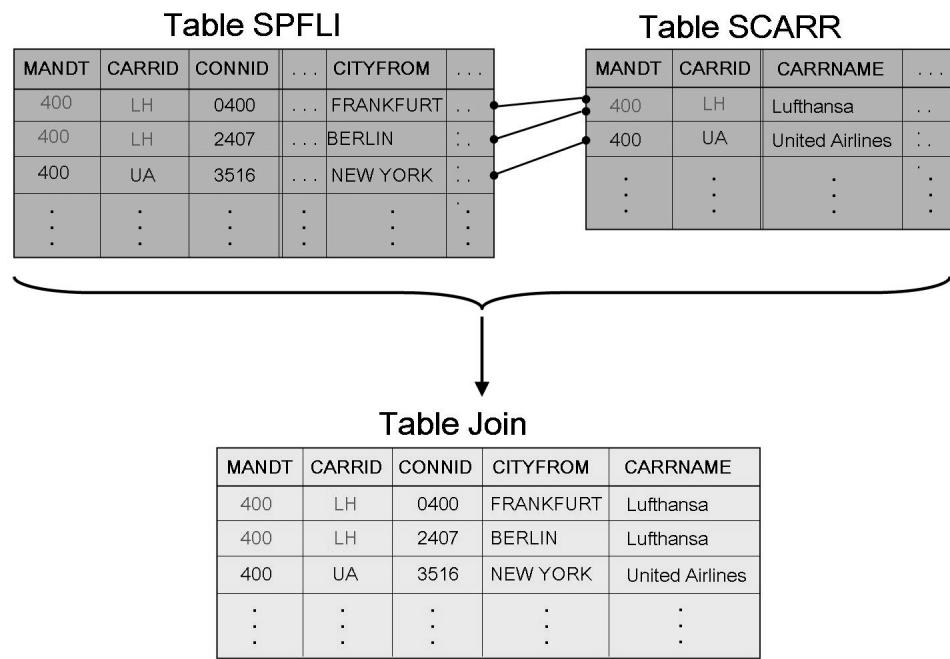


Figure 158: Example of a Table Join

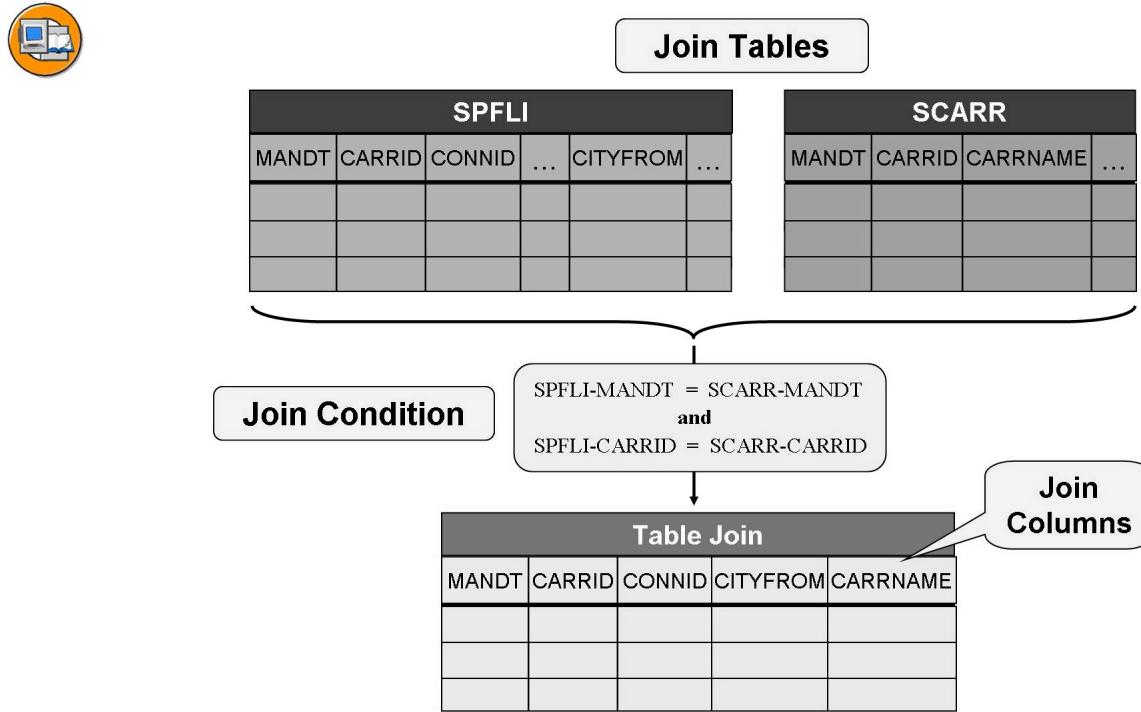


Figure 159: Entries for Defining a Table Join

You have to specify three things when defining a table join:

- **Join tables**
Which database tables should be accessed?
- **Join condition**
What is the condition under which the corresponding records from the join tables are summarized to a join record?
- **Join columns**
Which columns from the join tables should be available in the table join?

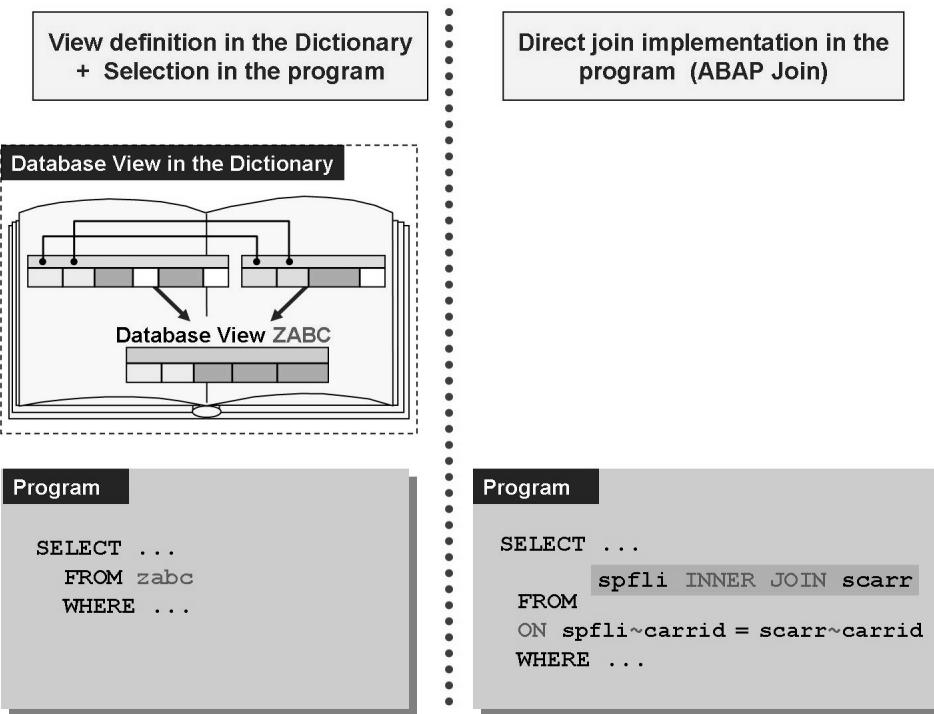


Figure 160: Implementation Options for Table Joins

There are two options for implementing a table join:

- In the *ABAP Dictionary* you create a **database view** that corresponds to a table join and select from it in your program.
For detailed information, refer to the online documentation for the *ABAP Workbench*, in the *ABAP Dictionary* section.
- In your program, you select from a join that is defined there (**ABAP join**). At runtime, the system dynamically generates an appropriate database query in the database interface.
For more information, see the keyword documentation for the FROM clause of the SELECT statement.



Hint: A table join is a view of the relevant database tables and does **not** contain the corresponding data redundantly. With a selection from a table join the data is read from the corresponding database tables.

You can join more than two tables in a join.

Database Change Accesses (Preview)

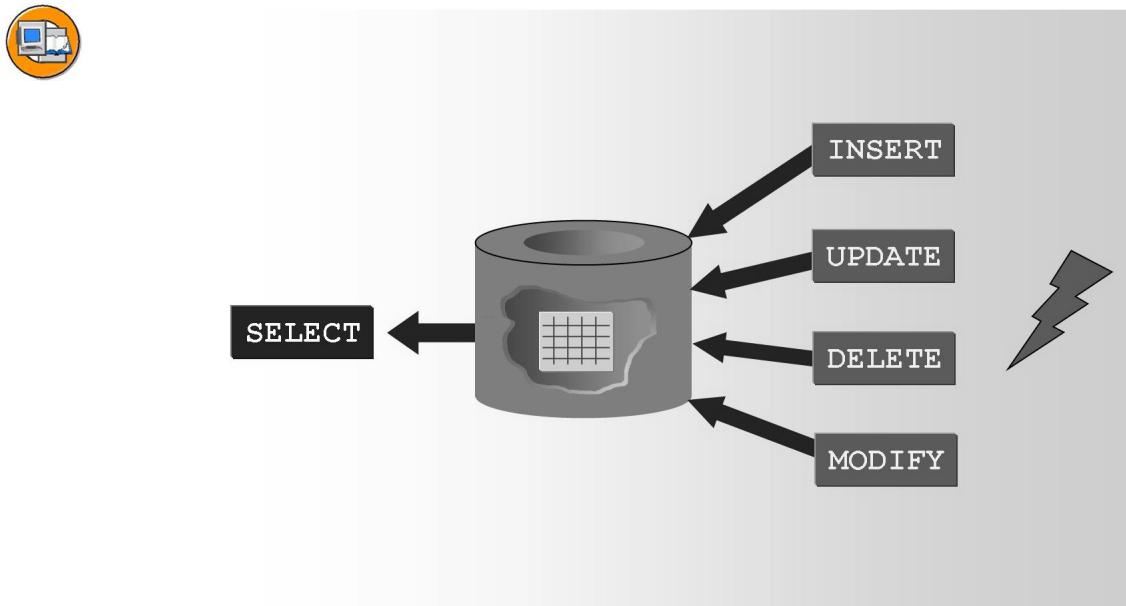


Figure 161: Database Change Accesses

In addition to the SELECT statement, Open SQL also contains the UPDATE, INSERT, DELETE, and MODIFY statements. However, the database change accesses that are possible in this way should **not** be used without knowing the SAP transaction concept as you might otherwise cause **data inconsistencies**.

The SAP transaction concept is taught in course BC414.

Exercise 16: Data Retrieval by Single Record Access

Exercise Objectives

After completing this exercise, you will be able to:

- Create a function group
- Create a function module
- Read from the database using single record access

Business Example

You want to develop a program in which data for a single flight connection is read from the database. You encapsulate the required database access by single record access in a function module, so that you can reuse it.

Template:

None

Solution:

BC400_DDS (function group)

BC400_DDS_CONNECTION_GET (function module)

Task 1:

Create a function group.

1. Create a new function group in your package (suggested name: **ZBC400_##**).

Task 2:

Create a new function module in the function group. Maintain the interface for the function module in such a way that it imports an airline ID and flight connection number and returns flight connection data. Create an exception too, for when no data can be found.

1. Create a new function module in your function group (suggested name: **Z_BC400_##_CONNECTION_GET**).
2. Create an export parameter for the detailed data for a flight connection (suggested name: **ES_CONNECTION**). Type the parameter with Dictionary structure type BC400_S_CONNECTION.

Continued on next page

3. Create a separate import parameter for the airline ID and connection number (suggested names: **IV_CARRID** and **IV_CONNID**). For typing, choose suitable components from the Dictionary structure type **BC400_S_CONNECTION**.
4. Create a (classic) exception (suggested name: **NO_DATA**).

Task 3:

In the source code of the function module, implement a single record access to database table SPFLI. Use the interface parameters of the function module to do this. Raise an exception if no data is found for the selection.

1. Implement a single record access (SELECT SINGLE) to database table SPFLI that fills the export parameter of the function module with values. List all the table fields that appear as components in the export parameter. Make sure that the order of the fields matches the order of the structure components.
2. Apply a selection condition (WHERE clause) to the SELECT statement. Use both import parameters of the function module to do this.
3. Raise the exception for the function module if the database access returns no data (sy-subrc \neq 0).

Task 4:

Activate and test your function module.

1. Activate the function module.
2. Test your function module.

Solution 16: Data Retrieval by Single Record Access

Task 1:

Create a function group.

1. Create a new function group in your package (suggested name: **ZBC400_##**).
 - a) In the navigation area, open the context menu for the package and choose *Create → Function Group*.
 - b) Enter the name of the function group and a short text and choose *Save*.
 - c) Assign the function group to your package and your request in the usual way.

Task 2:

Create a new function module in the function group. Maintain the interface for the function module in such a way that it imports an airline ID and flight connection number and returns flight connection data. Create an exception too, for when no data can be found.

1. Create a new function module in your function group (suggested name: **Z_BC400_##_CONNECTION_GET**).
 - a) In the navigation area, open the context menu for your function group and choose *Create → Function Module*.
 - b) Enter the name of the function module and a short text and choose *Save*.
2. Create an export parameter for the detailed data for a flight connection (suggested name: **ES_CONNECTION**). Type the parameter with Dictionary structure type BC400_S_CONNECTION.
 - a) Open the *Export* tab page. Enter the name of the parameter and the data type.
3. Create a separate import parameter for the airline ID and connection number (suggested names: **IV_CARRID** and **IV_CONNID**). For typing, choose suitable components from the Dictionary structure type BC400_S_CONNECTION.
 - a) Open the *Import* tab page. Enter the names of the parameters and the data types.

Continued on next page

4. Create a (classic) exception (suggested name: **NO_DATA**).
 - a) Open the *Exceptions* tab page. Enter the name of the exception and a short text. You must not select the *Exception Classes* field.

Task 3:

In the source code of the function module, implement a single record access to database table SPFLI. Use the interface parameters of the function module to do this. Raise an exception if no data is found for the selection.

1. Implement a single record access (SELECT SINGLE) to database table SPFLI that fills the export parameter of the function module with values. List all the table fields that appear as components in the export parameter. Make sure that the order of the fields matches the order of the structure components.
 - a) See source code extract from the model solution.
2. Apply a selection condition (WHERE clause) to the SELECT statement. Use both import parameters of the function module to do this.
 - a) See source code extract from the model solution.
3. Raise the exception for the function module if the database access returns no data (sy-subrc \neq 0).
 - a) See source code extract from the model solution.

Task 4:

Activate and test your function module.

1. Activate the function module.
 - a) In the *Function Builder* toolbar, choose the  pushbutton.

Continued on next page

2. Test your function module.
 - a) In the *Function Builder* toolbar, choose the  pushbutton.
 - b) Supply the import parameters with values and choose .
 - c) Check the result.



Hint: Check the raising of the exception if an error occurs too.

Result

Source code extract from the model solution:

```

FUNCTION BC400_DDS_CONNECTION_GET .
*"-"
**" Lokale Schnittstelle:
*"
* IMPORTING
*   REFERENCE(IV_CARRID) TYPE BC400_S_FLIGHT-CARRID
*   REFERENCE(IV_CONNID) TYPE BC400_S_FLIGHT-CONNID
* EXPORTING
*   REFERENCE(ES_CONNECTION) TYPE BC400_S_CONNECTION
* EXCEPTIONS
*   NO_DATA
*"-"

SELECT SINGLE carrid connid cityfrom airpfom
      cityto airpto fltime deptime arrtime
FROM spfli
INTO es_connection
WHERE carrid = iv_carrid
  AND connid = iv_connid.

IF sy-subrc <> 0.
  RAISE no_data.
ENDIF.

ENDFUNCTION.

```


Exercise 17: Data Retrieval Using a SELECT Loop

Exercise Objectives

After completing this exercise, you will be able to:

- Read from the database using a select loop
- Fill an internal table by record

Business Example

You want to develop a program in which all the flight times for a single flight connection is read from the database. You encapsulate the required database access by select loop in a function module, so that you can reuse it.

Template:

None

Solution:

BC400_DDS (function group)

BC400_DDS_FLIGHTLIST_GET (function module)

Task 1:

Create a function group, if one does not already exist.

1. Create a new function group in your package (suggested name: **ZBC400_##**).

Task 2:

Create a new function module in the function group. Maintain the interface for the function module in such a way that it imports an airline ID and flight connection number and returns a list of flight times. Create an exception too, for when no data can be found.

1. Create a new function module in your function group (suggested name: **Z_BC400_##_FLIGHTLIST_GET**).
2. Create an export parameter for the list of flight dates (suggested name: **ET_FLIGHTS**). Type the parameter with Dictionary table type BC400_T_FLIGHTS.

Continued on next page

3. Create a separate import parameter for the airline ID and connection number (suggested names: **IV_CARRID** and **IV_CONNID**). For typing, choose suitable components of the structure type BC400_S_FLIGHT (line type of the table type BC400_T_FLIGHTS).
4. Create a (classic) exception (suggested name: **NO_DATA**).

Task 3:

In the source code of the function module, implement a select loop to database table SFLIGHT. Make sure that the export parameter of the function module is filled with a new row in each loop pass. Raise an exception if no data is found for the selection.

1. Declare a local, structured data object in the function module (suggested name: **ls_flight**) that can be used as a work area for the export parameter (internal table).
2. Implement a select loop on database table SFLIGHT that fills the local data object with values on every loop pass. List all the table fields that appear as components in the line type of the export parameter. Make sure that the order of the fields matches the order of the structure components.
3. Apply a selection condition (WHERE clause) to the SELECT statement. Use both import parameters of the function module to do this.
4. Use the local data object to add a new row to the export parameter for each loop pass (APPEND statement).

Calculate the utilization for each data record as a percentage beforehand within the loop (structure component PERCENTAGE). To do this, use the maximum number of seats and number of occupied seats (structure components SEATSMAX and SEATSOCC).

5. Make sure that the export parameter does not contain any data before the first loop pass.



Hint: If you set call by value for the export parameter, the parameter is initial before each function module call. With table type parameters, call by reference is recommended for performance reasons, however. In this case, the export parameter can already contain data if the function module is called with a non-initial actual parameter.

6. Raise the exception for the function module if the database access returns no data (sy-subrc \neq 0).

Continued on next page

7. (OPTIONAL) After the select statement has been called successfully, sort the flight list in descending order according to the percentage utilization before the data is returned.

Task 4:

Activate and test your function module.

1. Activate the function module.
2. Test your function module.

Solution 17: Data Retrieval Using a SELECT Loop

Task 1:

Create a function group, if one does not already exist.

1. Create a new function group in your package (suggested name: **ZBC400_##**).
 - a) In the navigation area, open the context menu for the package and choose *Create → Function Group*.
 - b) Enter the name of the function group and a short text and choose *Save*.
 - c) Assign the function group to your package and your request in the usual way.

Task 2:

Create a new function module in the function group. Maintain the interface for the function module in such a way that it imports an airline ID and flight connection number and returns a list of flight times. Create an exception too, for when no data can be found.

1. Create a new function module in your function group (suggested name: **Z_BC400_##_FLIGHTLIST_GET**).
 - a) In the navigation area, open the context menu for your function group and choose *Create → Function Module*.
 - b) Enter the name of the function module and a short text and choose *Save*.
2. Create an export parameter for the list of flight dates (suggested name: **ET_FLIGHTS**). Type the parameter with Dictionary table type BC400_T_FLIGHTS.
 - a) Open the *Export* tab page. Enter the name of the parameter and the data type.
3. Create a separate import parameter for the airline ID and connection number (suggested names: **IV_CARRID** and **IV_CONNID**). For typing, choose suitable components of the structure type BC400_S_FLIGHT (line type of the table type BC400_T_FLIGHTS).
 - a) Open the *Import* tab page. Enter the names of the parameters and the data types.

Continued on next page

4. Create a (classic) exception (suggested name: **NO_DATA**).
 - a) Open the *Exceptions* tab page. Enter the name of the exception and a short text. Do not select the *Exception Classes* field.

Task 3:

In the source code of the function module, implement a select loop to database table SFLIGHT. Make sure that the export parameter of the function module is filled with a new row in each loop pass. Raise an exception if no data is found for the selection.

1. Declare a local, structured data object in the function module (suggested name: **ls_flight**) that can be used as a work area for the export parameter (internal table).
 - a) Declare the data object either with direct reference to the line type (Dictionary structure type BC400_S_FLIGHT) or relative to the export parameter (LIKE LINE OF). See source code extract from the model solution.
2. Implement a select loop on database table SFLIGHT that fills the local data object with values on every loop pass. List all the table fields that appear as components in the line type of the export parameter. Make sure that the order of the fields matches the order of the structure components.
 - a) See source code extract from the model solution.
3. Apply a selection condition (WHERE clause) to the SELECT statement. Use both import parameters of the function module to do this.
 - a) See source code extract from the model solution.
4. Use the local data object to add a new row to the export parameter for each loop pass (APPEND statement).

Calculate the utilization for each data record as a percentage beforehand within the loop (structure component PERCENTAGE). To do this, use the maximum number of seats and number of occupied seats (structure components SEATSMAX and SEATSOCC).

 - a) See source code extract from the model solution.

Continued on next page

5. Make sure that the export parameter does not contain any data before the first loop pass.



Hint: If you set call by value for the export parameter, the parameter is initial before each function module call. With table type parameters, call by reference is recommended for performance reasons, however. In this case, the export parameter can already contain data if the function module is called with a non-initial actual parameter.

- a) CLEAR statement. See source code extract from the model solution.
6. Raise the exception for the function module if the database access returns no data (sy-subrc \neq 0).
 - a) See source code extract from the model solution.
7. (OPTIONAL) After the select statement has been called successfully, sort the flight list in descending order according to the percentage utilization before the data is returned.
 - a) SORT statement. See source code extract from the model solution.

Task 4:

Activate and test your function module.

1. Activate the function module.
 - a) In the *Function Builder* toolbar, choose the pushbutton.
2. Test your function module.
 - a) In the *Function Builder* toolbar, choose the pushbutton.
 - b) Supply the import parameters with values and choose .
 - c) Check the result.



Hint: Check the raising of the exception if an error occurs too.

Result

Source code extract from the model solution:

```
FUNCTION bc400_dds_flightlist_get .  
*-----
```

Continued on next page

```

*** "Lokale Schnittstelle:
**  IMPORTING
**    REFERENCE(IV_CARRID) TYPE BC400_S_FLIGHT-CARRID
**    REFERENCE(IV_CONNID) TYPE BC400_S_FLIGHT-CONNID
**  EXPORTING
**    REFERENCE(ET_FLIGHTS) TYPE BC400_T_FLIGHTS
**  EXCEPTIONS
**    NO_DATA
**-----


DATA ls_flight TYPE bc400_s_flight.

CLEAR et_flights.

SELECT carrid connid fldate seatsmax seatsocc
      FROM sflight
      INTO ls_flight
      WHERE carrid = iv_carrid
            AND connid = iv_connid.

ls_flight-percentage = ls_flight-seatsocc / ls_flight-seatsmax * 100.
APPEND ls_flight TO et_flights.
ENDSELECT.

IF sy-subrc <> 0.
  RAISE no_data.
ELSE.
  SORT et_flights BY percentage DESCENDING.
ENDIF.

ENDFUNCTION.

```


Exercise 18: (Optional) Data Retrieval by Mass Access (Array Fetch)

Exercise Objectives

After completing this exercise, you will be able to:

- Read from the database using Array Fetch
- Fill an internal table completely

Business Example

You want to develop a program in which all the flight times for a single flight connection is read from the database. You encapsulate the required database access by Array Fetch in a function module, so that you can reuse it.

Template:

BC400_DDS (function group)

BC400_DDS_FLIGHTLIST_GET (function module)

Solution:

BC400_DDS (function group)

BC400_DDS_FLIGHTLIST_GET_OPT (function module)

Task 1:

Create a function group, if one does not already exist.

1. Create a new function group in your package (suggested name: **ZBC400_##**).

Task 2:

Copy the function module BC400_DDS_FLIGHTLIST_GET from function group BC400_DDS or your own function module Z_BC400_##_FLIGHTLIST_GET under a new name to your own function module.

1. Copy the function module (new name:
Z_BC400_##_FLIGHTLIST_GET_OPT)

Continued on next page

Task 3:

First, turn the SELECT loop into a comment and replace it with a database access in which the export parameter of the function module (internal table) is filled directly with data (Array Fetch). Then implement a loop through the internal table to calculate the utilization percentage for each flight time.

1. Turn the entire SELECT loop from SELECT to ENDSELECT into a comment.
2. Replace the SELECT loop with an Array Fetch, which fills the export parameter of the function module directly. Use the same field list and WHERE condition as for the SELECT loop.
3. Implement a loop through the internal table (LOOP ... ENDLOOP). The loop should only be executed if the database access has delivered data. Use the local structured data object as the work area and calculate the utilization percentage within the loop as before.
4. Use the MODIFY statement to write the changed line back to the internal table within the loop after the calculation.

Use the TRANSPORTING percentage addition to update only the changed field.

Use the INDEX sy-tabix addition to change the current line.



Hint: The INDEX sy-tabix addition is optional within a LOOP. To make your source code more readable, however, you should not use it.

Task 4:

Activate and test your function module.

1. Activate the function module.
2. Test your function module.

Solution 18: (Optional) Data Retrieval by Mass Access (Array Fetch)

Task 1:

Create a function group, if one does not already exist.

1. Create a new function group in your package (suggested name: **ZBC400_##**).
 - a) In the navigation area, open the context menu for the package and choose *Create → Function Group*.
 - b) Enter the name of the function group and a short text and choose *Save*.
 - c) Assign the function group to your package and your request in the usual way.

Task 2:

Copy the function module BC400_DDS_FLIGHTLIST_GET from function group BC400_DDS or your own function module Z_BC400_##_FLIGHTLIST_GET under a new name to your own function module.

1. Copy the function module (new name: **Z_BC400_##_FLIGHTLIST_GET_OPT**)
 - a) In the navigation area, display the function group that contains the function module to be copied.
 - b) Open the context menu for the function module and choose *Copy*.
 - c) Enter the name of the new function module as well as the function group in which it should be created, and choose *Copy*.

Continued on next page

Task 3:

First, turn the SELECT loop into a comment and replace it with a database access in which the export parameter of the function module (internal table) is filled directly with data (Array Fetch). Then implement a loop through the internal table to calculate the utilization percentage for each flight time.

1. Turn the entire SELECT loop from SELECT to ENDSELECT into a comment.
 - a) Select the corresponding area in the editor.
 - b) Open the context menu on this area and, depending on which editor is set, choose either *Comment* or *Format → Comment Lines*.

Alternatively, you can use the key combination *Ctrl + <* in both editors.
2. Replace the SELECT loop with an Array Fetch, which fills the export parameter of the function module directly. Use the same field list and WHERE condition as for the SELECT loop.
 - a) See source code excerpt from the model solution.
3. Implement a loop through the internal table (LOOP ... ENDLOOP). The loop should only be executed if the database access has delivered data. Use the local structured data object as the work area and calculate the utilization percentage within the loop as before.
 - a) See source code excerpt from the model solution.
4. Use the MODIFY statement to write the changed line back to the internal table within the loop after the calculation.

Use the TRANSPORTING percentage addition to update only the changed field.

Use the INDEX sy-tabix addition to change the current line.



Hint: The INDEX sy-tabix addition is optional within a LOOP. To make your source code more readable, however, you should not use it.

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

Continued on next page

Task 4:

Activate and test your function module.

1. Activate the function module.
 - a) In the *Function Builder* toolbar, choose the  pushbutton.
2. Test your function module.
 - a) In the *Function Builder* toolbar, choose the  pushbutton.
 - b) Supply the import parameters with values and choose .
 - c) Check the result.



Hint: Check the raising of the exception if an error occurs too.

Result

Source code extract from the model solution:

```
FUNCTION bc400_dds_flightlist_get_opt .
*"-"
*""* "Lokale Schnittstelle:
*"" IMPORTING
*""     REFERENCE(IV_CARRID) TYPE BC400_S_FLIGHT-CARRID
*""     REFERENCE(IV_CONNID) TYPE BC400_S_FLIGHT-CONNID
*"" EXPORTING
*""     REFERENCE(ET_FLIGHTS) TYPE BC400_T_FLIGHTS
*"" EXCEPTIONS
*""     NO_DATA
*"-"

DATA ls_flight TYPE bc400_s_flight.

*   SELECT carrid connid fldate seatsmax seatsocc
*       FROM sflight
*       INTO ls_flight
*       WHERE carrid = iv_carrid
*             AND connid = iv_connid.
*
*   ls_flight-percentage = ls_flight-seatsocc / ls_flight-seatsmax * 100.
*   APPEND ls_flight TO et_flights.
*   ENDSELECT.
```

Continued on next page

```
SELECT carrid connid fldate seatsmax seatsocc
      FROM sflight
      INTO TABLE et_flights
     WHERE carrid = iv_carrid
       AND connid = iv_connid.

IF sy-subrc <> 0.
  RAISE no_data.
ELSE.
  LOOP AT et_flights INTO ls_flight.
    ls_flight-percentage = ls_flight-seatsocc / ls_flight-seatsmax * 100.
    MODIFY et_flights FROM ls_flight
      INDEX sy-tabix
      TRANSPORTING percentage.
  ENDLOOP.

  SORT et_flights BY percentage DESCENDING.

ENDIF.
ENDFUNCTION.
```



Lesson Summary

You should now be able to:

- List different methods for searching relevant database tables
- Program read access to specific columns and rows within a particular database table
- List different methods for read accesses to several 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

Critical data and parts of the functional scope of the SAP System must be protected from unauthorized access. You have to implement authorization checks in your program so that the user may only access areas for which he or she is authorized. The following graphic illustrates the **SAP authorization concept**.

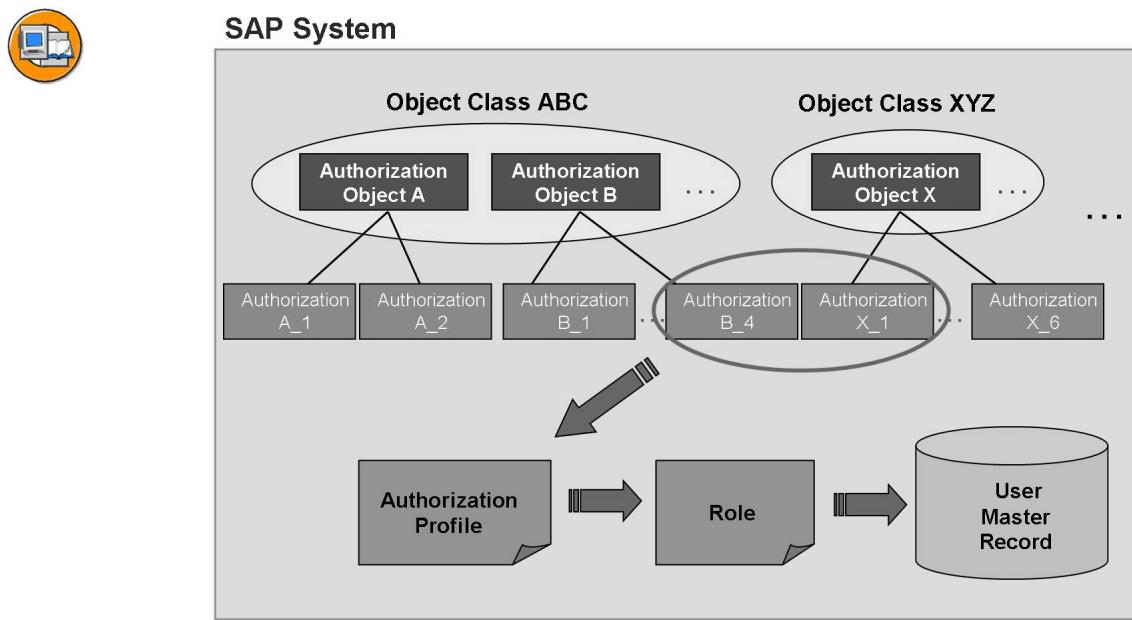
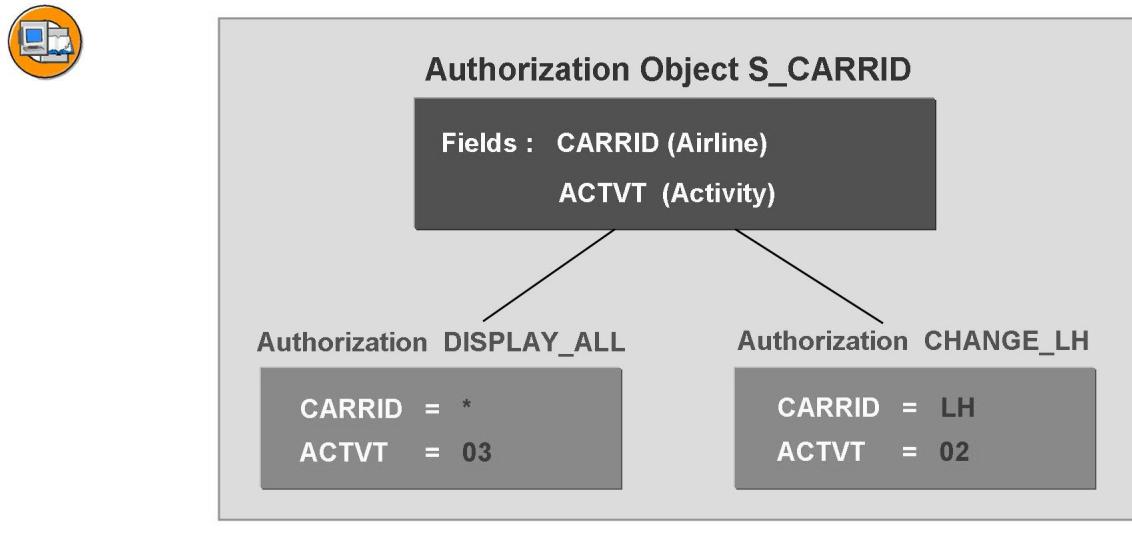


Figure 162: The SAP Authorization Concept



Possible ACTVT values:

- 01 = Create
- 02 = Change
- 03 = Display
- ...

Figure 163: Authorization Objects and Authorizations (Example)

Authorization objects can be defined within **object classes**. When you define an authorization object, you have to specify appropriate fields (without values). You create an actual **authorization** by subsequently assigning values to these fields. This authorization can be integrated into the required **user master records** by means of an **authorization profile**.

Several different authorizations (for the integration into different user master records) can be created for an authorization object.

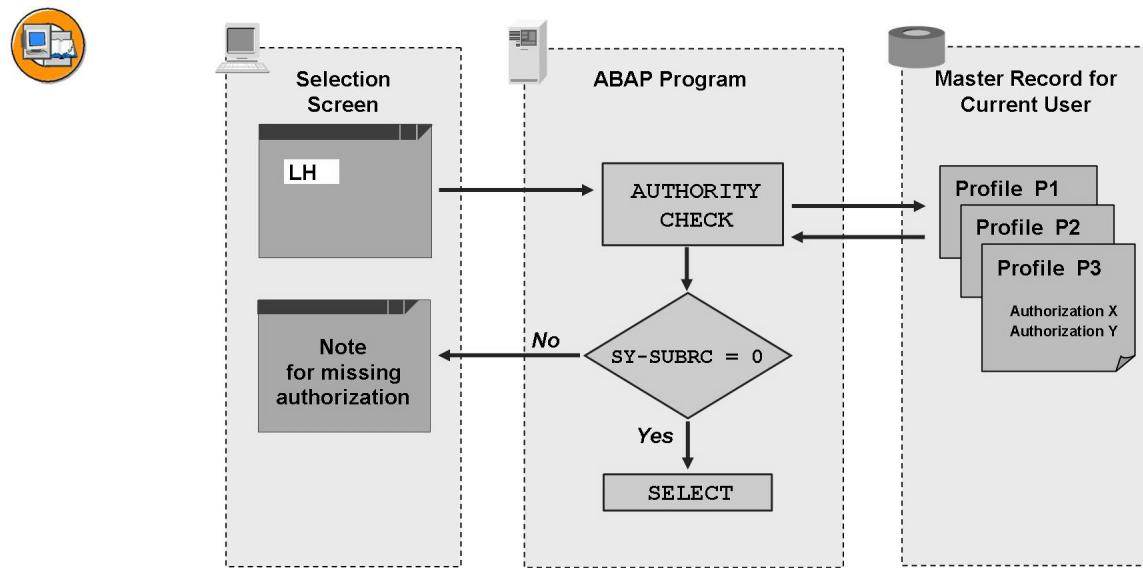


Figure 164: Authorization Check (Principle)

At runtime, you can use the **AUTHORITY-CHECK** statement to check whether the actual user has the authorization required for executing the function in his or her user master record. Depending on the **check result** (sy-subrc), you can continue your program accordingly:

sy-subrc = 0: User has required authorization Execute function (for example SELECT).

Else: Authorization not available -> Appropriate note to user, for example.



Hint: As well as the technique described, you also have the possibility of controlling access to entire programs and transactions using authorizations. However, such checks should only be considered as additions to, and not substitutions for the explicit authorization check by the developer.

Under normal circumstances, the definition of authorization objects is part of data modeling and the creation of database tables. Since we access an existing data model in this course, we use the authorization object S_CARRID that belongs to this data model.

Implementing the authorization concept is one of the tasks of the developer who programs access to the database tables.

The subsequent steps, such as defining the authorizations and profiles and designing the user master records, belong to the tasks of the administrator.

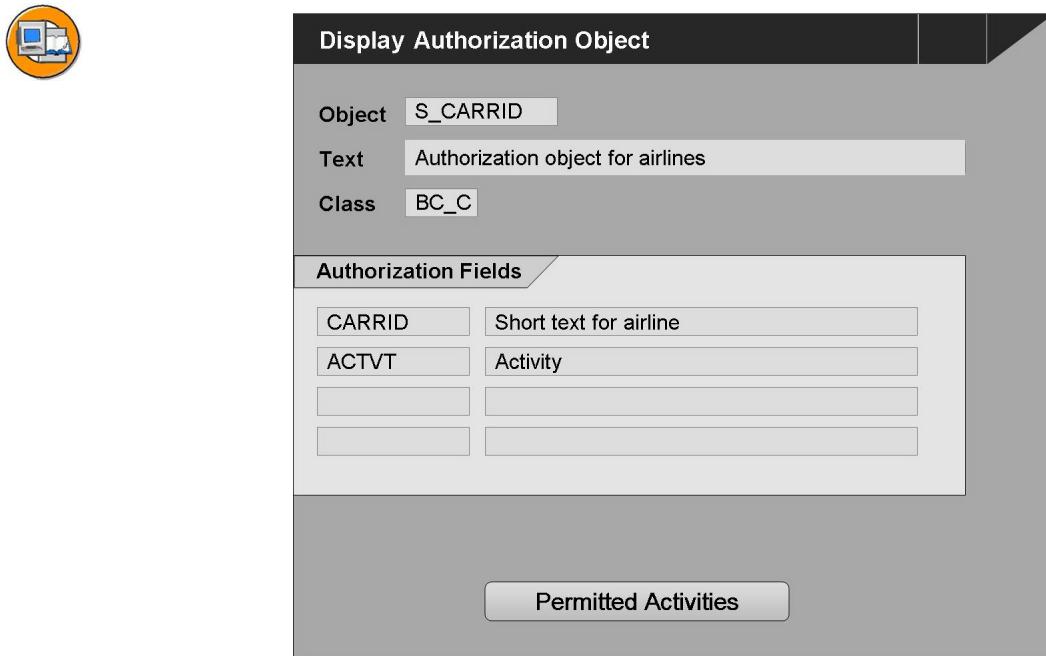


Figure 165: Displaying Authorization Objects

Before you can implement the required authorization check in your program, you must first determine the structure (the fields) of the respective authorization concept. An object usually consists of the ACTVT (activity) field and one other field, which specifies the data type to be protected (i.e., material number, airline and so on.) The values of these authorization fields specify what the user is authorized to do.

You can display an authorization object directly in the *ABAP Workbench* (for example, by using the *Other Object ...* pushbutton, or the *Other* tab page). You can also use transaction SU21. You will also get an overview of all the authorization objects created in the system here.

The following graphic shows how an authorization check is implemented.



Application Program

```

AUTHORITY CHECK
  OBJECT 'S_CARRID'
    ID 'CARRID' FIELD iv_carrid
    ID 'ACTVT'   FIELD '03'.
}

IF sy-subrc = 0.
  SELECT ...
ELSE.
  < Reaction to Missing Authorization >
ENDIF.

```

Check →



Figure 166: Authorization Check (Syntax Example)

For the authorization check in the program, you specify the authorization that is to be checked in the master record of the current user. The authorization is specified by specifying the authorization object, its fields, as well as the appropriate field values. Refer to the syntax in the above graphic.

In our example, there is a check as to whether the user has authorization for the **S_CARRID** object, in which the field **CARRID** (airline) contains the airline entered by the user, and the field **ACTVT** (activity) contains the value '03' (display).

After the AUTHORITY-CHECK statement, you should check the return code **sy-subrc** and monitor further processing of your program accordingly.



Hint: If you do not want to carry out a check for a field, either do not enter it in the AUTHORITY-CHECK statement or enter DUMMY as the field value. DUMMY is a predefined description entered without quotation marks.

An example of a suppressed field check: When a change transaction is called, the system should always check immediately whether the user has any change authorization for **any** airline. If the check fails, an appropriate message is to be output to the user immediately. Such a check can be implemented with the following syntax:

```

AUTHORITY-CHECK OBJECT 'S_CARRID'
  ID 'CARRID' DUMMY
  ID 'ACTVT'   FIELD '02'.

```

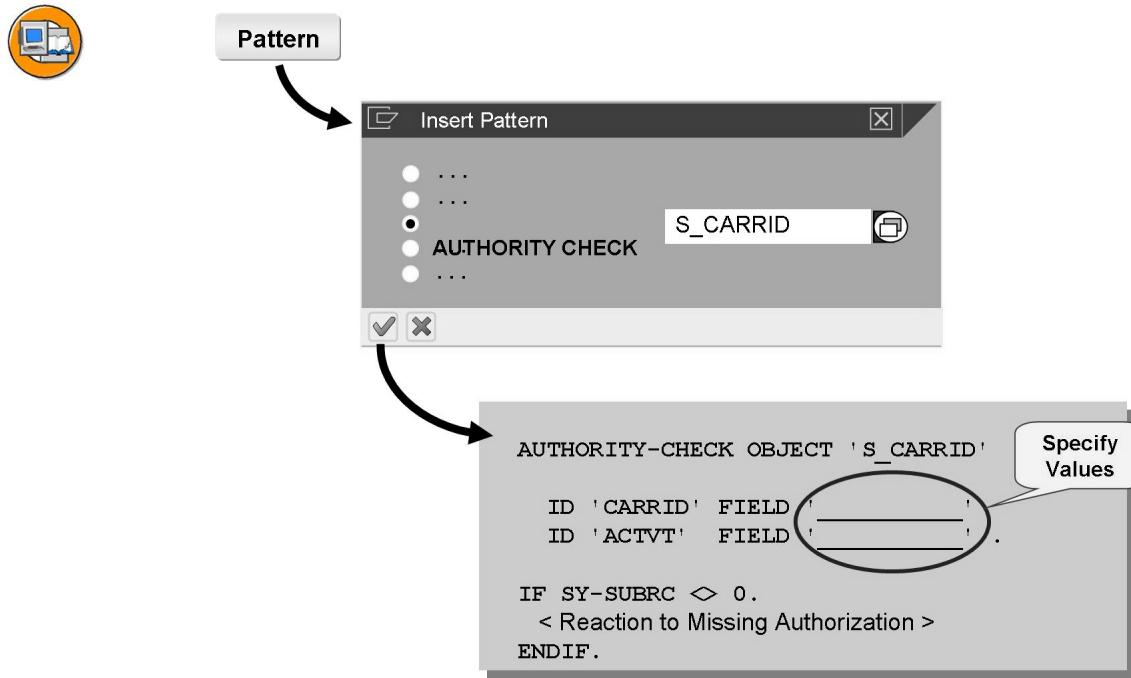


Figure 167: Implementing Authorization Checks in Programs

In order to avoid spelling errors in object and field names, you should have the AUTHORITY-CHECK statement generated into your source code by means of the **Pattern** button. After this, maintain the field values and implement the evaluation of sy-subrc.

Exercise 19: Authorization Check

Exercise Objectives

After completing this exercise, you will be able to:

- Implement authorization checks

Business Example

You want to develop a program in which data for the flight model is read from the database. Access to the data for this model is controlled using authorizations in the authorization object S_CARRID. You therefore need to perform an authorization check. To make the authorization check reusable, you encapsulate it in a function module.

Template:

None

Solution:

BC400_DDS_AUTH_CHECK (function module)

Task 1:

Create a function group, if one does not already exist.

1. Create a new function group in your package (suggested name: **ZBC400_##**).

Task 2:

Create a new function module in the function group. Maintain the interface for the function module in such a way that it imports an airline ID and an activity for the authorization check. Create two exceptions for the confirmation; one in case no authorization exists and another in case the caller has specified an invalid activity.

1. Create a new function module in your function group (suggested name: **Z_BC400_##_AUTH_CHECK**).

Continued on next page

2. Create a separate import parameter for the airline ID and activity (suggested names: **IV_CARRID** and **IV_ACTIVITY**). To type the data element, choose **S_CARR_ID** and **ACTIV_AUTH**.



Hint: The correct data elements are defined in the authorization object. To do this, open the authorization object **S_CARRID** in display mode and choose the input help (**F4 help**) for one of the authorization fields. You will find the relevant data element (second column) for each authorization field (first column) here.

3. Create two (classic) exceptions (suggested names: **NO_AUTH** and **WRONG_ACTIVITY**).

Task 3:

Implement an authorization check in the source code for the authorization object **S_CARRID**. Raise an exception if

- the authorization does not exist
- the caller has not specified any of the relevant activities.

1. Implement an authorization check for the authorization object **S_CARRID**. Supply the authorization fields with the import parameters for the function module. Evaluate the return value (sy-subrc) and raise an exception if the authorization is missing.



Hint: You should use the *Pattern* function, since the authorization check is very intolerant of typos and errors with the use of upper/lowercase characters.

2. Before the authorization check, make sure that the caller has specified only one of the the relevant activities. If necessary, raise an exception.



Hint: The relevant activities are defined in the authorization object. To do this, open the authorization object in display mode and choose the **Permitted Activities** pushbutton at the bottom of the screen.

Continued on next page

Task 4:

Activate and test your function module.

1. Activate the function module.
2. Test your function module.

Solution 19: Authorization Check

Task 1:

Create a function group, if one does not already exist.

1. Create a new function group in your package (suggested name: **ZBC400_##**).
 - a) In the navigation area, open the context menu for the package and choose *Create → Function Group*.
 - b) Enter the name of the function group and a short text and choose *Save*.
 - c) Assign the function group to your package and your request in the usual way.

Task 2:

Create a new function module in the function group. Maintain the interface for the function module in such a way that it imports an airline ID and an activity for the authorization check. Create two exceptions for the confirmation; one in case no authorization exists and another in case the caller has specified an invalid activity.

1. Create a new function module in your function group (suggested name: **Z_BC400_##_AUTH_CHECK**).
 - a) In the navigation area, open the context menu for your function group and choose *Create → Function Module*.
 - b) Enter the name of the function module and a short text and choose *Save*.
2. Create a separate import parameter for the airline ID and activity (suggested names: **IV_CARRID** and **IV_ACTIVITY**). To type the data element, choose **S_CARR_ID** and **ACTIV_AUTH**.



Hint: The correct data elements are defined in the authorization object. To do this, open the authorization object **S_CARRID** in display mode and choose the input help (**F4 help**) for one of the authorization fields. You will find the relevant data element (second column) for each authorization field (first column) here.

- a) Open the *Import* tab page. Enter the names of the parameters and the data types.

Continued on next page

3. Create two (classic) exceptions (suggested names: **NO_AUTH** and **WRONG_ACTIVITY**).
 - a) Open the *Exceptions* tab page. Enter the names of the exceptions and a short text. Do not set the *Exception Classes* indicator.

Task 3:

Implement an authorization check in the source code for the authorization object S_CARRID. Raise an exception if

- the authorization does not exist
 - the caller has not specified any of the relevant activities.
1. Implement an authorization check for the authorization object S_CARRID. Supply the authorization fields with the import parameters for the function module. Evaluate the return value (sy-subrc) and raise an exception if the authorization is missing.



Hint: You should use the *Pattern* function, since the authorization check is very intolerant of typos and errors with the use of upper/lowercase characters.

- a) See source code excerpt from the model solution.
2. Before the authorization check, make sure that the caller has specified only one of the the relevant activities. If necessary, raise an exception.



Hint: The relevant activities are defined in the authorization object. To do this, open the authorization object in display mode and choose the *Permitted Activities* pushbutton at the bottom of the screen.

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

Task 4:

Activate and test your function module.

1. Activate the function module.
 - a) In the *Function Builder* toolbar, choose the pushbutton.

Continued on next page

2. Test your function module.
 - a) In the *Function Builder* toolbar, choose the  pushbutton.
 - b) Supply the import parameters with values and choose .
 - c) Check the result.



Hint: Check the raising of the exception if an error occurs too.

Result

Source code extract from the model solution:

```

FUNCTION bc400_dds_auth_check.
* -----
* *** Lokale Schnittstelle:
* " IMPORTING
* "   REFERENCE(IV_CARRID) TYPE S_CARR_ID
* "   REFERENCE(IV_ACTIVITY) TYPE ACTIV_AUTH
* " EXCEPTIONS
* "   NO_AUTH
* "   WRONG_ACTIVITY
* -----


CASE iv_activity.
  WHEN '01' OR '02' OR '03'.

    AUTHORITY-CHECK OBJECT 'S_CARRID'
      ID 'CARRID' FIELD iv_carrid
      ID 'ACTVT'   FIELD iv_activity.

    IF sy-subrc <> 0.
      RAISE no_auth.
    ENDIF.

  WHEN OTHERS.
    RAISE wrong_activity.
ENDCASE.
ENDFUNCTION.

```



Lesson Summary

You should now be able to:

- Explain the SAP authorization concept
- Implement authorization checks



Unit Summary

You should now be able to:

- Explain the purpose and the benefits of using a data model in application development
- Describe the SAP flight data model
- Describe the meaning and the structure of a transparent table
- List different methods for searching relevant database tables
- Program read access to specific columns and rows within a particular database table
- List different methods for read accesses to several database tables
- Explain the SAP authorization concept
- Implement authorization checks

Related Information

... Refer to the online documentation in the *ABAP Dictionary* and for the relevant ABAP statements.



Test Your Knowledge

1. You can define variables in the program using Dictionary types (descriptive elements)

Determine whether this statement is true or false.

- True
- False

2. Database tables are stored together with their content in the *ABAP Dictionary*

Determine whether this statement is true or false.

- True
- False

3. What do you call the description of a database table in the *ABAP Dictionary* and what name does it have?

4. What is defined with the statement "DATA myvar TYPE dbtab." if dbtab is a transparent table?

Choose the correct answer(s).

- A A copy of the database table dbtab
- B An internal table with the same structure as the database table dbtab
- C An internal table with the same content as the database table dbtab
- D A structure variable with the same structure as a dbtab row

5. Why should "Open SQL" be generally used for database accesses instead of "Native SQL?"

6. Which statement is used to read a single record from a database table?

7. Does the client limitation have to be specified in the WHERE clause of a SELECT statement if the database table to be read has the client field?

8. What is the name of the high-performance technique that one should use to load the content of a database table into an internal table?

9. In which system field do you find the number of the records read using SELECT?

Choose the correct answer(s).

- A SY-SUBRC
- B SY-TABIX
- C SY-DBCNT
- D SY-INDEX

10. Which technique should be used for reading records that belong together from several database tables? This technique performs considerably better than nested SELECT statements.

11. Which of the following statements concerning authorization checks are correct?

Choose the correct answer(s).

- A In the SELECT statement, the corresponding authorization check is already integrated.
- B The SELECT statement functions without any problem, even if the user does not have the appropriate authorization for the read process.
- C To protect data from unwanted access, an authorization check must be implemented in the program.
- D An authorization check must always be implemented in the program.

12. Which ABAP statement is used to implement an authorization check and where is the check result to be found?

13. Which of the following statements concerning authorization and authorization checks are correct?

Choose the correct answer(s).

- A An authorization object has fields and field values.
- B An authorization is a real characteristic of an authorization object - that is, an authorization object with field values.
- C There are authorization objects in the user master record.
- D There are authorizations in the user master record.



Answers

1. You can define variables in the program using Dictionary types (descriptive elements)

Answer: True

You can use Dictionary types for typing variables.

2. Database tables are stored together with their content in the *ABAP Dictionary*

Answer: False

Only descriptions of database tables are stored in the *ABAP Dictionary*.

3. What do you call the description of a database table in the *ABAP Dictionary* and what name does it have?

Answer: A database table description in the *ABAP Dictionary* is called a **transparent table**. It has the same name as the database table.

4. What is defined with the statement "DATA myvar TYPE dbtab." if dbtab is a transparent table?

Answer: D

The specified syntax defines only a structure variable since a transparent table is specified for the type. This table basically describes the row structure of the corresponding database table.

5. Why should "Open SQL" be generally used for database accesses instead of "Native SQL"?

Answer: Because Open SQL is independent of the relational database system used.

6. Which statement is used to read a single record from a database table?

Answer: SELECT SINGLE

7. Does the client limitation have to be specified in the WHERE clause of a SELECT statement if the database table to be read has the client field?

Answer: No; if the client limitation is missing, only records of the executing client are read - that is, the client in which the program is called.

8. What is the name of the high-performance technique that one should use to load the content of a database table into an internal table?

Answer: Array Fetch

9. In which system field do you find the number of the records read using SELECT?

Answer: C

10. Which technique should be used for reading records that belong together from several database tables? This technique performs considerably better than nested SELECT statements.

Answer: Table join

11. Which of the following statements concerning authorization checks are correct?

Answer: B, C

12. Which ABAP statement is used to implement an authorization check and where is the check result to be found?

Answer: AUTHORITY-CHECK / SY-SUBRC

13. Which of the following statements concerning authorization and authorization checks are correct?

Answer: B, D

Unit 7

User dialogs

Unit Overview

The unit overview lists the individual lessons that make up this unit.



Unit Objectives

After completing this unit, you will be able to:

- List attributes and benefits of screens
- Implement simple screens with input/output fields and link them to a dialog application
- Explain and implement the program-internal processing for screen calls
- List the properties and usage scenarios of the ABAP Web Dynpro
- Explain the programming and runtime architecture of the ABAP Web Dynpro
- Implement simple Web Dynpro applications with input/output fields and pushbuttons
- List the properties and benefits of selection screens
- Implement the options for restricting selections on the selection screen
- Describe the attributes and benefits of ABAP lists
- Implement list and column headers
- Implement multilingual lists
- Describe the event-controlled processing of an executable ABAP program
- List the most important basic events and explain their purpose
- Use the SAP Grid Control (*SAP List Viewer*) to display an internal table on a screen

Unit Contents

Lesson: Screen	347
Exercise 20: Creating Screens and Dynamic Next Screen Processing	377
Exercise 21: Screen – Creating Input/Output Fields	385

Exercise 22: Screens – Data Transport	391
Lesson: ABAP Web Dynpro	398
Exercise 23: Web Dynpro:Navigation.....	423
Exercise 24: Web Dynpro: Data Transport and Layout	429
Lesson: Classic ABAP Reports	436
Exercise 25: Selection Screen and Classic ABAP List.....	455
Exercise 26: ABAP Events	463
Lesson: Displaying Tables with the SAP List Viewer.....	469
Procedure: Displaying an Internal Table in the ALV Grid Control on a Screen	477
Exercise 27: Displaying an Internal Table with the SAP List Viewer.....	481

Lesson: Screen

Lesson Overview

In this lesson you will learn how to design and program simple screens with input/output fields and pushbuttons.



Lesson Objectives

After completing this lesson, you will be able to:

- List attributes and benefits of screens
- Implement simple screens with input/output fields and link them to a dialog application
- Explain and implement the program-internal processing for screen calls

Business Example

You want to develop a program that allows data to be entered and output on screen layouts. Furthermore, pushbuttons should be available on the screen layouts that can be used to address the corresponding functions in your program.

The Screen

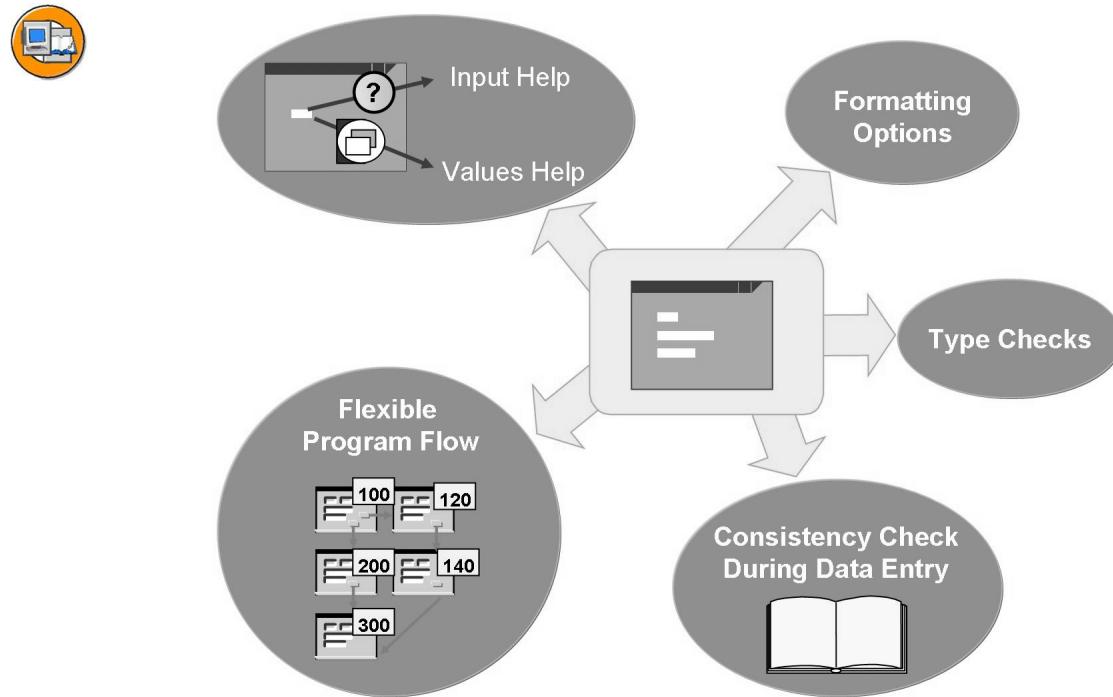


Figure 168: Features of Screens

A screen not only consists of its layout with input/output fields, pushbuttons and other screen elements but also a processing logic (source code excerpts that are processed as the pre-/postprocessing of the screen display).

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.

The layout can be designed very flexibly, with input fields, output fields, radio buttons, check fields and, most importantly, pushbuttons with which the corresponding functions of the program can be executed.

Screens have the same formatting options as lists and selection screens: Fixed point numbers and dates are formatted according to the settings in the user master record; the time is set to hh:mm:ss; sums of money are formatted according to the content of the currency field, and physical measures (lengths, weights, ...) are formatted according to the content of a unit field.

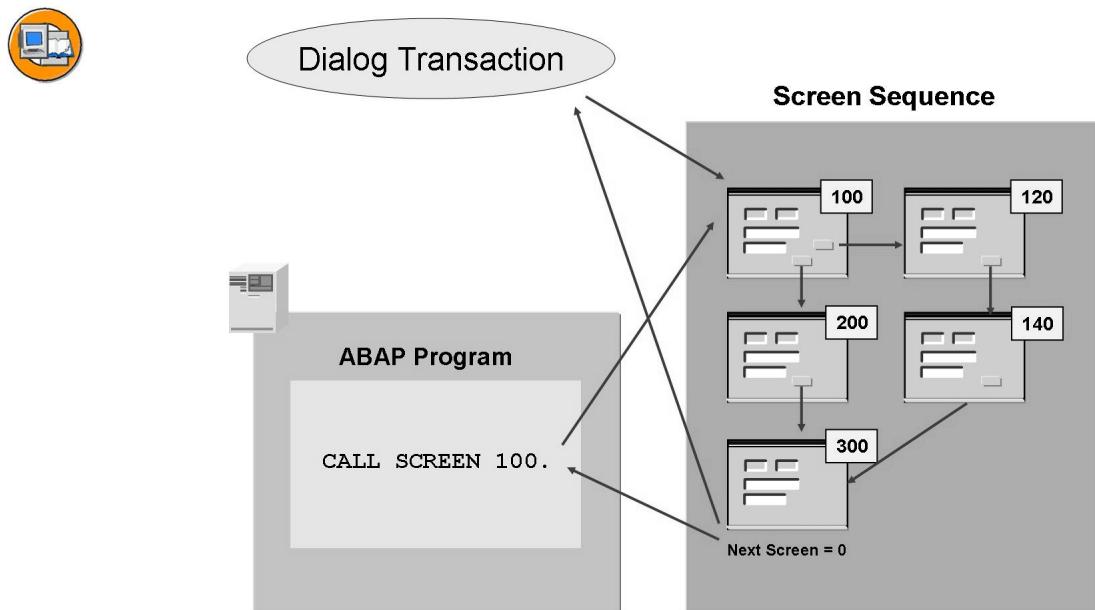


Figure 169: Screen Sequence

In principle, there are two options for starting a screen sequence:

1. By calling the first screen (CALL SCREEN statement) from a processing block in your program
2. By creating a transaction that references the program and the first screen (dialog transaction).

After a screen is processed, the statically or dynamically defined screen sequence is processed. The formal next screen 0 returns processing to the point where the screen was called or ends the dialog transaction.



Screens

appear in the following program types:

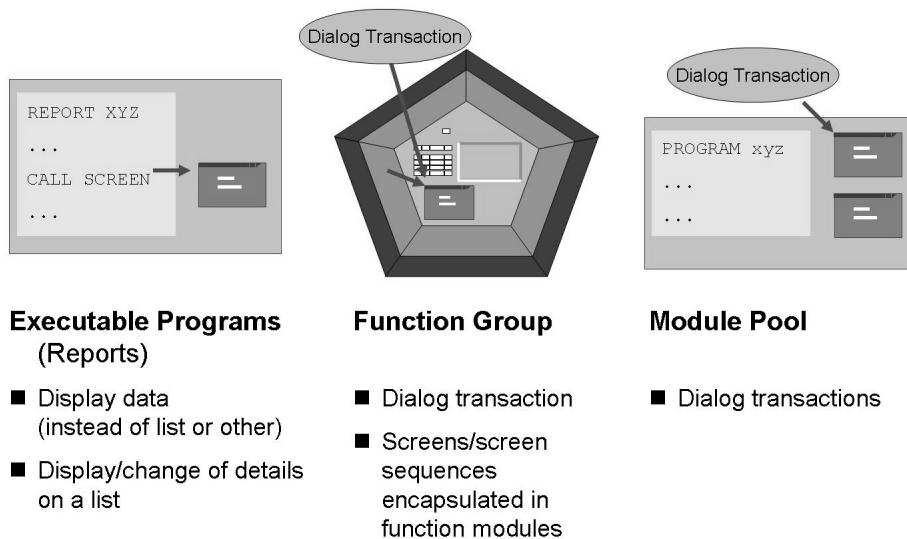


Figure 170: Screens and Program Types

Screens can be created in the following program types:

Executable program (report)

Screens are used in executable programs in order to display data in addition to the list output, or to replace the list output completely. In addition, there is also the possibility of using screens to enter and change data in the list. For the purpose of reusability and data encapsulation, you should no longer create screens directly in reports, but use screens in function groups instead.

function group

Screens in function groups can be addressed using dialog transactions. You can also start such screens from the source code of a function module using the `CALL SCREEN` statement. This enables you to make a screen or a screen sequence easily available with a defined interface for reuse. Function groups are the recommended program type for creating screens.

module pool

Screens in module pools can only be addressed using dialog transactions. In contrast to screens in function groups, you cannot encapsulate them and provide them with a well-defined external interface.

For this reason, you should avoid creating new module pools where possible. Always use function groups instead, even if addressing the screen using function modules has not (yet) been planned.

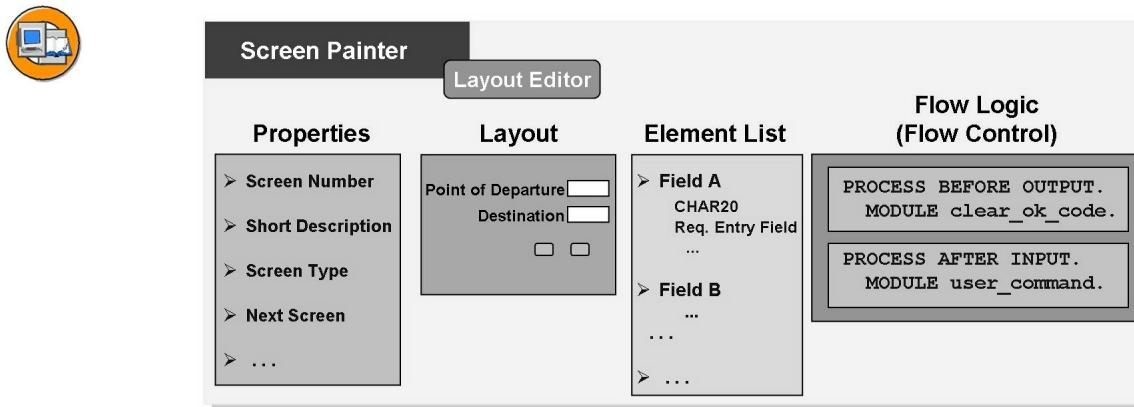


Figure 171: Components of a Screen

Each screen has the following components:

Properties (attributes)

Properties 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) and also specify the **default next screen**.

Layout

You can place input/output fields, texts, pushbuttons, and other elements on your screen. Such elements are called **screen elements**.

Element list

Lists all **screen elements including their attributes** such as position, size, data type, and so on.

Flow logic

The flow logic of a screen consists of the **PBO** (Process Before Output) and the **PAI** (Process After Input). PBO contains references to processing blocks (**PBO modules**) that are processed in preparation for the screen display (for example, data selection) before the screen is sent. PAI contains references to processing blocks (**PAI modules**) that are processed as a reaction to user input and actions (for example, save data).

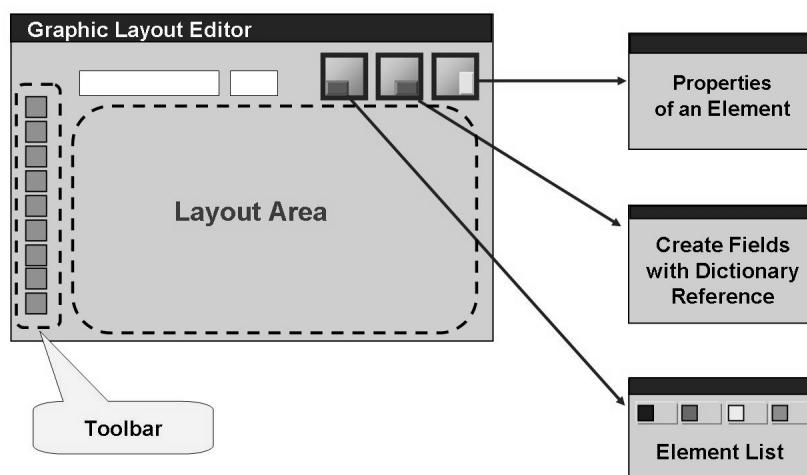


Figure 172: Graphic Layout Editor

Using the graphical *Screen Painter*, you can design the layout of the screen (toolbar). You also have the following maintenance functions using the three pushbuttons depicted in the above graphic:

Maintain element attributes

In a dialog box, all attributes for the selected screen element are displayed for maintenance.

Get Dictionary or program fields

You can use a dialog box to create screen fields with reference to fields from Dictionary structures or fields defined within the program.

Show element list

Shows all available screen elements including their attributes for maintenance.

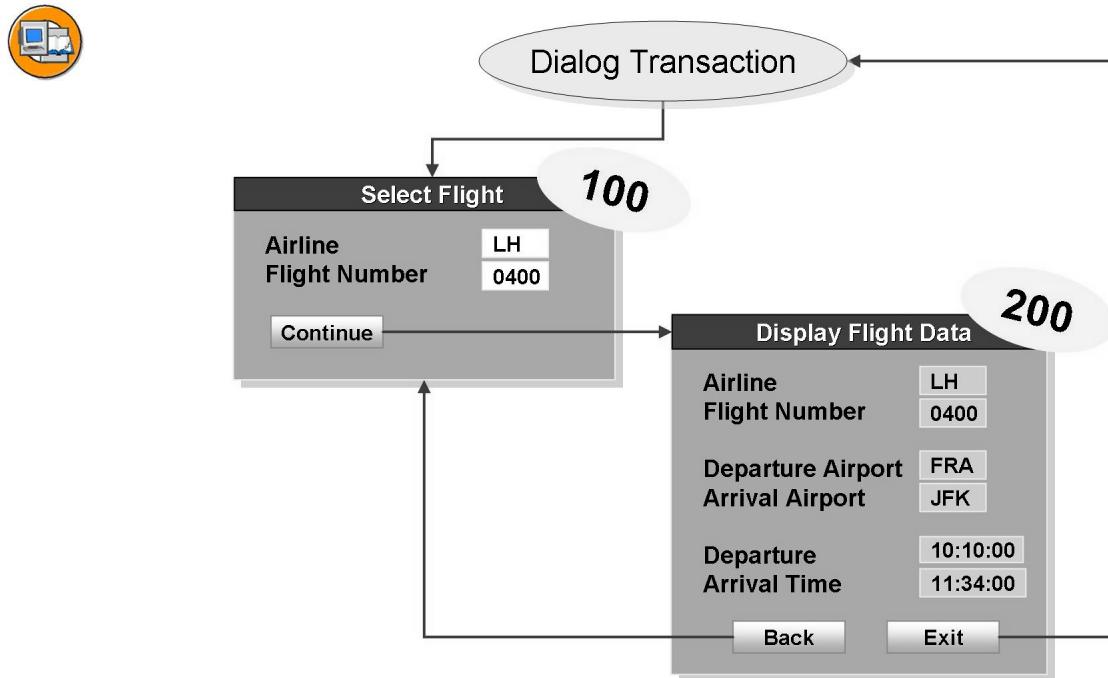


Figure 173: Application example

In the following sections, a program will be developed in several steps that will enable you to display the data for individual flight connections.

The program comprises a sequence of two screens. On the first screen, the user chooses a flight connection. On the second screen, the details for this connection are displayed.

The screen sequence is started by a dialog transaction. If the user chooses the *Continue* pushbutton on the first screen, the entries are processed, the data is read from the database and the system navigates to the second screen.

On the second screen, the flight connection data is displayed. If the user chooses the *Back* pushbutton, the system returns to the first screen, where another flight connection can be selected.

If the user chooses the *Exit* pushbutton, the system exists the screen sequence. In this case, the dialog transaction is cancelled.

Screens, Pushbuttons and Navigation

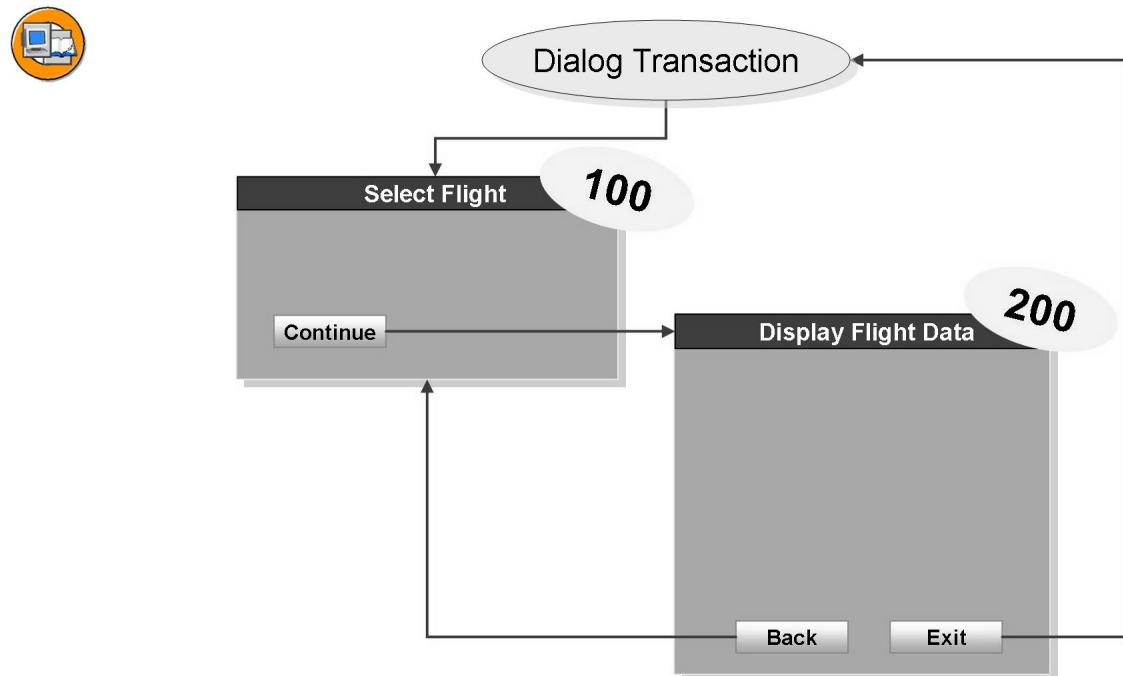


Figure 174: Realization Level 1: Creating Screens with Pushbuttons and Navigation

At the first level of our example, two screens will be created on which pushbuttons are located. The program is to be implemented in such a way that after a pushbutton has been selected, you navigate to the other screen, or exit the screen sequence. The screen sequence is to be started by a dialog transaction.

This section deals with the following:

- Creating screens
- Flow logic of a screen in PBO and PAI event blocks
- PBO and PAI modules as processing blocks for the corresponding events
- Implementing pushbuttons and evaluating user actions
- Creating dialog transactions

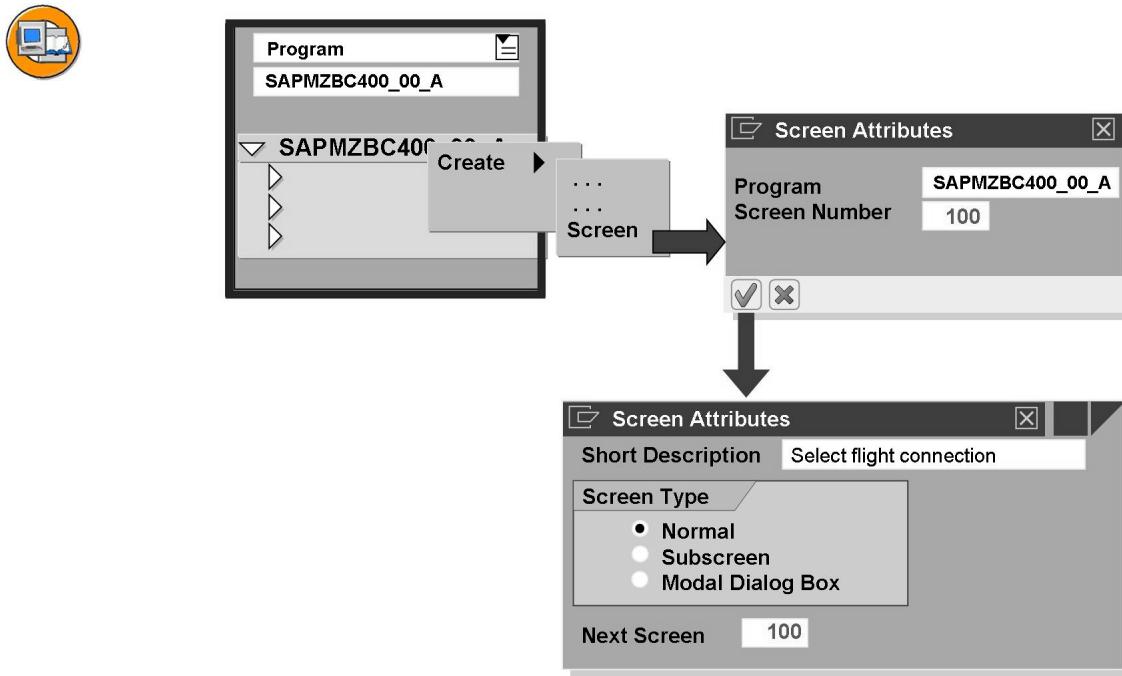


Figure 175: Creating a Screen

There are several ways to create a screen:

Using the Object Navigator

You can create a new program object (*screen*) for your program on the object list in the navigation area using the context menu for your program (see above graphic).

By forward navigation from the ABAP Editor

In the *ABAP Editor*, double-click the screen number to create a screen (in the CALL SCREEN nnnn statement). The system automatically opens the *Screen Painter* for maintaining the new screen.

When you create a screen, the system will first 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 the screen completely and then returns to processing at the point where the screen was called.

Caution: As zero is the initial value of the field; it is hidden when the screen attributes are displayed.

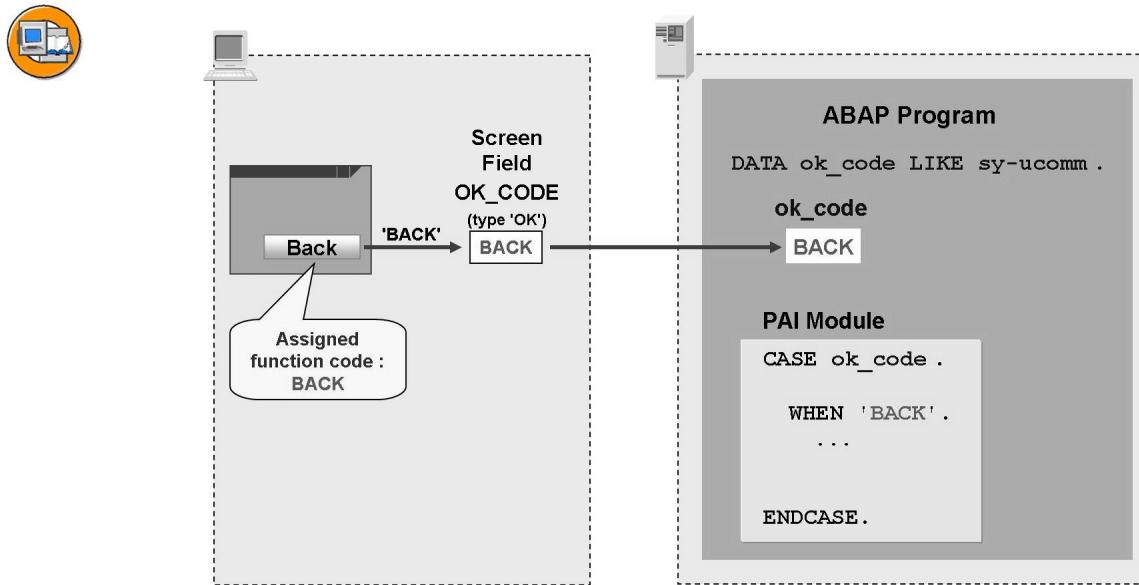


Figure 176: Flow for Choosing a Pushbutton

If the user chooses a pushbutton, the runtime system copies the assigned **function code** to a special screen field (of the OK type). This screen field is typically called the **ok_code**.

The content of this special screen field is automatically transported if there is a data object of the same name within the program. Following that, PAI processing is triggered, where you find out about the user action through the function code transferred to the program and carry out appropriate processing.

The following sections tell you how to define pushbuttons that use the special screen field of the 'ok' type, declare a data object of the same name within the program, and react to the respective user action.

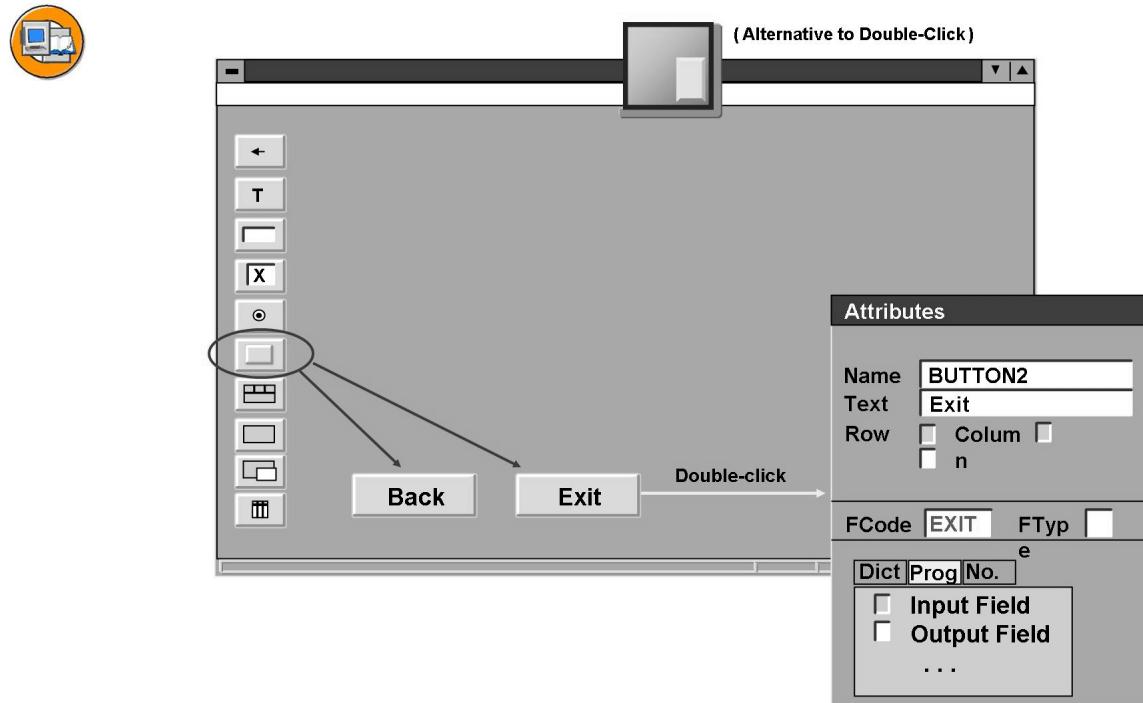


Figure 177: Defining Pushbuttons and Assigning Function Codes

The above graphic illustrates how you define pushbuttons in the *Graphical Screen Painter*. You have to assign a name and function code to each pushbutton. You can do so using the maintenance of the field attributes.

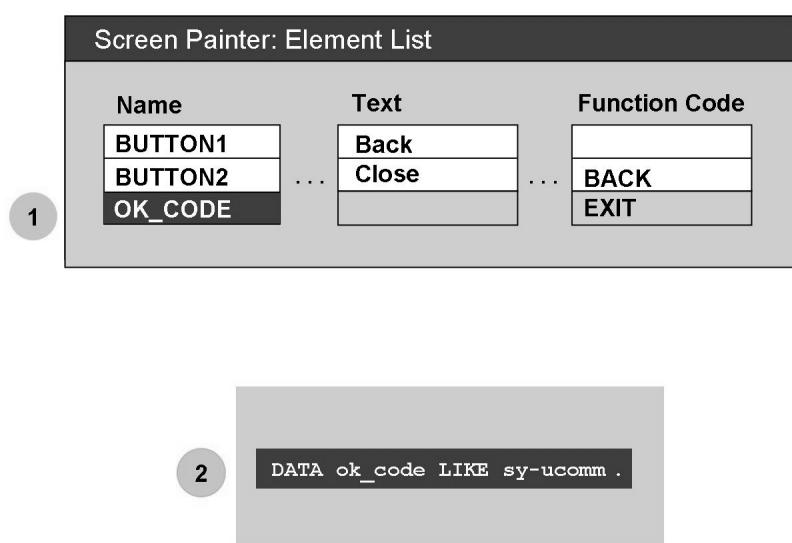


Figure 178: Implementing the Function Code Transport

The special screen field with which the respective function code is transported to the program is called the command field and is available on the screen by default. To use it, you must enter a name for it. Normally, the name **OK_CODE** is used.

You declare the program-internal data object with the same name by assigning the type to the system field **sy-ucomm** (see above graphic).

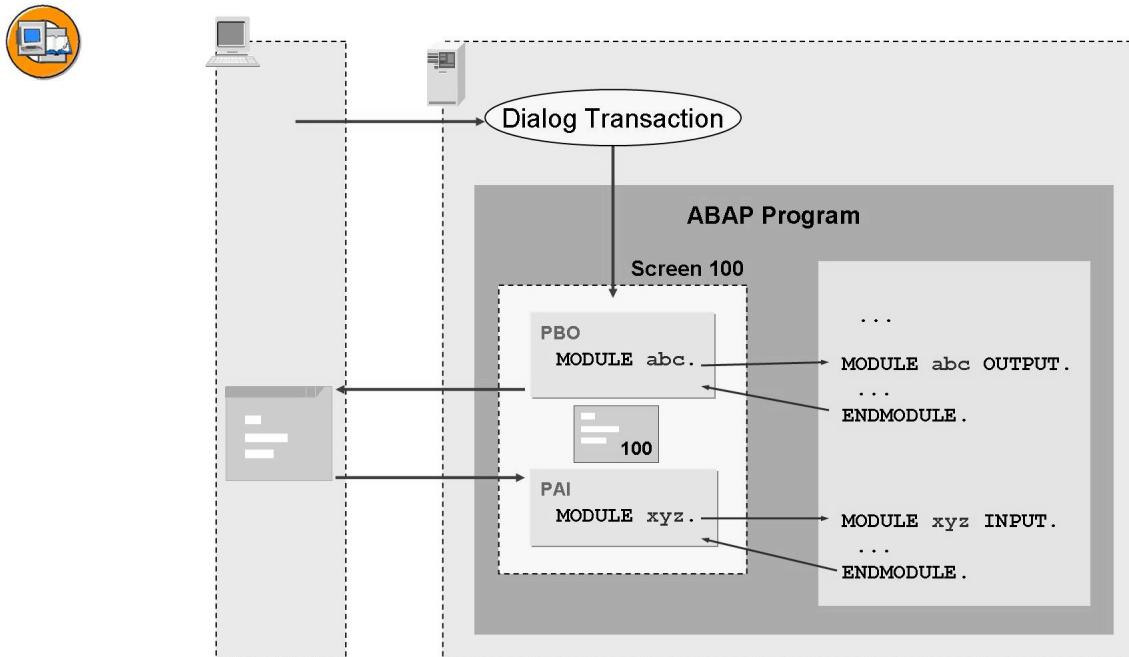


Figure 179: Runtime Architecture of the Screen Flow Control

The dialog transaction is used to trigger **screen processing**. This includes the following steps (that are executed automatically):

PBO processing

In preparation for the screen display, the PBO modules listed in the PBO block are processed successively.

Field transport from the program to the screen

After PBO has been completed, the data to be displayed is transported to the screen fields.

Screen display

The screen that has been assigned values is sent to the presentation server (SAPGui) and shown to the user.

Field transport from the screen to the program

Each user action on the screen triggers the transport of the screen field contents back to the program.

PAI processing

As a reaction to the user action, the modules listed in PAI are processed successively.

Modules are source code blocks without interface. They are enclosed by the ABAP statements MODULE and ENDMODULE. For each subfunction in PBO or PAI, a corresponding module should be implemented.



Caution: The **flow logic** of the screen (PBO/PAI) only contains **references to modules**, which are implemented in the program and consist of ABAP statements. ABAP source code may **not** be stored directly in the flow logic.

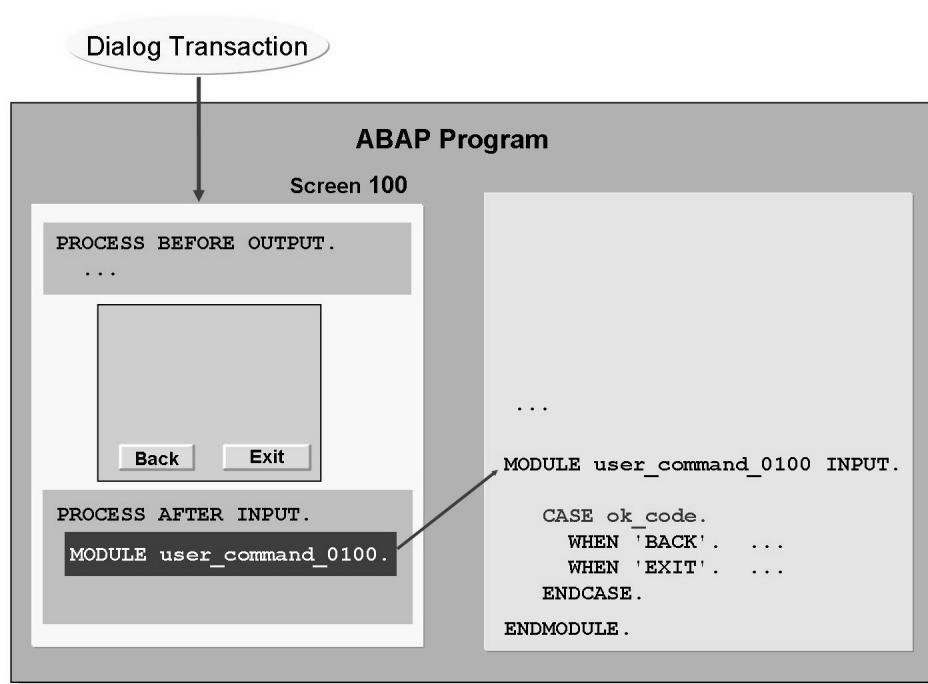


Figure 180: Evaluating the Function Code at the PAI Event

The above graphic shows the program's reaction to the user action. The corresponding function code that was transferred to the OK_CODE field immediately before PAI processing is evaluated.

Usually, the module with the **main processing in PAI** is called **USER_COMMAND_nnnn**, whereby nnnn represents the screen number.



Hint: In a module, you can access **all global data objects** of the program. Variables defined within the module are always **global**.

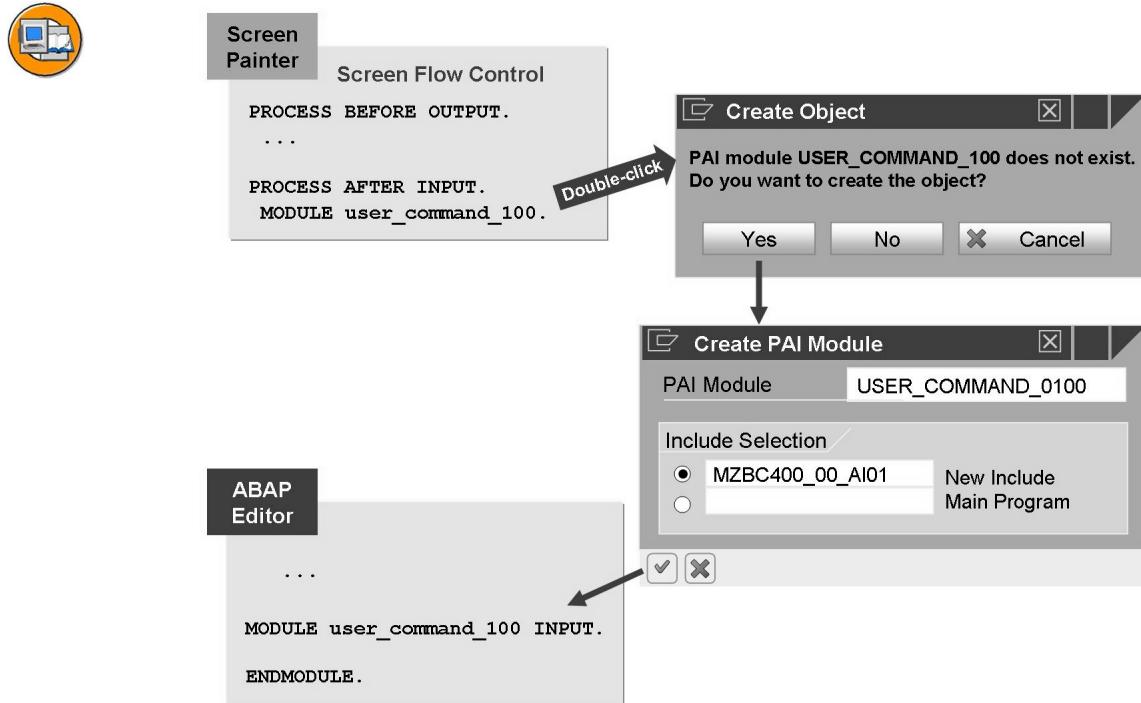


Figure 181: Creating Modules Using Forward Navigation

There are two ways to create a module:

Using forward navigation from the flow logic

Starting from the flow logic of your screen, you can double-click the module reference to create the corresponding module (see above graphic).

Using the navigation area of the Object Navigator

You can also create a module from the object list of your program. To do so, use the context menu of the program. In this case, note that you have to include a corresponding reference to the newly created module in the flow logic of your screen.

A module can be called from the flow logic of several screens (reusability).

Please note that modules starting with **MODULE ... OUTPUT** are **PBO modules** and can only be called by the PBO of a screen. Accordingly, **PAI modules** starting with **MODULE ... INPUT** can only be called by PAI.

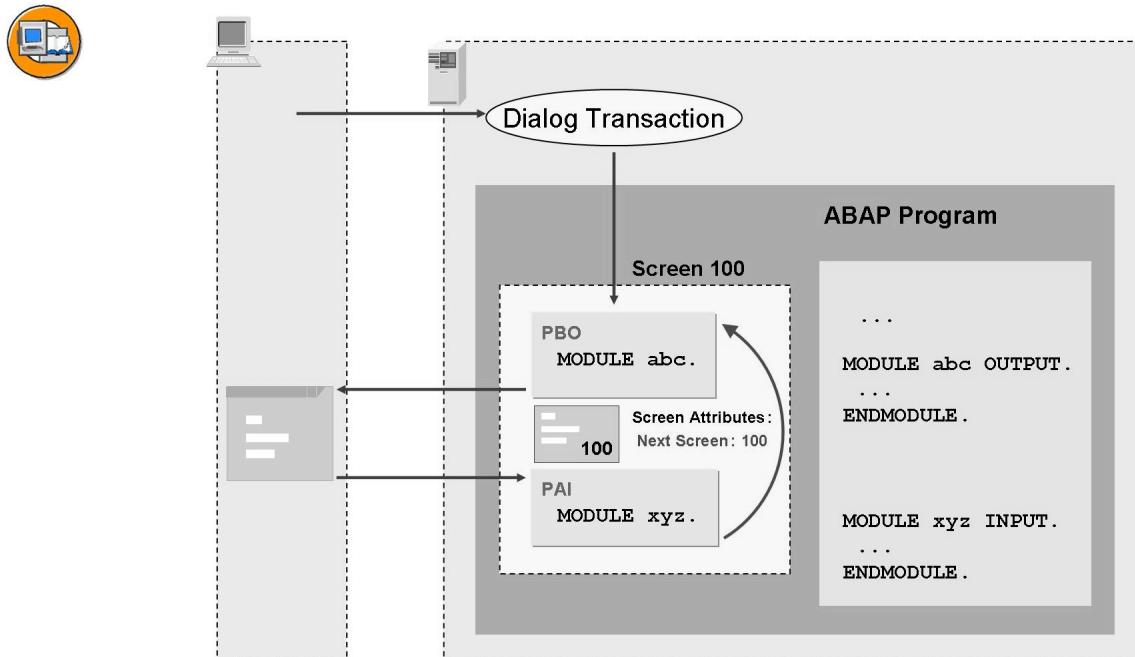


Figure 182: Next Screen“Same Screen” (Effect)

If a screen is assigned its own screen number as the next screen, it is processed again after it is closed.

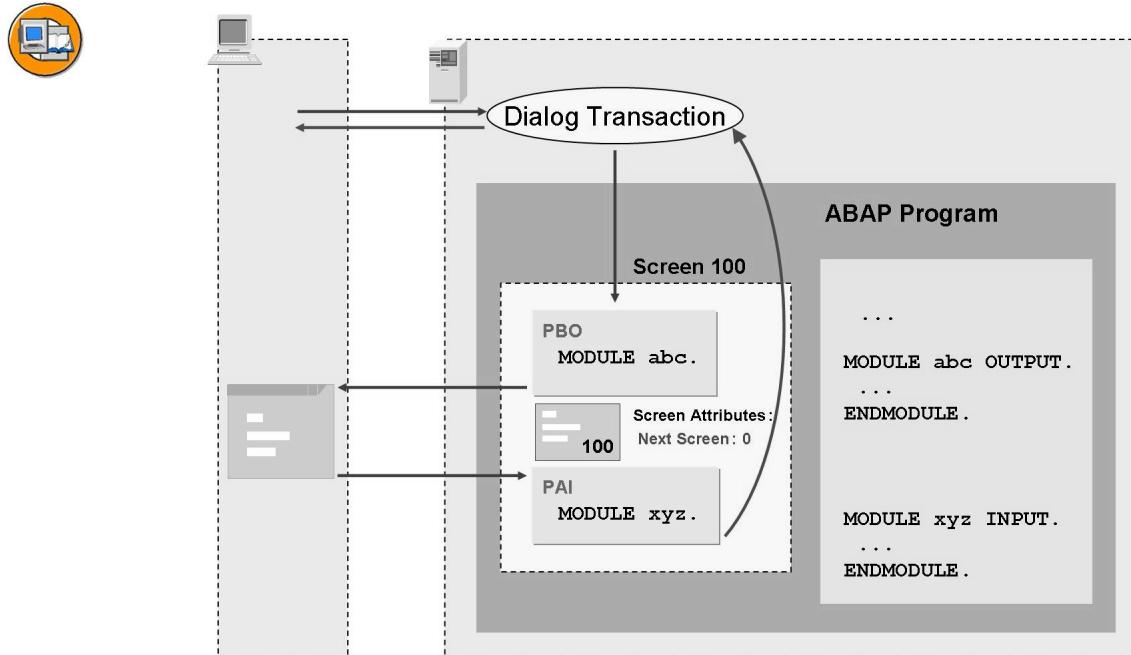


Figure 183: Next Screen 0 (Effect)

If you enter the screen attribute “next screen =0”, the system first processes the entire screen and then returns to processing at the point where the screen call is set.

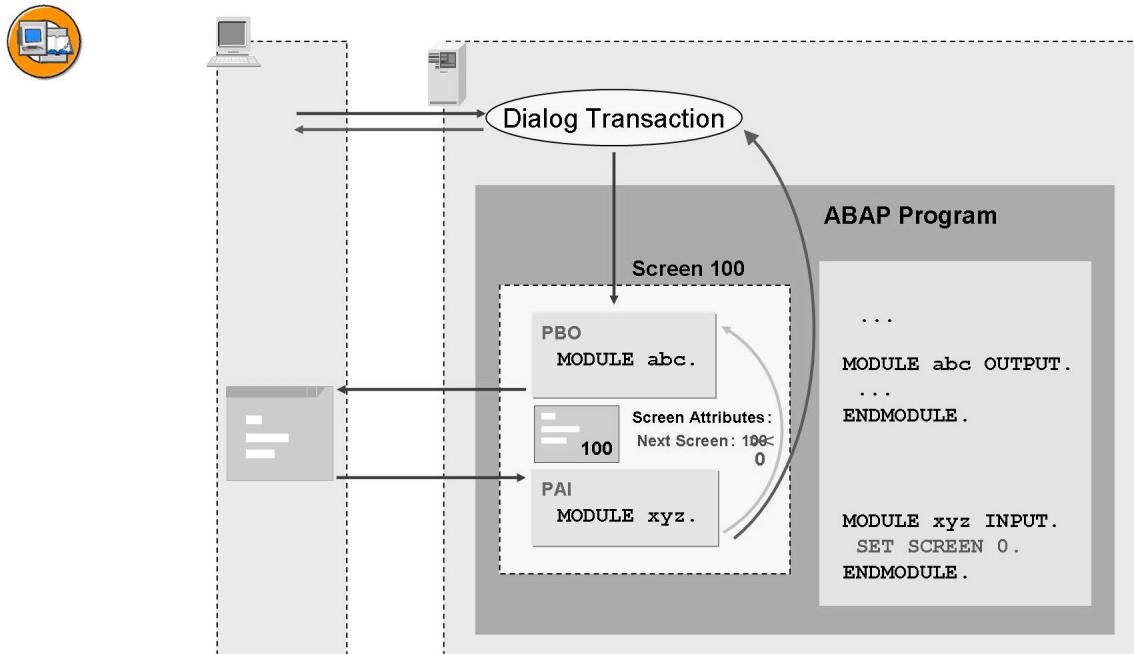


Figure 184: Dynamically Overwriting the Default Next Screen

You can use the ABAP statement `SET SCREEN` from a module (typically a PAI module) to **dynamically overwrite** the default next screen specified in the screen attributes (see above graphic). This option can be used for implementing the following SAP standard: When you choose **Enter** you return to the same screen; only certain pushbuttons take you to other screens. For your screen, enter its own screen number as the next screen (default next screen) and branch to other screens (using the `SET SCREEN` command) from PAI only if certain pushbuttons are pressed. (By default, the **Enter button** does **not** insert a function code into the command field of the screen). Hence, the initial value of the field (**Space**) is transported to the program as the function code.

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.

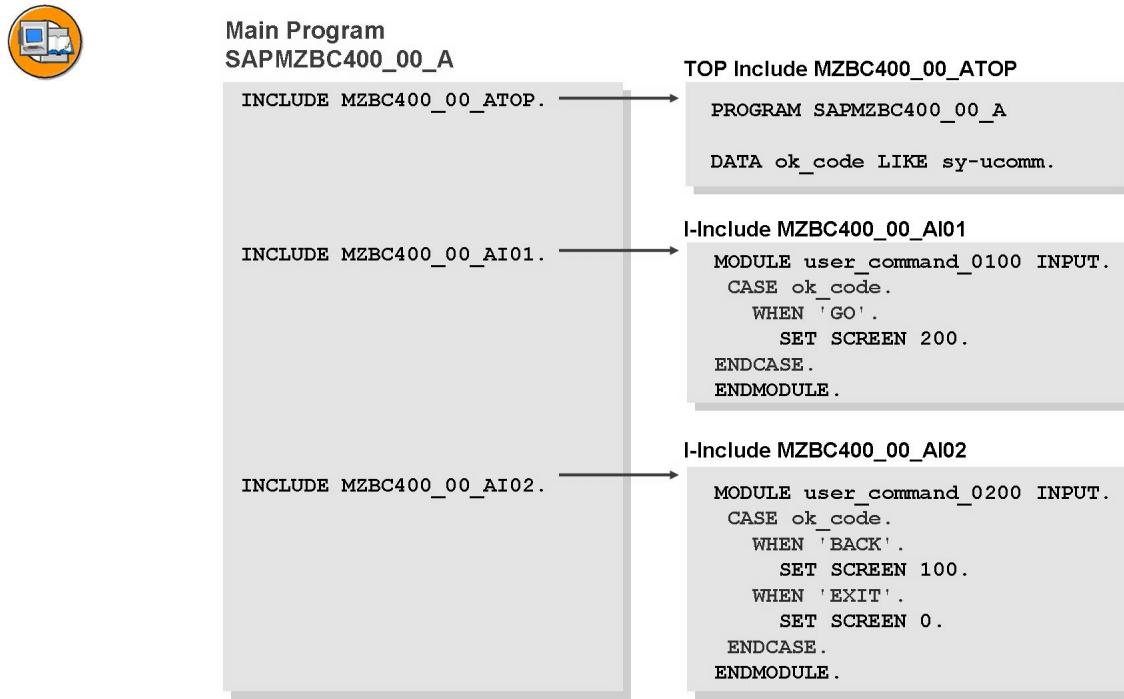


Figure 185: Syntax Example: Evaluating the Function Codes

For our scenario, one pushbutton should be caught on the first screen and two on the second:

- If you choose **Continue** (function code GO here) on the first screen, the next screen is set dynamically to 200 in order to get to the second screen.
- If you choose **Back** (function code BACK here) on the second screen, the next screen is set dynamically to 100 in order to return to the first screen.
- If you choose **Exit** (function code EXIT), the next screen is automatically set to zero to return to the call point of the screen. In our case, this means that the dialog transaction is terminated.

Screens, Creating Input/Output Fields

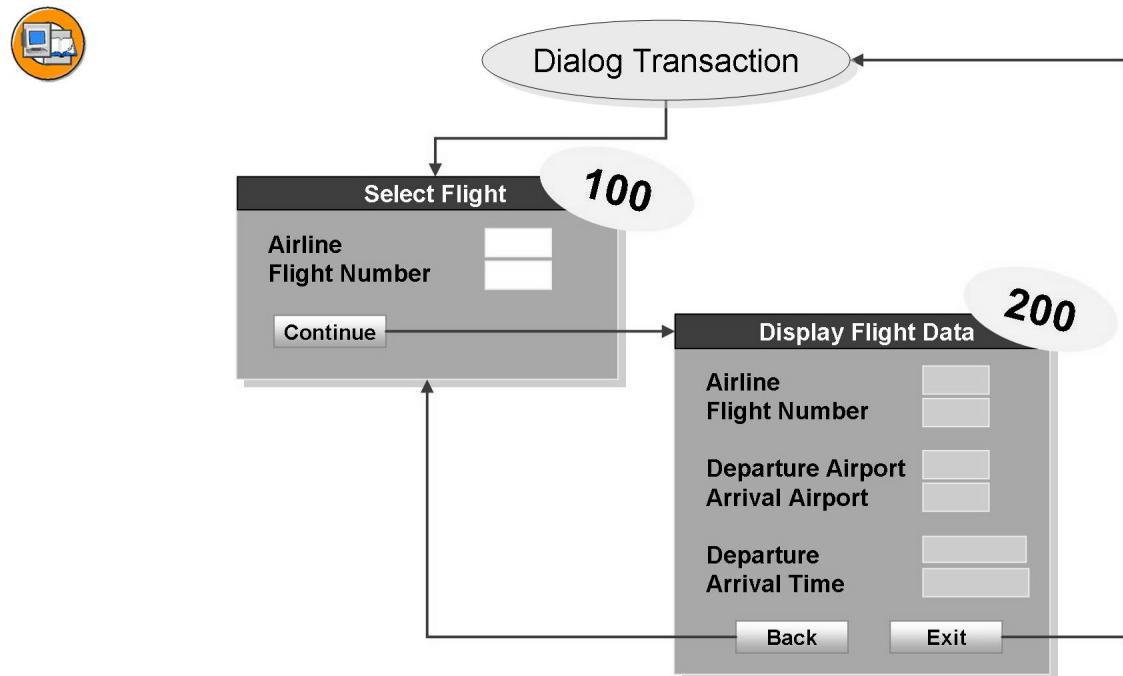


Figure 186: Realization Level 2: Creating Input/Output Fields

In the second step, input/output fields are to be created on both screens. The first screen should contain the fields *Airline* and *Flight Number* as ready-for-input fields. The second screen should contain only display fields. In addition to the *Airline* and *Flight Number* fields, it should also include the fields *Departure Airport*, *Destination Airport*, *Departure Time* and *Arrival Time*.

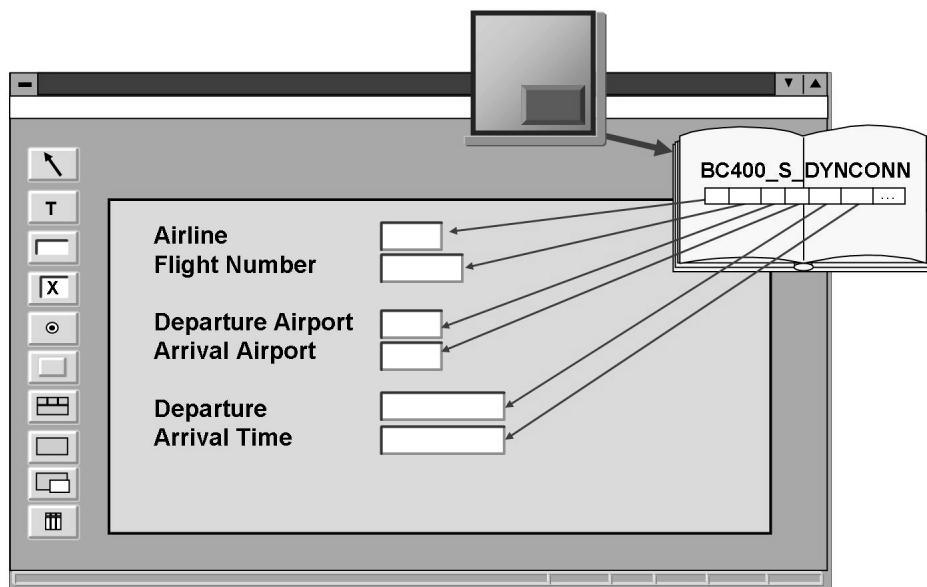


Figure 187: Creating Input/Output Fields (with Reference to Dictionary Fields)

You have two options for implementing the assignment of field attributes when you create screen fields:

Get from Dictionary:

When you create a screen field, you can copy the type as well as the field attributes of a Dictionary field. This makes all the information about the object available to you, including semantic information about the corresponding data element and foreign key dependencies. The field name is also copied from the *ABAP Dictionary*.

Get from program:

You can copy the field attributes of a program-internal data object for a screen field. To do so, a **generated** version of the program must be available (is automatically generated at the time of activation). The name of the data object is used as the field name.

The *Graphical Layout Editor* offers you easy options for defining further screen elements such as texts, frames, radio buttons, and checkboxes, for example. First, click on the required screen element in the tool bar to select it and then position it in the screen maintenance area.

You can delete screen elements 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.

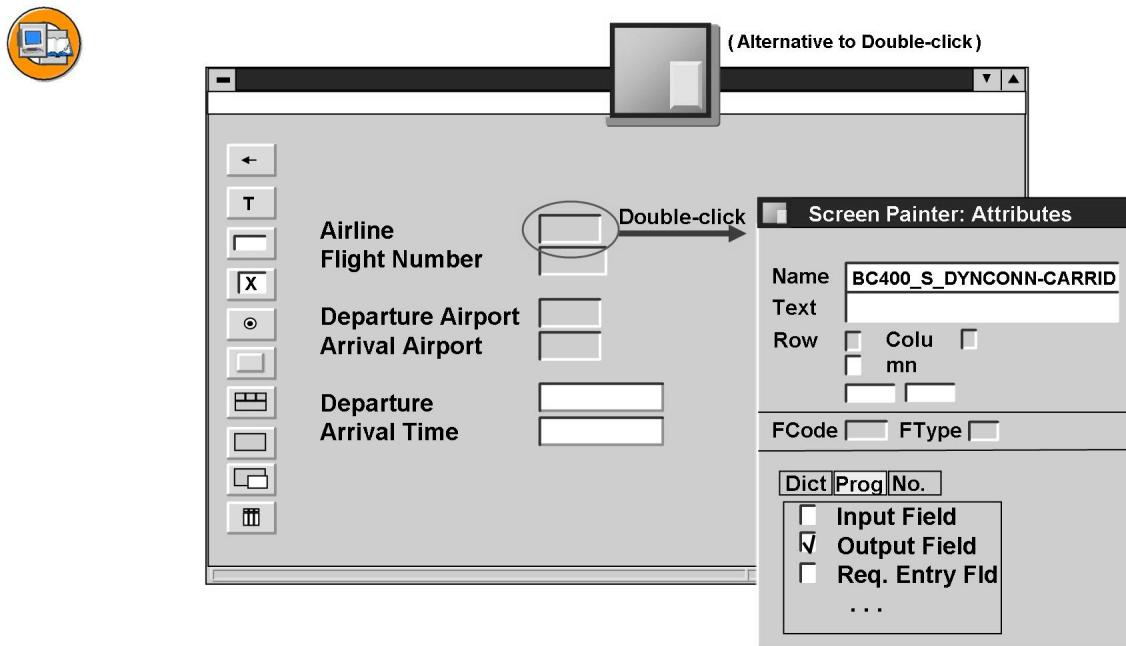


Figure 188: Maintaining the Attributes of a Screen Element

Double-click a screen element to maintain its attributes. Its attributes are displayed in an additional window for maintenance. (Instead of double-clicking, you can also select the element and then choose the *Attributes* pushbutton.)

To make input mandatory, assign the attribute *Required* to a screen field. At runtime, the field will be marked accordingly if it is blank. If the user does not fill out the field, an error dialog will follow any user action. After this dialog, the input fields are ready for input again.

Data Transports Between the Program and the Screen

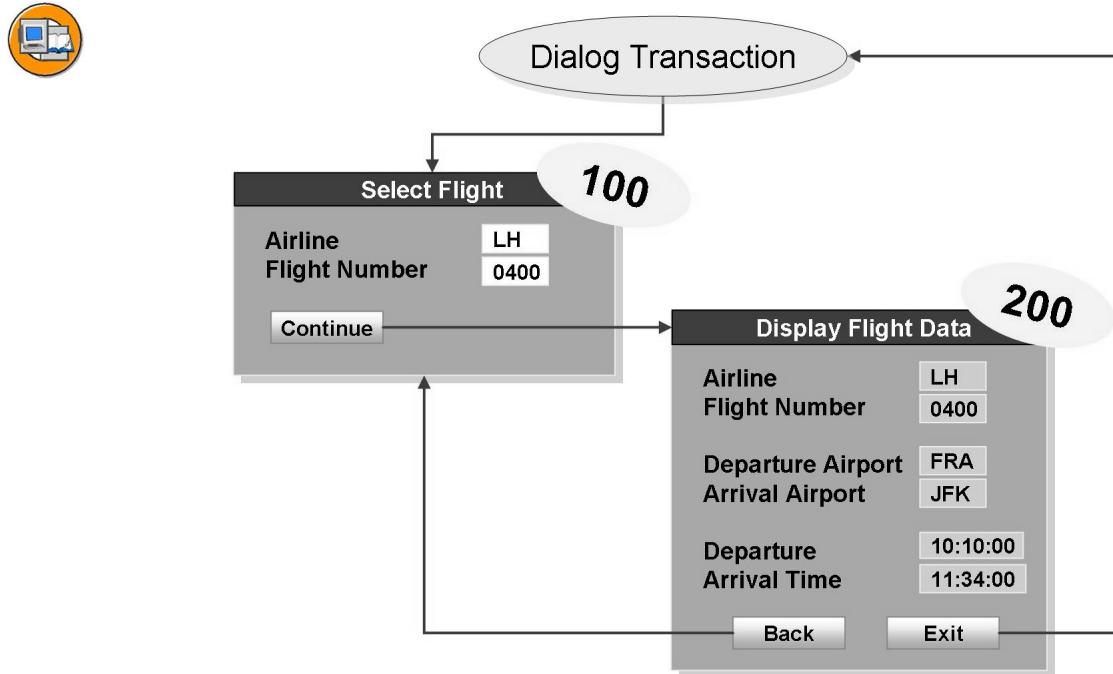


Figure 189: Realization Level 3: Displaying Data on the Screen

After the second realization level, it is possible to select a flight connection on the first screen and navigate to the second screen. However, only the *Airline* and *Flight Number* screen fields contain a value there. This is because they have exactly the same name as the input fields on the first screen. In the second level, the data transport must now be programmed, both from the (first) screen to the program and from the program to the second screen. This enables the user entries to be processed in the program and relevant detail information to be displayed on the second screen.

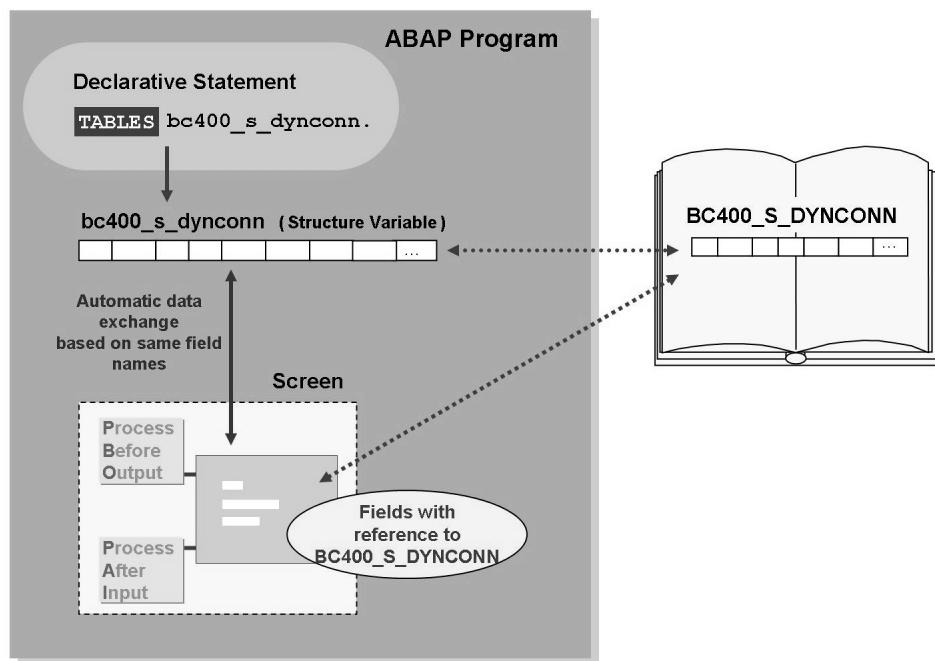


Figure 190: The TABLES Structure as an Interface to the Screen

The **TABLES** statement is used to define a structure variable of the same type and name with reference to the specified Dictionary structure (for example, a transparent table) within the program. This structure variable then serves as the interface between the program and the screen.

If the screen fields and the TABLES statement refer to the same structured type, then the data in their fields is **automatically** exchanged on the basis of the fields having the same names. After PBO, from the TABLES structure to the screen fields, and before PAI in the opposite direction.

A special structure is normally created in the *ABAP Dictionary* that contains the fields that are to be displayed on the screens of an application. This structure can also contain the fields of several database tables. On the screen, fields are defined with a reference to this structure and an interface structure is implemented within the program using the corresponding TABLES statement. This way, you still have a **clear** interface structure for the data exchange between program and screen, in spite of having fields from different tables.

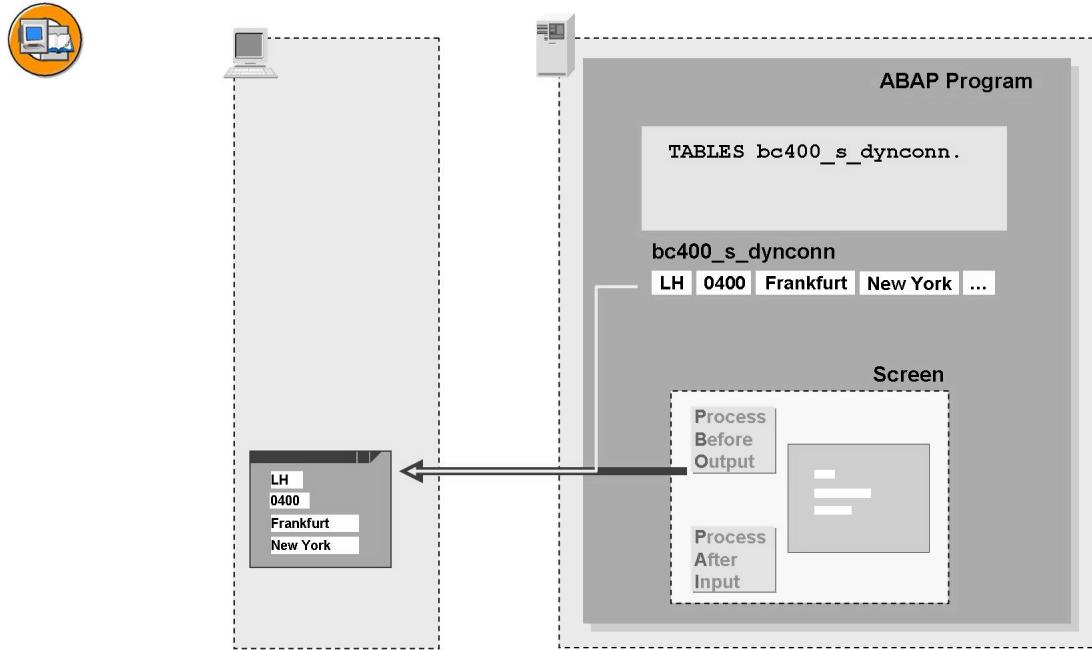


Figure 191: Data Transport from the Program to the Screen

After processing the **PBO** event and immediately before sending a screen to the presentation server, the field contents of the TABLES structure within the program are automatically copied to the screen fields with the same names. The moment of this automatic data transport makes sense as the data for the screen display in the TABLES structure is often prepared in PBO.

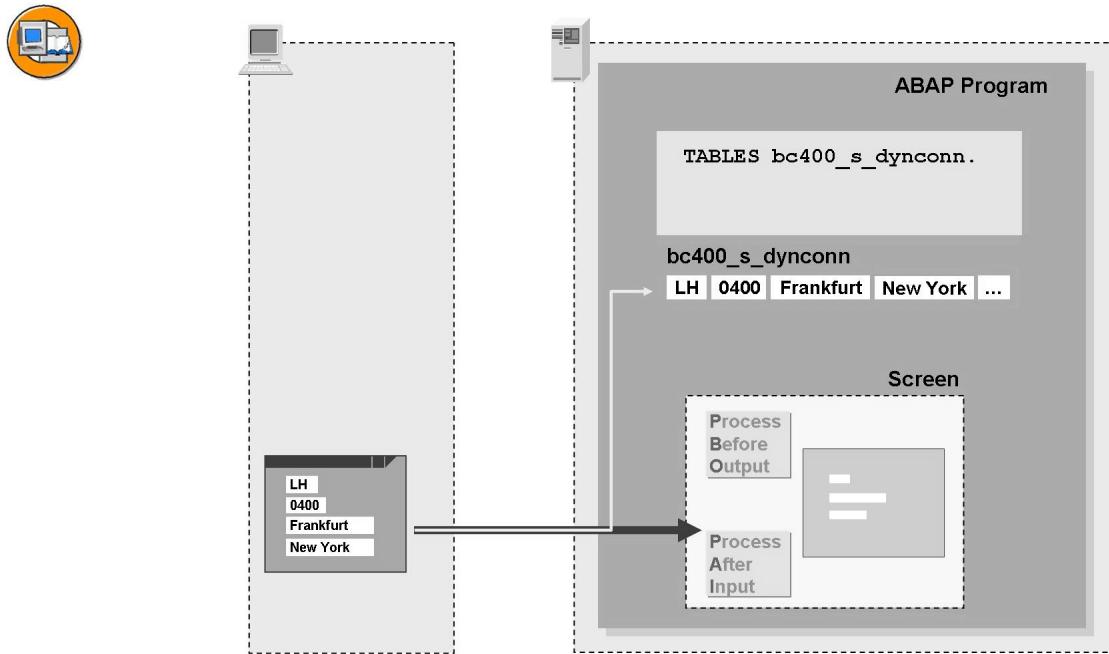


Figure 192: Data Transport from the Screen to the Program

For user actions on the screen, the contents of the screen fields are transported into TABLES structure fields with the same names **before** the PAI event is processed. The time of this automatic data transport makes sense, as the user entries are supposed to be processed further in PAI and thus have to be available in the program at that time.

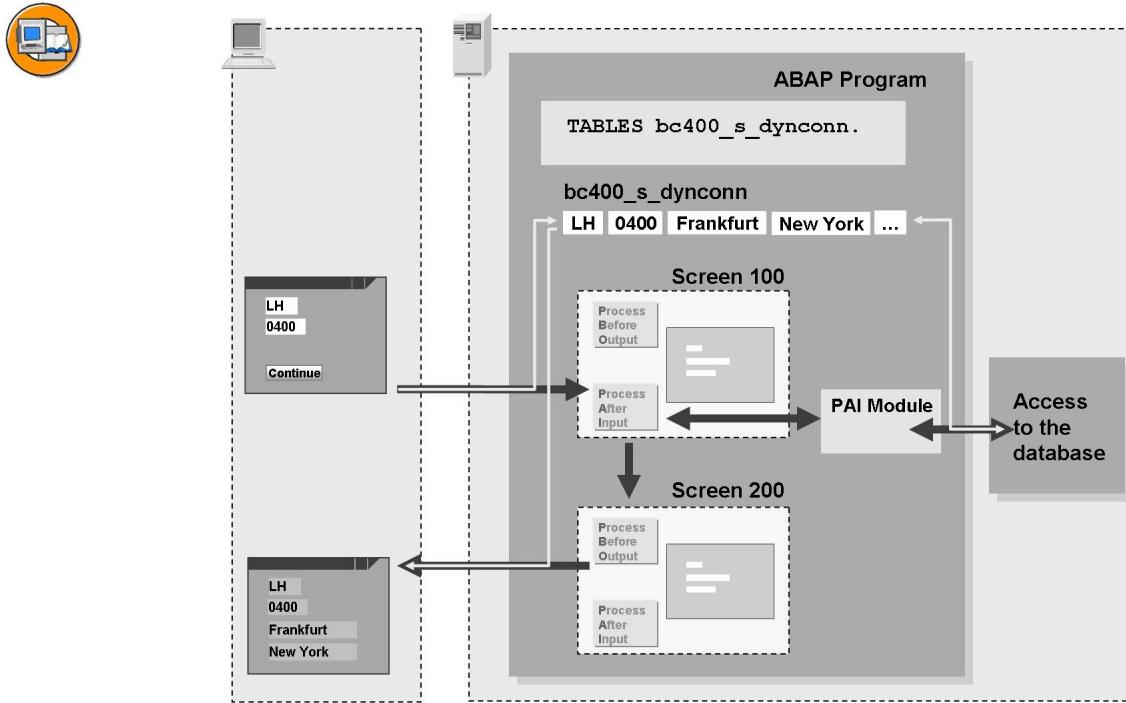


Figure 193: Transporting Data in the Example Program

In our example program, the process and data transport looks like this:

1. With a user action on screen 100, the user entries are automatically transported to the TABLES structure BC400_S_DYNCONN within the program.
2. After this, the PAI module USER_COMMAND_0100 is processed.
3. Depending on the function code, a function module for data retrieval is called in the PAI module. The user entries from the TABLES structure are transferred to the function module in the process. The data to be displayed on screen 200 is then placed in the same TABLES structure.
4. After PAI for screen 100, the system navigates to screen 200.
5. After PBO for the second screen, the data is automatically transported from the TABLES structure to the screen fields.



```

MODULE user_command_0100 INPUT.

CASE ok_code.
  WHEN 'GO'.
    MOVE-CORRESPONDING bc400_s_dynconn TO gs_connection.

    CALL FUNCTION 'BC400_DDS_CONNECTION_GET'
      EXPORTING
        iv_carrid      = gs_connection-carrid
        iv_connid      = gs_connection-connid
      IMPORTING
        es_connection = gs_connection
      EXCEPTIONS
        no_data        = 1
        OTHERS         = 2.

    IF sy-subrc <> 0.
      MESSAGE e042(bc400) WITH gs_connection-carrid
                                gs_connection-connid.
    ELSE.
      MOVE-CORRESPONDING gs_connection TO bc400_s_dynconn.
      SET SCREEN 200.
    ENDIF.
  ENDCASE.

ENDMODULE.          " USER_COMMAND_0100 INPUT

```

Figure 194: Syntax of the Example Program

The function code is evaluated in the PAI module USER_COMMAND_0100. If the user chooses the *Continue* pushbutton (function code GO), a function module for data retrieval is called.

The user entries from the TABLES structure are transferred to the function module. To separate the interface to the screen and the interface to the function module clearly from one another, the data is not transferred directly from the TABLES structure to the function module. It is first copied to the data object that serves as the interface to the function module (global, structured data object gs_connection, type BC400_S_CONNECTION).

If data was retrieved successfully, the result is copied back to the TABLES structure and navigation to the display screen is triggered. If the function module terminates with an exception, a corresponding error message is sent. This error message is displayed directly on screen 100.



Hint: If error messages (message type “E”) are output during the processing of a PAI module, the relevant screen is displayed again immediately, without PBO being triggered again. Using the corresponding programming techniques, screen fields can be ready-for-input again immediately afterwards. You can learn more about this in the training course BC410 “Programming User Dialogs”.



Hint: In our example, the data is read at PAI for the input screen. Alternatively, data could also be retrieved in a PBO module for the display screen, in other words, after navigation. The second option would be preferable in order to make reusing the display screen easier.

Exercise 20: Creating Screens and Dynamic Next Screen Processing

Exercise Objectives

After completing this exercise, you will be able to:

- Create classic screens
- Create pushbuttons for screens and link them to function codes
- Create dialog modules and process function codes
- Set the next screen dynamically
- Create a dialog transaction

Business Example

You want to develop a dialog application with two classic screens. You want to incorporate pushbuttons on the screens to allow the user to initiate navigation to the next screen.

Template:

None

Solution:

SAPMBC400_UDS_A

Task 1:

Create a module pool (type M program) with a TOP include.

1. Create a program and name it **SAPMZBC400##_A**. Leave the *With TOP Include* indicator set. Make sure that this automatically results in the system proposing the program type *M module pool*.

Task 2:

Create two screens in your module pool with the screen type *Normal*.

1. Create the first screen (suggested number: **100**). Enter a description (suggested short description: **Input screen**), make sure that the screen type has been selected correctly, and that the current screen has been entered as the next screen.
2. Create a second screen in the same way (suggested number: **200**, suggested short description: **Output screen**).

Continued on next page

Task 3:

Create navigation pushbuttons on the screens: On the first screen, create a pushbutton to get to the second screen, and on the second screen, create a pushbutton to return to the first screen. On the second screen, create an additional pushbutton to cancel the application.

1. Edit the first screen with the *Graphic Screen Painter* tool and create a pushbutton.
2. Assign a name to the pushbutton (suggestion: **BUTTON_GO**) and enter a text (suggestion: “**Continue**”).
3. Assign a function code to the pushbutton (suggestion **GO**) and save the screen.
4. Create two pushbuttons on the other screen in the same way. Suggested values:

Name	Text	Function code
BUTTON_BACK	“Back”	BACK
BUTTON_EXIT	“Exit”	EXIT

Task 4:

Create a global data object in the TOP include for the program that can carry the function code. Make sure that when you exit both screens the function code is written to this data object.

1. Edit the TOP include for the program and declare an *SYUCOMM* type data object there (suggested name: **OK_CODE**).
2. Edit the element list for the first screen and enter the name of your data objects with the *OK* type element.
3. Edit the element list for the second screen accordingly.

Task 5:

For each screen, create a PAI module in which the next screen is determined dependent on the function code.

1. Add a PAI module call to the flow logic of the first screen (suggested name: **USER_COMMAND_0100**)
2. Create the module using forward navigation. Generate a new include for the module as proposed by the system and assign a suitable name to it (suggested name: **MZBC400_##_AI01**).

Continued on next page

3. Implement the PAI module. Evaluate the content of the OK field and make sure that after you have chosen the pushbutton, the next screen is set dynamically to 200.
4. Create a PAI module for the second screen in the same way and implement it accordingly. To terminate the program, set the next screen dynamically to the value 0.

Task 6:

Create a dialog transaction to start the application with the first screen. Activate and test your program.

1. Create a transaction with the start object *Program and screen (dialog transaction)* (suggested name: **ZBC400_##_A**).

Solution 20: Creating Screens and Dynamic Next Screen Processing

Task 1:

Create a module pool (type M program) with a TOP include.

1. Create a program and name it **SAPMZBC400_##_A**. Leave the *With TOP Include* indicator set. Make sure that this automatically results in the system proposing the program type *M module pool*.
 - a) In the context menu for your package, choose *Create → Program*.
 - b) Enter the name of the program, leave the *With TOP Include* indicator set, and choose *Continue* .
 - c) Confirm the proposed name of the TOP include.
 - d) Check the program attributes and close the activity.
 - e) Assign the program and the includes to your package and transport request in the usual way.

Task 2:

Create two screens in your module pool with the screen type *Normal*.

1. Create the first screen (suggested number: **100**). Enter a description (suggested short description: **Input screen**), make sure that the screen type has been selected correctly, and that the current screen has been entered as the next screen.
 - a) In the context menu for the program, choose *Create → Screen* and enter the screen number.
 - b) Maintain the description, check the other details and close the activity.
2. Create a second screen in the same way (suggested number: **200**, suggested short description: **Output screen**).
 - a) Carry out this step in the same way as before.

Continued on next page

Task 3:

Create navigation pushbuttons on the screens: On the first screen, create a pushbutton to get to the second screen, and on the second screen, create a pushbutton to return to the first screen. On the second screen, create an additional pushbutton to cancel the application.

1. Edit the first screen with the *Graphic Screen Painter* tool and create a pushbutton.
 - a) Open the tool using the Layout pushbutton.
 - b) On the left of the toolbar, choose *Pushbutton* and place the pushbutton in the character area.
2. Assign a name to the pushbutton (suggestion: **BUTTON_GO**) and enter a text (suggestion: “**Continue**”).
 - a) Enter the values in the corresponding fields above the character area.
3. Assign a function code to the pushbutton (suggestion **GO**) and save the screen.
 - a) Open the attribute window by double-clicking on the pushbutton or by choosing the *F2* key.
 - b) In the attribute window, maintain the contents of the field *FctCode*
4. Create two pushbuttons on the other screen in the same way. Suggested values:

Name	Text	Function code
BUTTON_BACK	“Back”	BACK
BUTTON_EXIT	“Exit”	EXIT

- a) Carry out this step in the same way as before.

Task 4:

Create a global data object in the TOP include for the program that can carry the function code. Make sure that when you exit both screens the function code is written to this data object.

1. Edit the TOP include for the program and declare an *SYUCOMM* type data object there (suggested name: **OK_CODE**).
 - a) See source code excerpt from the model solution.

Continued on next page

2. Edit the element list for the first screen and enter the name of your data objects with the *OK* type element.
 - a) Edit the first screen and navigate to the *Element List* tab page.
 - b) Search for the *OK* type element.
 - c) In the *Name* column, enter the name of the data object.
3. Edit the element list for the second screen accordingly.
 - a) Carry out this step in the same way as before.

Task 5:

For each screen, create a PAI module in which the next screen is determined dependent on the function code.

1. Add a PAI module call to the flow logic of the first screen (suggested name: **USER_COMMAND_0100**)
 - a) See source code excerpt from the model solution.
2. Create the module using forward navigation. Generate a new include for the module as proposed by the system and assign a suitable name to it (suggested name: **MZBC400_##_AI01**).
 - a) See source code excerpt from the model solution.
3. Implement the PAI module. Evaluate the content of the OK field and make sure that after you have chosen the pushbutton, the next screen is set dynamically to 200.
 - a) See source code excerpt from the model solution.
4. Create a PAI module for the second screen in the same way and implement it accordingly. To terminate the program, set the next screen dynamically to the value 0.
 - a) See source code excerpt from the model solution.

Task 6:

Create a dialog transaction to start the application with the first screen. Activate and test your program.

1. Create a transaction with the start object *Program and screen (dialog transaction)* (suggested name: **ZBC400_##_A**).
 - a) In the context menu for the program to choose *Create → Transaction*.

Continued on next page

- b) Enter the name of the transaction and a short text and choose ✓.
- c) Enter the name of your program and the number of your first screen in both required entry fields. Set GUI support to *SAPGUI for Windows*.
- d) Save the transaction, activate all parts of the program, and then test the transaction.

Result

Screen 100 flow logic:

```
PROCESS BEFORE OUTPUT.
* MODULE STATUS_0100.
*
PROCESS AFTER INPUT.
MODULE USER_COMMAND_0100.
```

Screen 200 flow logic:

```
PROCESS BEFORE OUTPUT.
* MODULE STATUS_0200.
*
PROCESS AFTER INPUT.
MODULE USER_COMMAND_0200.
```

Source code extract from the model solution:

```
*&-----*
*& Include MBC400_UDS_ATOP      Modulpool      SAPMBC400_UDS_A
*&-----*
PROGRAM sapmbc400_uds_a.

DATA ok_code TYPE syucomm.

*-----*
***INCLUDE MBC400_UDS_AI01 .
*-----*
*&-----*
*&      Module   USER_COMMAND_0100  INPUT
*&-----*
```

Continued on next page

```
MODULE user_command_0100 INPUT.

CASE ok_code.
  WHEN 'GO'.
    SET SCREEN 200.
ENDCASE.

ENDMODULE.          " USER_COMMAND_0100  INPUT

*-----*
***INCLUDE MBC400_UDS_AI02 .
*-----*
*&-----*
*&      Module  USER_COMMAND_0200  INPUT
*&-----*

MODULE user_command_0200 INPUT.
CASE ok_code.
  WHEN 'BACK'.
    SET SCREEN 100.
  WHEN 'EXIT'.
    SET SCREEN 0.
ENDCASE.

ENDMODULE.          " USER_COMMAND_0200  INPUT
```

Exercise 21: Screen – Creating Input/Output Fields

Exercise Objectives

After completing this exercise, you will be able to:

- Create input/output fields that reference the Dictionary with the *Graphic Screen Painter* tool.
- Change the attributes of input/output fields

Business Example

You want to develop a dialog transaction and allow the user to input data as well as display data on the screen.

Template:

SAPMBC400_UDS_A

Solution:

SAPMBC400_UDS_B

Task 1:

Copy your program SAPMZBC400##_A or the template SAPMBC400_UDS_A and name it **SAPMZBC400##_B**. Make absolutely certain that the screen and the includes are copied as well.

1. Copy the program with all the screens and includes. Choose an appropriate name for the new includes (suggested name: **MZBC400##_BI01** or **MZBC400##_BI02**).

Task 2:

Create two input fields on the first screen, one for the airline ID and one for the flight number. Refer to the corresponding components of the Dictionary structure type BC400_S_DYNCONN. Maintain the properties of the fields in such a way that the user can only exit the screen after entries have been made in both fields.

1. Use the *Dict/Program Fields Window* function in the *Graphic Screen Painter* tool to create input fields for the *CARRID* and *CONNID* components of the Dictionary structure type.

Continued on next page

2. Set the *Input* attribute to the value *mandatory (required entry field)* for both input fields.
3. (OPTIONAL) Create a frame type element, maintain a header for it and shift the input fields into this frame.

Task 3:

Create input and output fields on the second screen for the airline ID, flight number, point of departure, destination, departure time, and arrival time. Refer to the Dictionary structure type **BC400_S_DYNCONN** here too. Maintain the properties of the fields in such a way that the user cannot make any entries here.

1. Create input and output fields in the same way you did on the first screen. Choose the components *CARRID*, *CONNID*, *CITYFROM*, *CITYTO*, *DEPTIME* and *ARRTIME* of the Dictionary structure type.
2. Set the *Input* attribute to the value *not possible* for all the input fields.
3. (OPTIONAL) Create a frame around the key fields for the flight connection and around the additional information.

Task 4:

Create a dialog transaction to start the application with the first screen. Activate and test your program.

1. Create a transaction with the start object *Program and screen (dialog transaction)* (suggested name: **ZBC400_##_B**).

Solution 21: Screen – Creating Input/Output Fields

Task 1:

Copy your program SAPMZBC400_##_A or the template SAPMBC400_UDS_A and name it **SAPMZBC400_##_B**. Make absolutely certain that the screen and the includes are copied as well.

1. Copy the program with all the screens and includes. Choose an appropriate name for the new includes (suggested name: **MZBC400_##_BI01** or **MZBC400_##_BI02**).
 - a) Display the program in the navigation area of the Workbench. In the navigation area, open the context menu for the program and choose *Copy*.
 - b) In the next dialog box, enter the name of the target program and choose *Copy*.
 - c) In the dialog box that follows, select all the fields and choose *Copy*.
 - d) Check the proposed name for the copied includes. Make sure that **all the includes have been selected** before you continue.
 - e) Assign the copied program and the includes to your package and transport request in the usual way.

Continued on next page

Task 2:

Create two input fields on the first screen, one for the airline ID and one for the flight number. Refer to the corresponding components of the Dictionary structure type BC400_S_DYNCONN. Maintain the properties of the fields in such a way that the user can only exit the screen after entries have been made in both fields.

1. Use the *Dict/Program Fields Window* function in the *Graphic Screen Painter* tool to create input fields for the *CARRID* and *CONNID* components of the Dictionary structure type.
 - a) Process the first screen and start the *Graphic Screen Painter* tool in the usual way.
 - b) Choose the *Dict/Program Fields Window* pushbutton to open the corresponding window.
 - c) Enter the name of the structure type and choose the *Get from Dictionary* button.
 - d) Select the structure components you want and choose .
 - e) Position the input fields and the relevant labels on the screen.
2. Set the *Input* attribute to the value *mandatory (required entry field)* for both input fields.
 - a) Double-click on the first input field to open the attribute window.
 - b) At the bottom of the screen, choose the *Program* tab page.
 - c) Maintain the *Input* attribute using the input help.
 - d) Follow the same procedure for the second field.
3. (OPTIONAL) Create a frame type element, maintain a header for it and shift the input fields into this frame.
 - a) From the toolbar on the left of the screen, choose the *Frame* tool and create the frame on the screen layout.
 - b) Maintain a name and description for the frame.
 - c) Select all the input fields and their labels (click on them with the **Ctrl** key pressed down) and shift them into the frame.

Continued on next page

Task 3:

Create input and output fields on the second screen for the airline ID, flight number, point of departure, destination, departure time, and arrival time. Refer to the Dictionary structure type **BC400_S_DYNCONN** here too. Maintain the properties of the fields in such a way that the user cannot make any entries here.

1. Create input and output fields in the same way you did on the first screen. Choose the components *CARRID*, *CONNID*, *CITYFROM*, *CITYTO*, *DEPTIME* and *ARRTIME* of the Dictionary structure type.
 - a) Proceed as you did with the first screen.
2. Set the *Input* attribute to the value *not possible* for all the input fields.
 - a) Proceed as you did with the first screen.
3. (OPTIONAL) Create a frame around the key fields for the flight connection and around the additional information.
 - a) Proceed as you did with the first screen.

Task 4:

Create a dialog transaction to start the application with the first screen. Activate and test your program.

1. Create a transaction with the start object *Program and screen (dialog transaction)* (suggested name: **ZBC400_##_B**).
 - a) In the context menu for the program to choose *Create → Transaction*.
 - b) Enter the name of the transaction and a short text and choose .
 - c) Enter the name of your program and the number of your first screen in both required entry fields and save the transaction.
 - d) Save the transaction, activate the program, and then test the transaction.

Exercise 22: Screens – Data Transport

Exercise Objectives

After completing this exercise, you will be able to:

- Fill the screen fields with data from the program
- Continue to process user entries in the program

Business Example

You want to develop a dialog transaction in which the entries made on the first screen are used to select data in data retrieval. The data that is found should then be displayed when you navigate to the second screen.

Template:

SAPMBC400_UDS_B

Solution:

SAPMBC400_UDS_C

Task 1:

Copy your program SAPMZBC400##B or the template SAPMBC400_UDS_B and name it **SAPMZBC400##C**. Make absolutely certain that the screen and the includes are copied as well.

1. Copy the program with all the screens and includes. Choose an appropriate name for the new includes (suggested name: **MZBC400##CI01** or **MZBC400##CI02**).

Task 2:

In your program, create a global, structured data object that can be used as an interface to the screens, whose components have the same name as the screen fields.

1. Edit the TOP include of your program. Use the TABLES statement here to create a structured data object based on the Dictionary structure type BC400_S_DYNCONN.

Continued on next page

Task 3:

Make sure that the flight connection details are read from the database and that this data then fills the fields of the TABLES structure before you navigate to the second screen. Use the function module BC400_DDS_CONNECTION_GET or the function module you developed yourself for data retrieval.

1. Edit the PAI module in which the function code for the first screen is evaluated. Implement the function module call before navigation. To do this, create a global, structured data object (suggested name: **gs_connection**) in the TOP include. You use this as the actual parameter for the EXPORT parameter of the function module. Transfer the user entries from the TABLES structure to the IMPORT parameter of the function module.



Hint: Although you can use the components of the TABLES structure directly as actual parameters, the user entries should first be transferred to the global data object and then transferred from there to the function module. In this way, the interface to the user interface (TABLES structure) and the interface to the function module (global data object) are clearly separated from one another.

2. Why can you not create the structured data object as a local data object in the PBO module?

3. Handle exceptions in the function module. If necessary, react with information message 042 of the message class BC400. Exit the input screen only if an exception has not been raised.
4. If the function module has run without errors, fill the fields of the TABLES structure with the result and initiate navigation to the second screen.
5. Activate your program, create a dialog transaction, and test the application.

Solution 22: Screens – Data Transport

Task 1:

Copy your program SAPMZBC400_##_B or the template SAPMBC400_UDS_B and name it **SAPMZBC400_##_C**. Make absolutely certain that the screen and the includes are copied as well.

1. Copy the program with all the screens and includes. Choose an appropriate name for the new includes (suggested name: **MZBC400_##_CI01** or **MZBC400_##_CI02**).
 - a) Carry out this step in the usual manner.

Task 2:

In your program, create a global, structured data object that can be used as an interface to the screens, whose components have the same name as the screen fields.

1. Edit the TOP include of your program. Use the TABLES statement here to create a structured data object based on the Dictionary structure type BC400_S_DYNCONN.
 - a) See source code excerpt from the model solution.

Task 3:

Make sure that the flight connection details are read from the database and that this data then fills the fields of the TABLES structure before you navigate to the second screen. Use the function module BC400_DDS_CONNECTION_GET or the function module you developed yourself for data retrieval.

1. Edit the PAI module in which the function code for the first screen is evaluated. Implement the function module call before navigation. To do this, create a global, structured data object (suggested name: **gs_connection**) in the TOP

Continued on next page

include. You use this as the actual parameter for the EXPORT parameter of the function module. Transfer the user entries from the TABLES structure to the IMPORT parameter of the function module.



Hint: Although you can use the components of the TABLES structure directly as actual parameters, the user entries should first be transferred to the global data object and then transferred from there to the function module. In this way, the interface to the user interface (TABLES structure) and the interface to the function module (global data object) are clearly separated from one another.

- a) See source code excerpt from the model solution.
 2. Why can you not create the structured data object as a local data object in the PBO module?
- Answer:** Local data objects only exist in subroutines, function modules and methods. You always create global program data objects with DATA statements in dialog modules. To avoid confusion, no data objects should be declared in dialog modules. Global data objects belong in the TOP include.
3. Handle exceptions in the function module. If necessary, react with information message 042 of the message class BC400. Exit the input screen only if an exception has not been raised.
 - a) See source code excerpt from the model solution.
 4. If the function module has run without errors, fill the fields of the TABLES structure with the result and initiate navigation to the second screen.
 - a) See source code excerpt from the model solution.
 5. Activate your program, create a dialog transaction, and test the application.
 - a) Perform this step in the same way as in previous exercises.

Result

Source code extract from the model solution:

```
*&-----*
*& Include MBC400_UDS_CTOP           Modulpool          SAPMBC400_UDS_C
*&-----*
```

```
PROGRAM sapmbc400_uds_c.
```

Continued on next page

```

DATA ok_code TYPE syucomm.

TABLES bc400_s_dynconn.

DATA gs_connection TYPE bc400_s_connection.

*-----*
***INCLUDE MBC400_UDS_CI01 .
*-----*
*&-----*
*&     Module   USER_COMMAND_0100   INPUT
*&-----*
MODULE user_command_0100 INPUT.

CASE ok_code.

WHEN 'GO'.

MOVE-CORRESPONDING bc400_s_dynconn TO gs_connection.

CALL FUNCTION 'BC400_DDS_CONNECTION_GET'
  EXPORTING
    iv_carrid      = gs_connection-carrid
    iv_connid      = gs_connection-connid
  IMPORTING
    es_connection = gs_connection
  EXCEPTIONS
    no_data        = 1
    OTHERS         = 2.

IF sy-subrc <> 0.
  MESSAGE i042(bc400) WITH gs_connection-carrid
                           gs_connection-connid. "flight not found
ELSE.

MOVE-CORRESPONDING gs_connection TO bc400_s_dynconn.

SET SCREEN 200.

ENDIF.

```

Continued on next page

```
ENDCASE.  
ENDMODULE.  
" USER_COMMAND_0100 INPUT
```



Lesson Summary

You should now be able to:

- List attributes and benefits of screens
- Implement simple screens with input/output fields and link them to a dialog application
- Explain and implement the program-internal processing for screen calls

Lesson: ABAP Web Dynpro

Lesson Overview

In this lesson you will gain an overview of the properties and usage scenarios as well as the programming and runtime architecture of ABAP Web Dynpro. You will also learn how to structure and program a simple ABAP Web Dynpro application with input/output fields and pushbuttons.



Lesson Objectives

After completing this lesson, you will be able to:

- List the properties and usage scenarios of the ABAP Web Dynpro
- Explain the programming and runtime architecture of the ABAP Web Dynpro
- Implement simple Web Dynpro applications with input/output fields and pushbuttons

Business Example

You should explain the properties and usage scenarios as well as the programming concept and runtime architecture of ABAP Web Dynpro.

ABAP Web Dynpro: Properties and Programming Model



ABAP Web Dynpro Properties

- Programming Model Independent of Client and Protocol
- Separation of Layout, Program Flow, and Business Logic
- Graphical Support in Layout Design and Program Flow Logic Design
- Development tools implemented in Object Navigator
- Reusable Components
- Any back-end systems through use of Web Services
- Support of Accessibility Standard Section 508

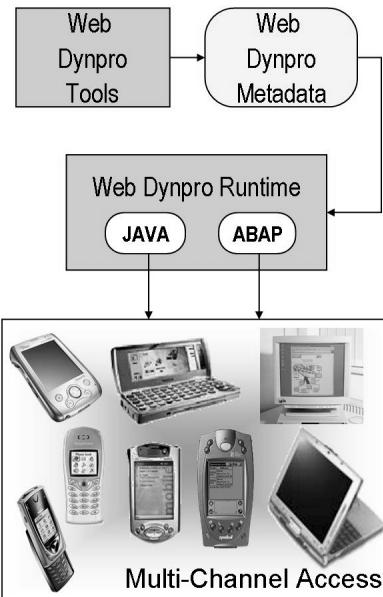


Figure 195: ABAP Web Dynpro Properties

With *SAP NetWeaver 7.0*, SAP introduced ABAP Web Dynpro - a new programming model that is independent of the implemented clients (for operating Web Dynpro applications) and the respective protocol. This independence is achieved through the use of metadata.

ABAP Web Dynpro will become the standard UI technology used by SAP for the development of Web applications in the ABAP environment.

In addition to having its own runtime environment, ABAP Web Dynpro - in particular - has a graphic development environment (*Web Dynpro Explorer*), whose tools are fully integrated into the ABAP development environment (*Object Navigator*). You use the *SAPGUI* to develop a Web Dynpro application in the SAP System. It is called through the respective URL or the Internet browser, or from the *SAP Portal*.

Maintenance and further development of ABAP Web Dynpro applications is simplified through the reusability of components and the strict separation of layout, program flow, and business logic. In addition, Web Dynpro applications can be made usable for visually impaired users since Web Dynpro supports Accessibility Standard 508 through keyboard input and screen reading functions.

ABAP Web Dynpro supports the use of web services, which enable you to access the business functions of various systems from any application, in a standardized, easy-to-use way.



Decoupling of Business Logic, Layout, and Program Flow

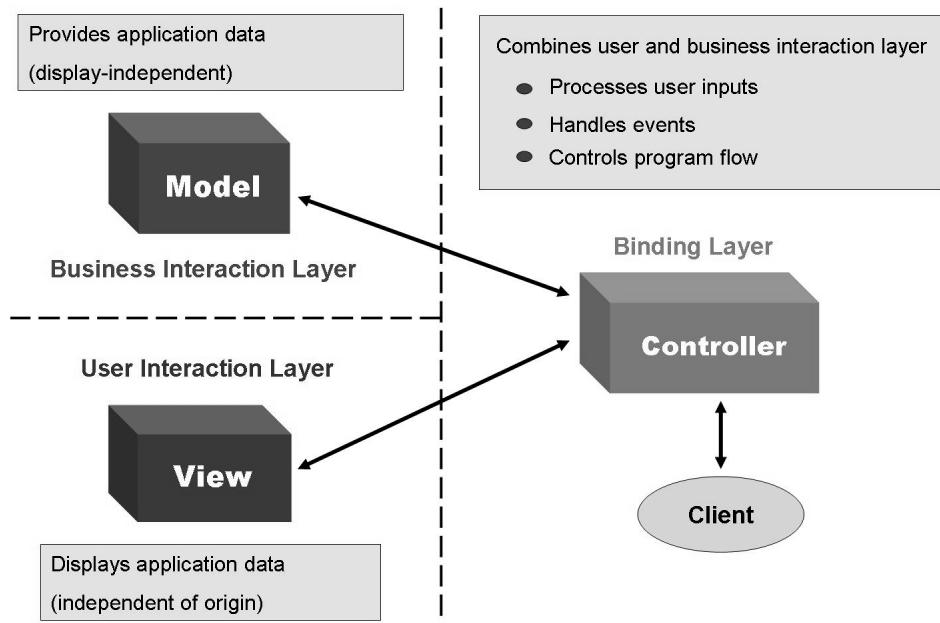


Figure 196: The Model View Controller Programming Model

ABAP Web Dynpro applications are structured according to the Model View Controller programming model (MVC programming model).

While the **model** in the *business interaction layer* is responsible for display-independent data access to the back-end system, the **view** takes care of the display of data (layout), irrespective of the source, in the *user interaction layer*. The **controller** in the *binding layer* is between the view and the model. It formats the model data that is to be displayed in the view, processes the user entries, and returns them to the model. In addition, it controls the flow of the program.

In ABAP Web Dynpro, the model can consist of application classes that encapsulate the data accesses. However, existing BAPIs, function modules, or web services can be used as well.

The view and the controller are implemented using graphic tools.

Elements of the ABAP Web Dynpro

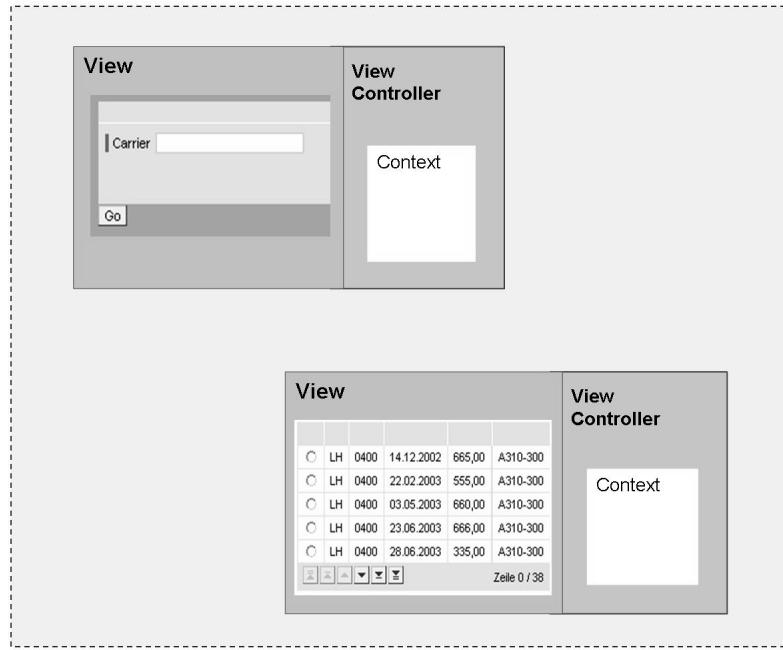


Figure 197: ABAP Web Dynpro Views

The layout of a **view** consists of a set of **UI elements** (screen elements such as input fields, tables, and pushbuttons) that are grouped together using a graphic tool. At runtime, views are displayed one after the other, not parallel, on the screen. Therefore, they correspond to classic screens, without flow logic.

As with the classic ABAP Dynpro, input checks and input helps (F4) do not have to be implemented manually, but can be used as UI services by referring to the ABAP Dictionary

You can integrate views into other views using the container technology. This enables modularization.

Each view has its own assigned view controller. Technically speaking, the view controller is an ABAP class. The source code for the view controller is only partially implemented directly by the developer. A large part is automatically generated in accordance with the developer's design specifications. The **context** of the view controller serves as a data container and contains the data to be displayed in the view (corresponds to the TABLES structure in the classic dynpro).

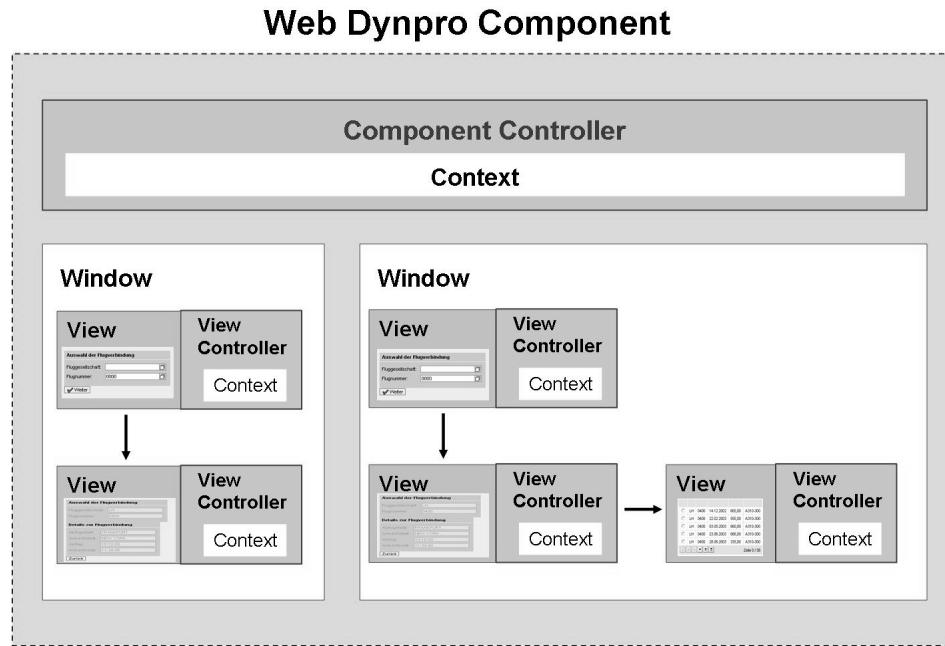


Figure 198: ABAP Web Dynpro Component

Windows are used to bundle several views and to define the navigation options between them (view sequences). A **component** contains one or several windows and has its own controller with context, where data that is to be displayed in several component views is stored.

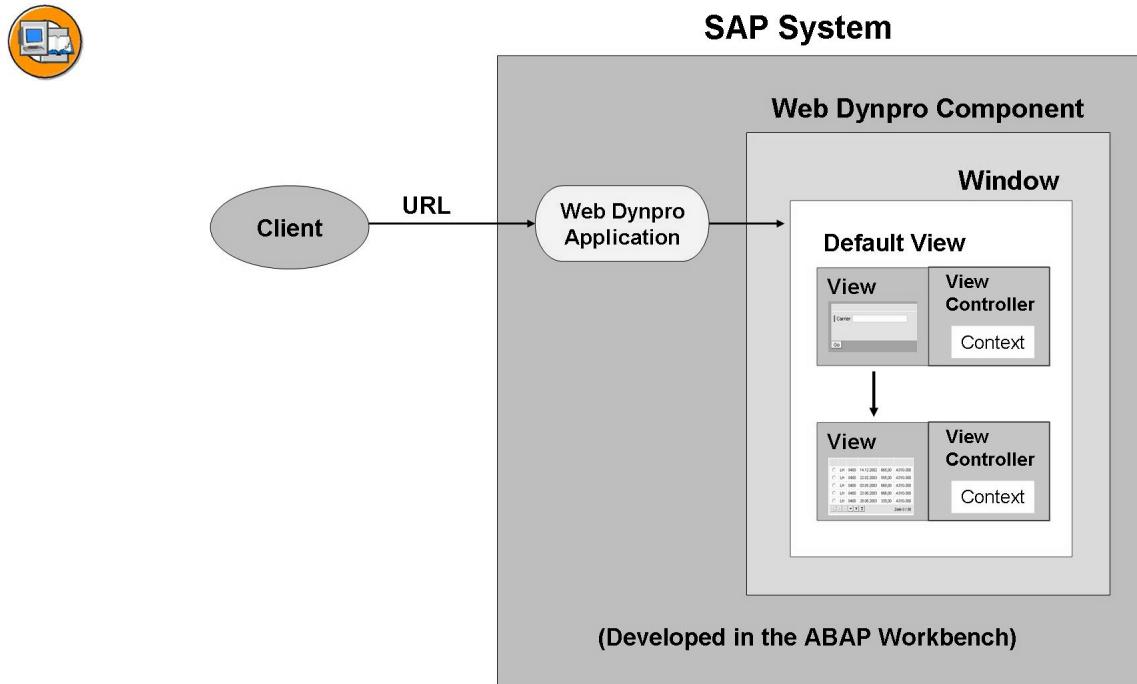


Figure 199: Development Environment and Runtime Architecture

A **Web Dynpro application** points to a window with a default view (start view), which is displayed when the application is called.

A component with appropriate windows is usable as a self-contained unit in different Web Dynpro applications. This reduces implementation and maintenance work to a minimum.

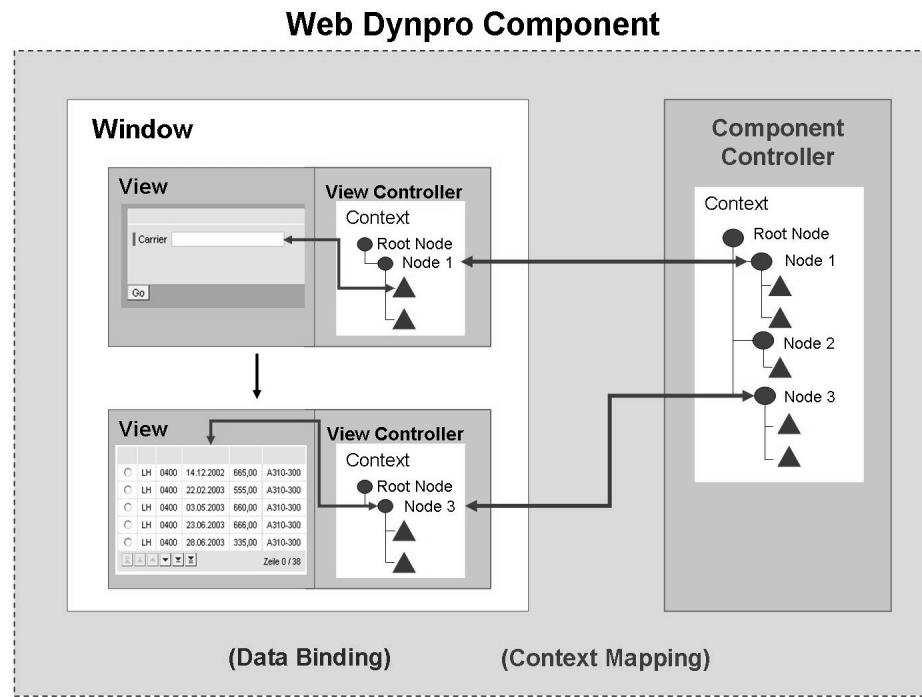


Figure 200: ABAP Web Dynpro Context

The **context** of each view controller contains data that is to be displayed in the view (data container). The data transport between the view context and the UI elements of the view takes place when the appropriate assignment is made (data binding).

A component, too, has a **context**. Here, data is stored that is to be displayed in the different views of the component. In such cases, references to component data are implemented mostly in the corresponding view contexts (context mapping).

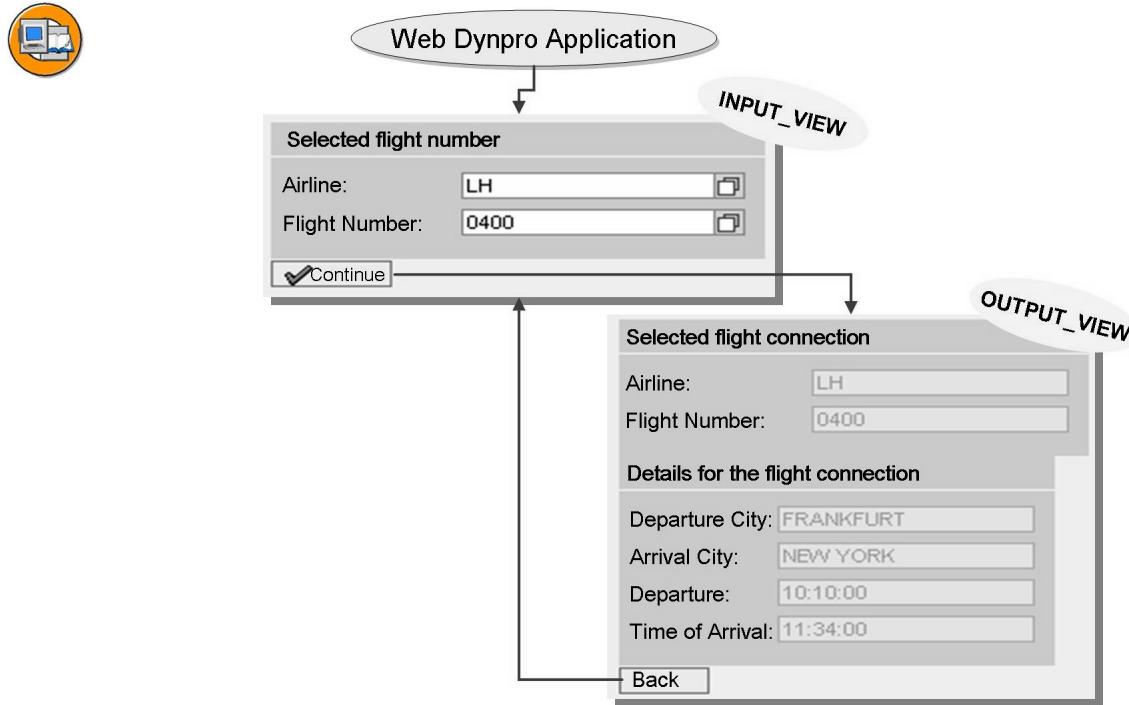


Figure 201: Application Example

In the following sections, an ABAP Web Dynpro application will be developed in several steps that will enable you to display the data for individual flight connections.

The program comprises a Web Dynpro component with two views. In the first view, the user chooses a flight connection. In the second view, the details for this connection are displayed.

Both views are embedded in a Web Dynpro window for the component and connected by navigation links in such a way that the user can navigate between the views using the pushbuttons.

The screen sequence defined in the window is started by way of the Web Dynpro application.

Web Dynpro, Pushbuttons and Navigation

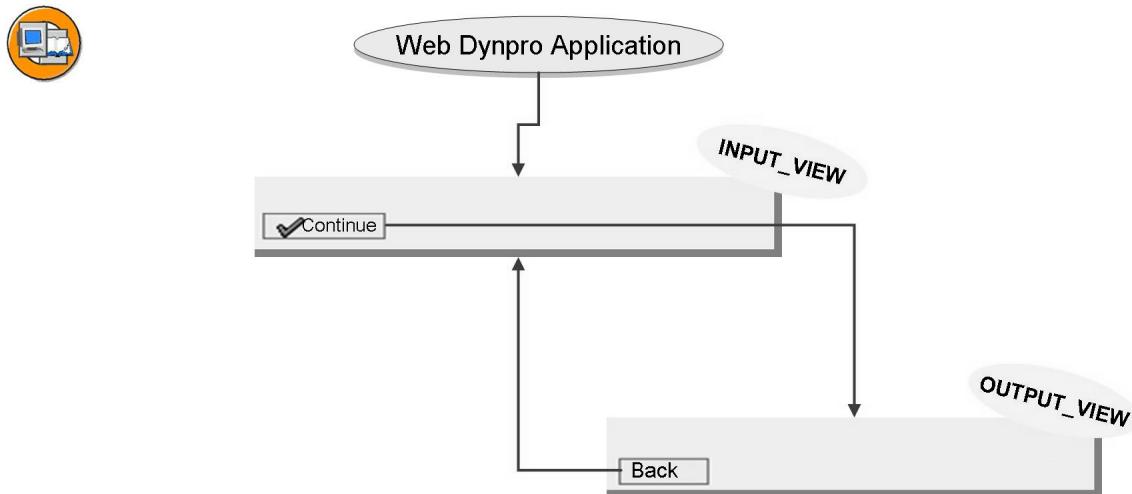


Figure 202: Realization Level 1: Web Dynpro Views with Pushbuttons and Navigation

At the first level of our example, we want to create a Web Dynpro component with two views. Each of the views should have one pushbutton. The program should be implemented in such a way that when you click on a pushbutton, you navigate to the other view. To start the view sequence, you need to create a Web Dynpro application.

This section deals with the following:

- Creating a Web Dynpro Component
- Creating Web Dynpro Views
- Embedding Views in Web Dynpro Windows
- Setting up the Navigation
- Creating Pushbuttons and Evaluating User Actions
- Creating Web Dynpro Applications

Creating a Web Dynpro Component

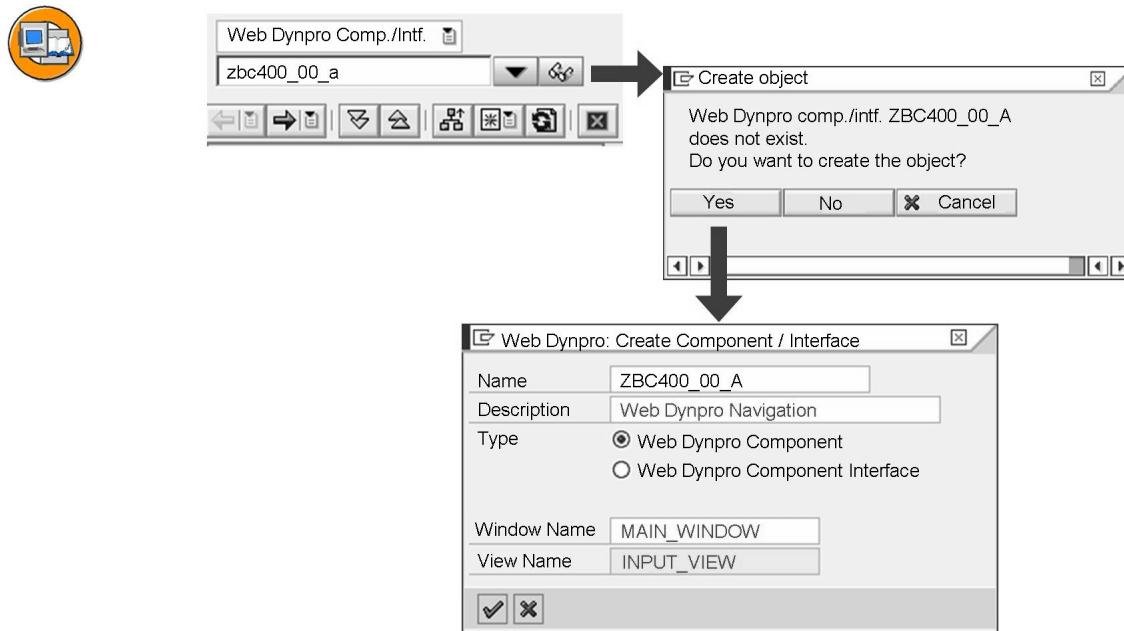


Figure 203: Creating a Web Dynpro Component

You create a Web Dynpro component in a similar manner to the way you create a new ABAP program, from the context menu in the Navigation Area of the *Object Navigator* (see graphic above).

When you create a new Web Dynpro component, a Web Dynpro window and Web Dynpro view are automatically created. At a later stage, you can create additional windows and views in the component.

Creating Web Dynpro Views and Embedding them in the Window

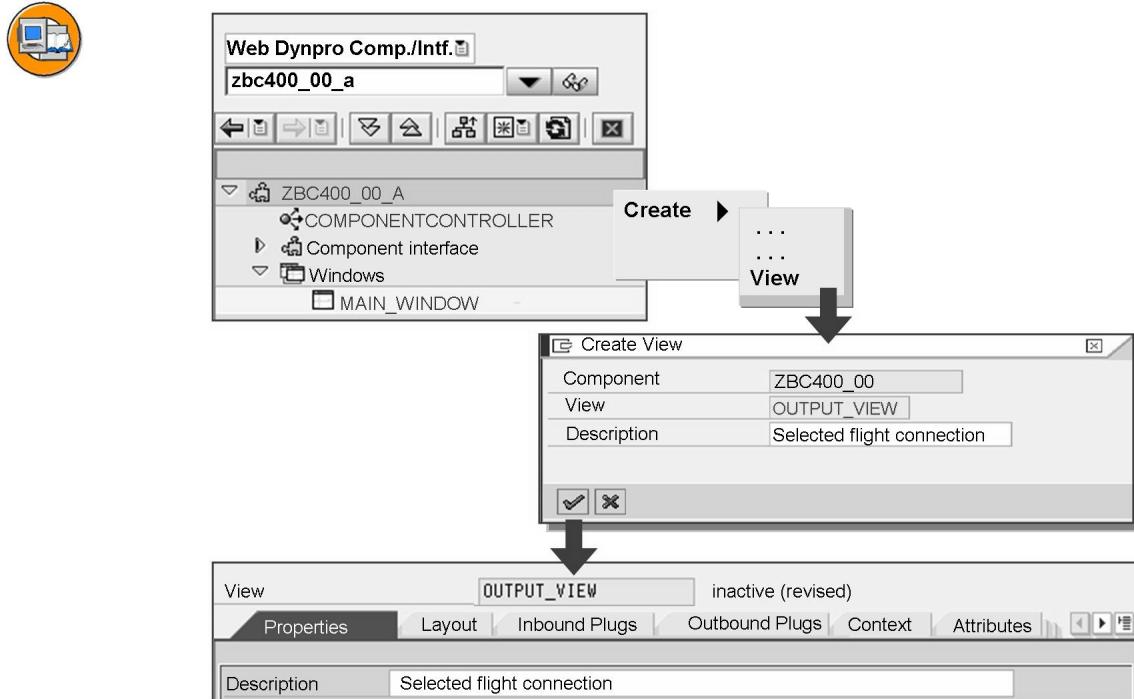


Figure 204: Creating a Web Dynpro View

To create a Web Dynpro view you also use the context menu in the Navigation Area of the *Object Navigator*. Assign a name to the view as well as a description.

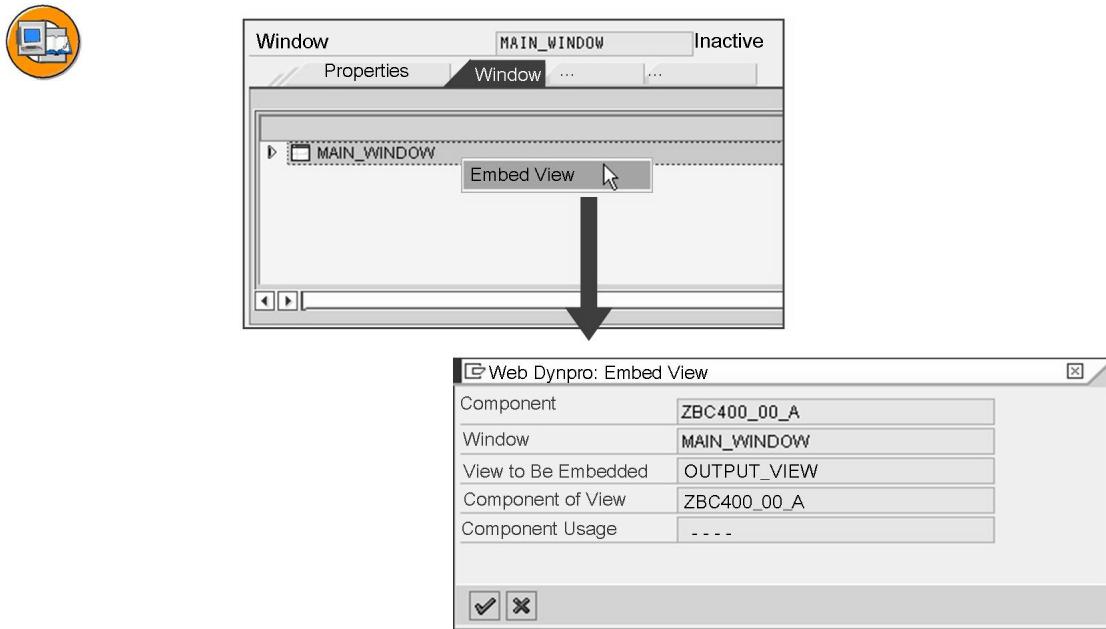


Figure 205: Embedding a Web Dynpro View

To embed a Web Dynpro view in a window, open the window in the *Object Navigator* and display the window structure. There are two options you can use to embed the views:

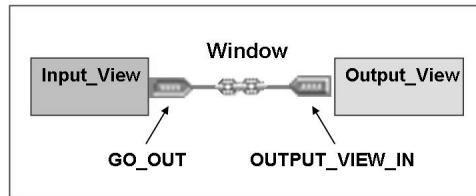
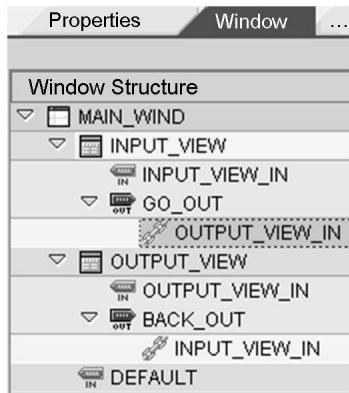
The context menu

In the window structure, the entry *Embed View* appears in the context menu for the name of the window (see above graphic). In the entry screen that follows, select the view to be embedded using the input help (*F4*).

Drag and drop from the Navigation Area

In the Navigation Area, if you open the *Views* node, you can drag the view to be embedded directly into the window structure and onto the window by drag and drop.

Setting up Navigation Between Web Dynpro Views



- Inbound Plug
- Outbound Plug
- Navigation Link

- If you want to define the navigation between two views, you must create exit points (outbound plugs) and entry points (inbound plugs) for each view.
- You can then use navigation links to determine the navigation flow.

Figure 206: Navigation Between Web Dynpro Views

To navigate within a window from one view to another, you have to create a **navigation link** between the views. A navigation link always leads from an **outbound plug** in one view to an **inbound plug** in the other.

The outbound plugs and inbound plugs of a view determine the possible “entrances” and “exits”. A view can have more than one inbound and outbound plug. The navigation link in the above graphic enables you to navigate to the OUTPUT_VIEW when you exit the INPUT_VIEW using its outbound plug *GO_OUT*. The OUTPUT_VIEW is then started by way of its inbound plug *OUTPUT_VIEW_IN*.

When you create the navigation links, this determines the view sequence within the respective window. This occurs on the basis of the existing plugs and independently of the implementation of the views.

You create the plugs very simply by processing the corresponding view and entering a name and a description for the plug on the *Inbound Plugs* or *Outbound Plugs* tab pages. You can also apply parameters to the outbound plugs for advanced applications.

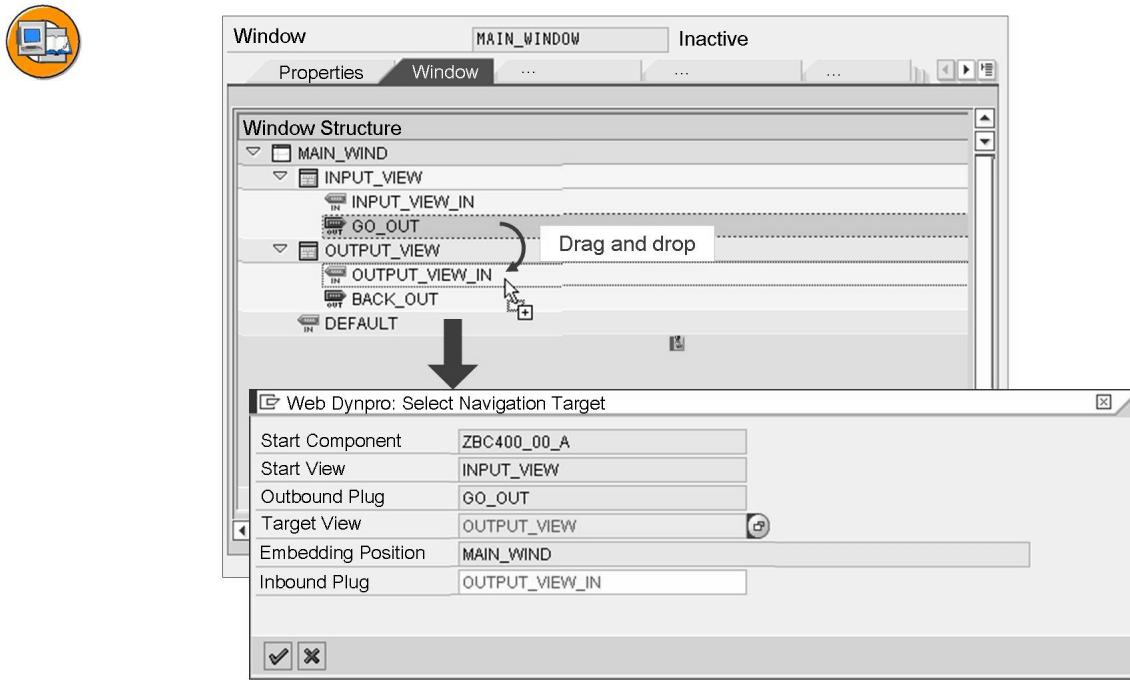


Figure 207: Creating a Navigation Link

After you have created the plugs, you can see them in the structure of a window next to the embedded views. You can create navigation links between these plugs quite simply using drag and drop (see above graphic).

Creating Pushbuttons and Triggering Navigation

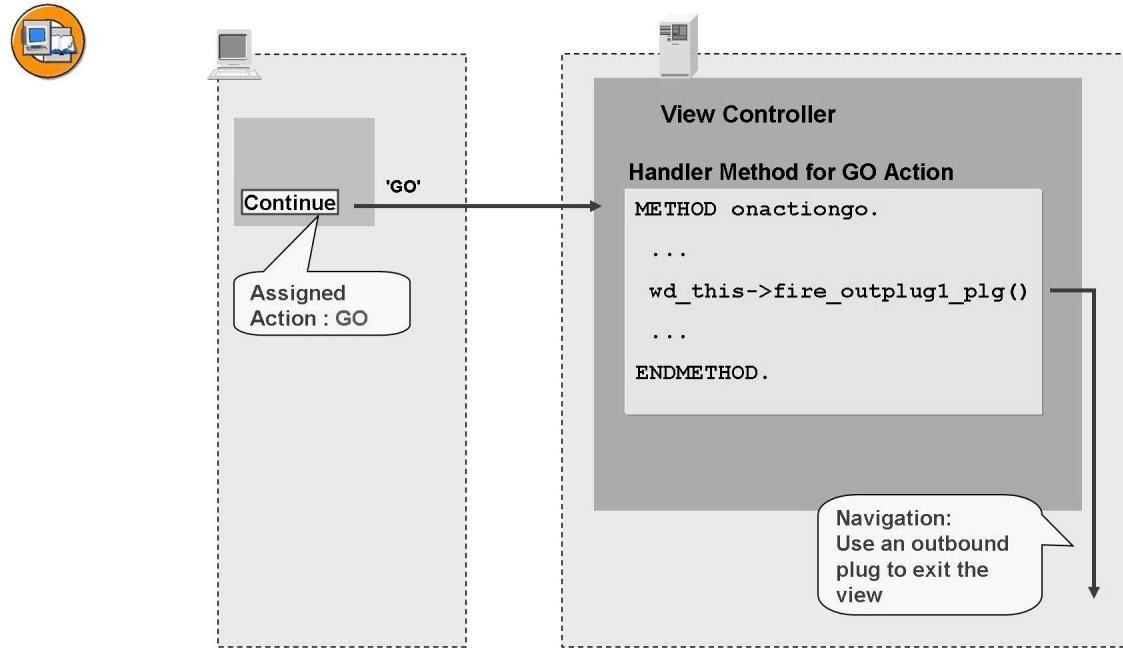


Figure 208: Process Flow when Pressing a Pushbutton

In our example program, navigation should in each instance be triggered by pressing a pushbutton. The process flow is as follows:

- When the user presses a pushbutton, an action code is transferred from the browser to the application server. This action code was defined by the developer for the respective view and assigned to the pushbutton.
- There is a special method in the view controller (action handler method) for each action code that is now executed in reaction to the action.
- In our case, only navigation to the next view should be triggered. For this to occur, the view is exited using the corresponding outbound plug. The expression used is that the outbound plug is “fired”.

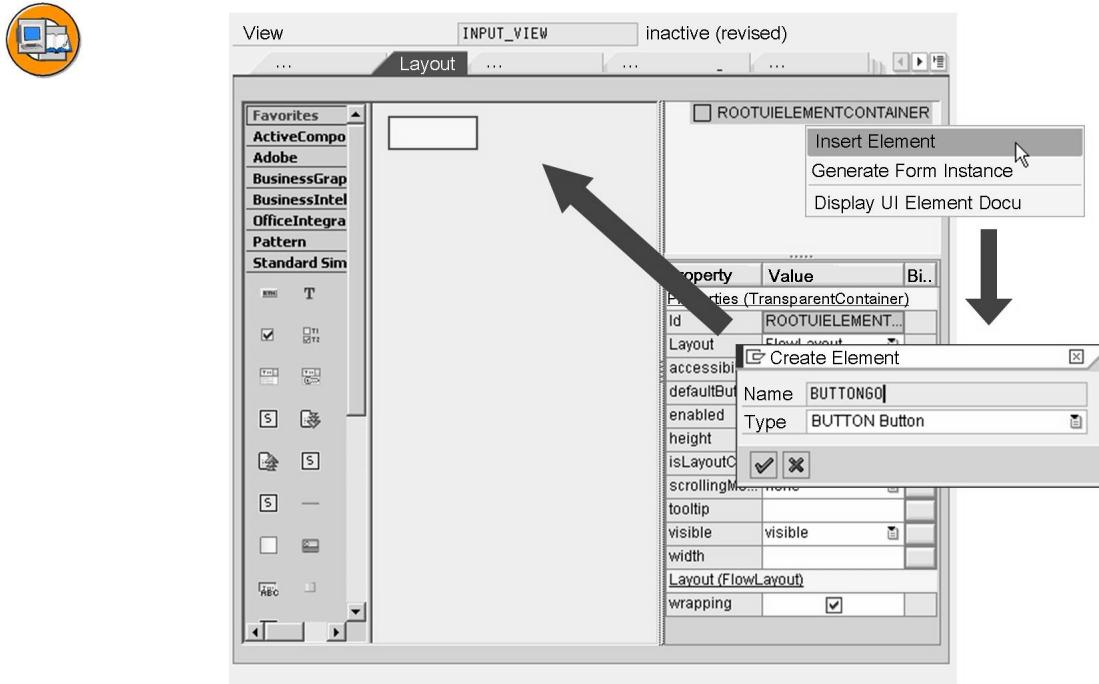


Figure 209: Creating a Pushbutton

Create a pushbutton in the layout view for the view. There are several options for this:

The context menu in the element hierarchy

In the element hierarchy, open the context menu for the *ROOTUIELEMENTCONTAINER* node at top right and choose *Insert Element*. Enter the name and type of the UI element (see above graphic).

The toolbar

From the toolbar, choose the *Button* tool on the left and position the pushbutton in the middle of the layout preview.

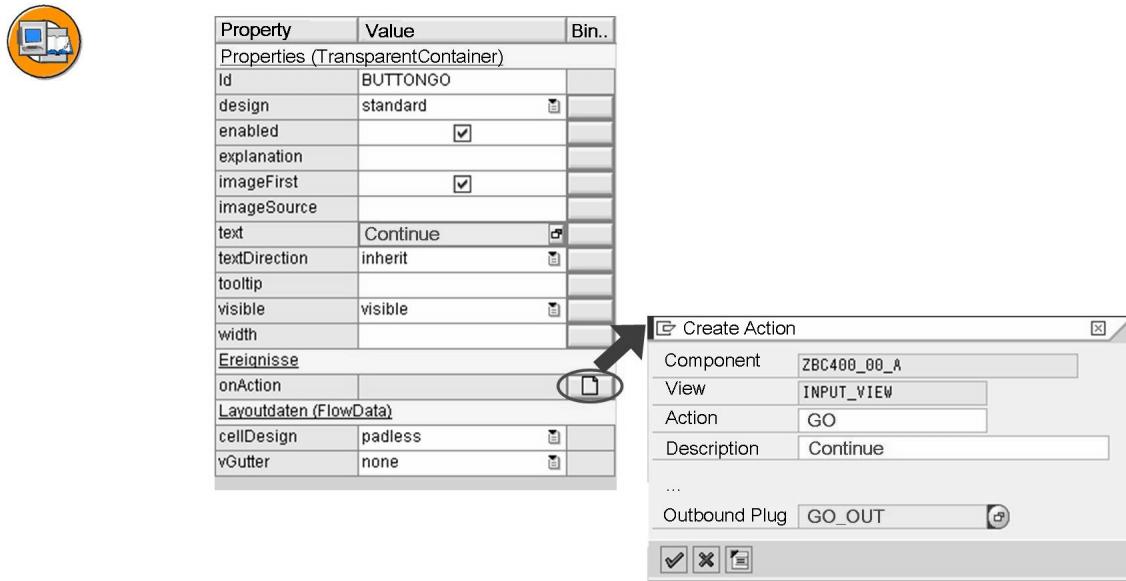


Figure 210: Maintaining Pushbutton Attributes

In the attribute window in the lower right corner the attributes are displayed for a specific element of the layout. You can enter a text for a pushbutton or assign an icon to it here too.

To assign an action to the pushbutton, go to the *onAction* attribute and click on the create icon in the third column (see above graphic). Assign an action to the pushbutton in the following dialog. If the action has not already been defined for the view (*Actions* tab page), you can create it here again directly.

In the *Outbound Plug* field, you also have to specify the outbound plug that is fired as a reaction to this action. The action handler method is then not created empty, but automatically contains the source code for firing the outbound plug.

Creating a Web Dynpro Application

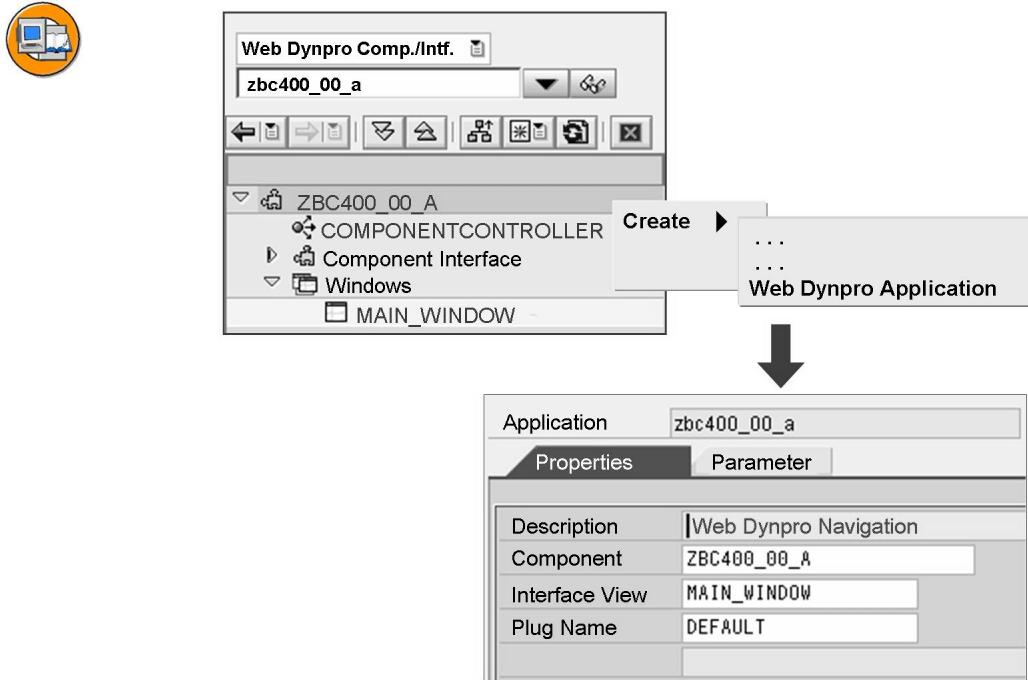


Figure 211: Creating a Web Dynpro Application

The entry point in a Web Dynpro view sequence, in other words, a Web Dynpro window, is always a **Web Dynpro application**. It points to the window or the **interface view** of the same name and the inbound plug *DEFAULT* that was automatically created for it.

You create a Web Dynpro application from the context menu of the *Object Navigator* (see above graphic). It is connected to the component, the interface view and the standard plug in the following dialog.

You can test a Web Dynpro directly from the development environment by way of a corresponding pushbutton. Alternatively, you can enter the assigned URL (Internet address) directly in the address bar of a browser.

Web Dynpro, Layout and Data Transport

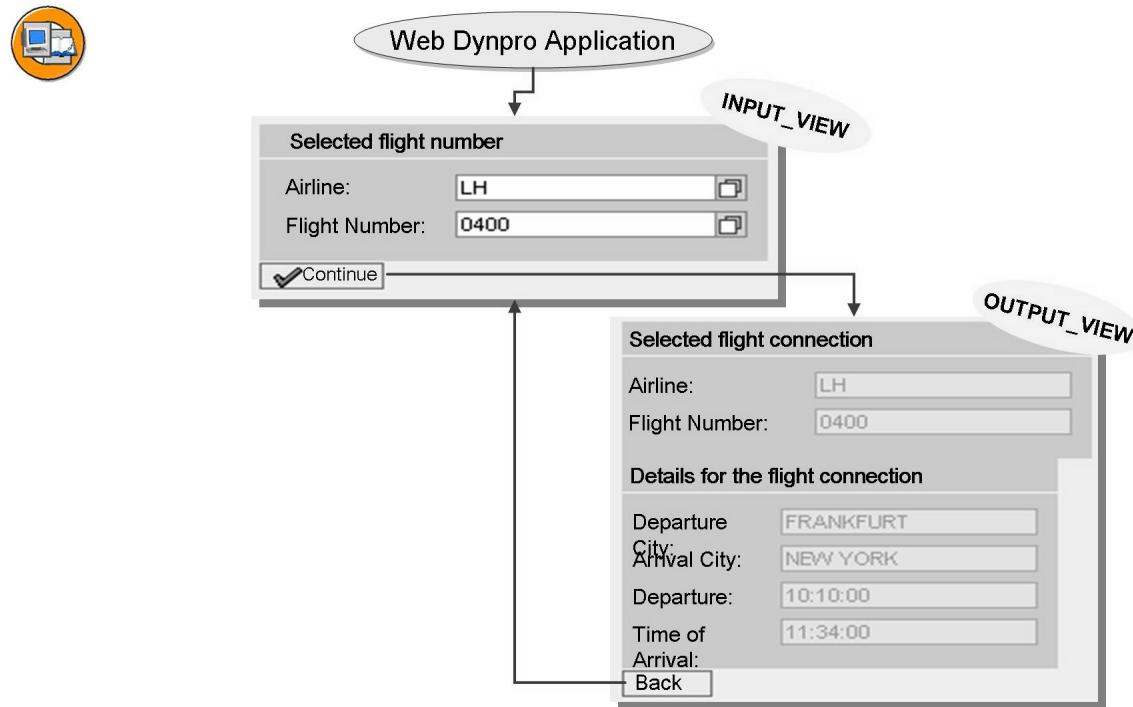


Figure 212: Realization Level 2: Creating Input/Output Fields and Displaying Data

At the second level of our example, the Web Dynpro views should be extended with the input/output fields, and data acquisition and transport should be implemented. In the first view, the user should select a flight connection using the airline ID and the flight number. Once the pushbutton has been pressed, this information should be further processed in the program and the detailed information on the airline read from the database. When the user navigates to the second view, the detailed information should be displayed there.

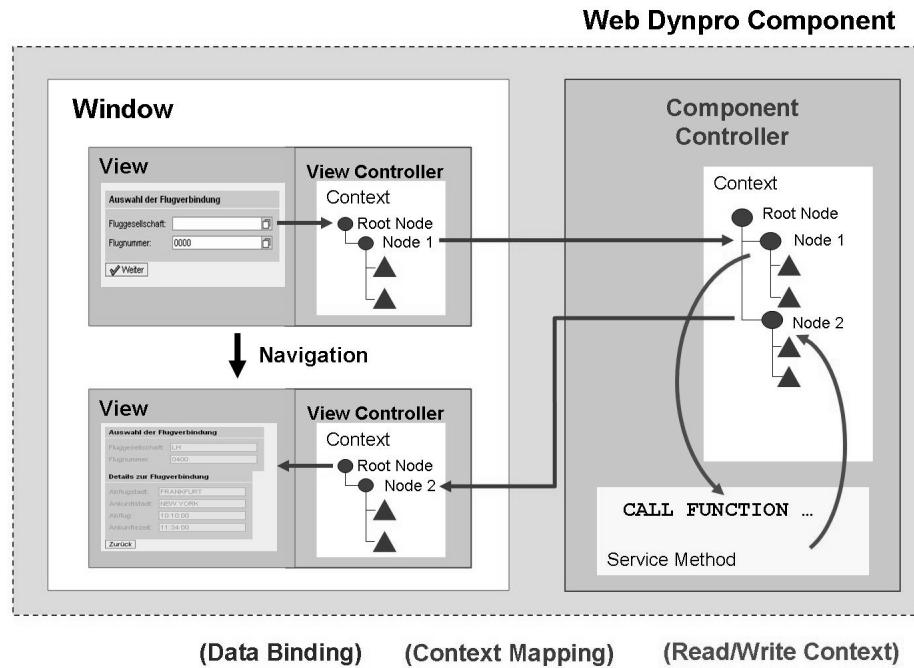


Figure 213: Data Retrieval and Data Transport

To correspond to the MVC programming model, each data retrieval of a Web Dynpro component should be encapsulated centrally in a component controller method. In our case, this **service call** should contain a function module call. The data required by the function module is taken from the context of the component controller. Accordingly, after the function module is called, the result is stored again in the context.

Data is transported between the central component controller and the individual view controllers using **context mapping**. This references from one context to another, so the data transport functions automatically in both directions.

Data is transported between the input/output fields of a view and the context for this view by **data binding**.

This section deals with the following:

- Creating a Web Dynpro service call
- Creating the controller context
- Setting up context mapping
- Creating input/output fields
- Setting up data binding
- Calling the service method for data retrieval

Creating a Web Dynpro Service Call and Generating the Controller Context

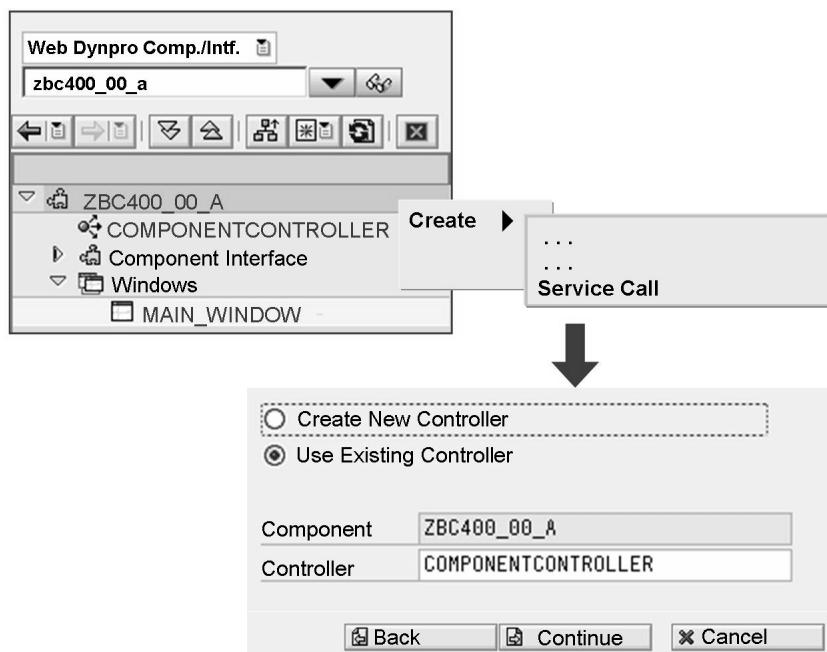


Figure 214: Creating a Service Call

The *ABAP Workbench* provides a *Web Dynpro Wizard* for creating service calls, which you can address in the Navigation Area of the *Object Navigator* using the context menu in the usual way. After you have started the wizard, you first have to select the controller in which you want to create the service call. In our case, this is the existing component controller. You have the option of creating an additional controller here too, though.

After this, the wizard still needs to know which function module is to be called, the name of the service method, and which function module parameters it needs to supply with actual parameters when the function module is called.

The wizard generates the following components in the selected controller:



- A service method (name begins with “EXECUTE_”)
- Context nodes for all function module parameters that have to be supplied during the call
- The function module call in the service method
- The source code for reading the data from the context
- The source code for filling the context with the result of the function module call

Setting up Context Mapping

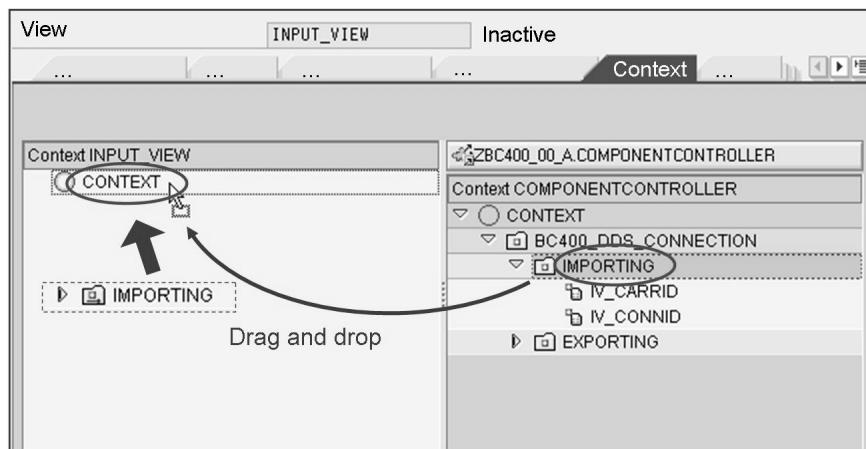


Figure 215: Setting up Context Mapping

To make the data from the component controller context available in the views too, you have to copy the corresponding nodes in the view controller context and map them to the component controller nodes. Context mapping is also necessary to make user entries available on a view in the component controller.

You can copy the nodes at the same time as you set up context mapping. To do this, open the corresponding view in change mode and go to the *Context* tab page. On the left, you will see the view controller context (still empty, apart from a root node named *CONTEXT*) and on the right the generated component controller context. You copy a node by dragging it from the right and dropping it on the root node for the view controller context on the left. In the following dialog, you confirm that the node should be both copied and mapped.

Creating Input/Output Fields and Setting up Data Binding

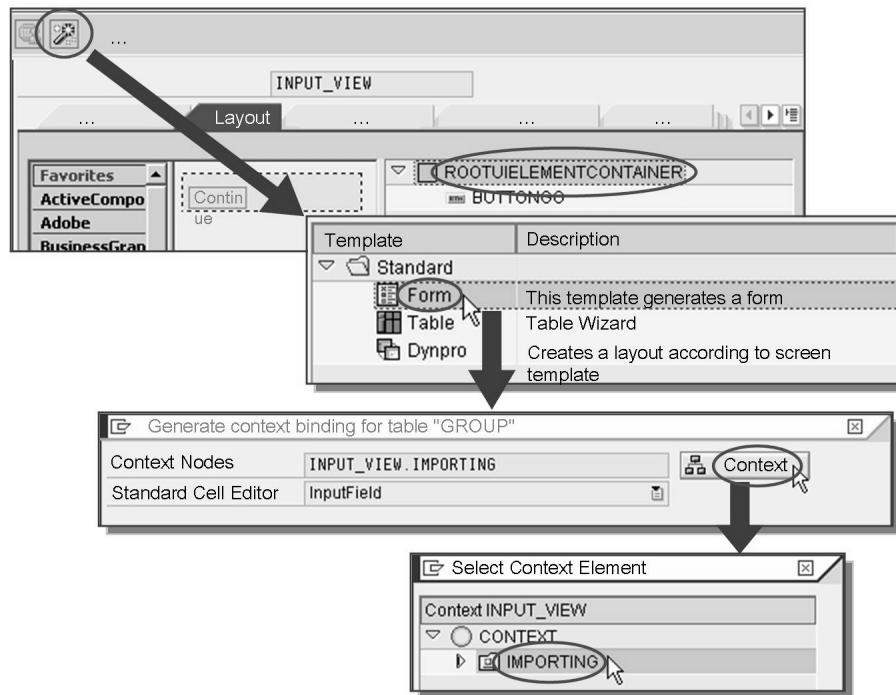


Figure 216: Creating a Layout Form

The *ABAP Workbench* also provides a wizard for creating input/output fields and setting up data binding. The prerequisite for this is that the node to be bound already exists in the view controller context. If this is the case, it is then very easy to create a layout form for this node.

Open the view in change mode and go to the *Layout* tab page. Select the entry *ROOTUIELEMENTCONTAINER* in the element hierarchy (top right). Start the *Web Dynpro Wizard* using the corresponding pushbutton (see above graphic) and select the *Form* template. Clicking the *Context* pushbutton takes you to a dialog in which you can select the node for which you want to create the form directly. If the node has several fields, you can then select or exclude these fields individually.

Calling the Service Method for Data Retrieval

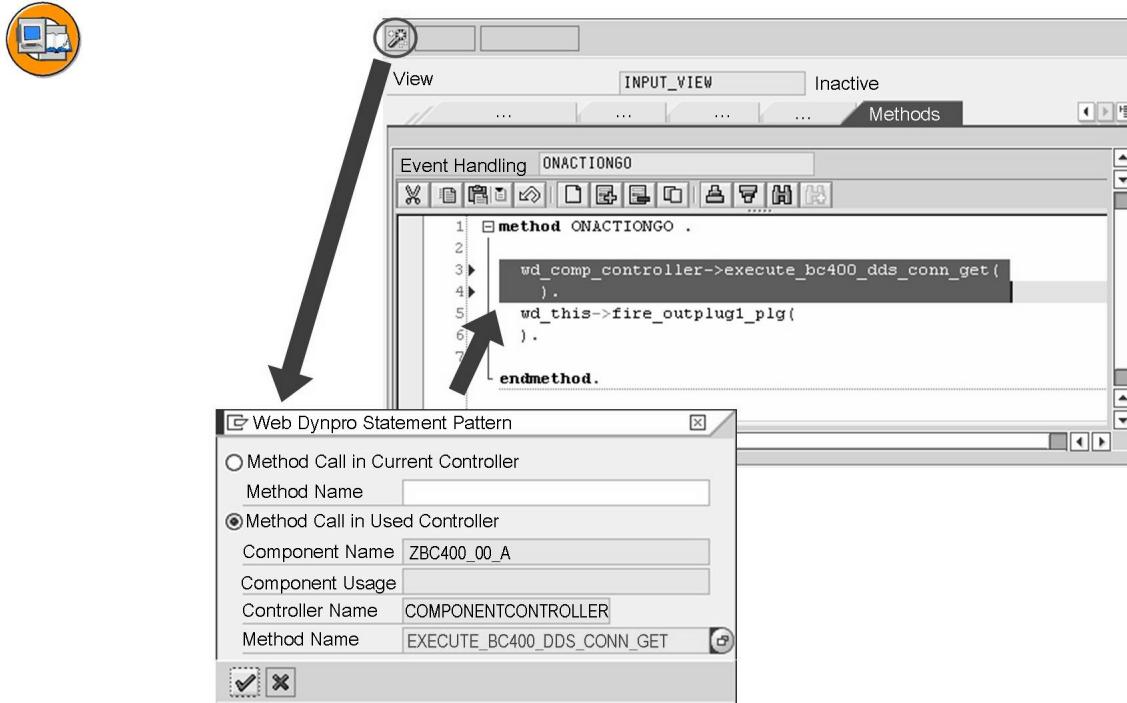


Figure 217: Implementing a Method Call

To enable the flight connection details to be displayed on the second view, the service method has to be called before you navigate to the view. In our example, this call is made in the action handler method (*ONACTION...* method) of the input view.

You can also generate method calls with the *Web Dynpro Wizard*. To do this, open the source code for the action handler method, position the cursor in front of the source code to fire the outbound plug and click on the pushbutton to launch the wizard (see above graphic).

On the following screen, choose *Method Call in Used Controller* and use the input help to enter the component controller as the used controller. Then use the input help again to choose the method you want to call.

To conclude the activity, the wizard generates the source code to call the method.

Exercise 23: Web Dynpro:Navigation

Exercise Objectives

After completing this exercise, you will be able to:

- Create a Web Dynpro component with a Web Dynpro window that contains two Web Dynpro views
- Create pushbuttons on Web Dynpro views
- Trigger actions through the pressing of pushbuttons by the user
- Implement navigation between Web Dynpro views
- Create a Web Dynpro application

Business Example

You want to develop a Web Dynpro application with two views. You want to incorporate pushbuttons on the views to allow the user to initiate navigation to the next view.

Template:

None

Solution:

BC400_WDS_A

Task 1:

Create a Web Dynpro component with a Web Dynpro window.

1. Create Web Dynpro component **ZBC400_##_A** with a **MAIN_WINDOW** window and a first view **INPUT_VIEW**.

Task 2:

Create a second Web Dynpro view in the Web Dynpro component and embed this in the window.

1. Create a second Web Dynpro view (suggested name: **OUTPUT_VIEW**) in your component.
2. Embed the Web Dynpro view in the Web Dynpro window as well.

Continued on next page

Task 3:

Create an outbound plug and an inbound plug for each view. Define two navigation links between the two views – each going from the outbound plug of one view to the inbound plug of the other view.

1. For your first view (**INPUT_VIEW**), create an outbound plug (suggested name: **GO_OUT**) and an inbound plug (suggested name: **INPUT_VIEW_IN**).
2. Do the same for your other view (suggested names for the plugs: **BACK_OUT** and **OUTPUT_VIEW_IN**).
3. Define a navigation link from the outbound plug of the first view to the inbound plug of the second view.
4. Define a navigation link from the outbound plug of the second view to the inbound plug of the first view.

Task 4:

Create a pushbutton on each of the two views and make sure the outbound plug of the respective view is fired when the pushbutton is pressed by the user.

1. Create a simple *BUTTON* type UI element (suggested name: **BUTTON_GO**) on the first view and maintain the text to be displayed on the pushbutton.
2. Create an action for the pushbutton (suggested name: **GO**) and reference it to the outbound plug of the view (**GO_OUT**).
3. Create a pushbutton on the second view (suggested name: **BUTTON_BACK**) with text and action (suggested name: **BACK**), and link to the outbound plug of this view.

Task 5:

Create a Web Dynpro application and test your program.

1. Activate the Web Dynpro component
2. Create a Web Dynpro application (suggested name: **ZBC400_##_A**) that references the main window of your Web Dynpro component.
3. Test the Web Dynpro application.

Solution 23: Web Dynpro:Navigation

Task 1:

Create a Web Dynpro component with a Web Dynpro window.

1. Create Web Dynpro component **ZBC400_##_A** with a **MAIN_WINDOW** window and a first view **INPUT_VIEW**.
 - a) In the navigation area, open the context menu for the package and choose *Create → WebDynpro → WebDynpro Component (Interface)*.
 - b) In the dialog box, enter the name of the component, a description, the name of the window and the name of the view.

Task 2:

Create a second Web Dynpro view in the Web Dynpro component and embed this in the window.

1. Create a second Web Dynpro view (suggested name: **OUTPUT_VIEW**) in your component.
 - a) In the context menu for the Web Dynpro component choose *Create → View*.
 - b) Enter the name of the view and a short description.
2. Embed the Web Dynpro view in the Web Dynpro window as well.
 - a) Edit the Web Dynpro window and open the *Window* tab.
 - b) In the window structure, open the context menu for the Web Dynpro window and choose *Embed View*.
 - c) In the dialog box, enter the name of the view to be embedded.

Continued on next page

Task 3:

Create an outbound plug and an inbound plug for each view. Define two navigation links between the two views – each going from the outbound plug of one view to the inbound plug of the other view.

1. For your first view (**INPUT_VIEW**), create an outbound plug (suggested name: **GO_OUT**) and an inbound plug (suggested name: **INPUT_VIEW_IN**).
 - a) Edit Web Dynpro view INPUT_VIEW.
 - b) Select the *Outbound Plugs* tab and enter the name of the outbound plug and a description in the uppermost table.
 - c) Select the *Inbound Plugs* tab and enter the name of the inbound plug and a description.
2. Do the same for your other view (suggested names for the plugs: **BACK_OUT** and **OUTPUT_VIEW_IN**).
 - a) Perform this step as before.
3. Define a navigation link from the outbound plug of the first view to the inbound plug of the second view.
 - a) Edit the Web Dynpro window.
 - b) Choose the *Window* tab and expand all the nodes of the window structure.
 - c) Drag the outbound plug of the first view and drop it on the inbound plug of the second view. Confirm the data in the dialog box.
4. Define a navigation link from the outbound plug of the second view to the inbound plug of the first view.
 - a) Perform this step as before.

Continued on next page

Task 4:

Create a pushbutton on each of the two views and make sure the outbound plug of the respective view is fired when the pushbutton is pressed by the user.

1. Create a simple **BUTTON** type UI element (suggested name: **BUTTON_GO**) on the first view and maintain the text to be displayed on the pushbutton.
 - a) Edit the first view and choose the *Layout* tab.
 - b) Open context menu for ROOTUIELEMENTCONTAINER and choose *Insert Element*.
 - c) Enter the name of the button and choose element type **BUTTON**.
 - d) Go to *Text* property and maintain the text to be displayed in the *Value* column.
2. Create an action for the pushbutton (suggested name: **GO**) and reference it to the outbound plug of the view (GO_OUT).
 - a) Go to the *OnAction* property for the pushbutton. In the *Binding* column, choose *Create*.
 - b) In the dialog box, enter the name of the action, a description, and the name of the outbound plug.
3. Create a pushbutton on the second view (suggested name: **BUTTON_BACK**) with text and action (suggested name: **BACK**), and link to the outbound plug of this view.
 - a) Perform this step as before.

Task 5:

Create a Web Dynpro application and test your program.

1. Activate the Web Dynpro component
 - a) Open the Web Dynpro **Component** and choose *Activate*.
2. Create a Web Dynpro application (suggested name: **ZBC400_##_A**) that references the main window of your Web Dynpro component.
 - a) In the navigation area, open the context menu for your Web Dynpro component and choose *Create → Web Dynpro Application*.
 - b) In the dialog box, check the name of the application (same name as the component) and enter a description.

Continued on next page

3. Test the Web Dynpro application.
 - a) Open the Web Dynpro **application** and press the  *Test/Execute* pushbutton.

Exercise 24: Web Dynpro: Data Transport and Layout

Exercise Objectives

After completing this exercise, you will be able to:

- Create a service call to call a function module and bind the parameters to the controller context
- Create UI elements on a view to enable data to be entered and displayed

Business Example

You want to develop a Web Dynpro application in which the entries made on the first view are used to select data for data retrieval. The data that is found should then be displayed after you navigate to the second view.

Template:

BC400_WDS_A

Solution:

BC400_WDS_B

Task 1:

Copy your Web Dynpro component ZBC400_##_A or the template BC400_WDS_A to Web Dynpro component **ZBC400_##_B**.

1. Copy the template.

Task 2:

In the component controller, create a service call for a function module that reads data for an individual flight connection. Use the function module BC400_DDS_CONNECTION_GET or the function module you developed yourself. By generating the service call, you create a method that encapsulates the function module call. In addition, you can trigger the system to create nodes for the interface parameters of the function module in the component controller context.

1. Create a service call for the function module in the existing component controller.

For all parameters of the function module, select the object type *Context (node/attribute)*.

Continued on next page

Accept the default name for the service method (EXECUTE_...) or shorten it, if necessary, in a recognizable way.

Task 3:

Copy the IMPORTING node in the component controller context to the context of the input view and define the context mapping. In the layout of the input view, create a group with input fields for the airline ID and flight number and link the input fields to the context of the view.

1. Copy the IMPORTING node from the component controller context to the context of the input view and define the mapping between the context nodes of the various controllers.
2. Create a form on the layout of the input view that comprises two input fields that correspond to both attributes of the IMPORTING context node. Use the *Web Dynpro Code Wizard* to create the form. The wizard also creates a *TransparentContainer* type UI element that includes the form fields.

Task 4:

Copy the ES_CONNECTION node in the component context to the context of the display view and define the context mapping. Use the *Web Dynpro Code Wizard* to create a form that displays some of the attributes of the context node.

1. Copy the ES_CONNECTION node from the component controller context to the context of the output view and define the mapping between the context nodes of the various controllers.
2. Create a form on the output view that displays the content of some of the attributes of the context node (suggestion: CARRID, CONNID, CITYFROM, CITYTO, ARRTIME and DEPTIME).

Task 5:

Use the *Web Dynpro Code Wizard* to implement a call of the service method EXECUTE_... before the navigation from the input view to the display view. Insert the call into the action handler method of the input view before the navigation plug is fired.

1. In the input view, edit the method for handling the action, (method with name ONACTION...). Start the *Web Dynpro Code Wizard* to implement a call of EXECUTE_... method of the component controller (method call in used controller).
2. Activate your component. Create a Web Dynpro application and test it.

Solution 24: Web Dynpro: Data Transport and Layout

Task 1:

Copy your Web Dynpro component ZBC400_##_A or the template BC400_WDS_A to Web Dynpro component **ZBC400_##_B**.

1. Copy the template.
 - a) Open the Web Dynpro component and, in the object list, open the context menu on the name of the component.
 - b) Choose *Copy...* and enter the name of the new component.

Task 2:

In the component controller, create a service call for a function module that reads data for an individual flight connection. Use the function module BC400_DDS_CONNECTION_GET or the function module you developed yourself. By generating the service call, you create a method that encapsulates the function module call. In addition, you can trigger the system to create nodes for the interface parameters of the function module in the component controller context.

1. Create a service call for the function module in the existing component controller.

For all parameters of the function module, select the object type *Context (node/attribute)*.

Continued on next page

Accept the default name for the service method (EXECUTE_...) or shorten it, if necessary, in a recognizable way.

- a) Open the context menu for the Web Dynpro component in the navigation area of the ABAP Workbench (left-hand side) and choose *Create* → *Service Call*.
- b) Choose *Continue* to confirm the first window of the wizard.
- c) Select *Use Existing Controller*, enter the component controller of your component, and choose *Continue*.
- d) Select *Function Module* and choose *Continue*.
- e) On the next screen, enter the name of the function module in the *Function Module* field and choose *Continue*.
- f) For all parameters that are to be stored in the controller context, select the object type *Context (node/attribute)* and choose *Continue*.
- g) Enter the name of the service method and a description and choose *Continue*.
- h) Exit the Web Dynpro wizard by choosing *Complete*.

Task 3:

Copy the IMPORTING node in the component controller context to the context of the input view and define the context mapping. In the layout of the input view, create a group with input fields for the airline ID and flight number and link the input fields to the context of the view.

1. Copy the IMPORTING node from the component controller context to the context of the input view and define the mapping between the context nodes of the various controllers.
 - a) Edit the Web Dynpro view and choose the *Context* tab page.
 - b) Drag the IMPORTING context node from the component controller context (right-hand side) and drop it on the top node of the view controller context (left-hand side).

Continued on next page

2. Create a form on the layout of the input view that comprises two input fields that correspond to both attributes of the IMPORTING context node. Use the *Web Dynpro Code Wizard* to create the form. The wizard also creates a *TransparentContainer* type UI element that includes the form fields.
 - a) Edit the Web Dynpro view and choose the *Layout* tab page.
 - b) Start the *Web Dynpro Code Wizard* by clicking the pushbutton in the application toolbar of the Web Dynpro Explorer. Select the template for creating forms.
 - c) Choose  *Context* and double-click the context node (IMPORTING) in the dialog box.
 - d) Confirm the entries by choosing .



Hint: You can rearrange the UI elements as follows:

- If you move the pushbutton by drag and drop to the generated TRANSPARENT_CONTAINER UI element, this will make it the last element in the group.
- If you set the *Layout Data* pushbutton property to *MatrixHeadData*, this will move the pushbutton to a new row in the group.
- If you maintain the *Text* property of the TEXT_VIEW sub-element, this adds a header to the group.

Task 4:

Copy the ES_CONNECTION node in the component context to the context of the display view and define the context mapping. Use the *Web Dynpro Code Wizard* to create a form that displays some of the attributes of the context node.

1. Copy the ES_CONNECTION node from the component controller context to the context of the output view and define the mapping between the context nodes of the various controllers.
 - a) Carry out this step as described above.
2. Create a form on the output view that displays the content of some of the attributes of the context node (suggestion: CARRID, CONNID, CITYFROM, CITYTO, ARRTIME and DEPTIME).
 - a) Create the form as before.

Continued on next page

Task 5:

Use the *Web Dynpro Code Wizard* to implement a call of the service method EXECUTE_... before the navigation from the input view to the display view. Insert the call into the action handler method of the input view before the navigation plug is fired.

1. In the input view, edit the method for handling the action, (method with name ONACTION...). Start the *Web Dynpro Code Wizard* to implement a call of EXECUTE_... method of the component controller (method call in used controller).
 - a) Edit the Web Dynpro view and choose the *Methods* tab page.
 - b) Open the handler method in the editor. In the application toolbar, choose  for the *Web Dynpro Code Wizard* and in the sample statement overview that follows, choose *Method Call in Used Controller*.
 - c) Select the component controller and the method to be called from the input help and choose .
2. Activate your component. Create a Web Dynpro application and test it.
 - a) Carry out this step as you did in the previous exercise.



Lesson Summary

You should now be able to:

- List the properties and usage scenarios of the ABAP Web Dynpro
- Explain the programming and runtime architecture of the ABAP Web Dynpro
- Implement simple Web Dynpro applications with input/output fields and pushbuttons

Lesson: Classic ABAP Reports

Lesson Overview

This lesson will discuss classic ABAP reports (executable programs). This type of ABAP program is closely linked with the classic **event control** technique as well as the special **selection screen** and **ABAP list** user dialogs. Although this type of programming is considered obsolete today and should be avoided, where possible, in new development, you will of course still find programs in the SAP standard system and in customer systems that are based on it. The technique will therefore be explained in brief here, without going into detail. For further information, refer to the documentation or the relevant training courses. At the end of the lesson you will see how to use the **ABAP List Viewer (ALV)** to replace classic ABAP lists simply and elegantly with a more modern and functional display.



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 on the selection screen
- Describe the attributes and benefits of ABAP lists
- Implement list and column headers
- Implement multilingual lists
- Describe the event-controlled processing of an executable ABAP program
- List the most important basic events and explain their purpose

Business Example

You want to develop a program that uses selections on a selection screen to read and list data from a database.

ABAP List

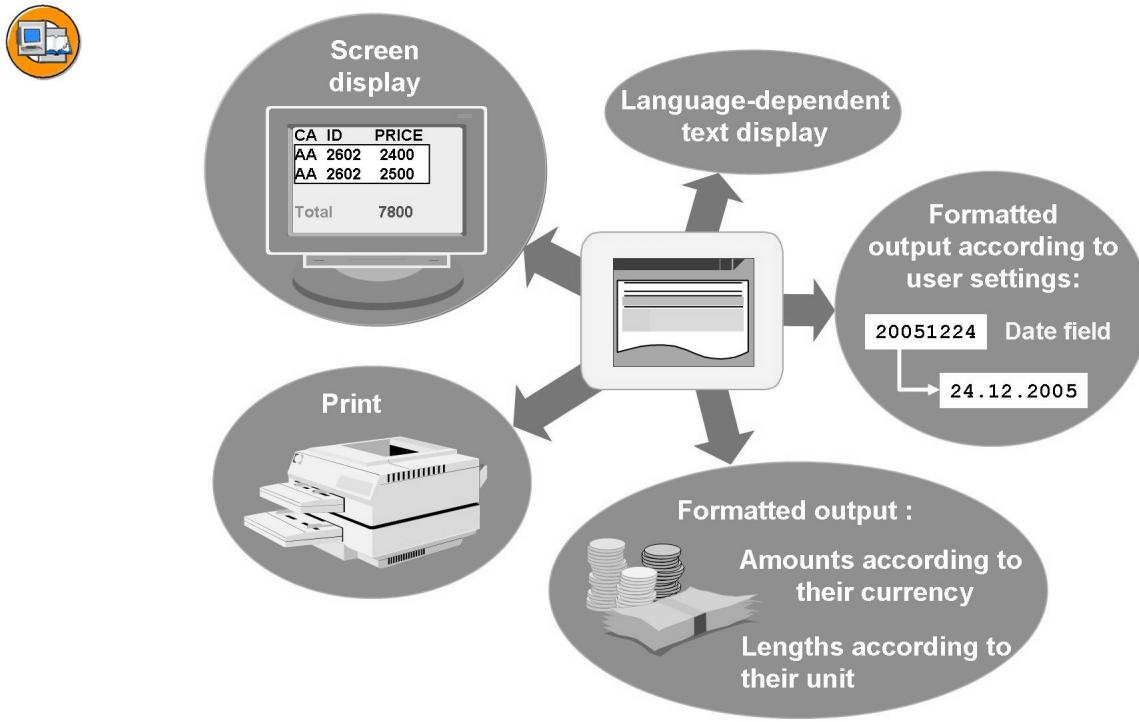


Figure 218: Features of Lists

The purpose of a list is to output data with the least possible programming effort. Lists also take the special requirements of business data into account:

- Lists can be designed for a number of languages: Texts and headers appear in the logon language whenever a corresponding translation is available.
- Lists can display monetary values in the appropriate currency.

The following options are available when outputting a list:

- Screen: You can add colors and icons here.
- Printer
- Internet or intranet: The system automatically converts the list to HTML format for this.
- Save: You can save lists within the SAP system as well as outside (for further processing, for example, using spreadsheet programs).

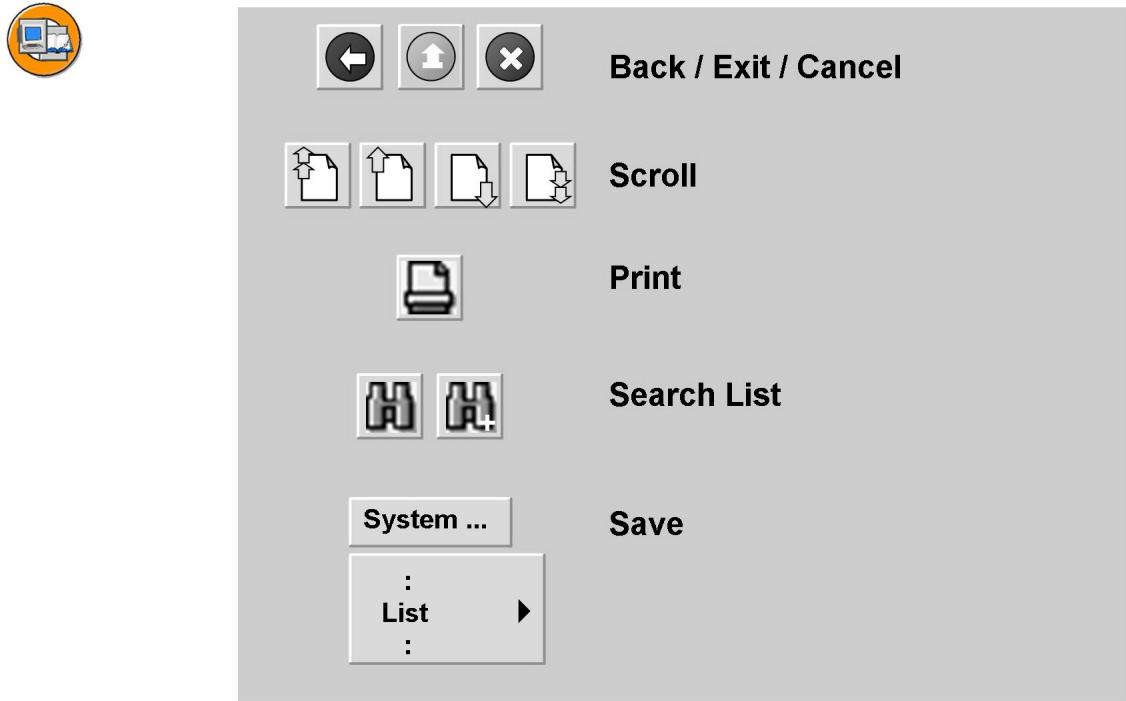


Figure 219: Standard List Functions

The standard user interface for a list offers a range of functions:

You can use the *Menu Painter* to adapt the default list interface to your own needs. More information is available in the documentation for this tool or in the corresponding training course.

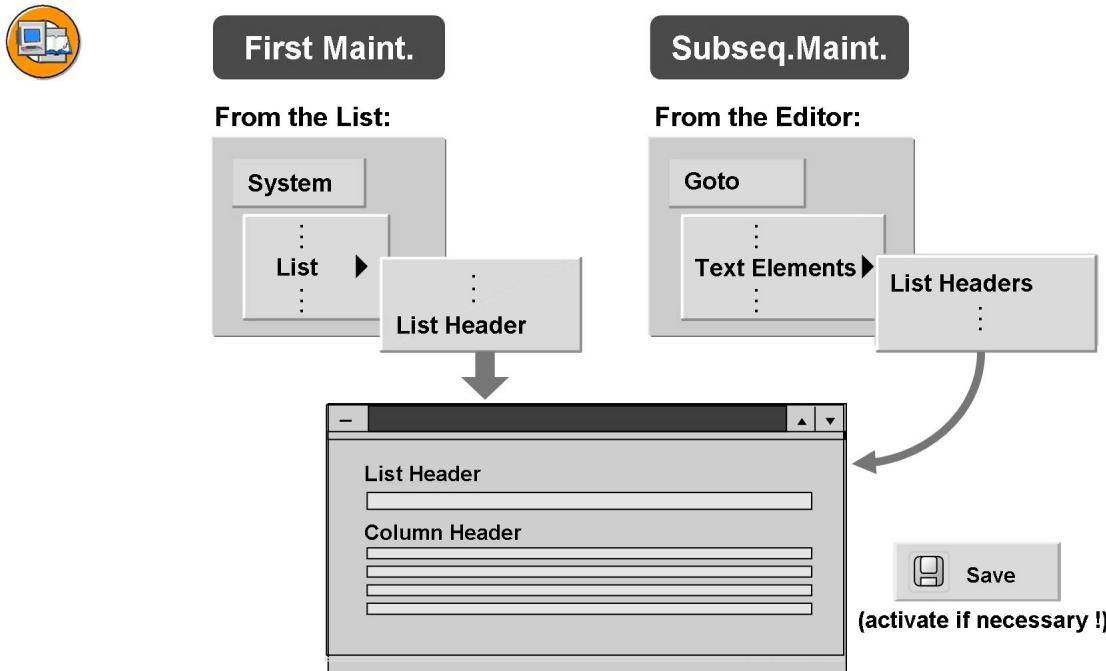


Figure 220: List and Column Headers

In an earlier lesson, you learned how you can implement translatable **text symbols** to make the interface of your program dependent on the logon language of the user. Text symbols are available in all types of program, for example, for module pools, function groups and global classes too.

For executable programs (reports) in particular, you have the option of maintaining a single line **list header** and up to four rows of **column headers** in addition to the text symbols for the classic ABAP list.

Together with text symbols and selection texts, list and column headers belong to the **text elements** of a program.

For **initial maintenance** of the headers, you first have to activate your program. Then create the list by executing the program. You can then maintain the headers directly above your list using the menu *System → List → List Header*. The next time you start the program, they will appear in the list automatically.

If you want to change the maintained header (**follow-up maintenance**), you do **not** have to start the program again and generate the list. Instead, starting from the editor into which the program is loaded, you can access the maintenance environment for changing the headers by choosing *Goto → Text Elements → List Headers*.

To translate list and column headers from within the *ABAP Editor*, choose *Goto → Translation*.

Selection Screen

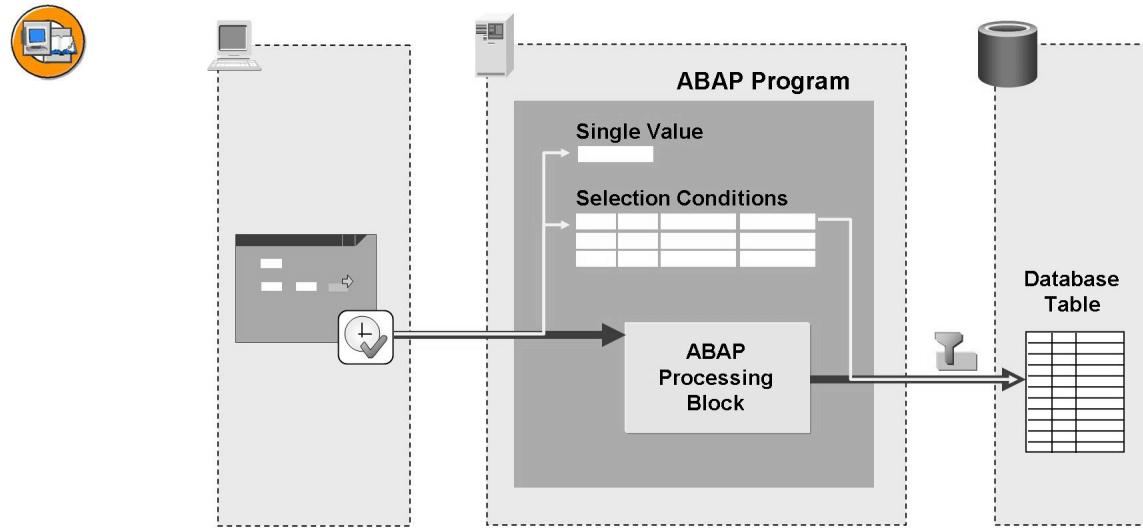


Figure 221: Architecture and Purpose of Selection Screens

In general, selection screens are used for entering selection criteria for data selection. For example, if the program creates a list of data from a very large database table, it often makes sense for users to select the data records they actually require and for the system to read only this data from the database. Apart from reducing the memory requirement, this also reduces the network load.

From a technical perspective, selection screens are dynpros. However, they are not designed by the developer directly with the *Screen Painter*, but generated in accordance with declarative statements in the source code.

Advantages of Selection Screens

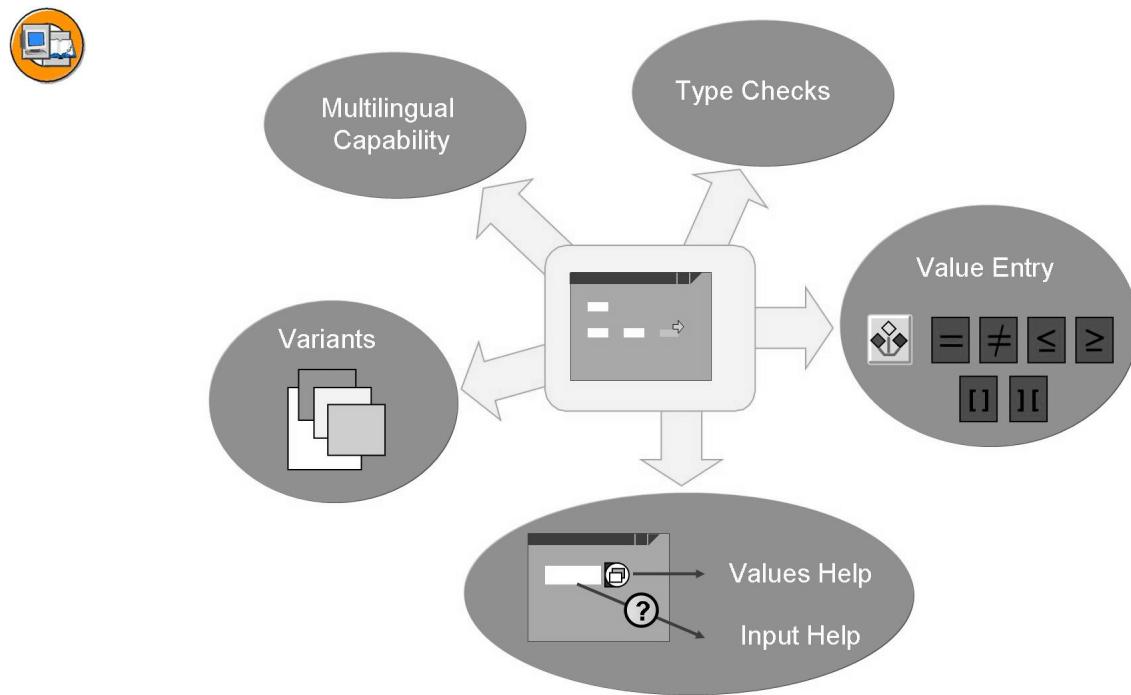


Figure 222: Selection Screen Attributes

The selection screen has the following standard functions:

- Texts on the selection screen (**selection texts**) can be maintained **in several languages**. At runtime the texts are automatically displayed in the user's logon language
- The system **checks types** automatically: If the user enters something that does not correspond to the type of the input field, the SAP GUI will ignore it, so that it does not even appear on the selection screen.
- In addition to single value entries (PARAMETERS), you can also implement **complex selections** (SELECT-OPTIONS) on the selection screen. The user can then enter intervals, comparative conditions or even patterns as restrictions.
- If the input field is defined using a Dictionary element (for example, data element), the **field documentation** (documentation of the data element) can be displayed on the input field using the **F1 (input help)** function key.

The search help attached to the Dictionary type **displaying possible inputs** can be called up using the **F4 (input help)** function key.

- You can easily save completed selection screens as **variants** for reuse or use in background operation.

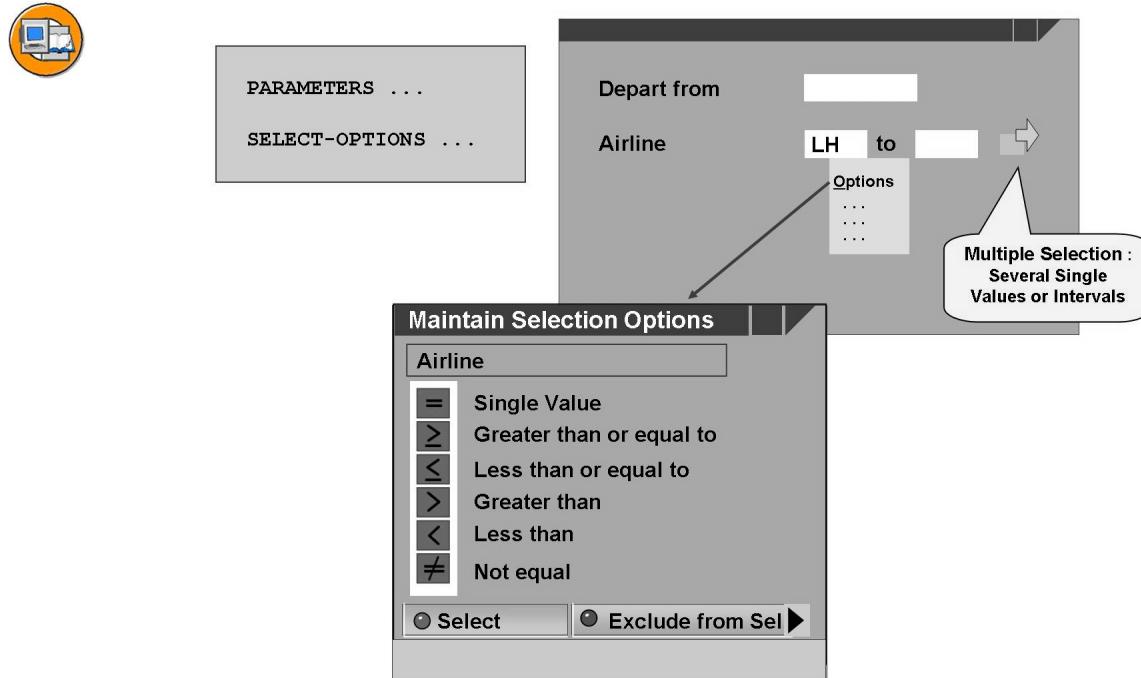


Figure 223: Using Selection Options

The above graphic illustrates the use of **selection options**, which enable complex entries. For more information on the operation in question, use the *Help on Screen* pushbutton on the additional screen for multiple selection.

You will learn more about defining selection options using the `SELECT-OPTIONS` statement later in this lesson.

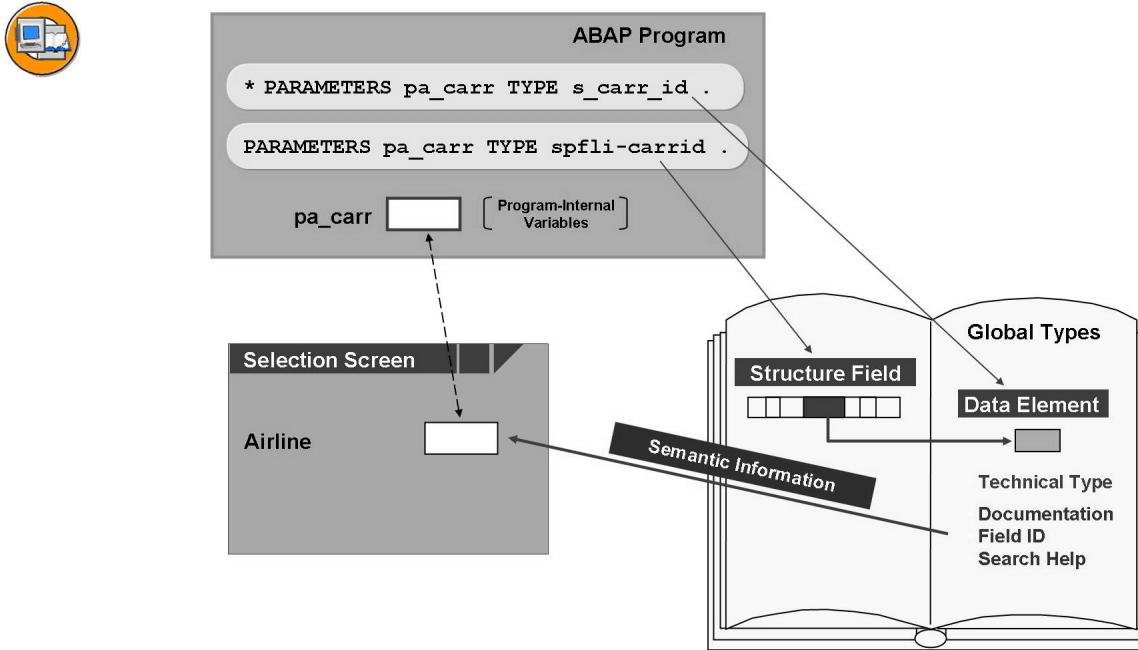


Figure 224: Semantic Information on Global Types on the Selection Screen

If an input field is typed with a **data element**, the following additional semantic information is available on the selection screen:

- The long **field label** of the data element can be adopted to describe the input field on the selection screen (**selection text**) (see next graphic).
- The **documentation** of the data element is automatically available as an **input help (F1 help)**.
- If a **search help** is linked to the data element, it is available as **input help (F4 help)**.

If the input field is typed with a **structure field** that, in turn, is defined using a data element, the above semantic information of the data element is available on the selection screen. If a search help is also attached to the structure field, this “overwrites” the search help of the data element.

For more information, refer to the online documentation for the *ABAP Dictionary*.

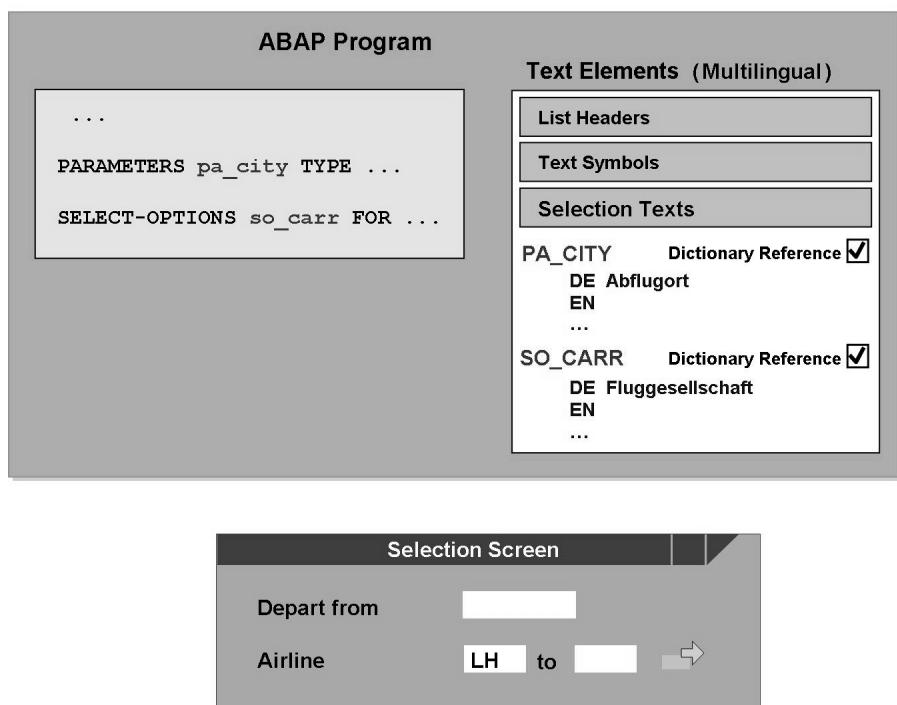


Figure 225: Selection Texts

On the selection screen, the names of the input fields appear as their description by default. However, you can replace these with corresponding **selection texts**, which you can then translate into any further languages you require. At runtime, the selection texts are then displayed in the logon language of the user (automatic language).

Just like the list headers and text symbols, selection texts belong to the text elements of the program. From the *ABAP Editor*, choose *Goto* → *Text Elements* → *Selection Texts* to maintain them. You can implement your translation using the menu *Goto* → *Translation*.

If the input field is typed directly or indirectly with a data element, you can take over the **long** field name from one of the texts stored in the Dictionary (“dictionary reference”). This provides you with an easy option for standardizing texts for all selection screens.

Input Options on Selection Screens

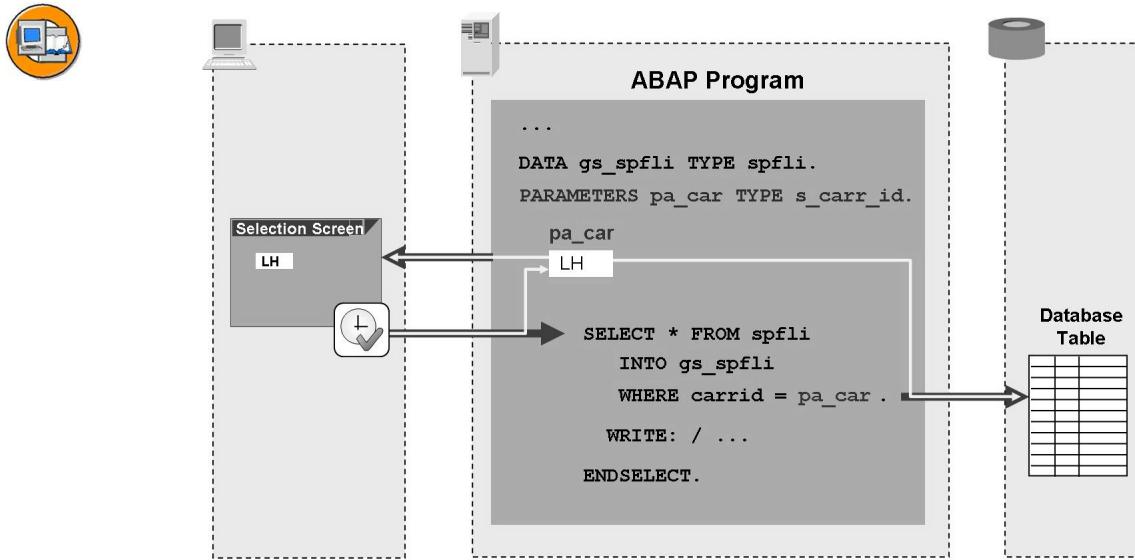


Figure 226: Single-Value Input (PARAMETERS Statement)

The above graphic shows the usage and the runtime behavior of an **input variable** defined using a `PARAMETERS` statement. Once again, the definition of such an input variable creates a variable in the system **and** implicitly generates a selection screen with a corresponding input option.

An input variable is defined in the same way as an ordinary variable. The only difference is that the `PARAMETERS` keyword is used instead of `DATA`.

You have to take three special aspects into consideration:

- The name of the input variable may be up to 8 characters long.
- It must not be typed with the standard types `F`, `STRING` and `XSTRING`.
- The assignment of a default value is **not** implemented with the `VALUE` addition but with the `DEFAULT` addition.

A default value assignment by means of the `DEFAULT` addition or value assignment **before** displaying the selection screen (`INITIALIZATION`) is displayed on the selection screen as a default value that can be overwritten.

If the user enters a value and chooses *Execute*, the input value is transferred to the internal variable and can be used to restrict the database selection, for example.

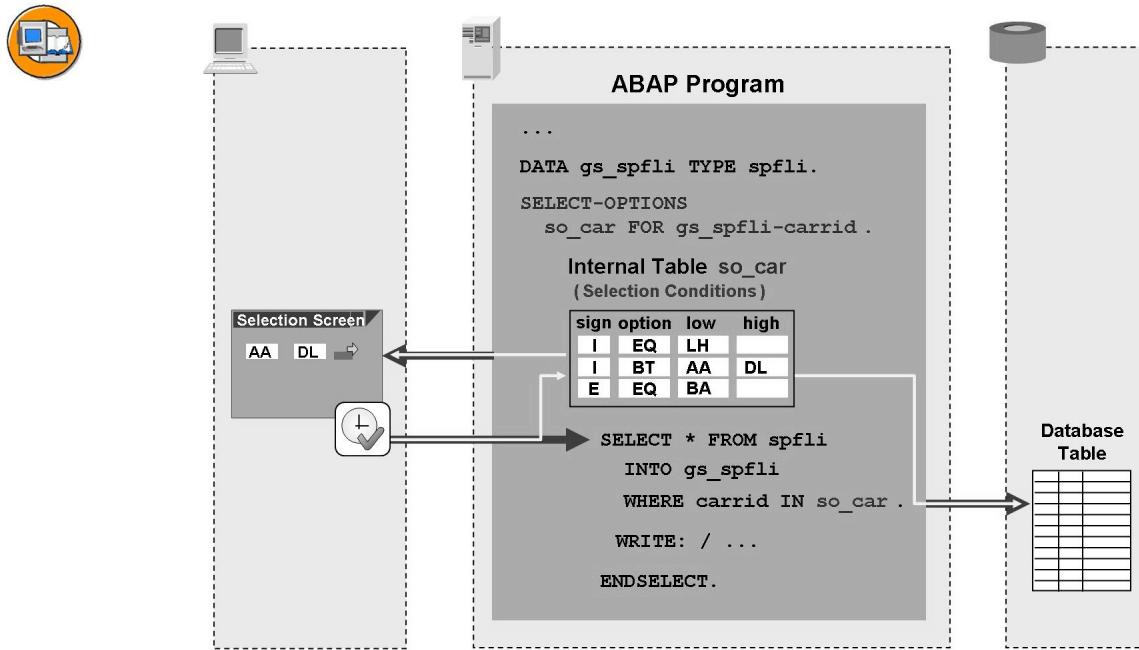


Figure 227: Entering Complex Selections (SELECT-OPTIONS Statement)

You can use the

```
SELECT-OPTIONS name FOR data_object.
```

statement to define a **selection option** for **entering complex selections**, where **name** is the name of the select option and **data_object** is a pre-defined variable. Such a definition creates an internal table of the specified name within the program (here: **so_car**) **and** generates a selection screen with an input option for limiting the specified variable (here: **gs_spfli-carrid**).

User entries are transported to the automatically generated internal table when *Execute* is chosen. This table always has four columns: **sign**, **option**, **low**, and **high**. The above graphic shows which entries are to be created for which user input:

- If 'LH' is entered, a row is generated with the values 'T' (inclusive), 'EQ' (equal), 'LH' and Space.
- If the interval 'AA' to 'DL' is entered, a row is generated with the values 'T' (inclusive), 'BT' (between), 'AA' and 'DL'.
- If 'BA' is entered as the single value to be excluded, a row is generated with the values 'E' (exclusive), 'EQ' (equal), 'BA' and Space.

The internal table filled with the entered selection criteria can be used to limit the database selection, as illustrated in the graphic. The table content is interpreted as follows:

If I_1, \dots, I_n and E_1, \dots, E_m are the inclusive/exclusive conditions of the internal table, the following composite condition is used to limit the data selection:

```
( I_1 OR ... OR I_n ) AND ( NOT E_1 ) AND ... AND ( NOT E_m )
```

If the table is empty, the WHERE condition for the relevant field is always met because there are no selections.



Hint: The IN operator can also be used in logical expressions:

```
IF gs_spfli-carrid IN so_car.
```

For the definition of a select option, the same three special features apply as for the PARAMETERS statement (see above).

If the internal table of the selection option is filled using the DEFAULT addition or APPEND statement **before** the selection screen is displayed (INITIALIZATION), its content is displayed on the selection screen as proposed conditions that can be overwritten.

For further details, refer to the keyword documentation for SELECT-OPTIONS.

ABAP Events

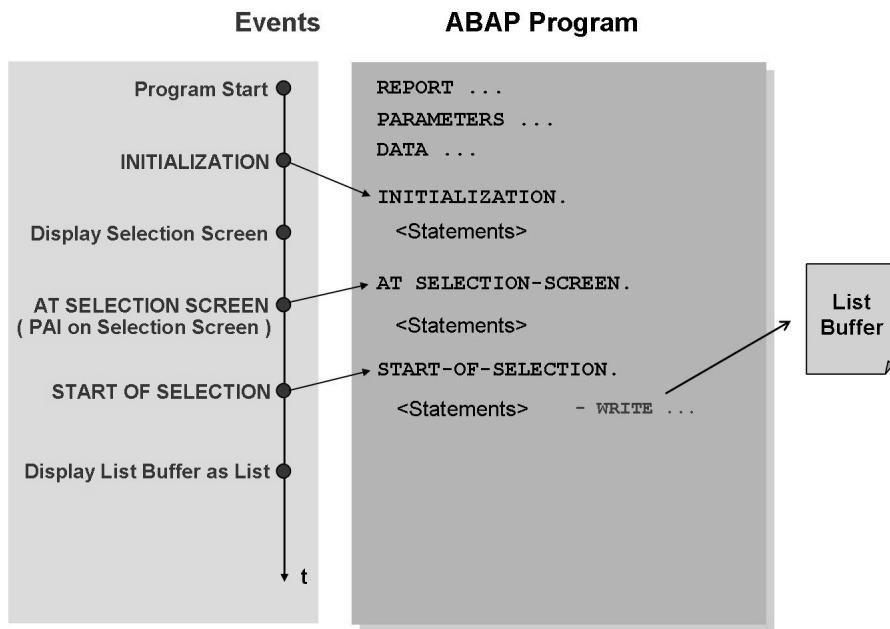


Figure 228: Structures and Procedures of an Executable ABAP Program

When you start an ABAP program, all global data objects of the programs are first created in the working memory (memory allocation). After that, the **runtime system** triggers various **events** in succession. If a **processing block** exists for a triggered event in the program, the statements in this block are executed in sequence.

The above graphic shows the basic events that are triggered, the sequence in which they are triggered, and how corresponding processing blocks are to be implemented in the program. An executable ABAP program is thus a collection of processing blocks that are processed for the respective events.

Outputs created by means of **WRITE** statements are first stored in **list buffers** and only displayed as a list once the **START-OF-SELECTION** block has been processed.

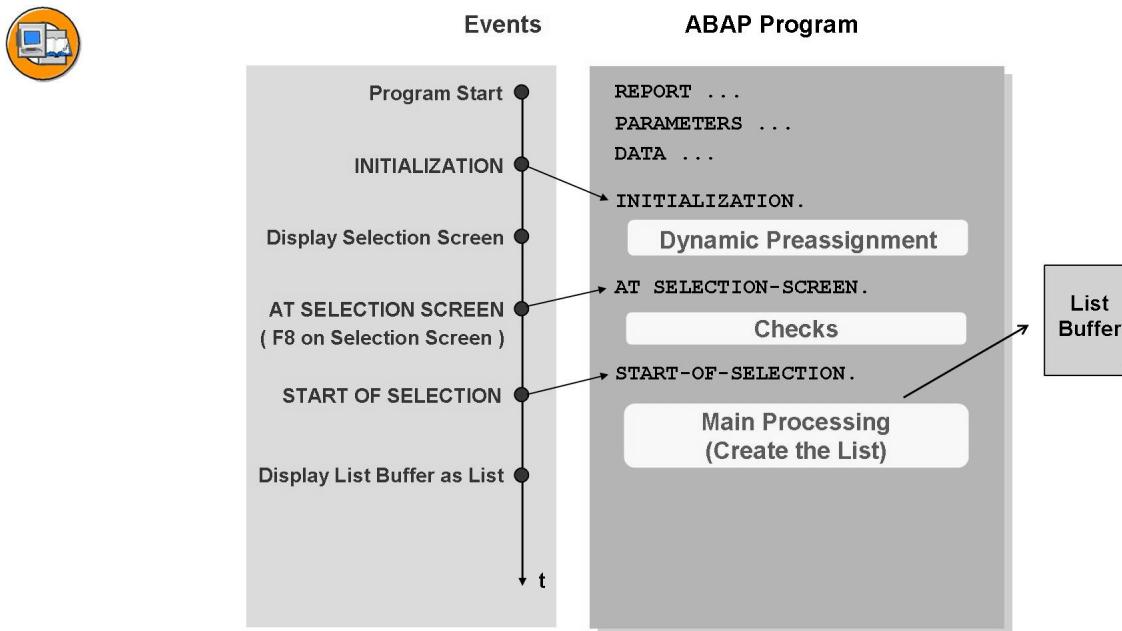


Figure 229: Using ABAP Events

If values are assigned to the PARAMETERS variables in the **INITIALIZATION** block, these are displayed as (changeable) default values in the input fields when the selection screen is subsequently displayed. You do have the option of specifying a default value for the respective input field in the PARAMETERS definition by means of the DEFAULT addition. However, you can use the value assignment described in the **INITIALIZATION** block to assign another default value (**dynamic prepopulation of the selection screen**) **dynamically** (that is, with reference to the situation).

If the user clicks a pushbutton on the selection screen (thus triggering PAI), the entries are transported into the corresponding internal PARAMETERS variables of the program and the **AT SELECTION-SCREEN** event is triggered. The corresponding processing block is therefore suitable for an **input or authorization check**. If a type 'E' message is sent in this event block, because a user does not have the required authorization, for example, this causes the selection screen to be displayed again with the error message, allowing users to correct their entries.

Only if no error message is sent in the **AT SELECTION-SCREEN** block is **START-OF-SELECTION** then triggered. The **main processing** of the program should then take place in the corresponding processing block.

Event Block Characteristics



- Introduced with an event keyword
 - Ends by beginning the next processing block
 - Cannot be nested
 - Existence not absolutely necessary
 - Sequence of event blocks unimportant
 - Implicit standard event block in the executable program :
START-OF-SELECTION

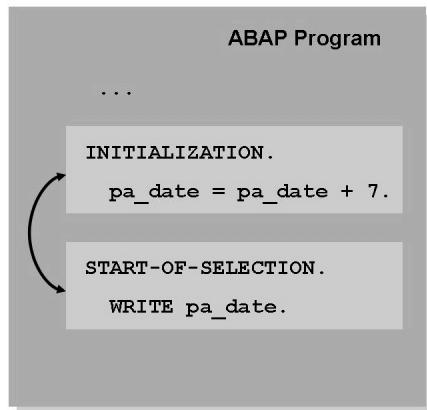


Figure 230: Event Block Characteristics

Processing blocks **cannot be nested**, since nesting would contradict the concept of ABAP events.

As mentioned, **if the processing block is missing** no statements are executed and the next event is triggered.

The ABAP runtime system controls the triggering of the events and thus the execution of the processing blocks. The sequence in which the event blocks are placed in the program is therefore **not important**.

If no blocks are implemented in the program, **all statements** are implicitly assigned to the **standard processing block START-OF-SELECTION**. This is also the reason why previous programs in this course could be executed.

The INITIALIZATION Event

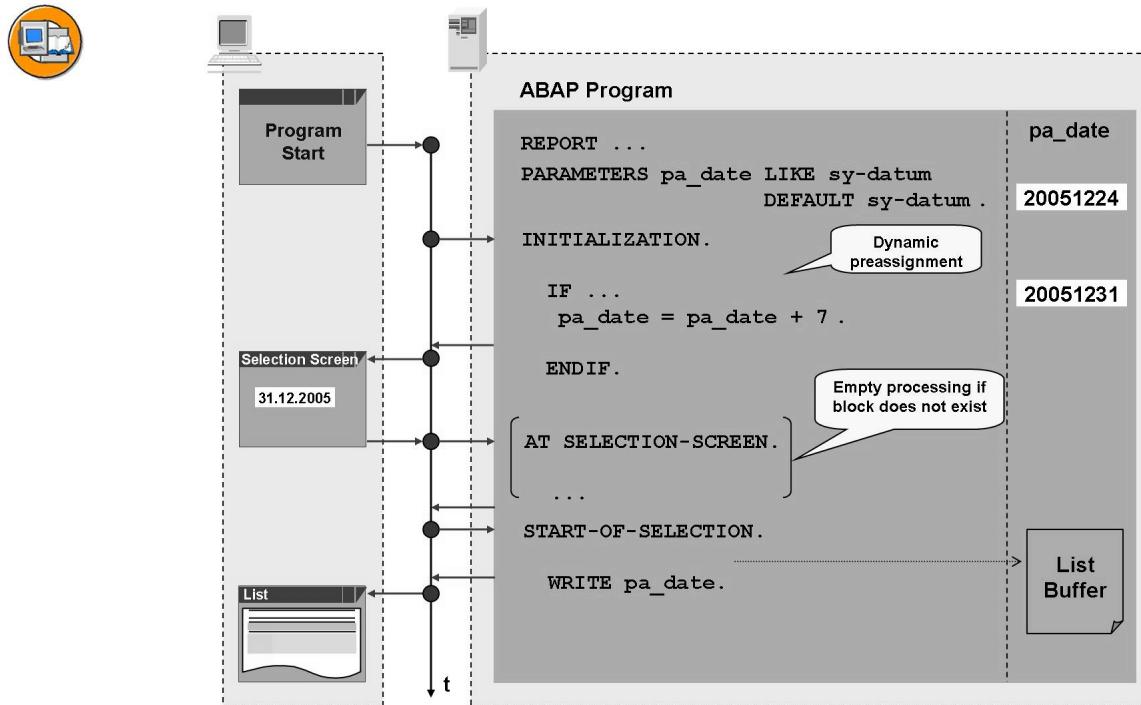


Figure 231: Example Program

The example given here contains a selection screen with an input field for a date. Under normal circumstances, the current date should appear as the default value (`DEFAULT sy-datum`). However, under certain conditions (`IF`) the date of the same weekday of the following week (`pa_date = pa_date + 7.`) is to be displayed as the default value.

The graphic above also shows how the runtime system reacts when the **processing block is missing**: There are simply no statements executed for the corresponding event and the next event is triggered.

The AT SELECTION-SCREEN Event

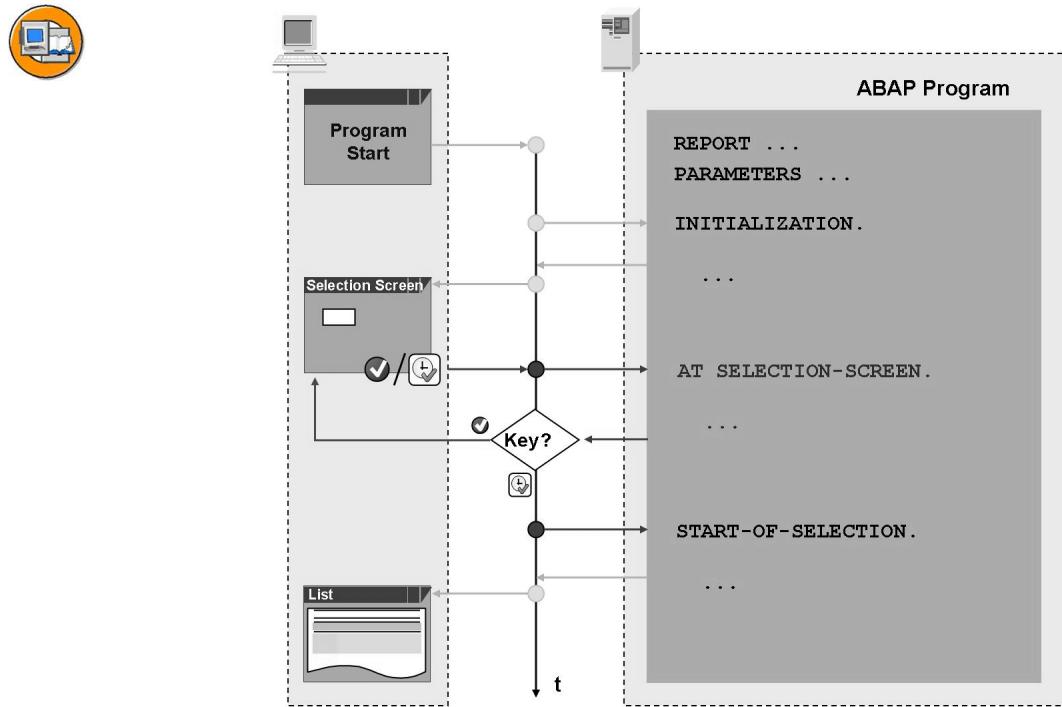


Figure 232: AT SELECTION-SCREEN Event

From the selection screen, the **AT SELECTION-SCREEN** event is triggered both by choosing **Enter** as well as **Execute (F8)**. After the corresponding processing block has been processed, the following START-OF-SELECTION event is triggered and the relevant processing started, if *Execute* was used. However, if **Enter** was chosen, the selection screen is displayed again.

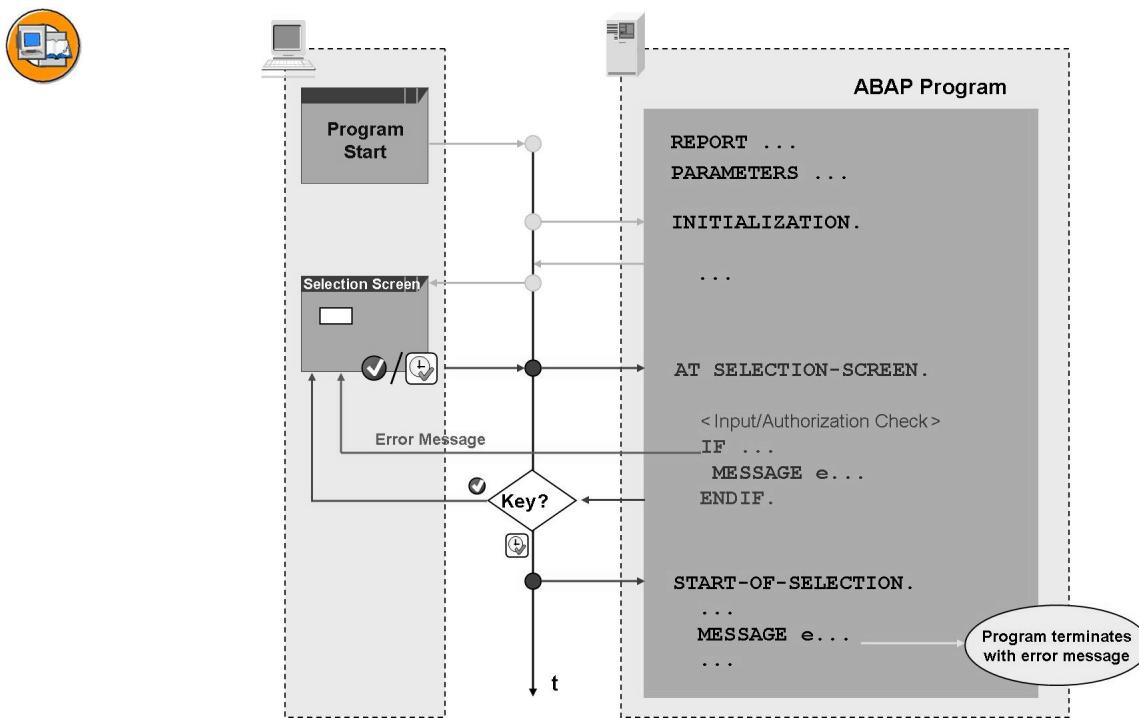


Figure 233: Error Messages in AT SELECTION-SCREEN

Immediately before processing the AT SELECTION-SCREEN block, the user entries are automatically transferred to the corresponding variables in the program. Hence, it makes sense to implement required input and authorization checks in this processing block. If the check result is negative, you can simply send out an error message to the user and have the selection screen displayed again. The user then has the chance to make new entries.

In contrast, an error message in the START-OF-SELECTION block would be displayed under program termination.

The following graphic illustrates a simple example program with authorization check and error dialog on the selection screen.



```
...
PARAMETERS pa_car TYPE s_carr_id.

CONSTANTS gc_actvt_display TYPE activ_auth VALUE '03'.

* Event processed after leaving the selection screen

AT SELECTION-SCREEN.

TRY.
  CALL METHOD cl_bc400_flightmodel=>check_authority
    EXPORTING
      iv_carrid = pa_car
      iv_activity = gc_actvt_display.
  CATCH cx_bc400_no_auth .
*   Show selection screen again with error message
*   MESSAGE e046(bc400) WITH pa_car.
ENDTRY.

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

Figure 234: Syntax Example: Authorization Check with Error Dialog

Additional information can be found in the keyword documentation for “AT SELECTION-SCREEN”.

Other ABAP Events

Apart from the events that have already been explained for executable programs, there are additional events for various other tasks. For example, there is the AT LINE-SELECTION event that is triggered when the user double-clicks on the list. This event can be used to display the detailed information for the row in the list that was clicked on, for example. You will find further information about this and other ABAP events in the ABAP documentation for the term “Event”.

Exercise 25: Selection Screen and Classic ABAP List

Exercise Objectives

After completing this exercise, you will be able to:

- Use the ABAP SELECT-OPTIONS statement to implement a complex selection option on the selection screen
- Transfer complex selections to a data retrieval method
- Use translatable text elements on the selection screen and in the ABAP list
- Use icons and colors in a classic list (optional)

Business Example

Instead of a single flight number, you want to be able to enter complex selection criteria for data selection on the selection screen.

Furthermore, you want to make the design of the the selection screen and the ABAP list more appealing by using translatable texts and, possibly, colors and icons.

Template:

BC400_RPT_REP_A

Solution:

BC400_RPS_REP_A

Task 1:

Copy the template

1. Copy the template to the name **ZBC400_##_REP_A**.

Continued on next page

Task 2:

Replace the parameter for the flight number on the selection screen with a complex selection (select option). Adjust the data retrieval in such a way that the select option is used. To do this, call a method of the class cl_bc400_flightmodel to which you can transfer the select option table.

1. Replace the declaration of the parameter pa_con with the declaration of a select option (suggested name: **so_con**).



Hint: Reference the same structure type in the Dictionary when typing the select option and the parameter pa_car so that the corresponding search help on the selection screen is context-dependent.

2. Replace the call of the method cl_bc400_flightmodel=>get_flights with a call of the method cl_bc400_flightmodel=>get_flights_range. Transfer the select option table to the method as an actual parameter.



Hint: Note that the data object so_con is an internal table **with a header**. The parameters can only be transferred if you explicitly transfer the “body” of the table (also so_con[]).

3. Activate and test your program.

Task 3:

Make sure that language-dependent texts are displayed on the selection screen instead of the names of the data objects.

1. Maintain the text elements of the program and define selection texts. Take the opportunity to use texts that have already been defined and translated in the *ABAP Dictionary*.
2. Activate and test your program.

Task 4:

If no data is available, output a translatable text in the list and create translatable headers.

1. Maintain a text symbol and output it with the WRITE statement if the method returns no data.
2. Maintain the headers for the list.

Continued on next page

Task 5: Optional:

Define the key fields for the flight times as light blue. Output an icon at the start of each row to show the utilization by means of a traffic light: Red for fully booked flights, yellow for well-booked flights, and green for flights with few or no bookings.

1. Provide the output of each of the key fields (CARRID, CONNID, FLDATE) with a COLOR addition and the color constant for key fields from the COL type group.
2. Output a traffic light icon at the start of each row. Use one of the constants *ICON_RED_LIGHT*, *ICON_YELLOW_LIGHT* and *ICON_GREEN_LIGHT* from the *ICON* type group, depending on the utilization percentage of the flight.

Solution 25: Selection Screen and Classic ABAP List

Task 1:

Copy the template

1. Copy the template to the name **ZBC400 ## REP_A**.
 - a) Carry out this step in the usual manner.

Task 2:

Replace the parameter for the flight number on the selection screen with a complex selection (select option). Adjust the data retrieval in such a way that the select option is used. To do this, call a method of the class cl_bc400_flightmodel to which you can transfer the select option table.

1. Replace the declaration of the parameter pa_con with the declaration of a select option (suggested name: **so_con**).



Hint: Reference the same structure type in the Dictionary when typing the select option and the parameter pa_car so that the corresponding search help on the selection screen is context-dependent.

- a) See source code excerpt from the model solution.
2. Replace the call of the method cl_bc400_flightmodel=>get_flights with a call of the method cl_bc400_flightmodel=>get_flights_range. Transfer the select option table to the method as an actual parameter.



Hint: Note that the data object so_con is an internal table **with a header**. The parameters can only be transferred if you explicitly transfer the “body” of the table (also so_con[]).

- a) See source code excerpt from the model solution.
3. Activate and test your program.
 - a) Carry out this step in the usual manner.

Continued on next page

Task 3:

Make sure that language-dependent texts are displayed on the selection screen instead of the names of the data objects.

1. Maintain the text elements of the program and define selection texts. Take the opportunity to use texts that have already been defined and translated in the *ABAP Dictionary*.
 - a) Open the maintenance of the selection screen by choosing *Goto → Text Elements → Selection Texts*.
 - b) Select the checkbox in the *Dictionary Reference* column for all the listed elements of the selection screen..



Hint: Note that only those parameters and select options are offered here that have been defined in the **active version** of the source code.

- c) Save and activate the texts.
2. Activate and test your program.



Hint: Note that you have to activate the texts (REPT type Repository object) before they are displayed when the program is executed.

Task 4:

If no data is available, output a translatable text in the list and create translatable headers.

1. Maintain a text symbol and output it with the WRITE statement if the method returns no data.
 - a) Open the maintenance of the text symbols by choosing *Goto → Text Elements → Text Symbols*.
 - b) Assign a three-character abbreviation and enter a text.
 - c) Save and activate the texts.
 - d) Output the text symbol on the list as shown in the source code excerpt from the model solution.

Continued on next page

2. Maintain the headers for the list.
 - a) Execute the program.
 - b) Branch from the list display to header maintenance (*System → List → List Header*).
 - c) Maintain the headers.
 - d) Restart the program for testing.



Hint: Note that the list will be buffered on the presentation server. The headers will only be displayed when you restart the program.

Task 5: Optional:

Define the key fields for the flight times as light blue. Output an icon at the start of each row to show the utilization by means of a traffic light: Red for fully booked flights, yellow for well-booked flights, and green for flights with few or no bookings.

1. Provide the output of each of the key fields (CARRID, CONNID, FLDATE) with a COLOR addition and the color constant for key fields from the COL type group.
 - a) See source code excerpt from the model solution.
2. Output a traffic light icon at the start of each row. Use one of the constants *ICON_RED_LIGHT*, *ICON_YELLOW_LIGHT* and *ICON_GREEN_LIGHT* from the *ICON* type group, depending on the utilization percentage of the flight.
 - a) See source code excerpt from the model solution.

Result

Source code excerpt from the model solution:

```
*&-----*
*& Report BC400_RPS REP_A
*&-----*
REPORT bc400_rps_rep_a.

TYPE-POOLS:
  icon,
  col.

CONSTANTS:
```

Continued on next page

```

gc_limit_red      TYPE s_flghtocc VALUE 98,
gc_limit_yellow  TYPE s_flghtocc VALUE 75.

DATA:
  gt_flights  TYPE bc400_t_flights,
  gs_flight   TYPE bc400_s_flight.

PARAMETERS:
  pa_car      TYPE bc400_s_flight-carrid.

SELECT-OPTIONS:
  so_con FOR gs_flight-connid.

TRY.
  CALL METHOD cl_bc400_flightmodel->get_flights_range
    EXPORTING
      iv_carrid  = pa_car
      it_connid  = so_con[] " brackets needed: so_con has header line!
    IMPORTING
      et_flights = gt_flights.
  CATCH cx_bc400_no_data.

    WRITE / 'Keine Flüge für die gewählte Flugverbindung' (non).
*   WRITE / text-non.  "alternative usage of text symbol
ENDTRY.

LOOP AT gt_flights INTO gs_flight.

  NEW-LINE.

  IF gs_flight-percentage >= gc_limit_red.
    WRITE icon_red_light AS ICON.
  ELSEIF gs_flight-percentage >= gc_limit_yellow.
    WRITE icon_yellow_light AS ICON.
  ELSE.
    WRITE icon_green_light AS ICON.
  ENDIF.

  WRITE: gs_flight-carrid    COLOR COL_KEY,
         gs_flight-connid  COLOR COL_KEY,
         gs_flight-fldate   COLOR COL_KEY,
         gs_flight-seatsmax,
```

Continued on next page

```
gs_flight-seatsocc,  
gs_flight-percentage.
```

```
ENDLOOP.
```

Exercise 26: ABAP Events

Exercise Objectives

After completing this exercise, you will be able to:

- Implement dynamic preassignment on selection screens
- Implement error dialogs for the standard selection screen

Business Example

You want to make a dynamic preassignment for the airline ID on the selection screen. If user enter an airline for which they do not have display authorization, an error message should be output and the selection screen displayed again.

Template:

BC400_RPS_REP_A

Solution:

BC400_RPS_REP_B

Task 1:

Copy your program ZBC400_##_REP_A or the copy template.

1. Copy the template to the name **ZBC400_##_REP_B**.

Task 2:

Create an ABAP event block that is run before the selection screen is displayed and implement a dynamic preassignment of the parameter for the airline ID.

1. Create the event block INITIALISATION and fill the parameter pa_car dynamically with the value “LH”.



Hint: Note that you have to assign the previous executable part of your program explicitly to the START-OF-SELECTION event block before you define the INITIALISATION event block.

Continued on next page

Task 3:

Make sure that after a user action on the selection screen, the system checks whether the user is authorized to display data for the airline that was entered. Implement the check in such a way that you can display the selection screen again with an error message if the authorization does not exist.

1. Create the AT SELECTION-SCREEN event block and implement an authorization check. To do this, use a method of the class `cl_bc400_flightmodel`.
 2. If the authorization is missing, send an appropriate message from the message class BC400.
 3. Which message type do you have to use so that the selection screen is displayed again and the user cannot access the START-OF-SELECTION processing block without authorization.
-
-
-
-

Solution 26: ABAP Events

Task 1:

Copy your program ZBC400_##_REP_A or the copy template.

1. Copy the template to the name **ZBC400_##_REP_B**.
 - a) Carry out this step in the usual manner.

Task 2:

Create an ABAP event block that is run before the selection screen is displayed and implement a dynamic preassignment of the parameter for the airline ID.

1. Create the event block INITIALISATION and fill the parameter pa_car dynamically with the value “LH”.



Hint: Note that you have to assign the previous executable part of your program explicitly to the START-OF-SELECTION event block before you define the INITIALISATION event block.

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

Task 3:

Make sure that after a user action on the selection screen, the system checks whether the user is authorized to display data for the airline that was entered. Implement the check in such a way that you can display the selection screen again with an error message if the authorization does not exist.

1. Create the AT SELECTION-SCREEN event block and implement an authorization check. To do this, use a method of the class cl_bc400_flightmodel.
 - a) See source code excerpt from the model solution.
2. If the authorization is missing, send an appropriate message from the message class BC400.
 - a) See source code excerpt from the model solution.
3. Which message type do you have to use so that the selection screen is displayed again and the user cannot access the START-OF-SELECTION processing block without authorization.

Answer: Error message, message type “E”.

Continued on next page

Result

Source code excerpt from the model solution:

```
*&-----*
*& Report BC400_RPS REP_B
*&-----*
REPORT bc400_rps_rep_b.

TYPE-POOLS:
  icon,
  col.

CONSTANTS:
  gc_limit_red      TYPE s_flightocc VALUE 98,
  gc_limit_yellow   TYPE s_flightocc VALUE 75,
  gc_actvt_display  TYPE activ_auth VALUE '03'.

DATA:
  gt_flights TYPE bc400_t_flights,
  gs_flight   TYPE bc400_s_flight.

PARAMETERS:
  pa_car TYPE bc400_s_flight-carrid.

SELECT-OPTIONS:
  so_con FOR gs_flight-connid.

INITIALIZATION.

pa_car = 'LH'.

AT SELECTION-SCREEN.

TRY.
  CALL METHOD cl_bc400_flightmodel=>check_authority
    EXPORTING
      iv_carrid  = pa_car
      iv_activity = gc_actvt_display.
  CATCH cx_bc400_no_auth .
    MESSAGE e046(bc400) WITH pa_car.
  *  Keine Anzeige-Berechtigung für Fluggesellschaft &1
```

Continued on next page

```
ENDTRY.

START-OF-SELECTION.

TRY.
  CALL METHOD cl_bc400_flightmodel->get_flights_range
    EXPORTING
      iv_carrid = pa_car
      it_connid = so_con[]      " brackets needed: so_con has header line!
    IMPORTING
      et_flights = gt_flights.
  CATCH cx_bc400_no_data.

  WRITE / 'Keine Flüge für die gewählte Flugverbindung' (non).

  ENDTRY.

  LOOP AT gt_flights INTO gs_flight.

    NEW-LINE.

    IF gs_flight-percentage >= gc_limit_red.
      WRITE icon_red_light AS ICON.
    ELSEIF gs_flight-percentage >= gc_limit_yellow.
      WRITE icon_yellow_light AS ICON.
    ELSE.
      WRITE icon_green_light AS ICON.
    ENDIF.

    WRITE: gs_flight-carrid      COLOR COL_KEY,
           gs_flight-connid     COLOR COL_KEY,
           gs_flight-fldate     COLOR COL_KEY,
           gs_flight-seatsmax,
           gs_flight-seatsocc,
           gs_flight-percentage.

  ENDLOOP.
```



Lesson Summary

You should now be able to:

- List the properties and benefits of selection screens
- Implement the options for restricting selections on the selection screen
- Describe the attributes and benefits of ABAP lists
- Implement list and column headers
- Implement multilingual lists
- Describe the event-controlled processing of an executable ABAP program
- List the most important basic events and explain their purpose

Lesson: Displaying Tables with the SAP List Viewer

Lesson Overview

The *SAP List Viewer* is a tool you can use to display an internal table on a screen. As well as the graphic formatting of the data, the *SAP List Viewer* offers a whole range of standard functions, such as sort and filter, for example. An SAP standard class is used to control the *SAP List Viewer*.

Note that this lesson can only offer an introduction to the topic. Detailed knowledge of programming the *SAP List Viewer* is provided in the training courses BC405 and BC412. Knowledge about object-oriented programming in ABAP is given in the training course BC401.



Lesson Objectives

After completing this lesson, you will be able to:

- Use the SAP Grid Control (*SAP List Viewer*) to display an internal table on a screen

Business Example

You want to present data that exists in an internal table in the user dialog in an interesting way, and provide the user with additional functions with the minimum of effort.

The SAP List Viewer

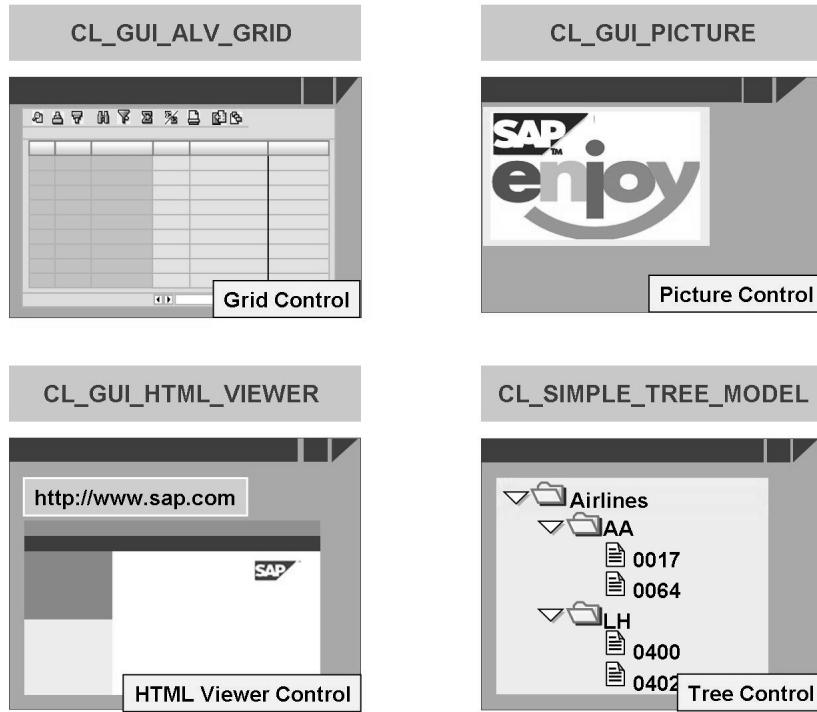


Figure 235: Examples of *EnjoySAP Controls*

With release 4.6, SAP delivered a large number of so-called ***EnjoySAP Controls***, which you can use to design screens more ergonomically and interestingly. The above graphic shows a selection of these controls:

Grid Control

For displaying an internal table on a screen with many interesting functions, such as sort, filter, and totals, for example.

Picture Control

For displaying a picture on the screen

HTML Viewer control

For displaying an HTML file or Web page on the screen

Tree Control

For depicting a hierarchical list in the form of a tree structure on the screen

Classes and methods that are delivered with the SAP standard system are used to communicate between these controls and an ABAP program. In the following, we will use the class CL_GUI_ALV_GRID by way of example to address the most interesting and popular *EnjoySAP Control*.

For detailed information on all EnjoySAP controls as well as interaction between them, refer to course **BC412**.

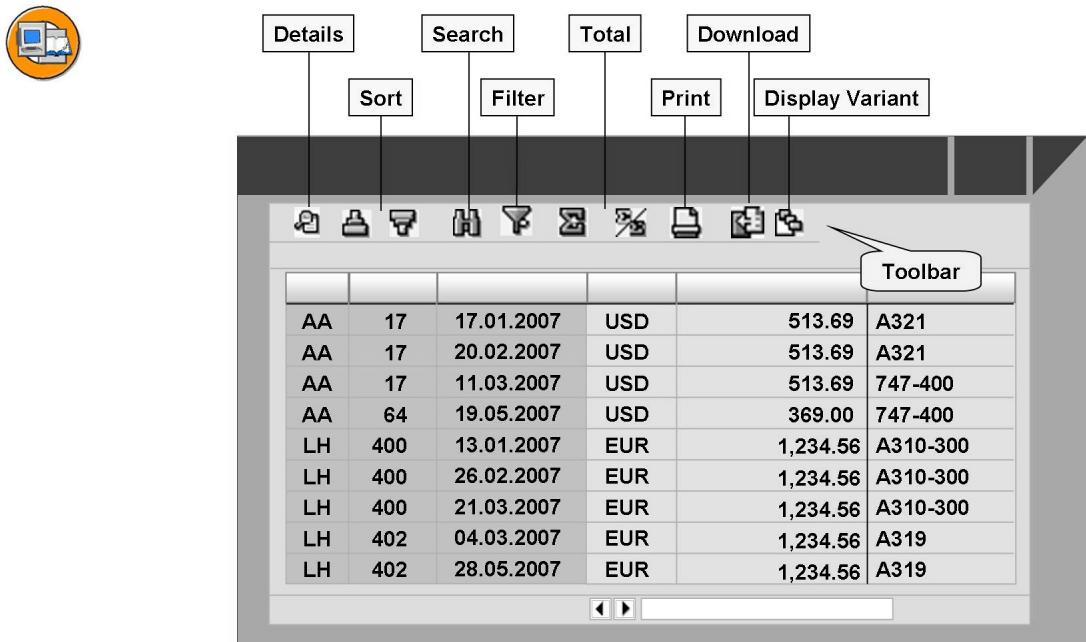


Figure 236: Application Example:SAP Grid Control

The **ALV Grid Control**, also known as the SAP List Viewer or ABAP List Viewer (**ALV**), is used to display an internal table on a screen. It has numerous user functions:

On the screen, the user can vary the width of the columns, or the width can be automatically adjusted to the current data. The display position of the columns can also be changed by drag and drop.

The standard pushbuttons of the control can be used to execute, among others, the following functions:

The detailed display shows the fields which were previously selected with the cursor in a modal dialog window.

The sort function provides the user with the option of specifying complex sorting criteria for the columns.

Within the selected area, you can use the search function for searching for a character string in rows or columns.

You can form totals for one or each of several numerical columns. You can then use the *Subtotals* function to set up control level lists: Select the non-numeric columns that you want to include before choosing this function. The corresponding control level totals are then displayed.

You can also *Print* and *Download* with the corresponding pushbuttons.

The user can save his or her settings in the Grid Control as a display variant and reuse them at a later time.

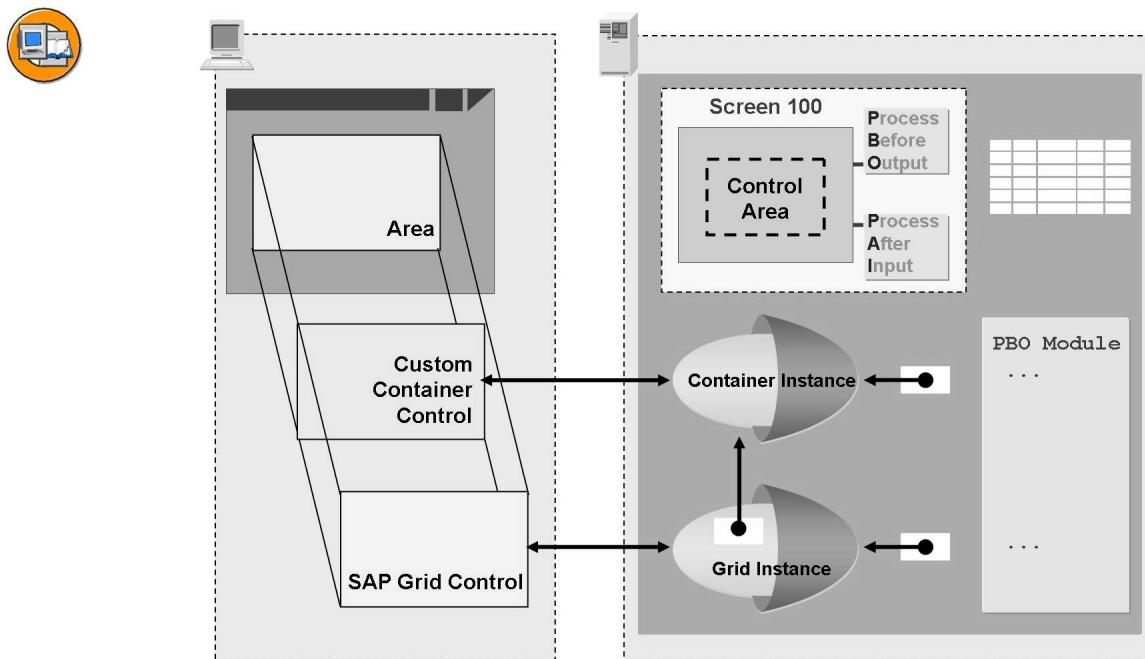


Figure 237: Runtime Architecture of the ALV Grid Control

An *EnjoySAP Control* is always displayed together with a screen. There are various options for the optical attachment of the *EnjoySAP Control* to the screen. The simplest of these is to use the *Graphic Screen Painter* to create a special **control area** on the layout of the screen (see the left side of the above graphic). An **SAP Container Control** (referred to from now on as **container**) is used to link the *EnjoySAP Control* and the control area on the screen. The *EnjoySAP Control* is embedded in the container, which in turn is integrated in the control area.

In order to implement the grid control and the container on the GUI, corresponding instances have to be created as “substitutes” in the ABAP program. You can use these instances to address the elements in the GUI. To do so, your SAP system has standard classes from which you can generate the container and grid control instance.

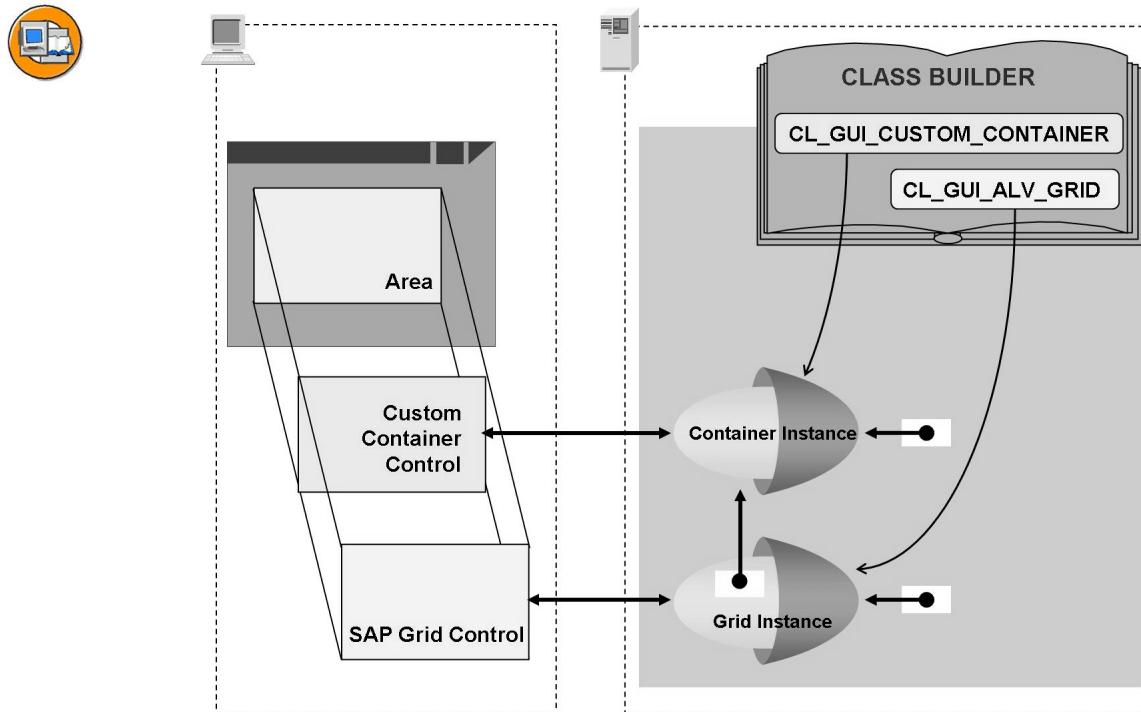
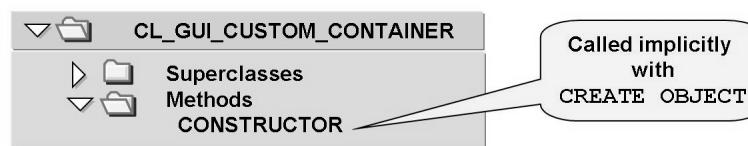


Figure 238: Classes for Controlling Containers and Grid Controls

When you create an instance, the **class-specific constructor** (special method **CONSTRUCTOR** of the class) is called implicitly. The task of this method is to use its own input parameters to fill the attributes of the instance to be created. You must therefore supply the required import parameters of the constructor with values when you create an instance (**CREATE OBJECT** statement).



Parameter	Value	Optional	Associated Type	Description
PARENT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
CONTAINER_NAME	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
...	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		

Figure 239: The CONSTRUCTOR Method of the Container Class

To obtain detailed information on a global class or method, you can navigate to the *Class Builder*:

Display the object list of the class in the navigation area of the *Object Navigator*. Double-clicking on the class takes you to the detailed display in the *Class Builder*. (Alternatively, you can also go to the *Class Builder* by double-clicking the class names from within an ABAP program.) Select the required method with the cursor and press the *Parameter* pushbutton to display the interface parameters of the method.

Method **CONSTRUCTOR** of global class **CL_GUI_CUSTOM_CONTAINER** (class for the container) has the required parameter **CONTAINER_NAME**. When you create the container, you therefore have to supply at least this parameter with data, namely with the name of the control area on the screen.

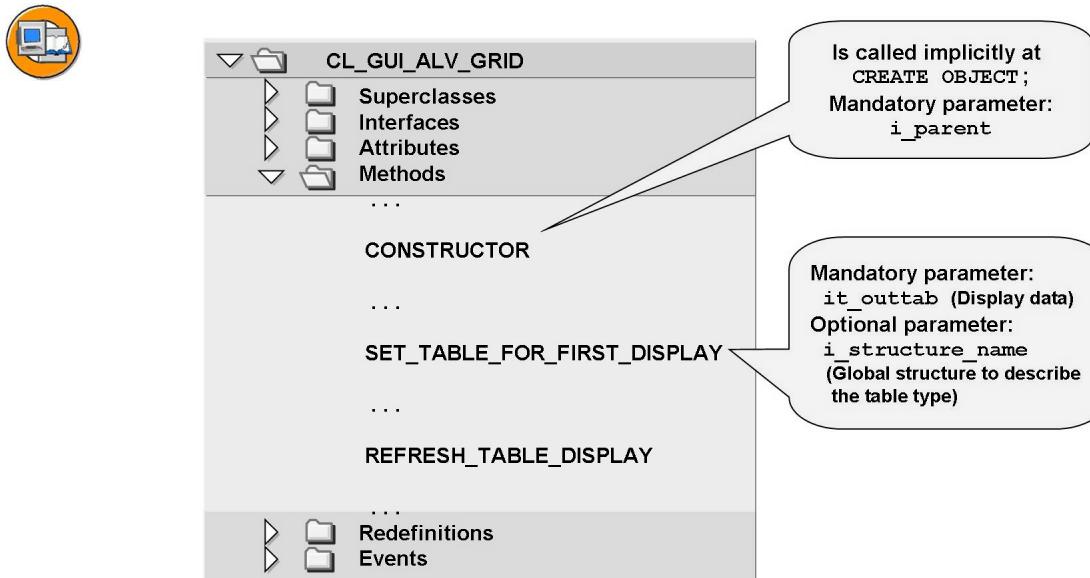


Figure 240: Important Methods for the Grid Control Class

The global class **CL_GUI_ALV_GRID** has numerous methods that can be called for the corresponding grid control functions. To display the contents of an internal table with an SAP Grid Control, it is sufficient to know three methods in more detail:

CONSTRUCTOR

The grid class has a constructor too. The only required parameter is **`i_parent`**, to which the previously created container instance - in the form of a pointer - has to be transferred (when the grid control instance is created).

SET_TABLE_FOR_FIRST_DISPLAY

This method of the created grid control instance is used for transferring data and settings to the grid control. The internal table to be displayed must be transferred to the parameter **IT_OUTTAB**. This must be a standard table (see typing of the parameter).

Furthermore, technical information is required for the formatting of the grid columns. The easiest thing to do is to use a Dictionary structure or transparent table as the row description of the internal table. In this case, all you have to do is transfer the name of the Dictionary object to the parameter **I_STRUCTURE_NAME**. (Alternatively, you can also set up a field catalog and transfer it to the parameter **IT_FIELDCATALOG**.)

REFRESH_TABLE_DISPLAY

You only need to call this method if the table content, which has changed in the meantime, is to be loaded to the grid control to update the display if the screen is processed again.



Displaying an Internal Table in the ALV Grid Control on a Screen

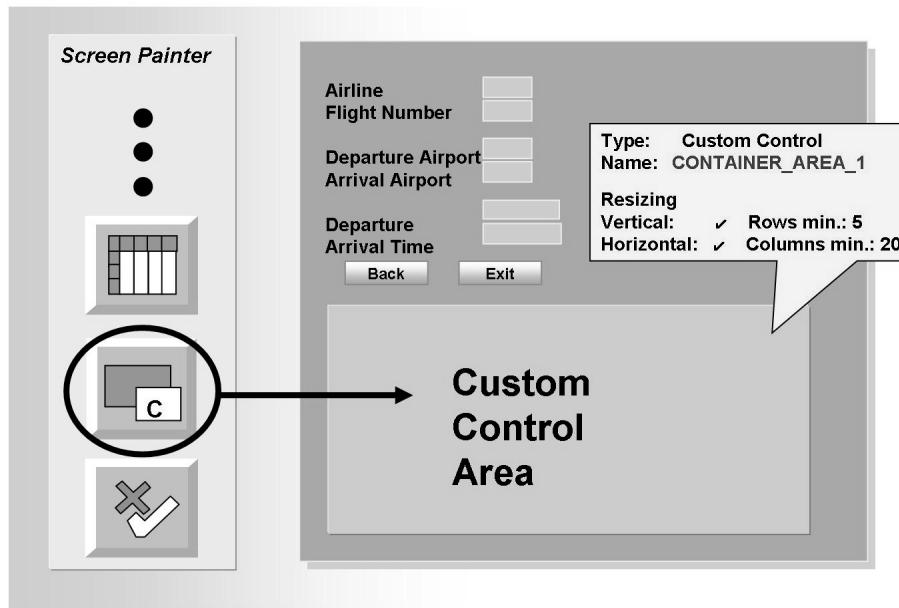


Figure 241: Creating the Screen Element "Custom Control Area"

1. In the *Graphic Layout Editor*, you can define a control area on your screen. To do so, choose the *Custom Control* pushbutton from the toolbar. Now select it and specify the size and position of the area on the screen as follows:

Click the editing area where you want to place the top left corner of the custom control and hold down the mouse key. 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.

Enter a name for the new screen element (here: **CONTAINER_AREA_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. Lines* and *Min. Columns*.

Continued on next page

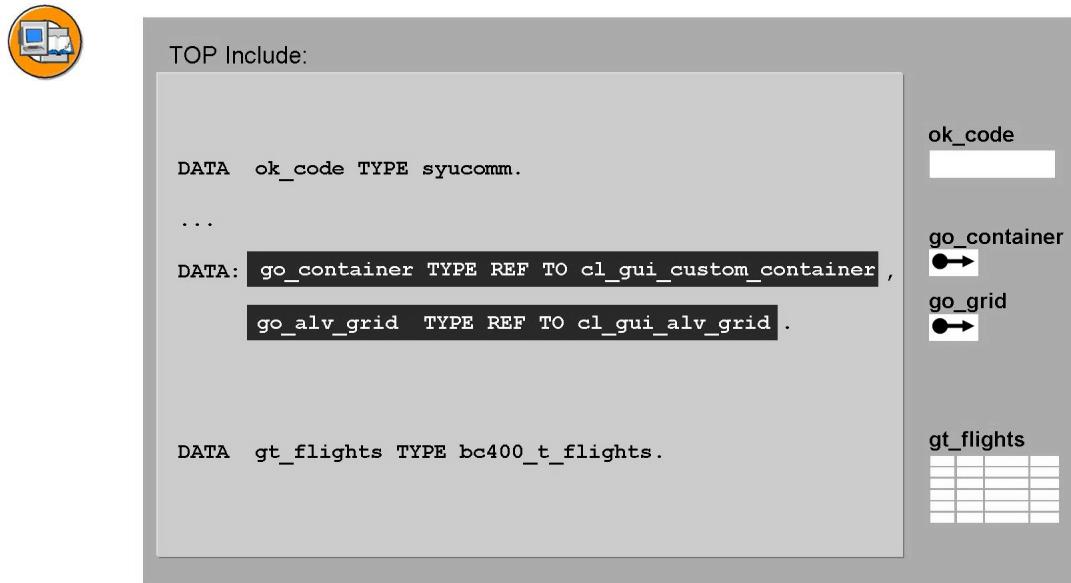


Figure 242: Defining Reference Variables

2. Two reference variables are required in the ABAP program:
 - A reference variable that should point to a container instance that has yet to be created (name here: **GO_CONTAINER**)
 - A reference variable that should point to a grid control instance that has yet to be created (name here: **GO_ALV_GRID**)

Continued on next page



O Include:

```

MODULE init_control_processing_0200 OUTPUT.

IF go_container IS INITIAL.

    CREATE OBJECT go_container
        EXPORTING container_name = 'CONTROL_AREA_1'.

    CREATE OBJECT go_alv_grid
        EXPORTING i_parent      = go_container.

    ...

ENDIF.

ENDMODULE.
```

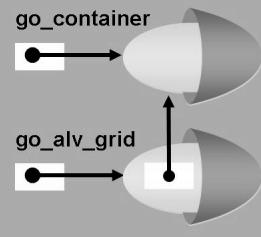


Figure 243: Creating Instances

3. You create the instances with the `CREATE OBJECT` statement. You should definitely have your statement generated in your source code in order to avoid typing errors and omissions. To do so, display the object list of the respective class in the navigation area of the *Object Navigator*, and then drag and drop the class name in your source code.

In the generated call, you must now insert the name of the reference variable for `xxxxxxxx` and supply the parameters with values. The call syntax is very similar to that of the function module. However, the parameters to be supplied with values with `CREATE OBJECT` are the interface parameters of the respective constructor.

It makes sense to implement the creation of the control instances **before** displaying the screen, in other words, in a **PBO module**.

You only have to instantiate the controls on a screen once. This means that this step is omitted when a screen is processed again. This is easily done by querying one of the two reference variables:

```
IF go_container IS INITIAL.
```

Continued on next page



O Include:

```

MODULE init_control_processing_0200 OUTPUT.

IF go_container IS INITIAL.

  CREATE OBJECT go_container
    EXPORTING container_name = 'CONTROL_AREA_1'.

  CREATE OBJECT go_alv_grid
    EXPORTING i_parent      = go_container.

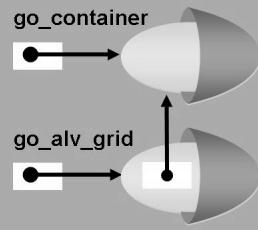
  CALL METHOD
    go_alv_grid->set_table_for_first_display
    EXPORTING i_structure_name = 'BC400_S_FLIGHT'
    CHANGING it_outtab       = gt_flights.

ELSE.

  CALL METHOD
    go_alv_grid->refresh_table_display .

ENDIF.

ENDMODULE.
```

**Figure 244: Calling Methods**

4. To transfer the content of an internal table and its row description to the SAP Grid Control, call the method **SET_TABLE_FOR_FIRST_DISPLAY** of the grid control instance. Here, you should also generate the call by drag and drop. In the generated call, you then have to use the name of the grid control reference variable instead of **xxxxxxxx** and supply the parameters with values:
- You then transfer the filled internal table to parameter **IT_OUTTAB**.
 - Since the internal table has the global structure type **BC400_S_FLIGHT** as a line type here, it is sufficient to pass this name to the **I_STRUCTURE_NAME** parameter. The corresponding Dictionary information is then loaded automatically and transferred to the control.

If the **content** of the internal table changes during the program, you must call the **REFRESH_TABLE_DISPLAY** method in order to update the grid display **before** the next screen is displayed.

Exercise 27: Displaying an Internal Table with the SAP List Viewer

Exercise Objectives

After completing this exercise, you will be able to:

- Output an internal table with an SAP grid control

Business Example

You want to extend a program that outputs detailed data for a flight connection on a screen. In addition to the detailed data, you want to display a list of flight times and their respective utilization.

Template:

SAPMBC400_UDS_C

Solution:

SAPMBC400_UDS_D

Task 1:

Copy your program SAPMZBC400_##_C or the template to the new name **SAPMZBC400_##_D**.

1. Copy the program with all the screens and includes. Choose an appropriate name for the new includes (suggested name: **MZBC400_##_DTOP**, **MZBC400_##_DI01**, and so on).

Task 2:

Prepare the second screen (screen 200) for displaying a control. To do this, create a control area on the screen (suggested name: **CONTROL_AREA_FLIGHTS**).

1. Edit the screen layout with the *Graphic Screen Painter* tool and create a control area below the pushbuttons.
2. Maintain the attributes of the control area. Activate *Resizing* (horizontal and vertical) for the control area and enter the minimum size for the control area (suggestion: at least 5 rows and at least 25 columns).

Continued on next page

Task 3:

Make sure that in addition to the detailed data for the flight connection,a list of flight times is also retrieved. Store this data with the global data for the program in the form of an internal table. Use the function module BC400_DDS_FLIGHTLIST_GET or the function module you developed yourself for data retrieval.

1. Create a global internal table in the Top include (suggested name: **gt_flights**), that you can use as an actual parameter for the function module call.
2. Edit the PAI module in which the function code for the first screen (screen 100) is evaluated. Implement a call of the function module for reading the flight list if the existing function module call was successful.

Task 4:

Create a new PBO module for the second screen (suggested name: init_control_processing_0200).In this module, generate an instance for the container and one for the grid control, and transfer the global internal table with the flight times to the instance of the grid control.

1. In the TOP include, define a reference variable for the class CL_GUI_CUSTOM_CONTAINER and one for the class CL_GUI_ALV_GRID (suggested names: **go_container** and **go_alv_grid**).
2. Create a new PBO module for the second screen and implement the generation of the container and control instances in it. Supply only the mandatory parameters, that is CONTAINER_NAME for the container instance and I_PARENT for the grid control instance.



Caution: The CONTAINER_NAME parameter must be supplied with the **name** of the control area on the screen, that is, a character literal in quotation marks. Make sure you spell the name correctly using uppercase only.

The parameter I_PARENT, on the other hand, must be supplied with a reference to the container instance. You cannot use quotation marks here.

3. After the instance has been generated, call the instance method SET_TABLE_FOR_FIRST_DISPLAY for the instance of the grid control. Supply the parameter I_STRUCTURE_NAME with the name of the line type of the internal table, and the parameter IT_OUTTAB with the internal table itself.

Continued on next page



Caution: The I_STRUCTURE_NAME parameter must be supplied with the **name** of the data type, that is, a character literal in quotation marks. Make sure you spell the name correctly using uppercase only.

The parameter IT_OUTTAB must be supplied with the **content** of the internal table. You cannot use quotation marks here.

4. Ensure that the instances are created and the method called only when the screen is processed for the first time. Otherwise, call the instance method REFRESH_DISPLAY for the grid control instance.
5. Activate the program, create a dialog transaction, and test your application.

Solution 27: Displaying an Internal Table with the SAP List Viewer

Task 1:

Copy your program SAPMZBC400_##_C or the template to the new name **SAPMZBC400_##_D**.

1. Copy the program with all the screens and includes. Choose an appropriate name for the new includes (suggested name: **MZBC400_##_DTOP**, **MZBC400_##_DI01**, and so on).
 - a) Carry out this step in the usual manner.

Task 2:

Prepare the second screen (screen 200) for displaying a control. To do this, create a control area on the screen (suggested name: **CONTROL_AREA_FLIGHTS**).

1. Edit the screen layout with the *Graphic Screen Painter* tool and create a control area below the pushbuttons.
 - a) Open the screen in change mode.
 - b) Start the *Graphic Screen Painter* tool.
 - c) In the toolbar on the left, choose the *Custom Control* button and then determine the position and size of the rectangular control area on the layout.
 - d) Enter a name for the new control area in the *Name* field at the top.
2. Maintain the attributes of the control area. Activate *Resizing* (horizontal and vertical) for the control area and enter the minimum size for the control area (suggestion: at least 5 rows and at least 25 columns).
 - a) Open the attribute window by double-clicking on the new control area.
 - b) Under *Resizing*, select the *vertical* and *horizontal* fields and enter a minimum size for rows and columns.

Continued on next page

Task 3:

Make sure that in addition to the detailed data for the flight connection,a list of flight times is also retrieved. Store this data with the global data for the program in the form of an internal table. Use the function module BC400_DDS_FLIGHTLIST_GET or the function module you developed yourself for data retrieval.

1. Create a global internal table in the Top include (suggested name: **gt_flights**), that you can use as an actual parameter for the function module call.
 - a) See the source code excerpt from the model solution.
2. Edit the PAI module in which the function code for the first screen (screen 100) is evaluated. Implement a call of the function module for reading the flight list if the existing function module call was successful.
 - a) The module you have to edit should be called **user_command_0100**.
See the source code excerpt from the model solution.

Task 4:

Create a new PBO module for the second screen (suggested name: **init_control_processing_0200**).In this module, generate an instance for the container and one for the grid control, and transfer the global internal table with the flight times to the instance of the grid control.

1. In the TOP include, define a reference variable for the class **CL_GUI_CUSTOM_CONTAINER** and one for the class **CL_GUI_ALV_GRID** (suggested names: **go_container** and **go_alv_grid**).
 - a) See the source code excerpt from the model solution.
2. Create a new PBO module for the second screen and implement the generation of the container and control instances in it. Supply only the mandatory parameters, that is **CUSTOMER_NAME** for the container instance and **I_PARENT** for the grid control instance.



Caution: The **CUSTOMER_NAME** parameter must be supplied with the **name** of the control area on the screen, that is, a character literal in quotation marks. Make sure you spell the name correctly using uppercase only.

Continued on next page

The parameter I_PARENT, on the other hand, must be supplied with a reference to the container instance. You cannot use quotation marks here.

- See the source code excerpt from the model solution.



Hint: There is a pattern for the CREATE OBJECT statement that you can address by choosing the *Pattern* pushbutton and the following path: *Pattern* → *Pattern for ABAP Objects* → *Create Object*.

- After the instance has been generated, call the instance method SET_TABLE_FOR_FIRST_DISPLAY for the instance of the grid control. Supply the parameter I_STRUCTURE_NAME with the name of the line type of the internal table, and the parameter IT_OUTTAB with the internal table itself.



Caution: The I_STRUCTURE_NAME parameter must be supplied with the **name** of the data type, that is, a character literal in quotation marks. Make sure you spell the name correctly using uppercase only.

The parameter IT_OUTTAB must be supplied with the **content** of the internal table. You cannot use quotation marks here.

- See the source code excerpt from the model solution.



Hint: There is a pattern for the CALL METHOD statement that you can address by choosing the *Pattern* pushbutton: *Pattern* → *Pattern for ABAP Objects* → *Call Method*.

- Ensure that the instances are created and the method called only when the screen is processed for the first time. Otherwise, call the instance method REFRESH_DISPLAY for the grid control instance.
 - See the source code excerpt from the model solution.
- Activate the program, create a dialog transaction, and test your application.
 - Carry out this step in the usual manner.

Result

Screen 100 flow logic:

```
PROCESS BEFORE OUTPUT.
```

Continued on next page

```

* MODULE STATUS_0100.
*
PROCESS AFTER INPUT.
MODULE USER_COMMAND_0100.
```

Screen 200 flow logic:

```

PROCESS BEFORE OUTPUT.
* MODULE STATUS_0200.
*
MODULE init_control_processing_0200.

PROCESS AFTER INPUT.
MODULE user_command_0200.
```

Source code excerpt from the model solution:

```

*&-----*
*& Include MBC400_UDS_DTOP           Modulpool      SAPMBC400_UDS_D
*&
*&-----*
PROGRAM sapmbc400_uds_d.

DATA ok_code TYPE syucomm.

TABLES bc400_s_dynconn.

DATA gs_connection TYPE bc400_s_connection.

DATA: go_container TYPE REF TO cl_gui_custom_container,
      go_alv_grid  TYPE REF TO cl_gui_alv_grid.

DATA gt_flights TYPE bc400_t_flights.

*-----*
***INCLUDE MBC400_UDS_DI01 .
*-----*
*&
*&     Module   USER_COMMAND_0100   INPUT
```

Continued on next page

```

*-----*
MODULE user_command_0100 INPUT.

CASE ok_code.

WHEN 'GO'.

MOVE-CORRESPONDING bc400_s_dynconn TO gs_connection.

CALL FUNCTION 'BC400_DDS_CONNECTION_GET'
  EXPORTING
    iv_carrid      = gs_connection-carrid
    iv_connid      = gs_connection-connid
  IMPORTING
    es_connection = gs_connection
  EXCEPTIONS
    no_data        = 1
    OTHERS         = 2.

IF sy-subrc <> 0.
  CLEAR bc400_s_dynconn.

MESSAGE i042(bc400) WITH gs_connection-carrid
                     gs_connection-connid.
ELSE.
  MOVE-CORRESPONDING gs_connection TO bc400_s_dynconn.

  CALL FUNCTION 'BC400_DDS_FLIGHTLIST_GET'
    EXPORTING
      iv_carrid  = gs_connection-carrid
      iv_connid  = gs_connection-connid
    IMPORTING
      et_flights = gt_flights
    EXCEPTIONS
      no_data    = 1
      OTHERS     = 2.

  IF sy-subrc <> 0.
    CLEAR gt_flights.
  ENDIF.

  SET SCREEN 200.

```

Continued on next page

```
ENDIF.

ENDCASE.

ENDMODULE.          " USER_COMMAND_0100  INPUT

*-----*
***INCLUDE MBC400_UDS_D001 .
*-----*
*&-----*
*&      Module  init_control_processing_0200  OUTPUT
*&-----*
MODULE init_control_processing_0200 OUTPUT.

IF go_container IS INITIAL.

CREATE OBJECT go_container
EXPORTING
  container_name      = 'CONTROL_AREA_FLIGHTS'.

CREATE OBJECT go_alv_grid
EXPORTING
  i_parent            = go_container.

CALL METHOD go_alv_grid->set_table_for_first_display
EXPORTING
  i_structure_name    = 'BC400_S_FLIGHT'
CHANGING
  it_outtab           = gt_flights.

ELSE.
  CALL METHOD go_alv_grid->refresh_table_display.
ENDIF.

ENDMODULE.          " init_control_processing_0200  OUTPUT
```



Lesson Summary

You should now be able to:

- Use the SAP Grid Control (*SAP List Viewer*) to display an internal table on a screen



Unit Summary

You should now be able to:

- List attributes and benefits of screens
- Implement simple screens with input/output fields and link them to a dialog application
- Explain and implement the program-internal processing for screen calls
- List the properties and usage scenarios of the ABAP Web Dynpro
- Explain the programming and runtime architecture of the ABAP Web Dynpro
- Implement simple Web Dynpro applications with input/output fields and pushbuttons
- List the properties and benefits of selection screens
- Implement the options for restricting selections on the selection screen
- Describe the attributes and benefits of ABAP lists
- Implement list and column headers
- Implement multilingual lists
- Describe the event-controlled processing of an executable ABAP program
- List the most important basic events and explain their purpose
- Use the SAP Grid Control (*SAP List Viewer*) to display an internal table on a screen

Related Information

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



Test Your Knowledge

1. Which of the following statements concerning the dynpro are correct?
Choose the correct answer(s).
 - A The static next screen of a screen cannot be overridden dynamically.
 - B A screen has a flow control setup, in addition to the attributes, layout, and element list.
 - C You will find ABAP statements in the flow control of a screen.
 - D In the dynpro flow control, you will find references to modules that are contained in the program and include ABAP statements.
 - E The automatic data transport between internal program variables and dynpro fields takes place only with fields of the same name.

2. Which is the most important attribute of a pushbutton?

3. What is the name of the statement that overrides the static next screen of the current screen?

4. What is the main advantage of using text symbols instead of literals?

5. What syntax do you use to output a text symbol named abc in a list?

Choose the correct answer(s).

- A WRITE abc.
- B WRITE 'abc' .
- C WRITE 'text-abc' .
- D WRITE text-abc .

6. Which of the following statements concerning text elements in a program are correct?

Choose the correct answer(s).

- A As well as text symbols, text elements also comprise list and column headers.
- B Text elements are stored with reference to a program.
- C Not all text elements can be translated.
- D Text elements in a program have to be activated before they can be used.
- E Maintaining a list header also changes the short description of the program.

7. What is the purpose of the selection screen?

8. Which of the following statements concerning selection texts are correct?

Choose the correct answer(s).

- A No selection texts can be maintained for select options.
- B Just as list headers can be maintained directly in the list, selection texts can also be maintained directly on the selection screen.
- C Selection texts cannot be translated.
- D Selection texts belong to the text elements of a program, just like text symbols and list headers, and can therefore also be translated.

9. Which of the following statements concerning selection screens are correct?

Choose the correct answer(s).

- A All the input fields of a selection screen have an F1 help available.
- B All the input fields of a selection screen have an F4 help available.
- C Typically, authorization checks are executed at the AT SELECTION-SCREEN event.
- D A type E user message triggers the display of the message text with program termination if it is sent at the AT SELECTION-SCREEN event.

10. Which of the following statements concerning event processing blocks are correct?

Choose the correct answer(s).

- A If a program has no explicit event processing blocks, all the statements belong implicitly to the AT-SELECTION-SCREEN block.
- B The sequence of event processing blocks in the program has no effect on the program flow.
- C Event processing blocks can be nested inside one another.
- D Event processing blocks must each be closed explicitly.

11. Why is the INITIALIZATION event suitable for making (dynamic) default settings for entry fields on the selection screen?

12. What is the name of the event for the main processing of an executable program?

13. Which of the following statements concerning classes/methods are correct?

Choose the correct answer(s).

- A In the SAP standard version, classes that contain useful functions are already available.
- B You can search for classes using the standard search tools "Repository Information System" and "SAP Application Hierarchy".
- C You can use a standard class to control the SAP Grid Control (ALV).
- D To call up other EnjoySAP controls (for example, tree control), you must define your own specific classes.



Answers

1. Which of the following statements concerning the dynpro are correct?

Answer: B, D, E

2. Which is the most important attribute of a pushbutton?

Answer: Its function code, which is transmitted to the program when the pushbutton is operated.

3. What is the name of the statement that overrides the static next screen of the current screen?

Answer: SET SCREEN

4. What is the main advantage of using text symbols instead of literals?

Answer: Text symbols can be translated into different languages.

5. What syntax do you use to output a text symbol named abc in a list?

Answer: D

6. Which of the following statements concerning text elements in a program are correct?

Answer: A, B, D

7. What is the purpose of the selection screen?

Answer: To enter criteria for data selection

8. Which of the following statements concerning selection texts are correct?

Answer: D

Selection texts cannot be maintained directly on the selection screen. They belong to the text elements of the program and are maintained there. You call up these texts in the *Object Navigator* through the context menu of the respective program or through the menu *Goto* in the Editor.

9. Which of the following statements concerning selection screens are correct?

Answer: C

10. Which of the following statements concerning event processing blocks are correct?

Answer: B

The implicit event processing block is called START-OF-SELECTION.

11. Why is the INITIALIZATION event suitable for making (dynamic) default settings for entry fields on the selection screen?

Answer: Because this event is processed before the selection screen is displayed. When the selection screen is then displayed, the corresponding variable contents are transported as default entry values to the selection screen.

12. What is the name of the event for the main processing of an executable program?

Answer: START-OF-SELECTION

13. Which of the following statements concerning classes/methods are correct?

Answer: A, B, C

Unit 8

Tools for Program Analysis

Unit Overview

The unit overview lists the individual lessons that make up this unit.



Unit Objectives

After completing this unit, you will be able to:

- Explain the purpose and use of the *Code Inspector*
- Describe the most important attributes of the *Code Inspector*
- Use the *Code Inspector* for simple analysis of your programs

Unit Contents

Lesson: The Code Inspector	500
----------------------------------	-----

Lesson: The Code Inspector

Lesson Overview

In this lesson, you will learn about the basic function and purpose of the *Code Inspector*. This is only a brief introduction to the tool. You will find more information in the relevant documentation or by attending the advanced courses.



Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose and use of the *Code Inspector*
- Describe the most important attributes of the *Code Inspector*
- Use the *Code Inspector* for simple analysis of your programs

Business Example

You are to check your programs for performance, typical semantic programming errors, and security gaps.

Introduction

The *Code Inspector* offers you the option of analyzing your programs with regard to performance, security, and typical semantic errors. In order to explain these three aspects in more detail, the section below lists some check criteria for each:

Performance

- Are indexes used for database access?
- Are SELECT statements embedded in loops?

Security

- Is data read from a client other than the login client?
- Is the database table or the WHERE clause dynamically specified in the SELECT statement?

Typical semantic errors

- Is the sy-subrc field checked after each AUTHORITY-CHECK statement?
- Is a client actually specified for CLIENT SPECIFIED?
- Are several messages of type E (E messages) sent in direct succession?

Calling the Code Inspector

There are several different ways of calling the *Code Inspector*.

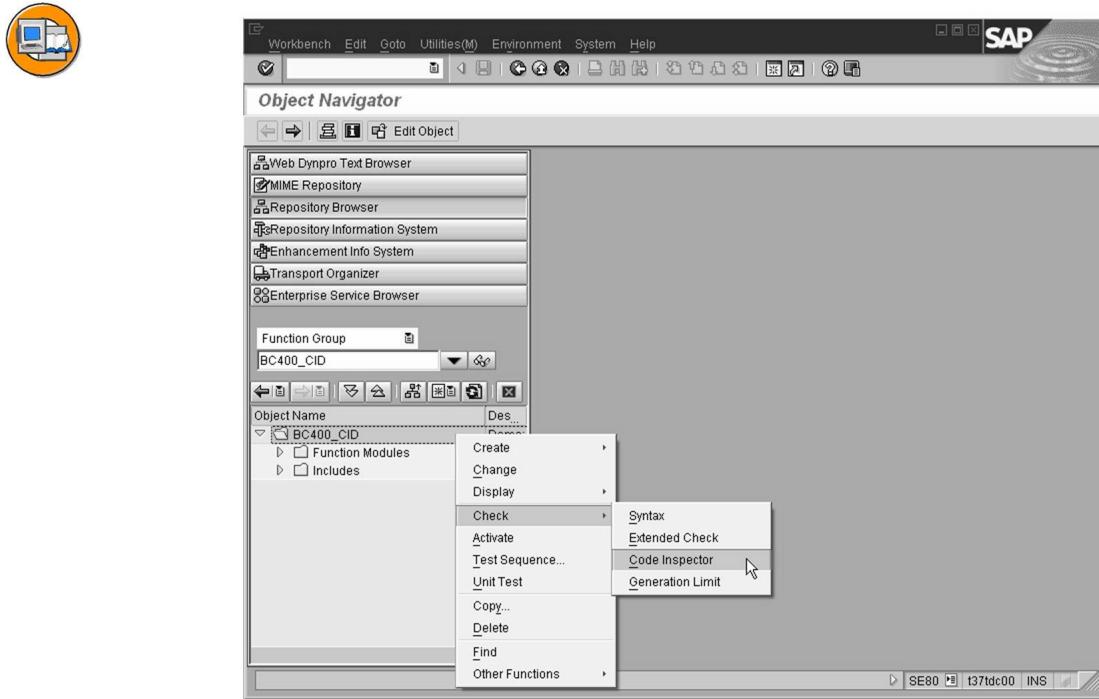


Figure 245: Calling the Code Inspector

The graphic shows one possibility. Call the *Code Inspector* from the context menu of the object list by choosing *Check* → *Code Inspector* and carry out a standard inspection. You will get the same call if you choose the menu path *Function Module* → *Check* → *Code Inspector*.

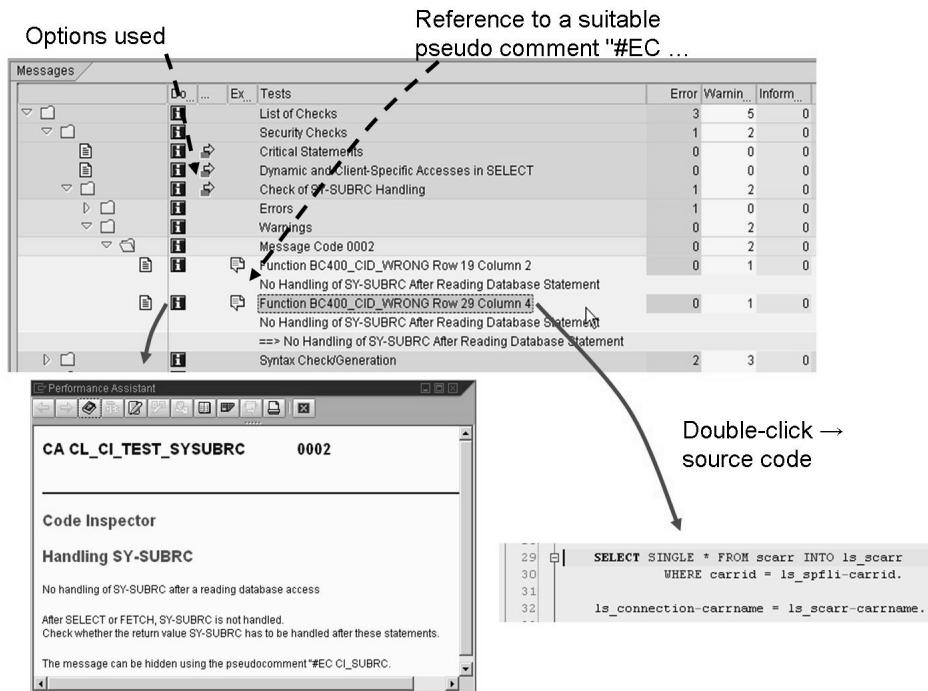


Figure 246: Inspection Result

As the result of an inspection you receive a list of error and warning messages. The **I** button that belongs to the message shows a detailed error description as well as suggestions for improvements. By double-clicking on the error text, however, you branch to the corresponding program statement.

Sometimes, it is practical, or even essential to write ABAP source code that the *Code Inspector* would query, such as a transaction call. As is the case with the extended syntax check too, you can provide a pseudo comment "#EC * for such commands.

With a standard check called from the Editor, the *Code Inspector* uses a default check variant in which the checks to be performed are predefined.

You can overwrite this default check variant by creating a new check variant with the name **DEFAULT**. Note, however, that this new check variant will override the standard variant for your user (in all clients).

If you delete your default check variant, the standard variant is used automatically for future checks. For this reason, you should always define your own check variants instead of overwriting the default variant.

Defining Check Variants, Object Sets and Inspections

To define individual checks, start the *Code Inspector* with transaction code SCI, or from the *SAP Easy Access* menu by choosing *Tools* → *ABAP Workbench* → *Test* → *Code Inspector*. The initial screen contains three areas:

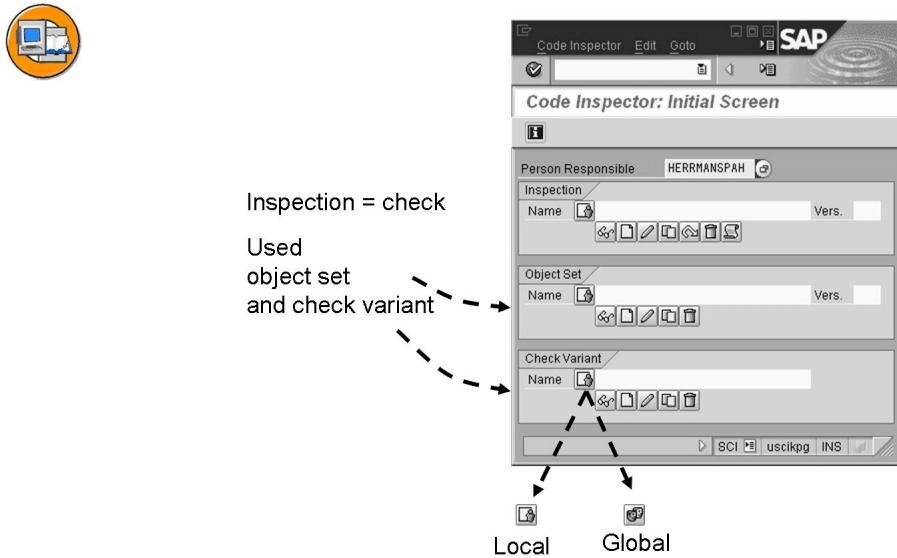


Figure 247: Advanced Code Inspector Check

Check Variant

Determines which checks are to be carried out, for example, the programming conventions or performance check.

object set

Determines which (Repository) objects should be checked.

inspection

Name of the actual check. If you want to save the inspection and its results, or execute it in the background, you have to give it a name. If you do not enter a name for the inspection, the results are not saved.

Normally, an inspection uses a previously defined check variant and object set. However, it is also possible to create temporary object sets and check variants that are only valid for one inspection, and that cannot be re-used in other inspections.



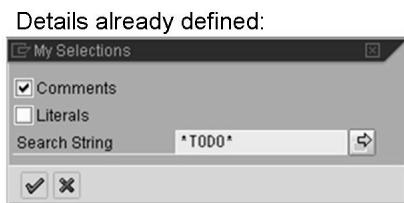
Hint: You can create check variants, object sets, and inspections either as **private** or **public**. To switch between these two categories you can use the pushbutton that is always placed in front of the input field. Note that only you can use your private objects whereas public objects are available to all users of the system.

A check variant consists of one or more check categories, which in turn consist of one or more single checks (inspections). These single checks can be parameterized, for example, with a keyword or an indicator for a specific partial aspect of the check. In most cases, single checks only investigate a specific object type – such as the "Check of Table Attributes", which only examines DDIC tables.



Selection	Docu...	Attributes	Tests
	i		List of Checks
	i		General Checks
	i		Performance Checks
	i		Security Checks
	i		Search Functs.
	i		Search of ABAP Tokens
	i		Search ABAP Statement Patterns
	i		Find Unwanted Language Elements

Details still missing



Details already defined:

Figure 248: Check Variant

The single checks are assigned to various check categories. The most important check categories are described below:

- *General Checks* contain data formatting, such as the list of table names from SELECT statements
- *Performance Checks* contain checks for performance and resource usage, such as
 - analysis of the WHERE condition for SELECT / UPDATE and DELETE
 - SELECT statements that bypass the table buffer
 - low-performance access to internal tables
- *Security Checks* contain checks of critical statements, cross-client queries, and insufficient authority checks
- *Syntax Check/Generation* contain the ABAP syntax check, an extended program check, and generation
- *Programming Conventions* contain checks of naming conventions
- *Search Functions* contain searches for tokens (words) and statements in ABAP source code



Lesson Summary

You should now be able to:

- Explain the purpose and use of the *Code Inspector*
- Describe the most important attributes of the *Code Inspector*
- Use the *Code Inspector* for simple analysis of your programs

Related Information

- For more information, refer to the online documentation for the tools of the *ABAP Workbench*.



Unit Summary

You should now be able to:

- Explain the purpose and use of the *Code Inspector*
- Describe the most important attributes of the *Code Inspector*
- Use the *Code Inspector* for simple analysis of your programs

Related Information

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



Test Your Knowledge

1. Which of the following statements on the *Code Inspector* are true?
Choose the correct answer(s).
 - A When a Code Inspector check is carried out, a check variant must always be specified.
 - B To define the level of detail for a check, you can create an appropriate check variant and specify it for the inspection.
 - C The execution of a *Code Inspector* check through the (context) menu of the *Object Navigator* uses a standard check variant.
 - D Check variants, object sets, and inspections can be defined using transaction SCI.

2. Which aspects are taken into consideration by the Code Inspector when a program is examined?
Choose the correct answer(s).
 - A Syntax check
 - B Typical semantic errors (for example, AUTHORITY-CHECK statement without subsequent SY-SUBRC check)
 - C Performance (for example, nested SELECT statements)
 - D Security (for example, cross-client data accesses)
 - E Formatting of the source code (for example, indenting of the program lines within loops)



Answers

1. Which of the following statements on the *Code Inspector* are true?

Answer: B, C, D

2. Which aspects are taken into consideration by the Code Inspector when a program is examined?

Answer: B, C, D

Unit 9

Adjusting the SAP Standard Software (Overview)

Unit Overview

The unit overview lists the individual lessons that make up this unit.



Unit Objectives

After completing this unit, you will be able to:

- Explain the terms **original, copy, correction, repair, customizing, modification, and enhancement**
- Describe which options are available for adjusting the SAP standard software to your company's requirements
- List the disadvantages of modifications and the advantages of SAP enhancements
- List and explain the different enhancement types

Unit Contents

Lesson: Adjusting the SAP Standard Software (Overview) 512

Lesson: Adjusting the SAP Standard Software (Overview)

Lesson Overview

This lesson provides an overview of the options for adjusting the SAP standard software to your company's requirements.



Lesson Objectives

After completing this lesson, you will be able to:

- Explain the terms **original, copy, correction, repair, customizing, modification, and enhancement**
- Describe which options are available for adjusting the SAP standard software to your company's requirements
- List the disadvantages of modifications and the advantages of SAP enhancements
- List and explain the different enhancement types

Business Example

You want to make adjustments to the SAP standard but, in doing so, you want to avoid modifications because of their disadvantages.

Basic Terms and Options of SAP Software Adjustment

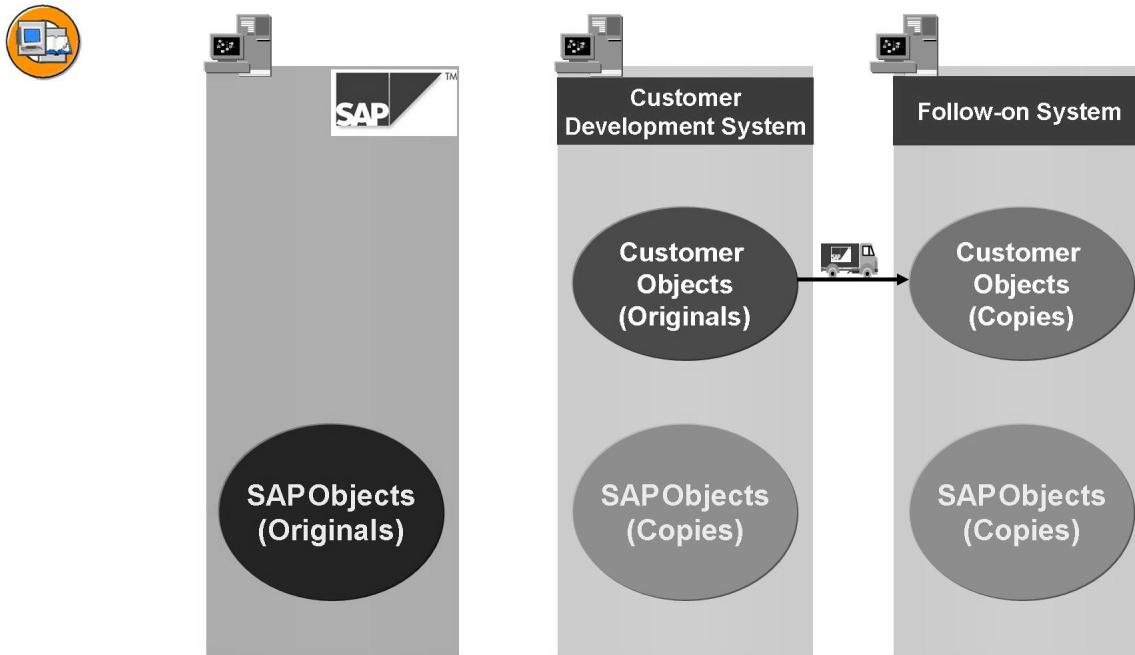


Figure 249: Originals and Copies

In the system in which it was developed, a Repository object is called **original**. In another system, in which the object arrives by means of transport, it only exists as a **copy**. The original system (the “birth place”) of a Repository object is noted in the attributes of the object.

Usually, changes are only to be made to the original. These changes are then transported to subsequent systems to change the corresponding copies. This ensures that Repository objects are consistent in all systems. It is possible to change copies, but you should avoid that in order to avoid the systems becoming inconsistent if the changes are not made to the respective original.

Originals can never be overwritten in transports.

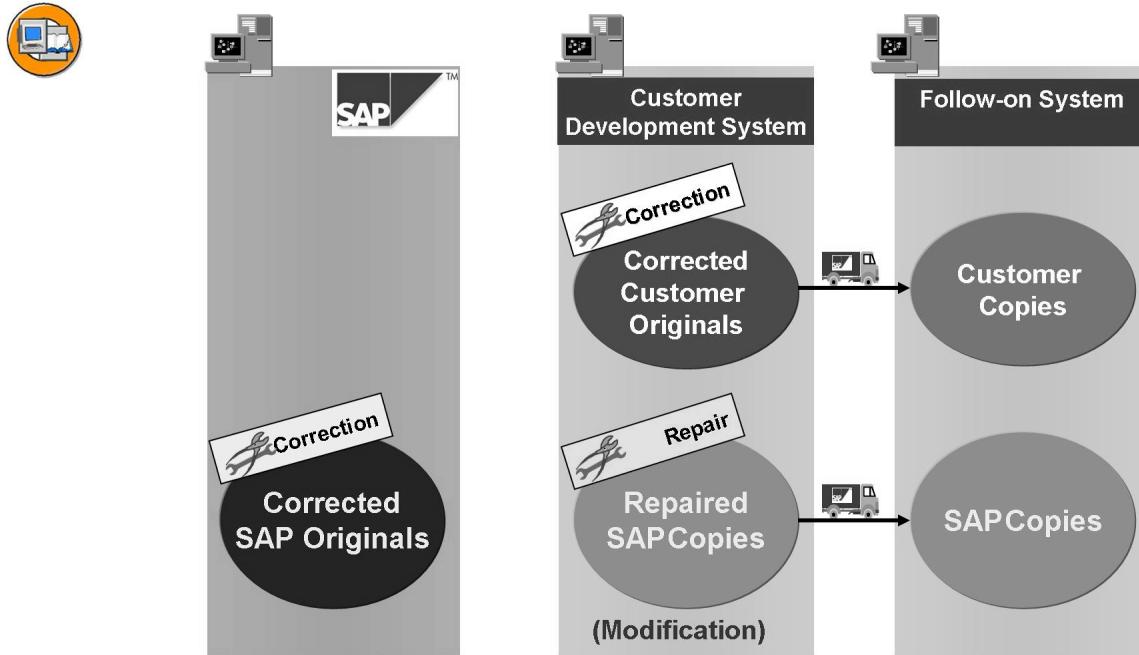


Figure 250: Corrections and Repairs

Changing an original is called a **correction**. Corrections are made in *development/correction* type tasks.

In contrast, changes to a copy are called **repair**. Repairs are made in *repair* type tasks.

After a customer object has been repaired (for example, as a consequence of an emergency in the production system), the change should always be reproduced in the original to ensure the systems remain consistent.

Repairing an SAP object in a customer system is also called **modification**.

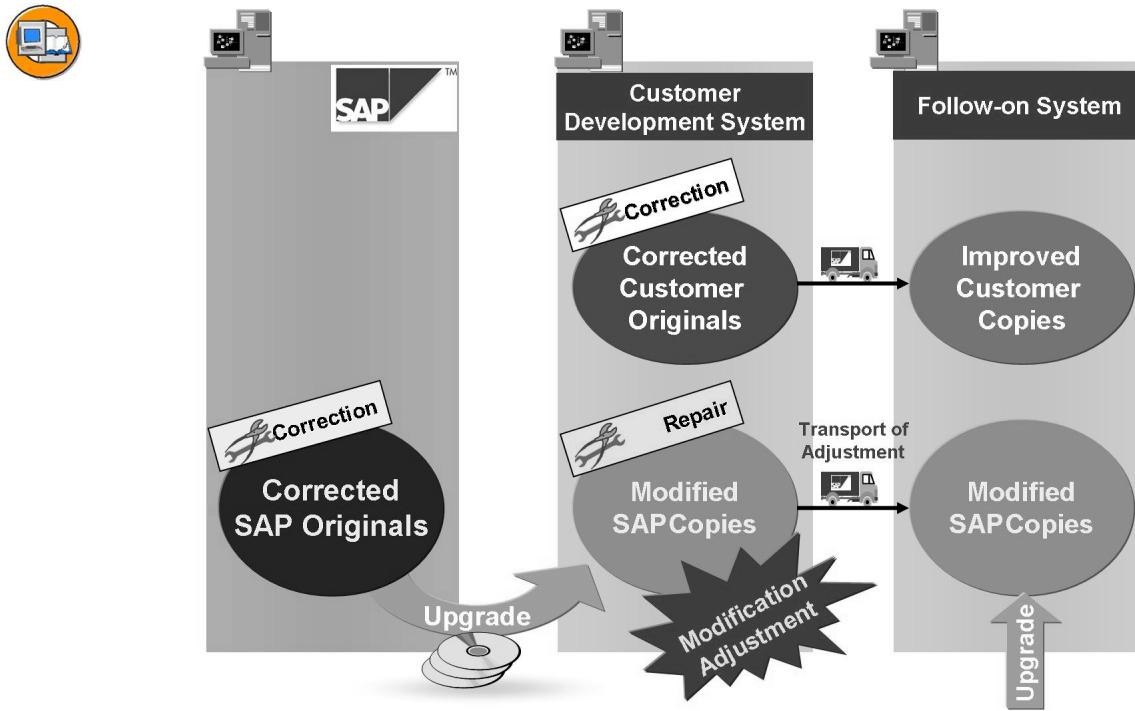


Figure 251: Modifications During Upgrade

During an upgrade, the modified SAP objects in your system might be overwritten by new versions of the objects that are supplied by SAP. You need to make a **modification adjustment** to get your modification back into the system. When you do this, you must compare your old, modified version with the newly supplied version and copy earlier changes to the SAP standard into the new version.

Such a comparison can be very **time-consuming**. We therefore strongly recommend you use an **SAP enhancement** instead of a modification. Enhancements also provide the option of adjusting the SAP functionality. However, they **do not depend on the release** and are thus “low maintenance”. You will be provided with an overview of existing enhancement types later on in this lesson.

We recommend only making the **modification adjustment in the development system** and then transporting the adjusted objects to subsequent systems (by releasing the request you used). This ensures consistency between the development system and the subsequent systems.

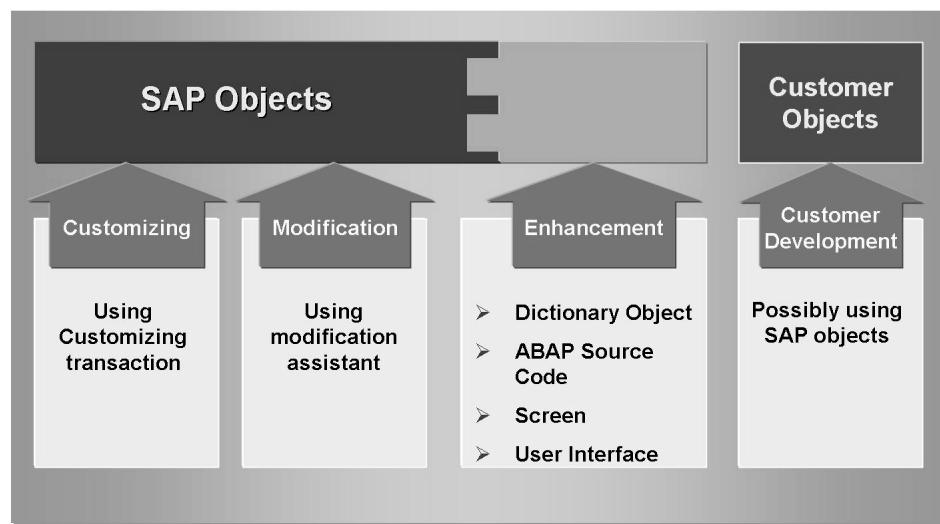


Figure 252: Options of SAP Software Adjustment

The illustration above shows the four options for adjusting the SAP software to customer requirements:

- **Customer developments**

You can develop your own Repository objects under consideration of the customer name space. This may be necessary when there is not a similar functionality to the functionality you require available in the SAP standard.

- **Customizing**

You can set certain system properties and functionalities by means of appropriate maintenance transactions. SAP has planned for and organized such adjustments. Customizing is a mandatory part of setting up an SAP system.

- **Enhancements**

SAP Repository objects can be adjusted without modifications by means of enhancements. Enhancements are **release-independent** and do not require **any adjustments**. However, not all customer requirements are covered by enhancement options.

- **Modifications**

Modifications should only be made when there is no appropriate enhancement option in the system as they might cause a significantly increased workload due to the adjustments that might be required. To support an organized modification and to facilitate the subsequent adjustment, the Modification Assistant has been integrated in the system since SAP R/3 4.6b. For detailed information on the Modification Assistant, refer to the online documentation for the *ABAP Workbench (changes to the SAP standard)*.

The following section explains the different types of enhancements.

Enhancement Types

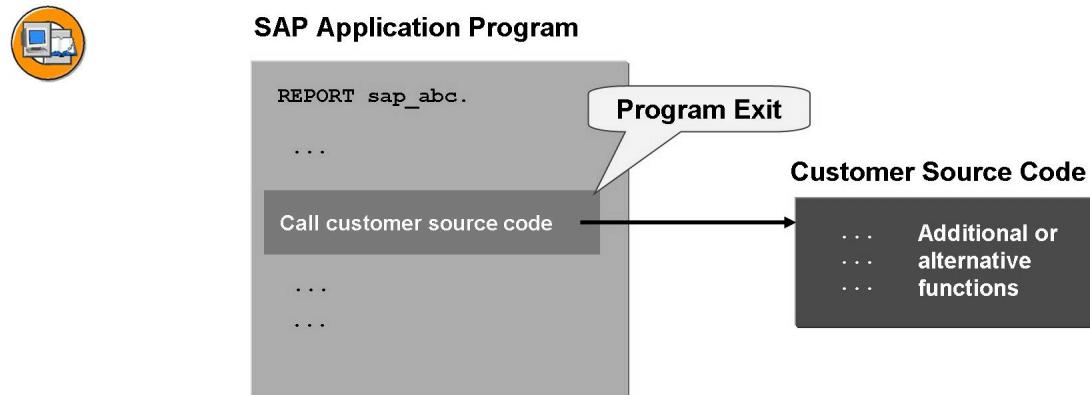


Figure 253: The Concept of a Program Exit

SAP has implemented **program exits** in some SAP programs, to which you as the customer can attach the appropriate source code. This is also processed at the runtime of the SAP program. This way, you have the option of realizing additional or alternative functions **without changing the SAP program**. Program exits are the most important type of enhancement.

SAP uses various techniques to implement program exists: User Exits, Customer Exits, Business Transaction Events (BTEs), and Business Add-Ins (BAdIs). There is a special search function for each enhancement technique, which you can use to find the program exits prepared by SAP. You can find out more about this in course BC425.



Screen of an SAP Program

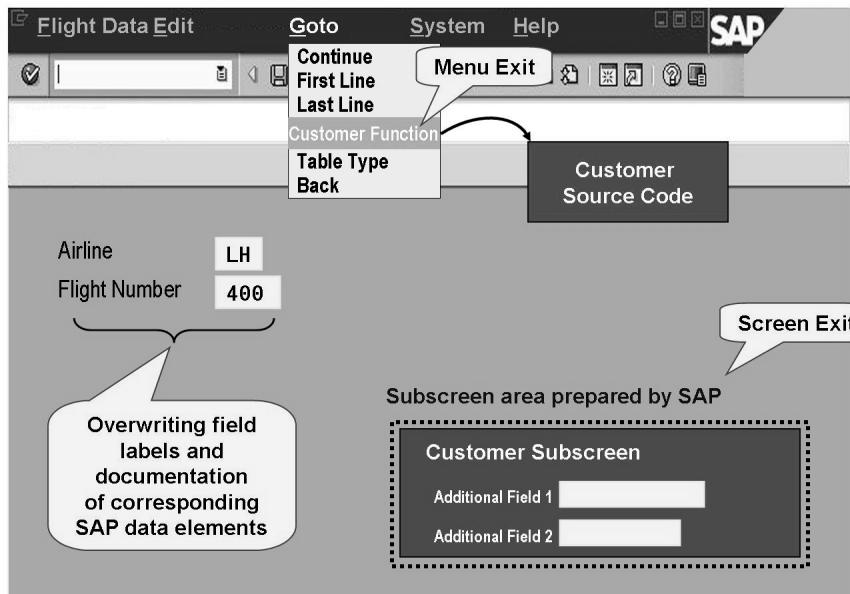


Figure 254: Other Enhancement Types

In addition to program exits, the following enhancement types exist:

Menu exit

Some SAP menus contain entries prepared by SAP, which you can link to your own functionality (customer source code) and activate.

Screen exit

Some screens contain SAP subscreens areas in which you can integrate your own screens.

Furthermore you can overwrite the field documentation and field labels of an SAP data element in the *ABAP Dictionary*:

Overwriting the field documentation

You can overlay the SAP field documentation that is displayed when you press **F1** with your own texts.

Overwriting the field labels

You can overwrite the different field labels of an SAP data element with your own texts.

Apart from the enhancement type already mentioned, there are also others, such as:



Append structure

You can attach an append structure with the required additional fields to most transparent table. With that you can implement additional columns in the database table of SAP applications without modifications.

Source texts areas commented out by SAP

Some SAP programs contain source code areas that have been commented out. If recommended by SAP, you can remove the comments to obtain additional functionality. Technically, however, this is a modification.

The courses BC425 and BC427 provide detailed information on all the enhancement types.

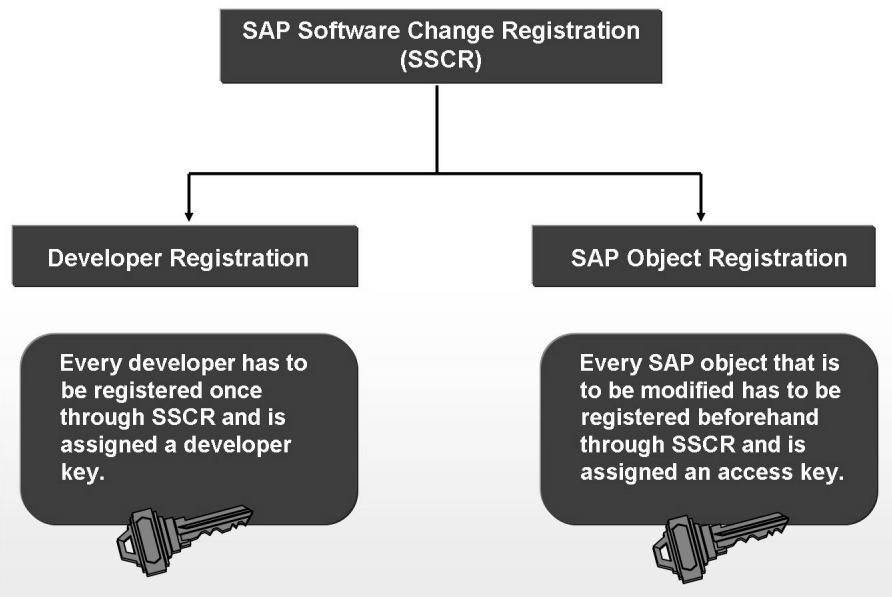


Figure 255: SAP Software Change Registrations

As soon as a developer creates or changes a Repository object for the first time, the system will prompt him for his developer key. He can obtain this key by means of a corresponding one-time SSCR developer registration. The developer key is linked to the developer's user ID and the license number of the SAP system.

You need an object-related access key (object key) for every SAP Repository object that is to be **modified**. You can get this by means of SSCR object registration. When you register, you must specify the Repository object name, the object type, and the SAP system license number and its release.



Lesson Summary

You should now be able to:

- Explain the terms **original, copy, correction, repair, customizing, modification, and enhancement**
- Describe which options are available for adjusting the SAP standard software to your company's requirements
- List the disadvantages of modifications and the advantages of SAP enhancements
- List and explain the different enhancement types



Unit Summary

You should now be able to:

- Explain the terms **original, copy, correction, repair, customizing, modification, and enhancement**
- Describe which options are available for adjusting the SAP standard software to your company's requirements
- List the disadvantages of modifications and the advantages of SAP enhancements
- List and explain the different enhancement types

Related Information

... Refer to the articles “Transport Organizer ” and “Changing the SAP Standard” in the online documentation.



Test Your Knowledge

1. What are modifications?

2. Why are enhancements, not modifications, preferred for customer-specific adjustments to the SAP software?



Answers

1. What are modifications?

Answer: A direct change to the SAP software

2. Why are enhancements, not modifications, preferred for customer-specific adjustments to the SAP software?

Answer: Enhancement do not cause any adjustment effort whenever there is a change in a release.



Course Summary

You should now be able to:

- Create an ABAP program containing user dialogs and database accesses
- Describe the different types of development objects and their typical intended purposes
- Use appropriate tools to create simple examples of the development objects presented

Related Information

... Refer to the online documentation for all the concepts, keywords and tools addressed.

Index

A

ABAP documentation, 42
ABAP syntax, 39
ABAP Web Dynpro, 399
ABAP Web Dynpro application,
 403
activate, 45
active, 44
actual parameter, 123
APPEND, 240
Application Component, 22
application hierarchy, 24
Application server level, 3
AT SELECTION-SCREEN,
 452
AUTHORITY-CHECK, 326
authorization, 326
Authorization Check, 324
authorization object, 326
Authorization profile, 326

B

BAPI, 155, 285
 standard BAPIs, 156
BAPI Explorer, 157
basic event, 448
branches
 conditional, 82
breakpoint, 88, 91
business object, 155
Business Object Repository,
 155

C

Calculation, 80
call by reference, 124

call by value, 124
call by value and result, 124
CALL FUNCTION, 149
CALL METHOD, 191
 CHANGING, 210
 EXPORTING, 210
 IMPORTING, 210
CASE, 82
chained statement, 41
Change request, 32
check variant, 503
CLASS, 209
Class Builder, 474
CLEAR, 79, 247
Code Inspector, 500
COLLECT, 241
column header, 439
comment, 41
component, 402
COMPUTE, 80
Constant, 75
CONSTANTS, 75
 VALUE addition, 75
Constructor, 191
CONSTRUCTOR, 191
container control, 472
context, 404
context menu, 26
Conversion rule, 79
Copy, 513
Correction, 514
CREATE OBJECT, 191
Customizing, 517

D

DATA, 73, 222, 238

- BEGIN-OF-addition, 222
- LIKE addition, 73
- TYPE REF TO, 190
- VALUE addition, 74
- data encapsulation, 116
- data model, 267
- data object
 - fixed, 75
 - global, 129
 - local, 129
 - structured, 221
- Data object, 68
- data transport, 117
- data type
 - global, 71
 - local, 71
- Data Type, 68
- Database, 267
- database level, 3
- database management system
 - relational, 3
- database table, 271
- Debugging mode, 88, 224
- DEFAULT, 210
- DELETE, 241–242
- Dialog message, 87
- DO, 84
- documentation
 - ABAP, 42
 - keyword, 42
- E**
- "#EC, 502
- encapsulation, 116
- enhancement
 - append structure, 520
 - field documentation, 519
 - field label, 519
 - menu exit, 519
 - program exit, 518
 - screen exit, 519
- Enhancement Concept, 517
- EnjoySAP controls, 470
- entity, 267
- entity relationship model, 267
- ERM, 267
- event, 448
- exception, 151
- expression
 - arithmetical, 80
- F**
- flight data model, 268
- FORM, 126
 - CHANGING addition, 126
 - USING addition, 126
- formal parameter, 123
 - typing, 127
- FREE, 247
- function
 - predefined, 80
 - STRLEN, 80
- Function code, 356
- function group, 143
- function module, 143–144
- Function module, 285
- G**
- generate, 46
- global class
 - test, 186
- global variable, 123
- H**
- hashed table, 236
- header, 248
- I**
- IF, 82
- inactive, 44
- Index access, 235
- INSERT, 241, 243
- inspection, 503
- interface
 - changing parameter, 144
 - exception, 144
 - export parameter, 144

- import parameter, 144
- method, 210
- of a subroutine, 123
- subroutine, 123
- internal table
 - deleting contents, 246–247
 - header row, 248
 - index access, 244
 - inserting rows, 243
 - key, 234
 - key access, 245
 - local data type, 238
 - properties, 234
 - row type, 234
 - row types, 233
 - row-by-row output, 243
 - sorting entries, 246
 - table kind, 234
- K**
 - key field, 270
 - keyword documentation, 42
- L**
 - list buffer, 448
 - list header, 439
 - Literal, 75
 - logical database, 285
 - Loop, 84
 - LOOP, 242–243
 - INDEX addition, 244
 - INTO clause, 242
 - WHERE addition, 245
- M**
 - memory allocation, 448
 - Message, 87
 - MESSAGE, 87
 - WITH addition, 87
 - method
 - parameter, 210
 - signature, 210
 - Model View Controller, 400
 - modification, 514, 517
- adjustment, 515
- Modification Assistant, 517
- MODIFY, 241, 243
- modularization
 - global modularization, 116
 - local modularization, 115
 - modularization unit, 114
- Module, 360
- MOVE, 79
- MOVE-CORRESPONDING, 223
- MVC programming model, 400
- N**
 - Native SQL, 283
- O**
 - Object class, 326
 - Object Navigator, 26
 - Open SQL, 283
 - OPTIONAL, 210
 - Original, 513
- P**
 - package, 22, 33
 - PAI Module, 352
 - parameter, 117
 - subroutine, 123
 - PARAMETERS, 445
 - PBO module, 352
 - performance, 505
 - Presentation server level, 3
 - Pretty Printer, 40
 - Primary index, 294
 - primary key, 270
 - processing block, 448
 - program
 - activate, 45
 - generate, 46
 - pseudo comment, 502
- R**
 - RDBMS, 3
 - READ TABLE, 241

- INDEX addition, 244
- WITH addition, 245
- Record, 221
- REFRESH, 246
- registration, 520
- Repair, 514
- Repository, 21
- Repository Information System, 22
- Repository object
 - create object, 29
- RETURNING, 210
- Reuse Components, 284
- runtime object, 46
- runtime system, 448
- S**
- SAP enhancements
 - Enhancement, 515
- SAP Grid Control, 471
- scalability, 3
- screen, 348
 - call, 349
 - component, 351
 - create, 355
 - data transport, 369
 - function code field, 356
 - layout, 367
 - module, 360
 - OK code field, 356
 - pushbutton, 354
 - TABLES, 370
- Screen processing, 359
- Secondary index, 295
- SELECT, 286
- SELECT-OPTIONS, 446
- selection screen, 440
- selection texts, 444
- SET TA-
 - BLE_FOR_FIRST_DIS-
PLAY, 475
- shadow rule, 129
- Signature, 210
- SORT, 246
 - AS TEXT addition, 246
 - STABLE addition, 246
- sorted table, 236
- SQL, 283
- SSCR
 - registration, 520
- Standard table, 236
- structure
 - data object, 221
- Structure component, 222
- sy-tabix, 243
- system field, 85
 - SY-DBCNT, 290–291
 - SY-SUBRC, 86
- System field
 - SY-INDEX, 84
- T**
- table
 - internal, 233
- Table join, 297
- table key, 270
- Table type, 237
- TABLES, 370
- Task, 32
- text element
 - list headers, 439
- text elements
 - selection texts, 444
 - text symbols, 76
- text pool, 76
- text symbols, 76
- tokens, 505
- transparent table, 270
- transport layer, 35
- Transport Organizer, 32
- Type conflict, 79
- Type conversion, 79
- TYPES, 71, 221
 - BEGIN-OF addition, 221

V

variable
 reference, 190
Variable, 68
version
 active, 44
 inactive, 44
view, 401

W

watchpoint, 92
WHILE, 84
wndow, 402
Work area, 240
worklist, 45

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.