# BC400 ABAP Workbench Concepts and Tools

BC400

Release 46C 12.12.2002


- R/3 System
- Release 4.6C
- Material Number 5004 2281
- Januar 2001

# Copyright

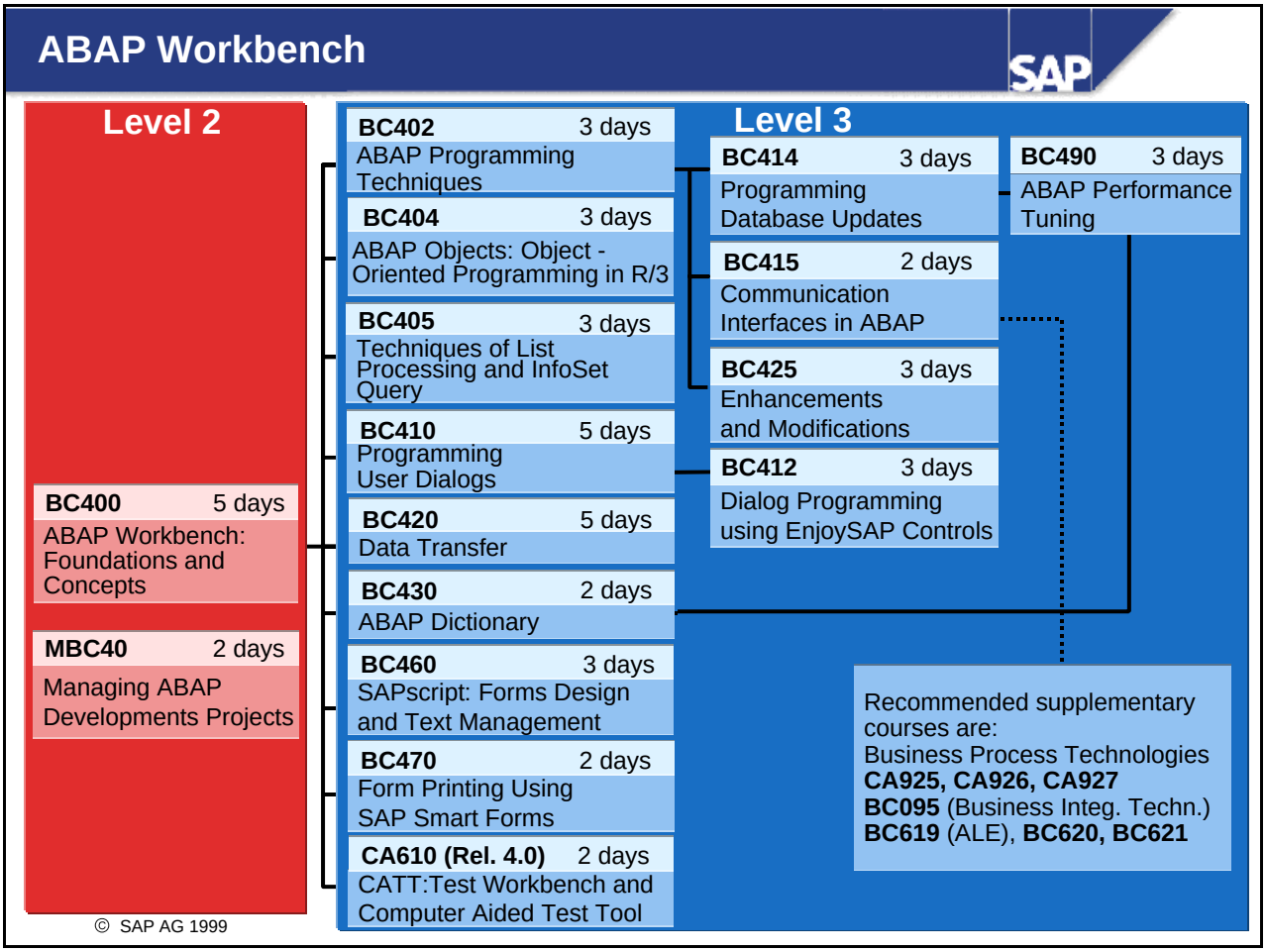


**Trademarks:**

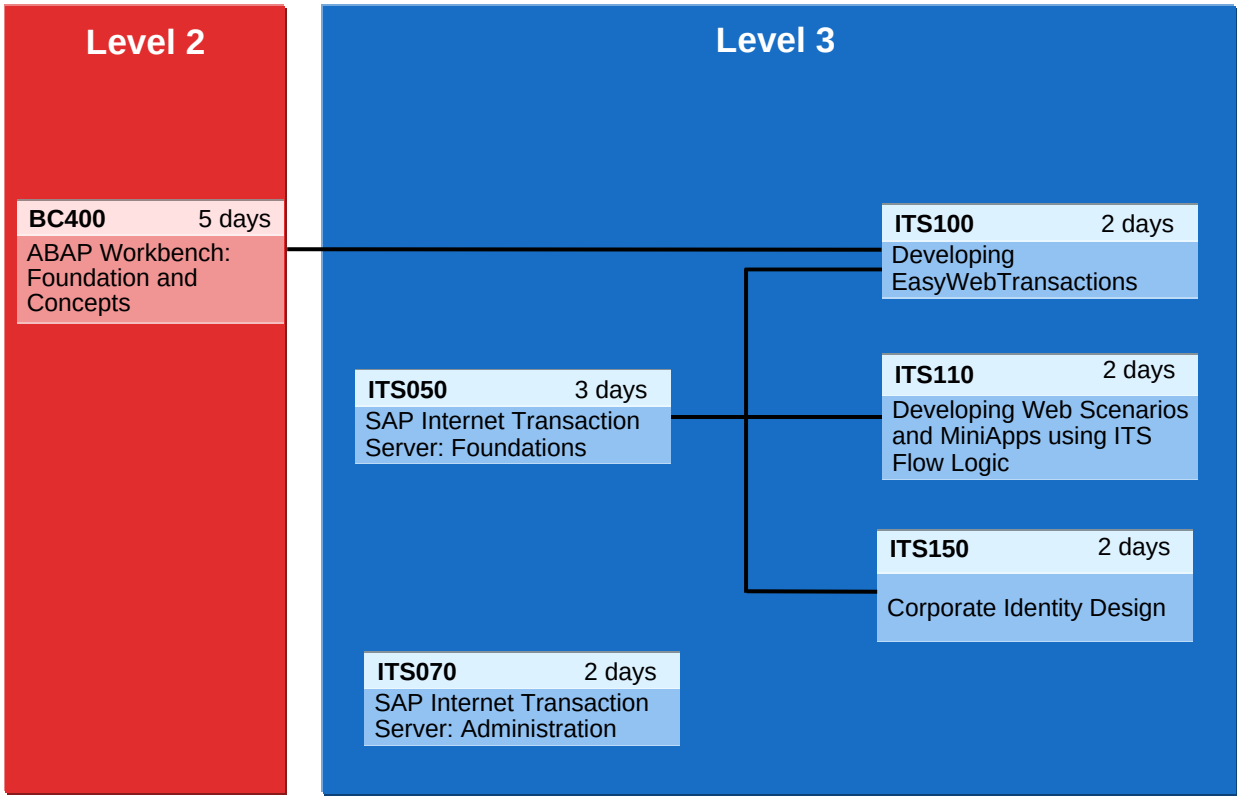
- Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.
- Microsoft®, WINDOWS®, NT®, EXCEL®, Word®, PowerPoint® and SQL Server® are registered trademarks of Microsoft Corporation.
- IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, 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 Server™ are registered trademarks of Informix Software Incorporated.
- UNIX®, X/Open®, OSF/1®, and Motif® are registered trademarks of the Open Group.
- 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, ABAP, 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.

# ABAP Workbench



# ITS



# Prerequisites

- **Programming knowledge**
- **Training course SAP 50: Basis Technology**

# Target Group

- **Participants:**
  - **Project members**
- **Duration: 5 days**



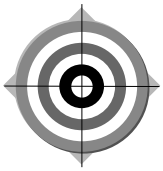
Notes to the user

- The training materials are **not teach-yourself programs.** They **complement the course instructor's explanations.** On the sheets, there is space for you to write down additional information.
- There may not be time during the course itself for you to complete all the exercises. The exercises are intended as additional examples of the topics discussed during the course. Participants can also use them as an aid to enhancing their knowledge after the course has finished.

# ABAP Workbench Foundations and Concepts: Contents

- Course goal
- Course objectives
- Course content
- Course overview diagram
- Main business scenario
- Course introduction

# Course Goal



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

- **Understand the various uses of the ABAP Workbench, including:**
  - **The different methods available for facilitating user dialog, and**
  - **How to carry on dialog with the database**

# Course Objectives

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

- **Create an ABAP program containing user dialogs and database dialogs**
- **Describe various development objects (Repository objects) and how they are used**
- **Create basic examples of those Repository objects introduced in the course using the appropriate ABAP Workbench tools**

## Course Content

| | |
|---|---|
| Unit 1 | **Introduction** |
| Unit 2 | **Program Flow in an ABAP Program** |
| Unit 3 | **Introduction to the ABAP Workbench** |
| Unit 4 | **ABAP Statements and Data Declarations** |
| Unit 5 | **Database Dialogs I (Reading Database Tables)** |
| Unit 6 | **Internal Program Modularization** |
| Unit 7 | **User Dialogs: List** |
| Unit 8 | **User Dialogs: Selection Screen** |
| Unit 9 | **User Dialogs: Screen** |
| Unit 10 | **Interfaces** |

## Course Content

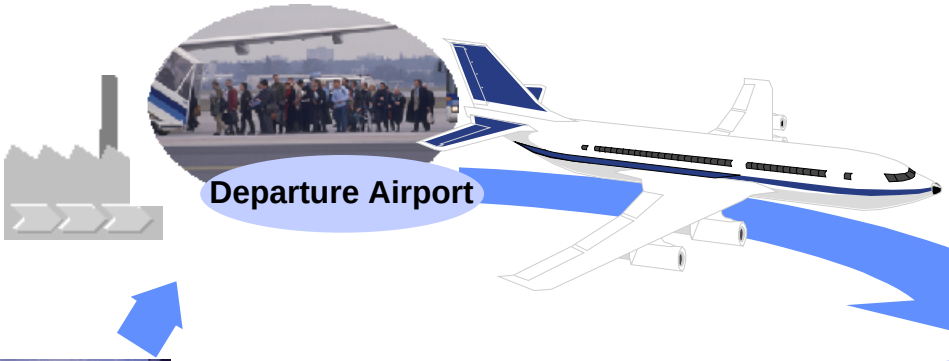
| | |
|---|---|
| Unit 11 | **Reuse Components** |
| Unit 12 | **Database Dialogs II (Making Changes to the Database)** |
| Unit 13 | **Software Logistics and Software Adjustment** |

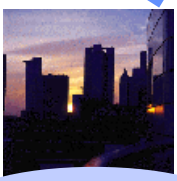
**Exercises**

**Solutions**

**Appendices**

# Main Business Scenario




**In this course, you will develop several programs meant to assist travel agencies. Some of their typical needs include:**

- **Determining flight connections on specific dates**
- **Processing bookings for specific flights**
- **Evaluating additional flight information, such as**
  - **Price**
  - **Capacity**

**Destination**

# Important Database Tables for the Flight Data Model

| | | |
|---|---|---|
| **SCARR** | **CARRID:** | **Airline ID** |
| | **CARRNAME:** | **Airline** |
| | **CURRCODE:** | **Local currency of airline** |
| **SPFLI** | **CARRID:** | **Airline ID** |
| | **CONNID:** | **Flight connection ID** |
| | **COUNTRYFR:** | **Country key for departure city** |
| | **CITYFROM:** | **Departure city** |
| | **AIRPFROM:** | **Departure airport** |
| | **COUNTRYTO:** | **Country key for arrival city** |
| | **CITYTO:** | **Destination City** |
| | **AIRPTO:** | **Destination airport** |
| **SFLIGHT** | **CARRID:** | **Airline ID** |
| | **CONNID:** | **Flight connection ID** |
| | **FLDATE:** | **Flight date** |
| | **PRICE:** | **Price** |
| | **CURRENCY:** | **Currency** |
| | **SEATSMAX:** | **Maximum number of seats on flight** |
| | **SEATSOCC:** | **Current number of occupied seats on flight** |

# Program Flow in an ABAP Program

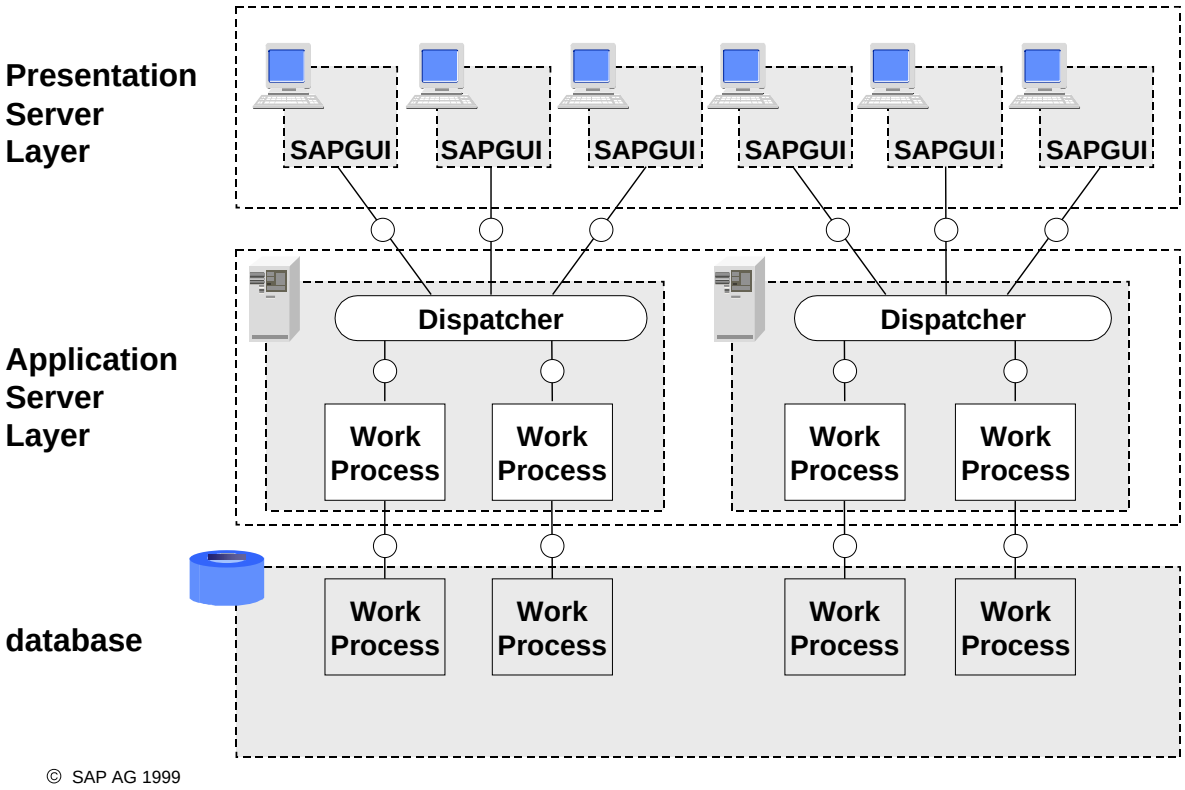
## Contents:

- Client / server architecture
- Sample program with data displayed in list form
- Sample program with data displayed on a screen
- Which ABAP program components are discussed in which units?

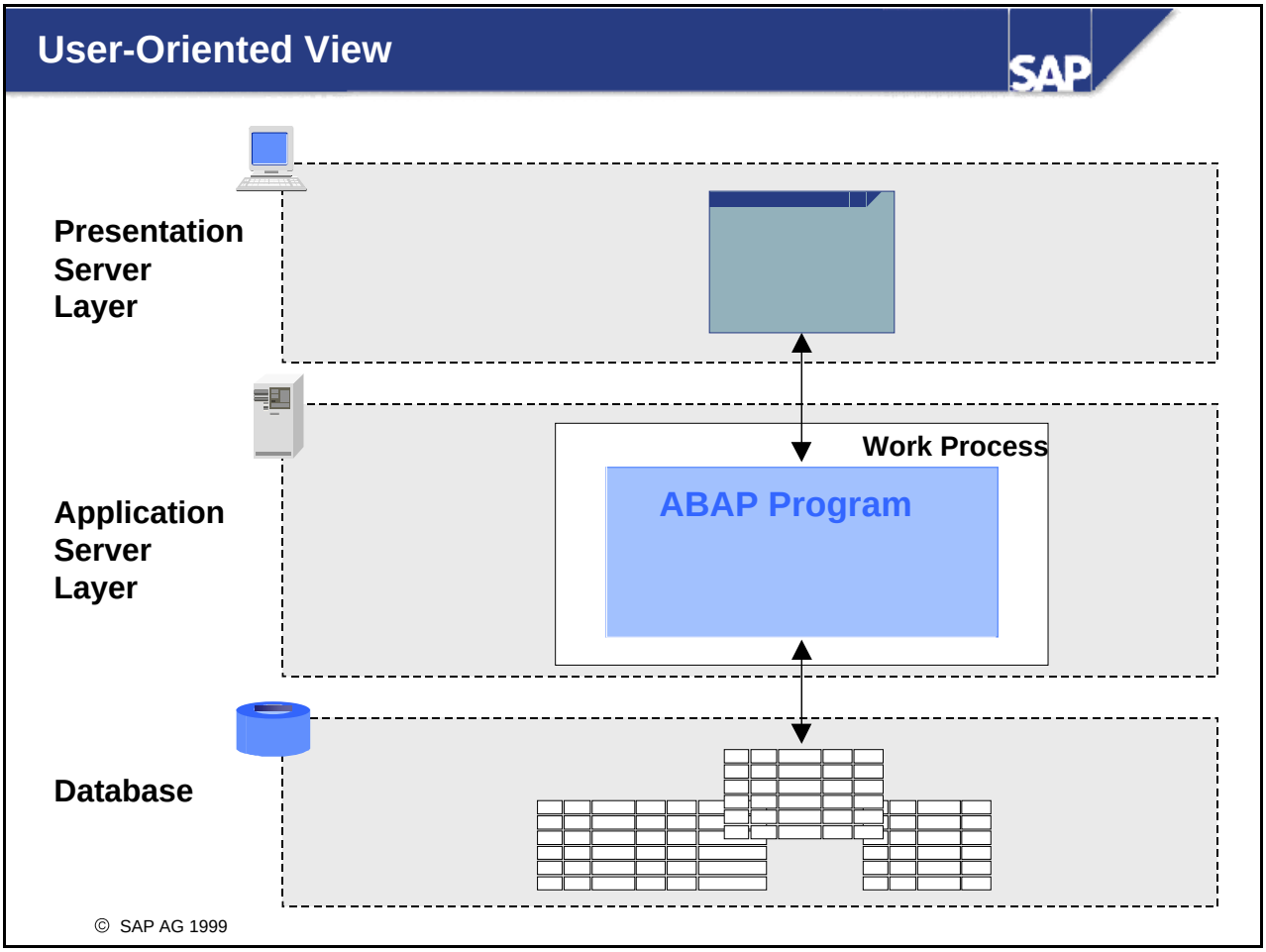
# Overview

- Client / server architecture
- Sample program with data displayed in list form
- Sample program with data displayed on the screen
- Which ABAP program components are discussed in which units?

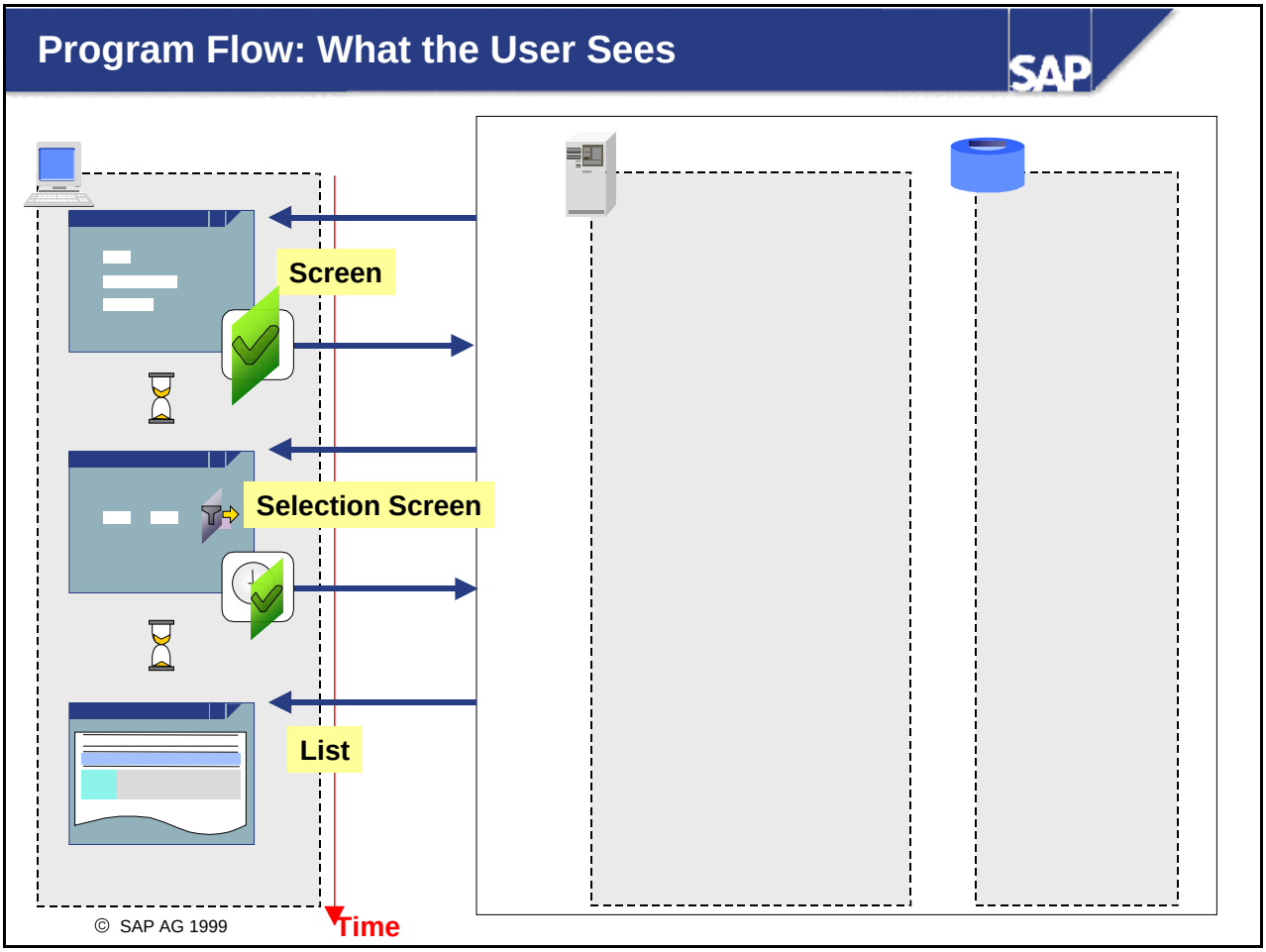
## Client / Server Architecture



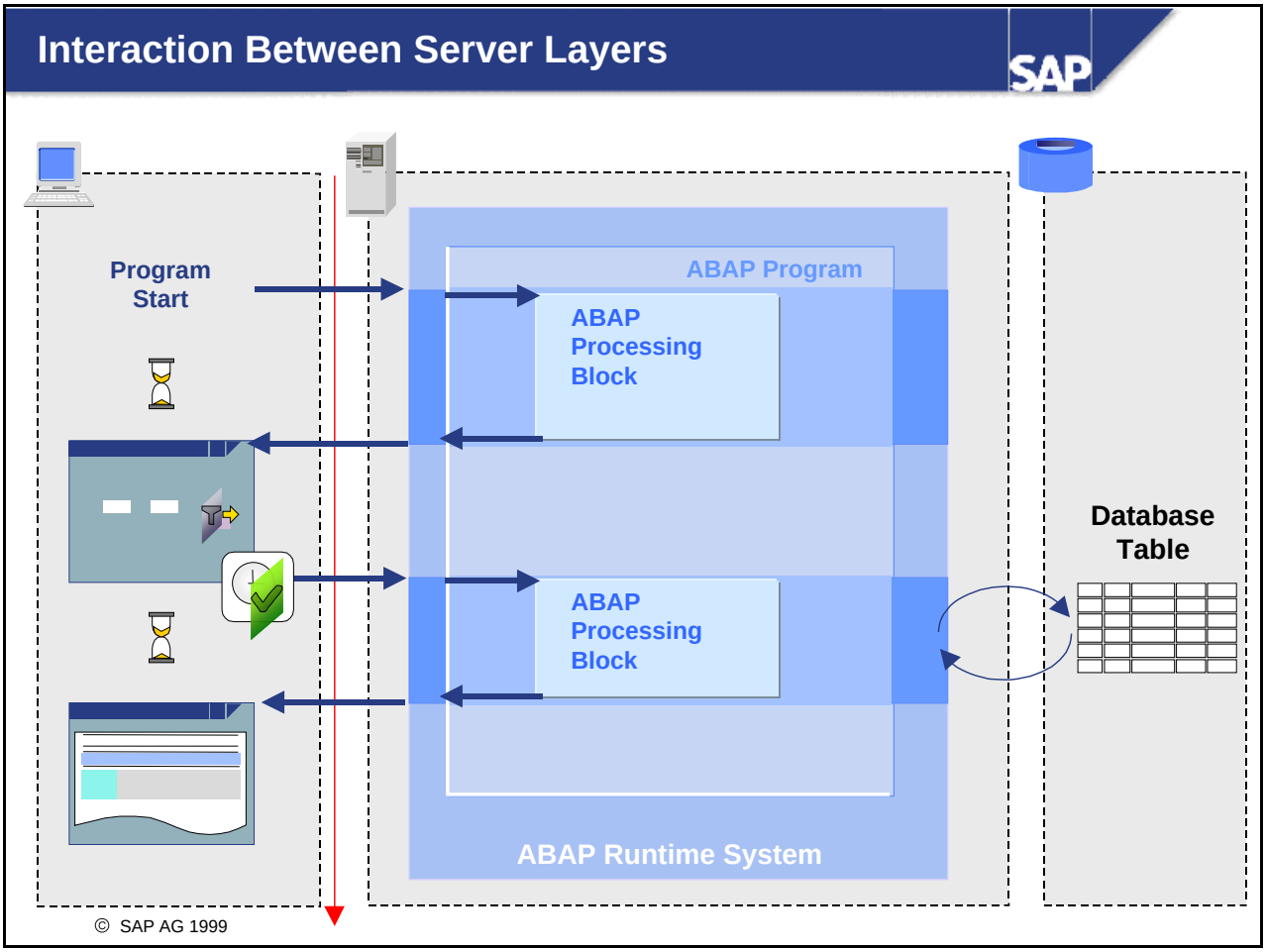

- The R/3 System has a modular software architecture that follows **software-oriented** client/server principles.
- The R/3 System allocates presentation, applications, and data storage to different computers. This serves as the basis for the **scalability** of the R/3 system.
- The lowest level is the **database level**. Here data is managed with the help of a relational database management system (RDBMS). In addition to master data and transaction data, programs and the metadata that describe the R/3 System are stored and managed here.
- ABAP programs run at the **application level**, both the applications provided by SAP and the ones you develop yourself. ABAP programs work with data called up from the database level and store new data there as well.
- The third level is the **presentation level** (SAPGUI). This level contains the user interface, in which an end user can access an application, 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, application and presentation level components can be divided among any number of computers. The horizontal distribution of database components, however, depends on the type of database installed.




- This graphic can be simplified for most topics discussed during this course. The interaction between ABAP programs and their users 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 the **user dialogs** and the **database dialogs** is therefore of particular importance when writing application programs.




- The user is primarily interested in how his or her business transaction flows and in how data can be input into and displayed from the transaction. Technical details, such as whether a single program is running or multiple programs are called implicitly, or the technical differences between the kind of screens being displayed, are usually less important to the user. The user does not need to know the precise flow of the ABAP program on the application server. Users see the R/3 System with application servers and database as a black box.
- There are, however, three technically distinct screen types (screens, selection screens, and lists) that offer the user different services. It is the developer's job to determine which type of user dialog is most suitable to the user's needs.




- When the user performs a user action (choosing *Enter,* a function key, a menu function or a pushbutton, for example), control is handed over from the presentation server to the application server and certain parts of the ABAP program are processed. If further user dialog is triggered within the ABAP program, the system sends a screen to the presentation server and control is once again handed over to the presentation server.

## Overview

Client / server architecture

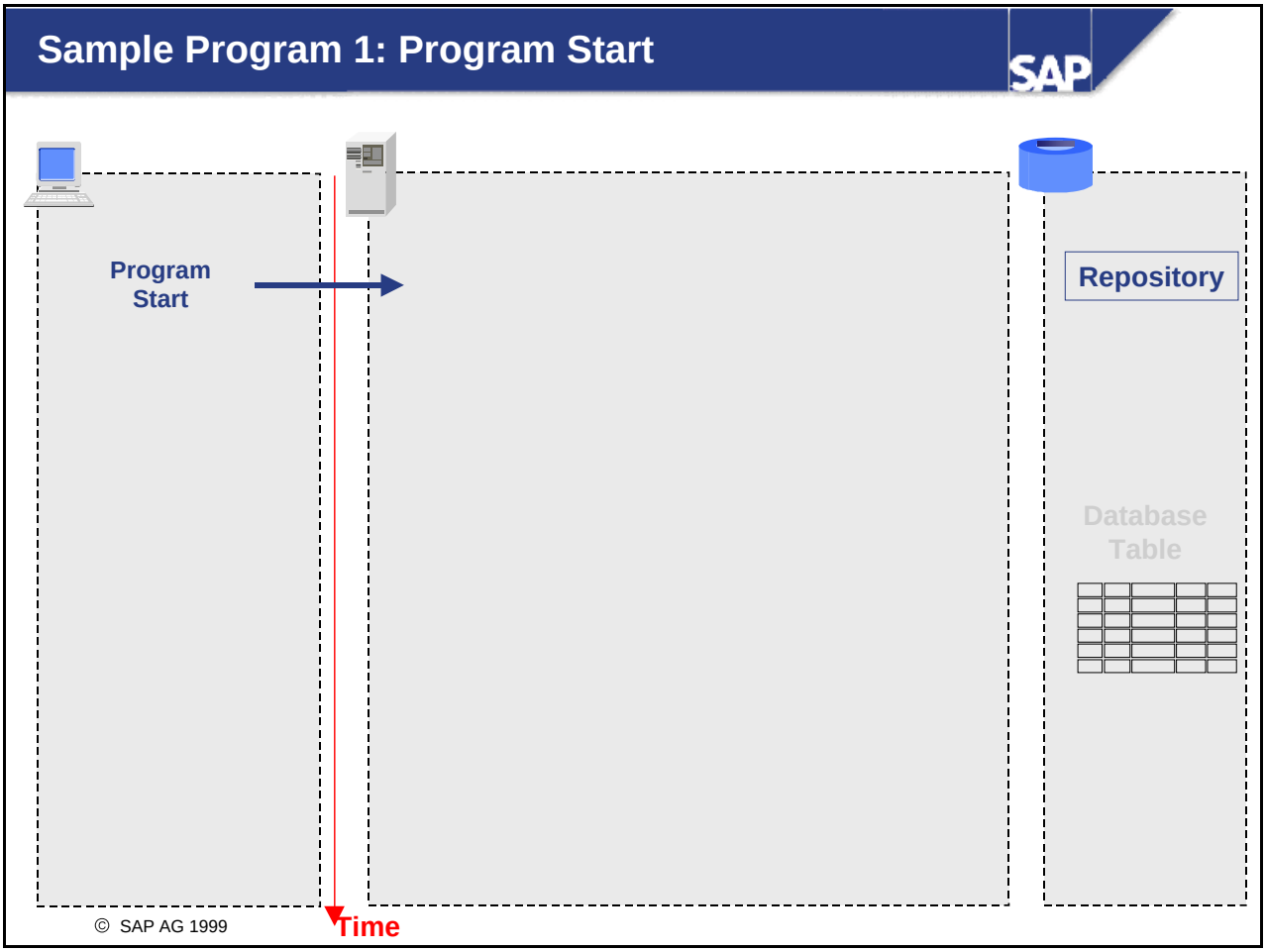
Sample program with data displayed in list form

Sample program with data displayed on the screen

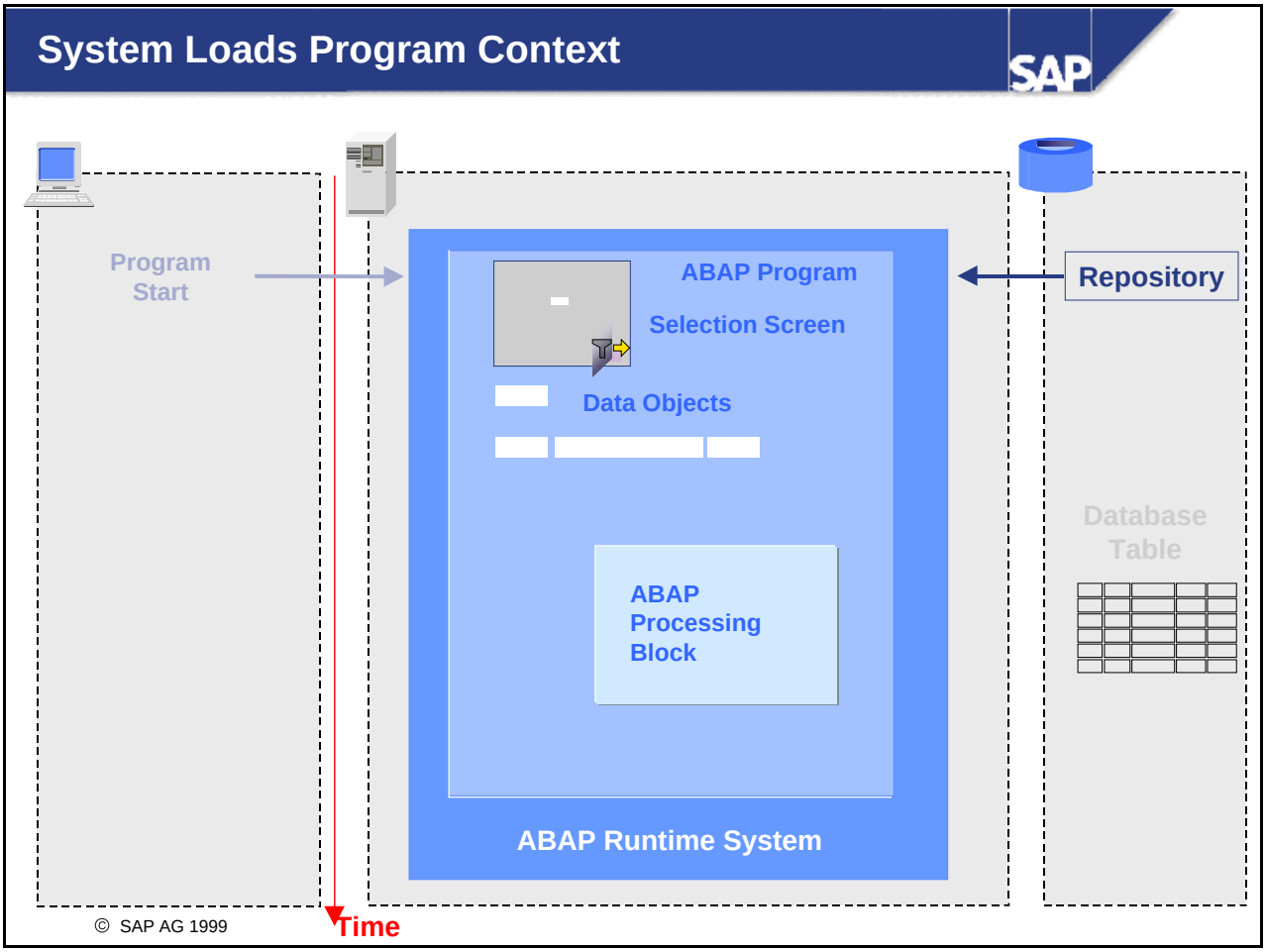
Which ABAP program components are discussed in which units?



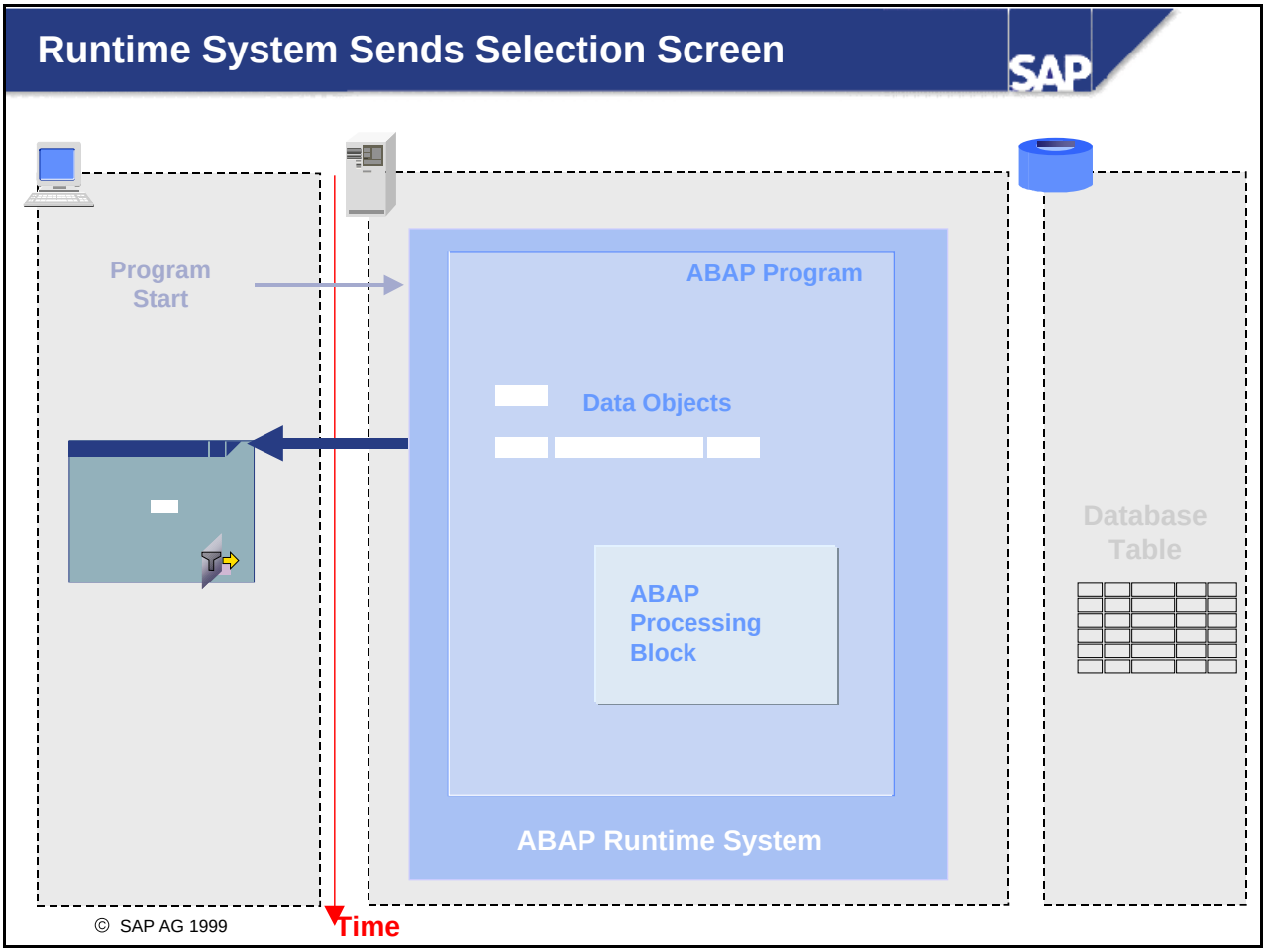
- In this part of the unit, the user has chosen to start a program where an airline ID can be entered on the initial selection screen. The program subsequently uses this information to retrieve the 'Long name of airline' and the 'Local currency of airline' from the database and display them for the user in **list form**.



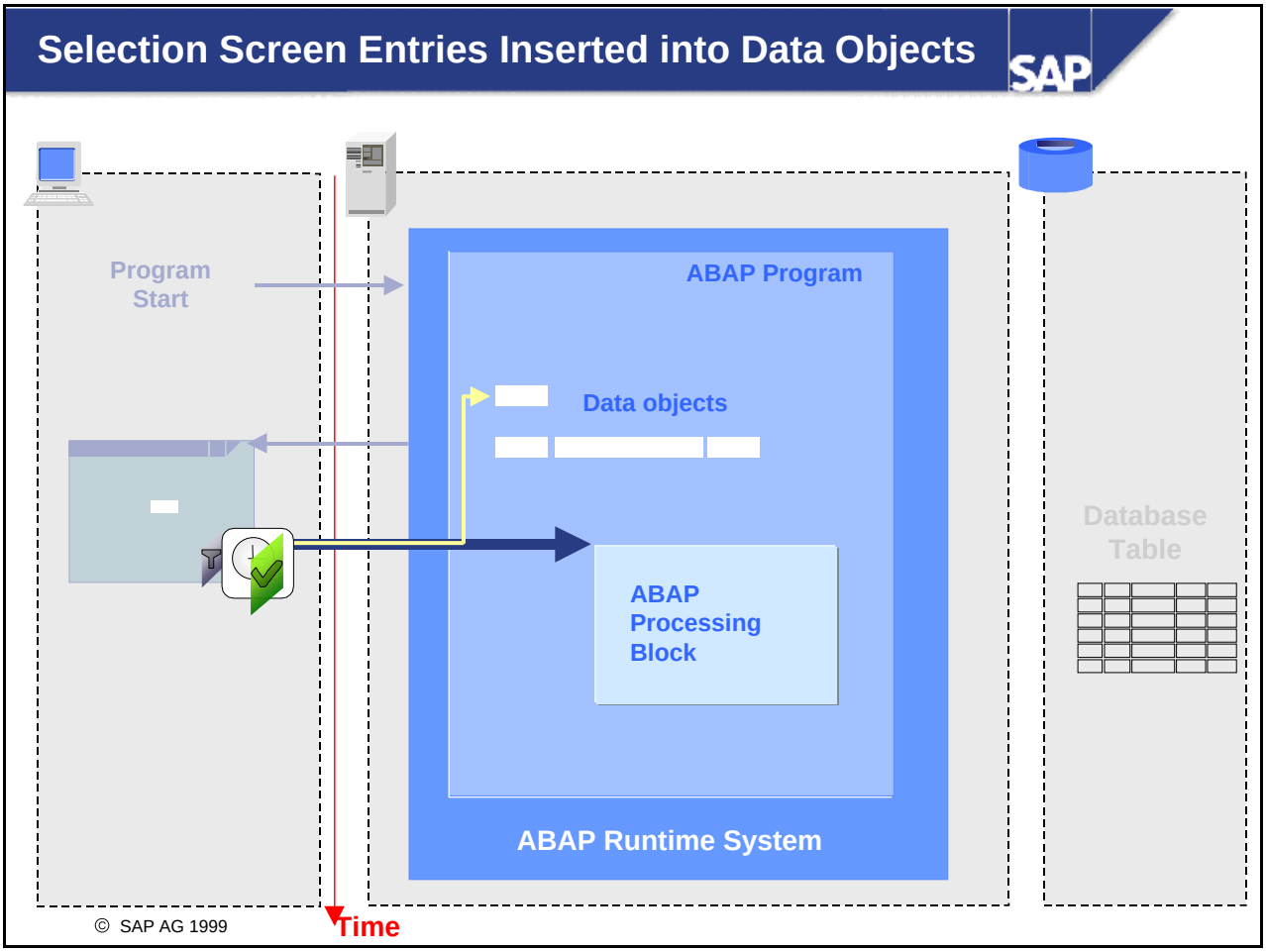

- Whenever a user logs on to the system, a screen is displayed. From this screen, the user can start a program by using its menu path.



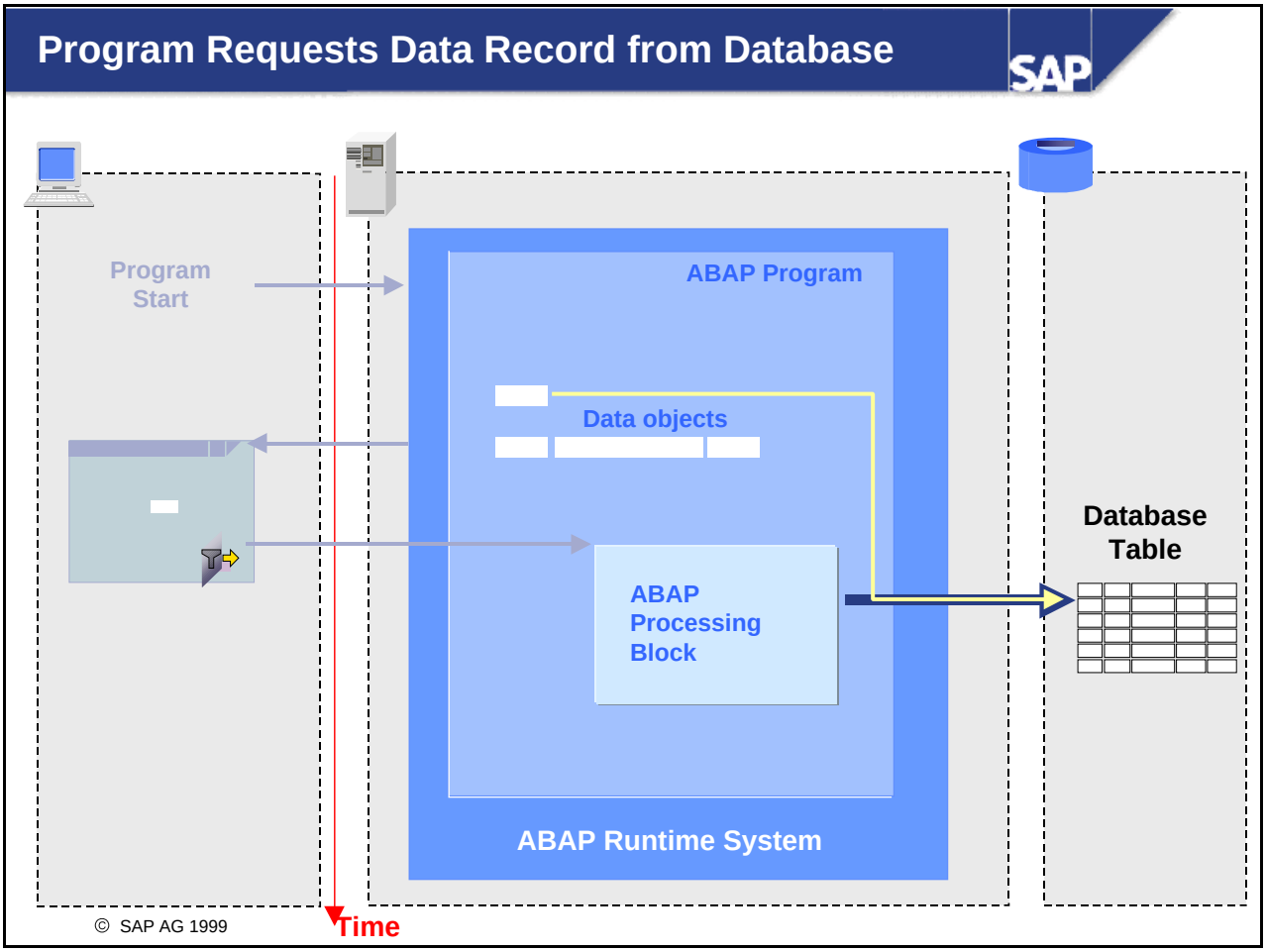

- If the user has triggered a program with a user action, then the program context is loaded on the application server. The program context contains memory areas for variables and complex data objects, information on the screens for user dialogs and ABAP processing blocks. The runtime system gets the program information from the Repository, which is a special part of the database.
- The sample program has a selection screen as the user dialog, a variable and a structure as data objects and one ABAP processing block. The list that is used to display the data is created dynamically at runtime.
- The subsequent flow of the program is controlled by the ABAP runtime system.



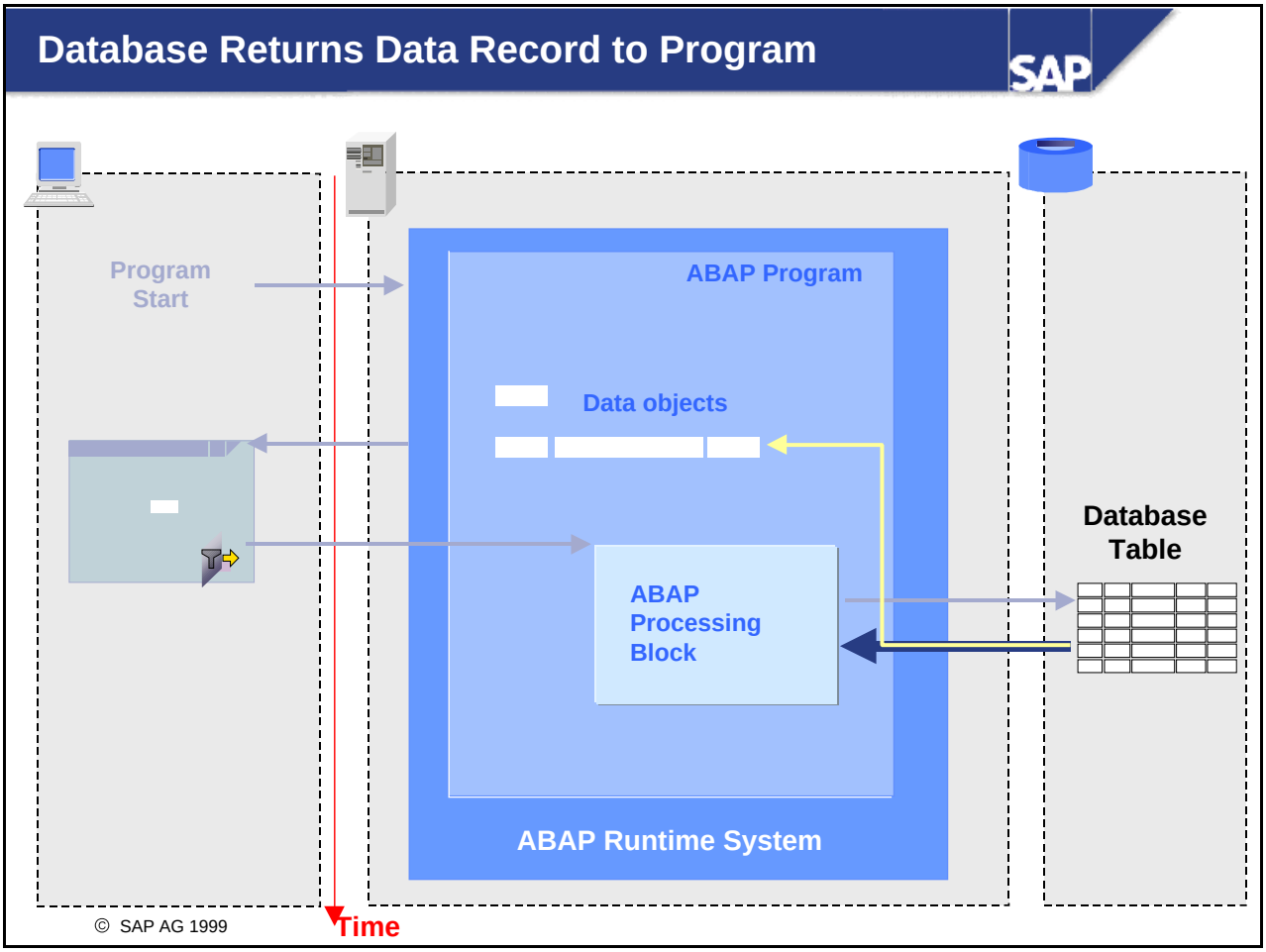

- Since the program contains a selection screen, the ABAP runtime system sends it to the presentation server at the beginning of program processing. The presentation server controls the program flow for as long as the user fills in the input fields.
- Selection screens allow users to enter selection criteria required by the program.



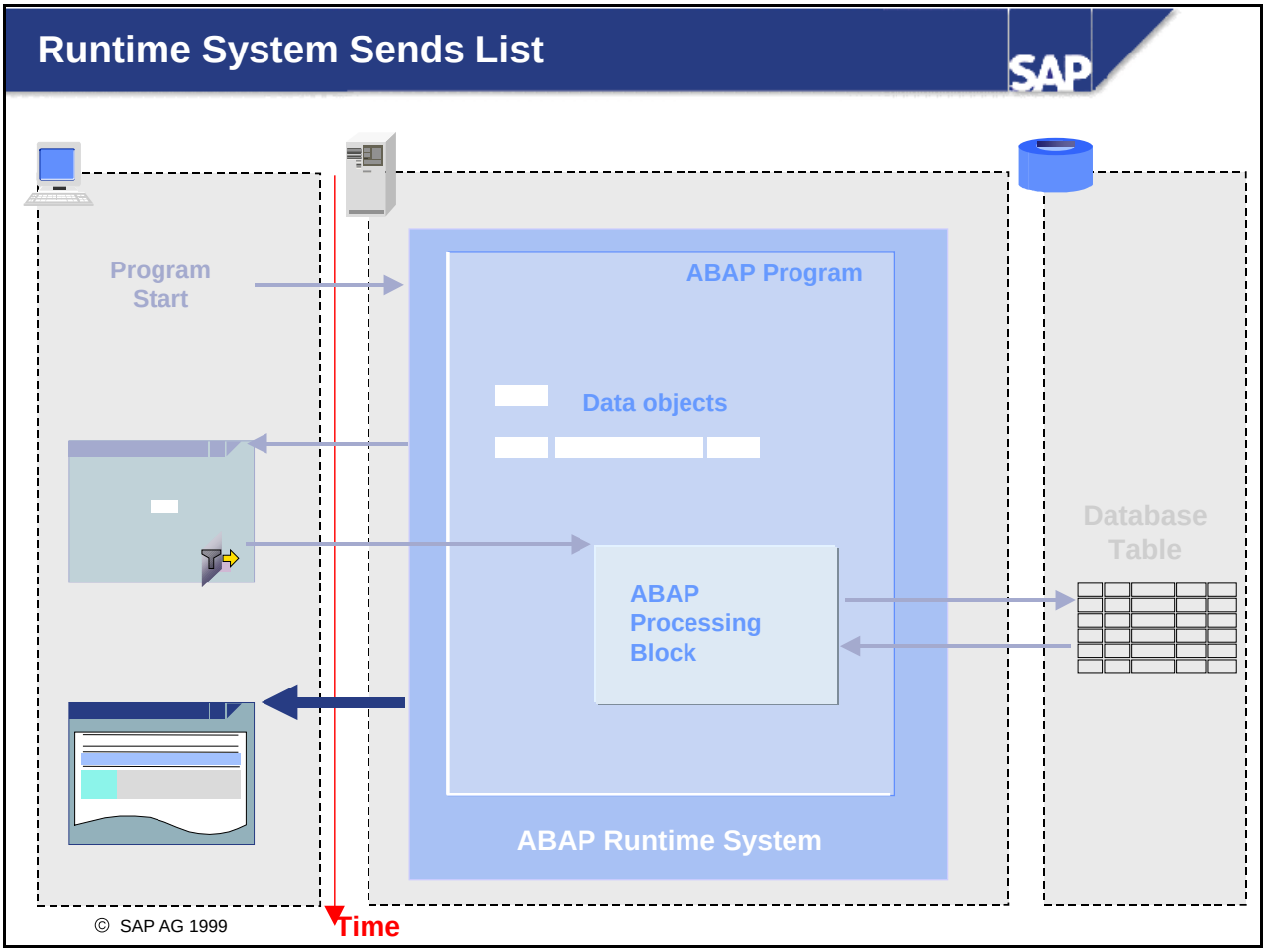

- As soon as the user has finished entering data on the selection screen, he or she can trigger further processing by choosing 'Execute'. All data input on the selection screen is the automatically placed in its corresponding data object in the program and the ABAP runtime system resumes control of processing. Our sample program contains only one ABAP processing block. The runtime system triggers sequential processing of this ABAP processing block.
- If the entries made by the user do not have the correct type, then an error message is automatically triggered. The user must correct his/her entries.



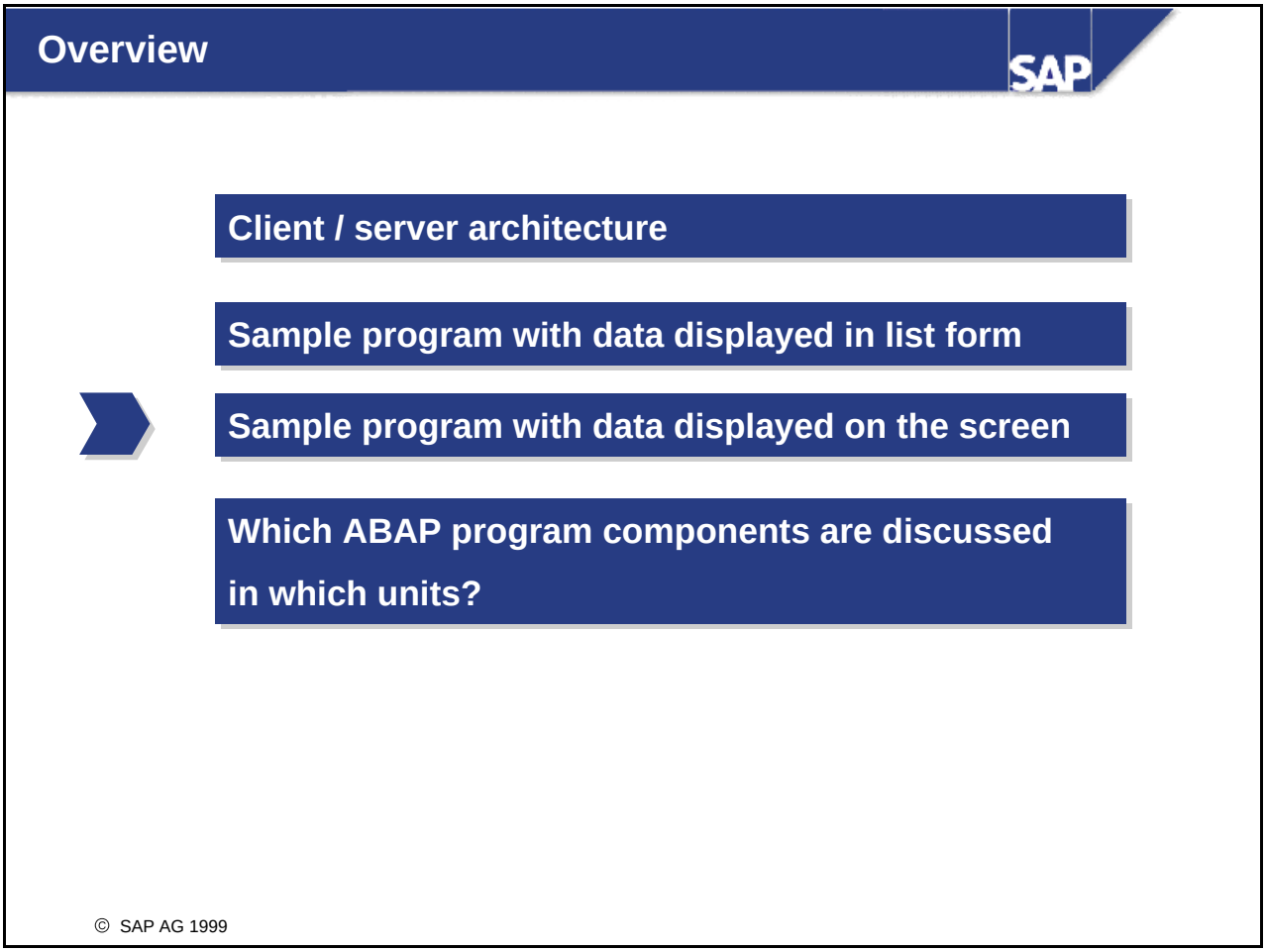

- The ABAP processing block contains a read access to the database that has been programmed into it. The program also passes the database information about which database table to access and which line in the table to read.



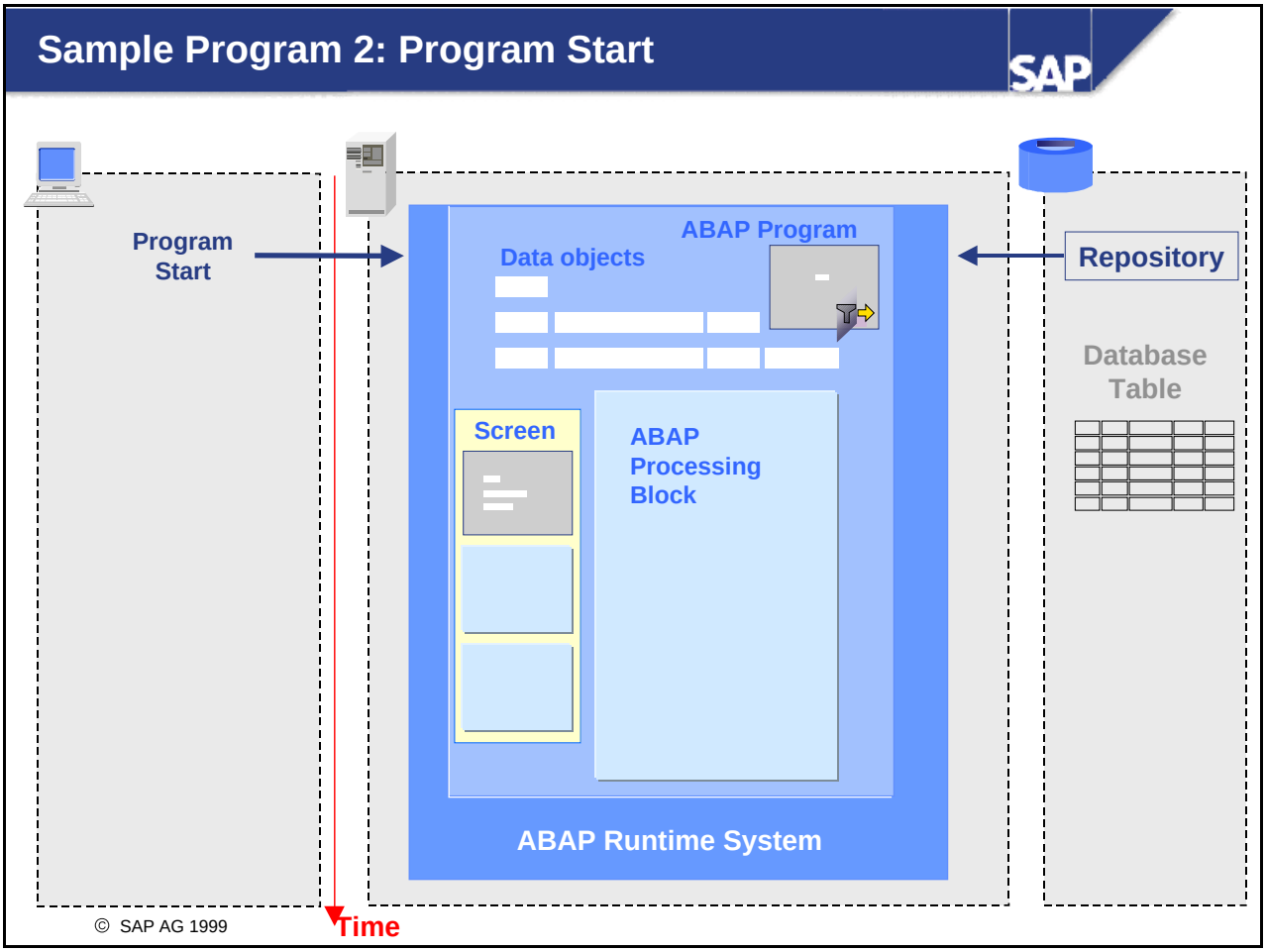

- The database returns the requested data record to the program and the runtime system ensures that this data is stored in the appropriate data objects. Normally a structure is the target field when a single record is accessed. The structure contains variables for all fields requested from the database.



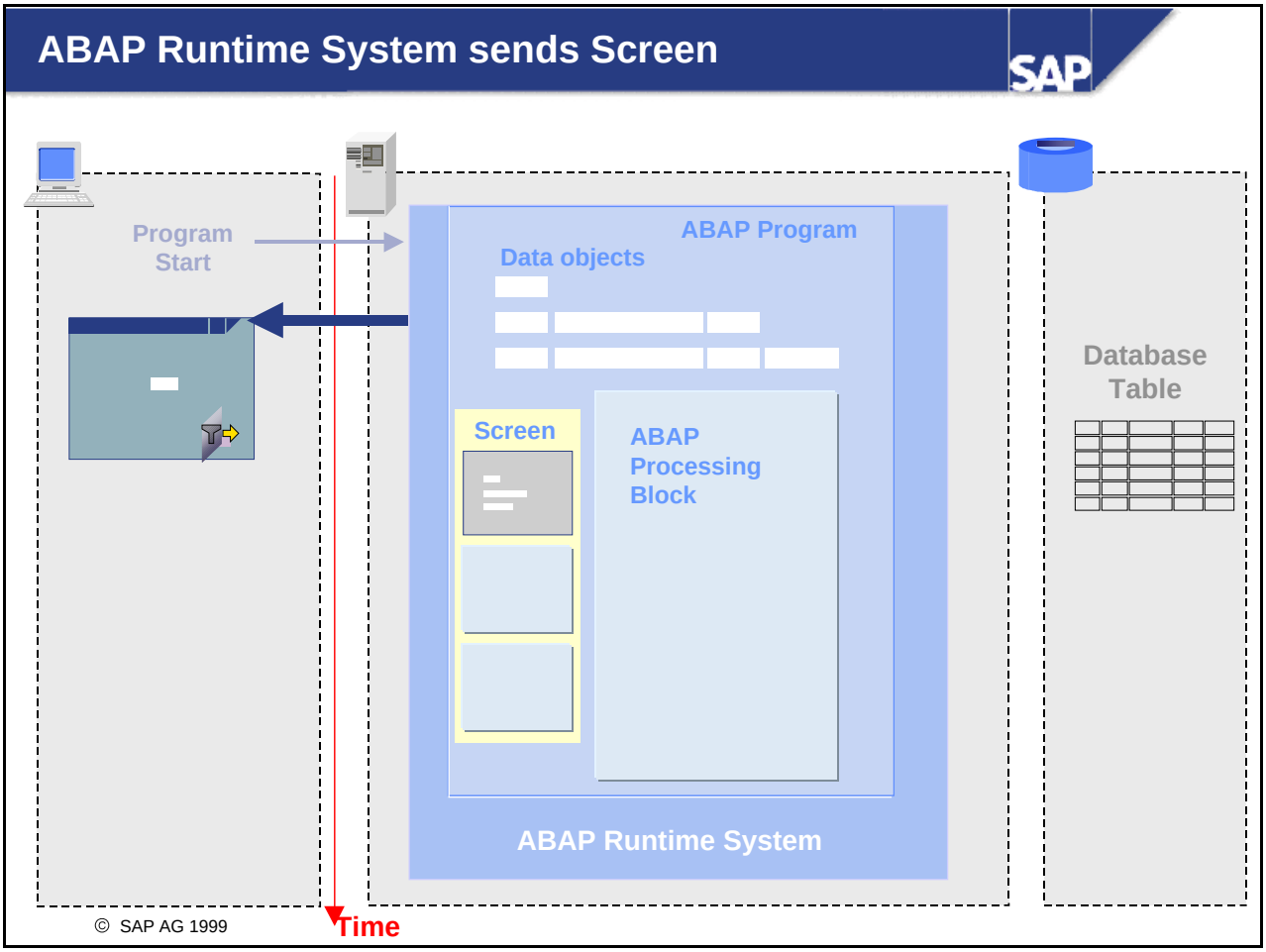

- The layout of the subsequent list display has also been programmed into the processing block. After all processing has ended, the runtime system sends the list screen to the presentation server.



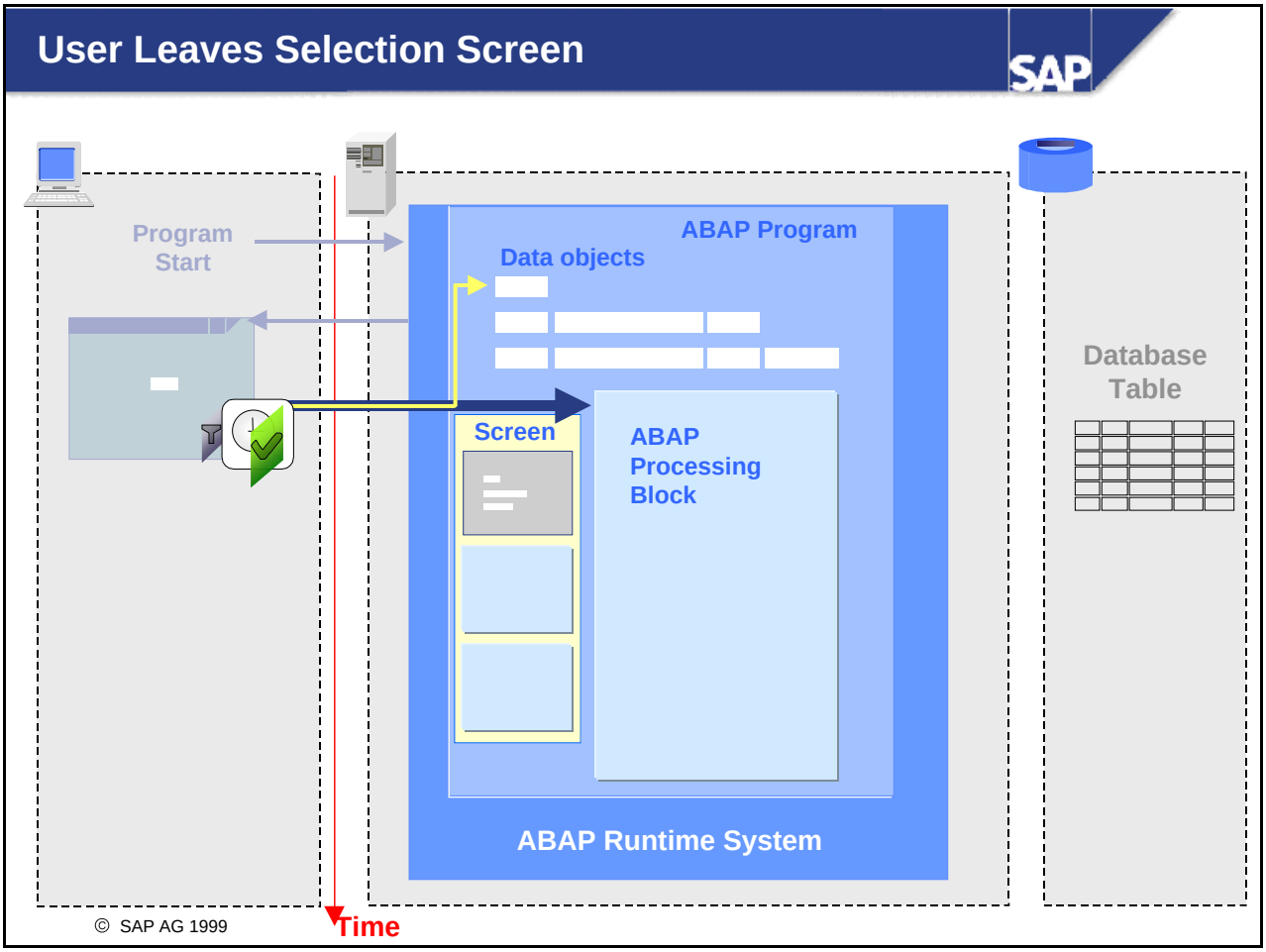

- In this part of the unit, the user starts a second sample program where an airline ID can be entered on the initial selection screen. This program subsequently uses the information input on the selection screen to retrieve the 'Long name of airline' and the 'Local currency of airline' from the database and display them for the user on a **screen**.



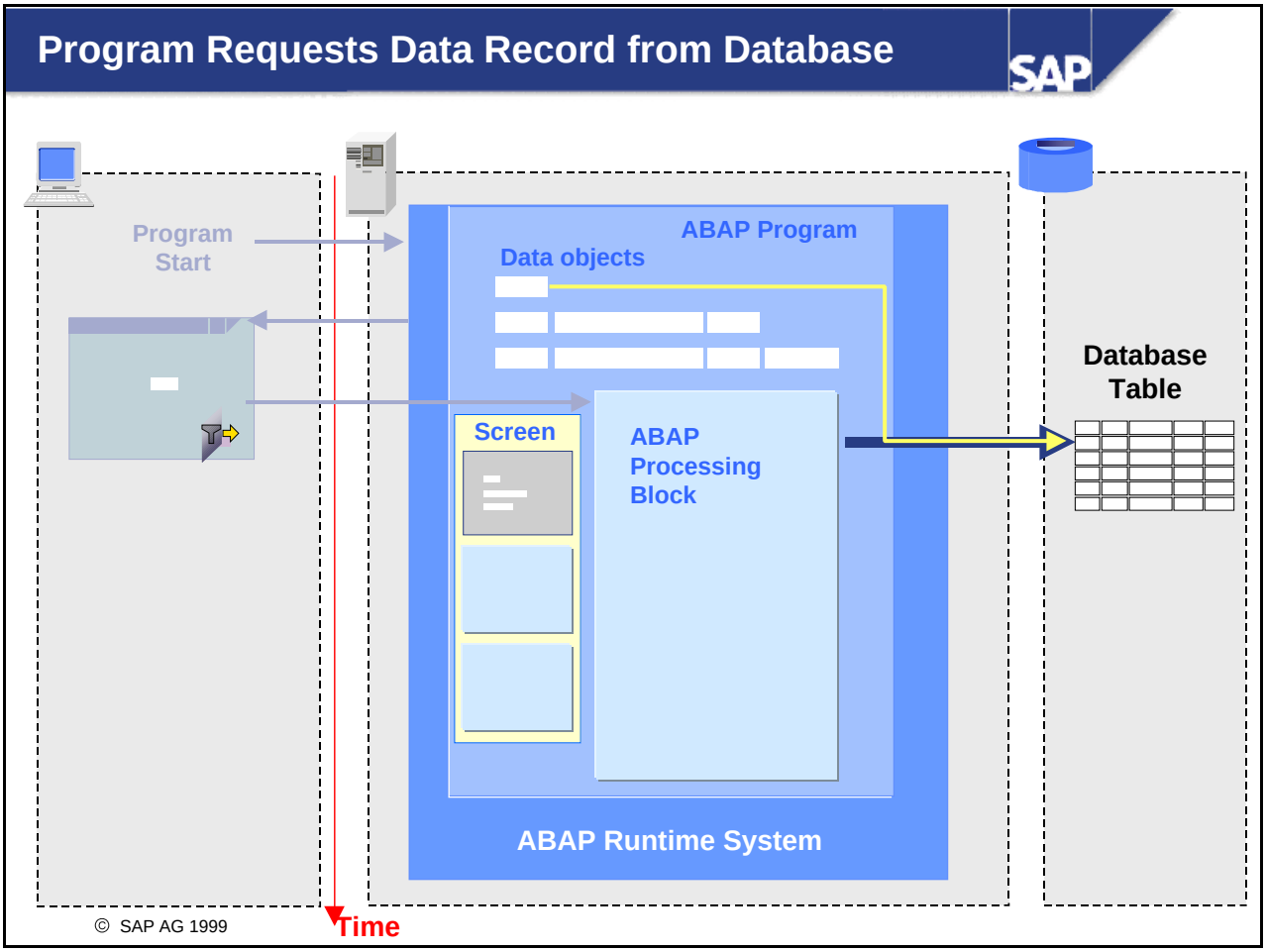

- When the user starts the program, the program context is loaded first. This time, however, our sample program contains three processing blocks, a selection screen, and a screen, and a variable and two structures as its data objects.



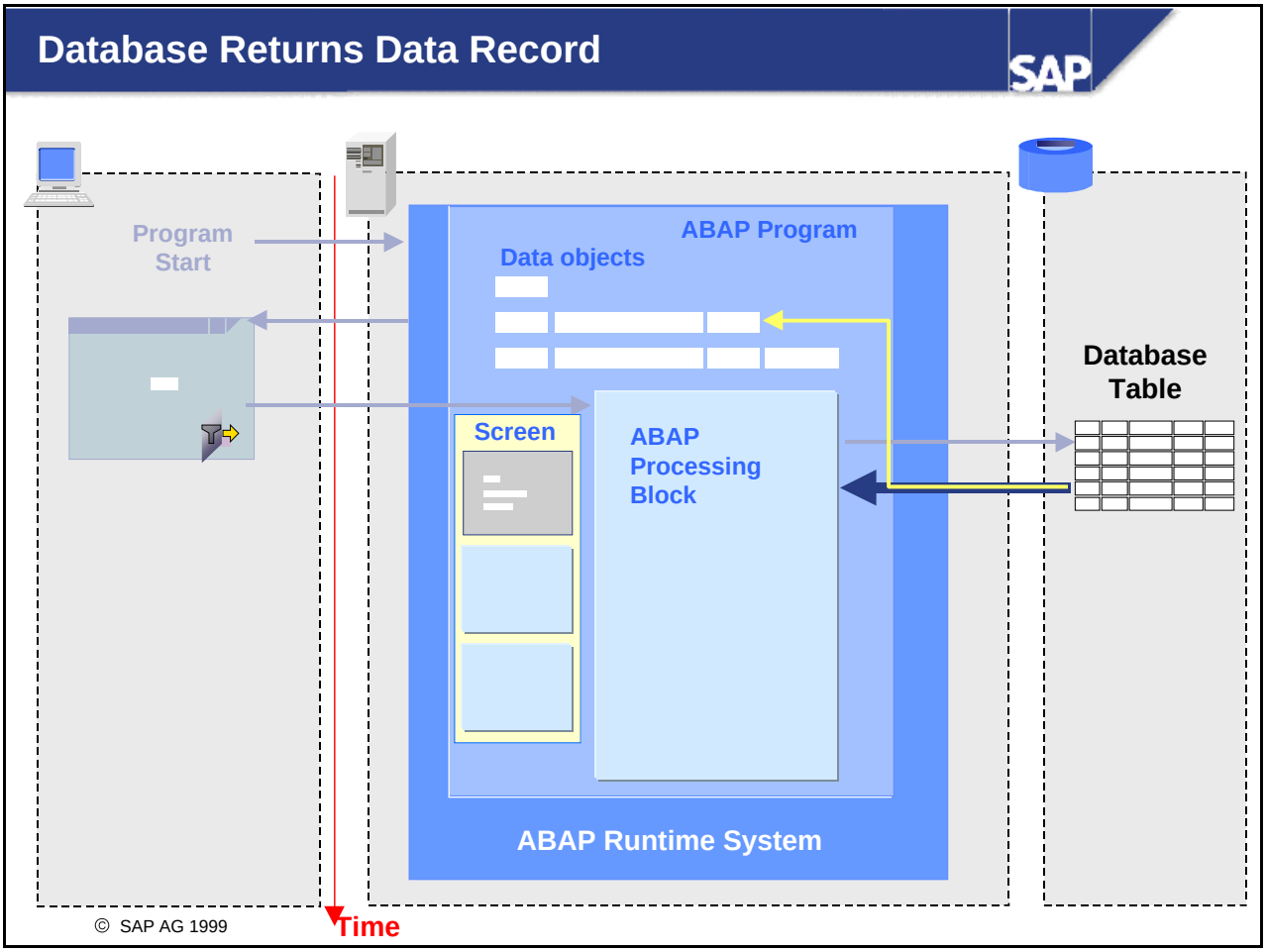

- Since the program contains a selection screen, the ABAP runtime system sends it to the presentation server at the beginning of program processing.



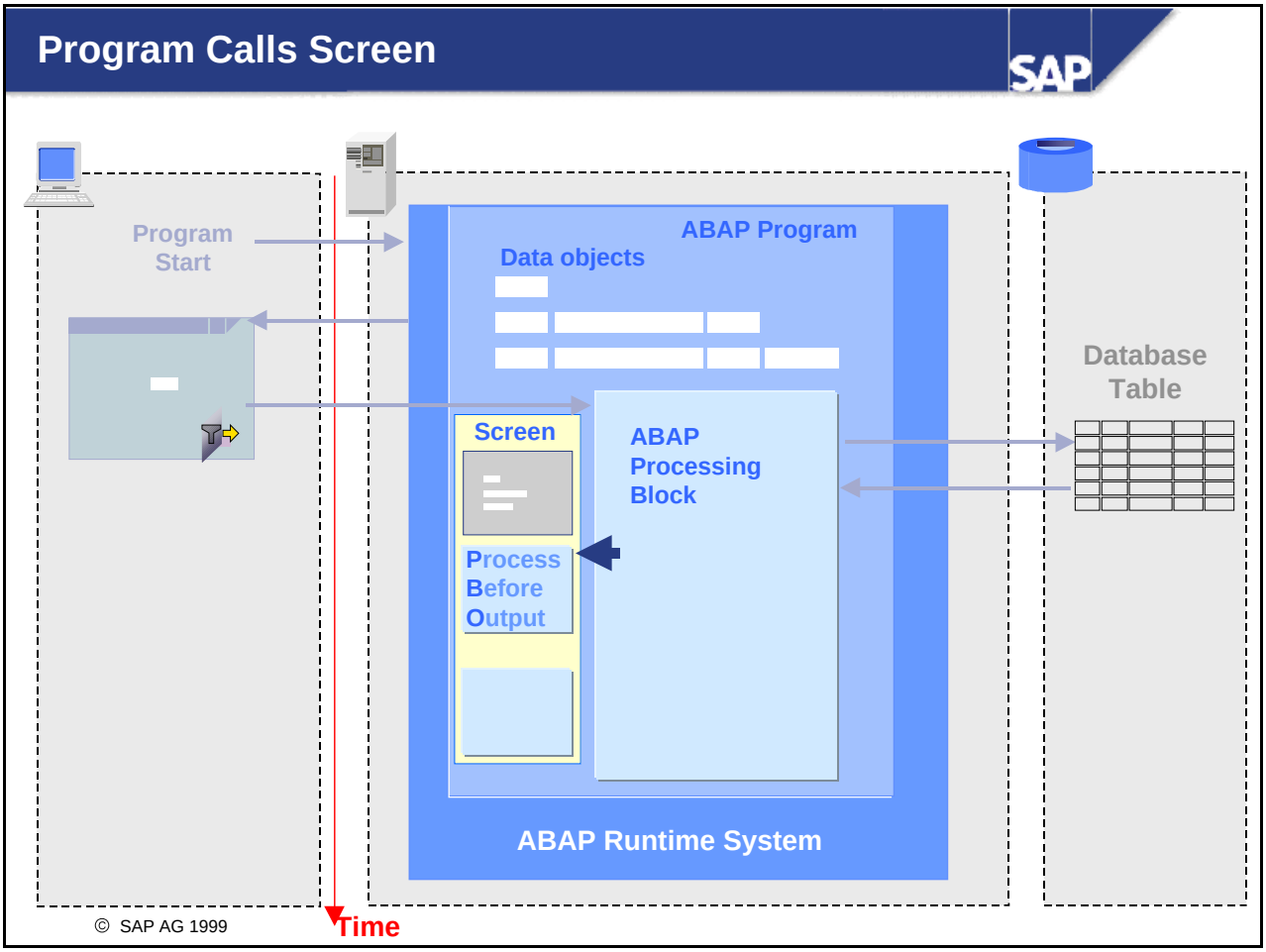

- As soon as the user has finished entering data on the selection screen, he or she can trigger further processing by choosing 'Execute'. All data input on the selection screen is then automatically placed in its corresponding data object in the program and the ABAP runtime system resumes control of processing. The runtime system then triggers sequential processing of the ABAP processing block that comes after the selection screen.



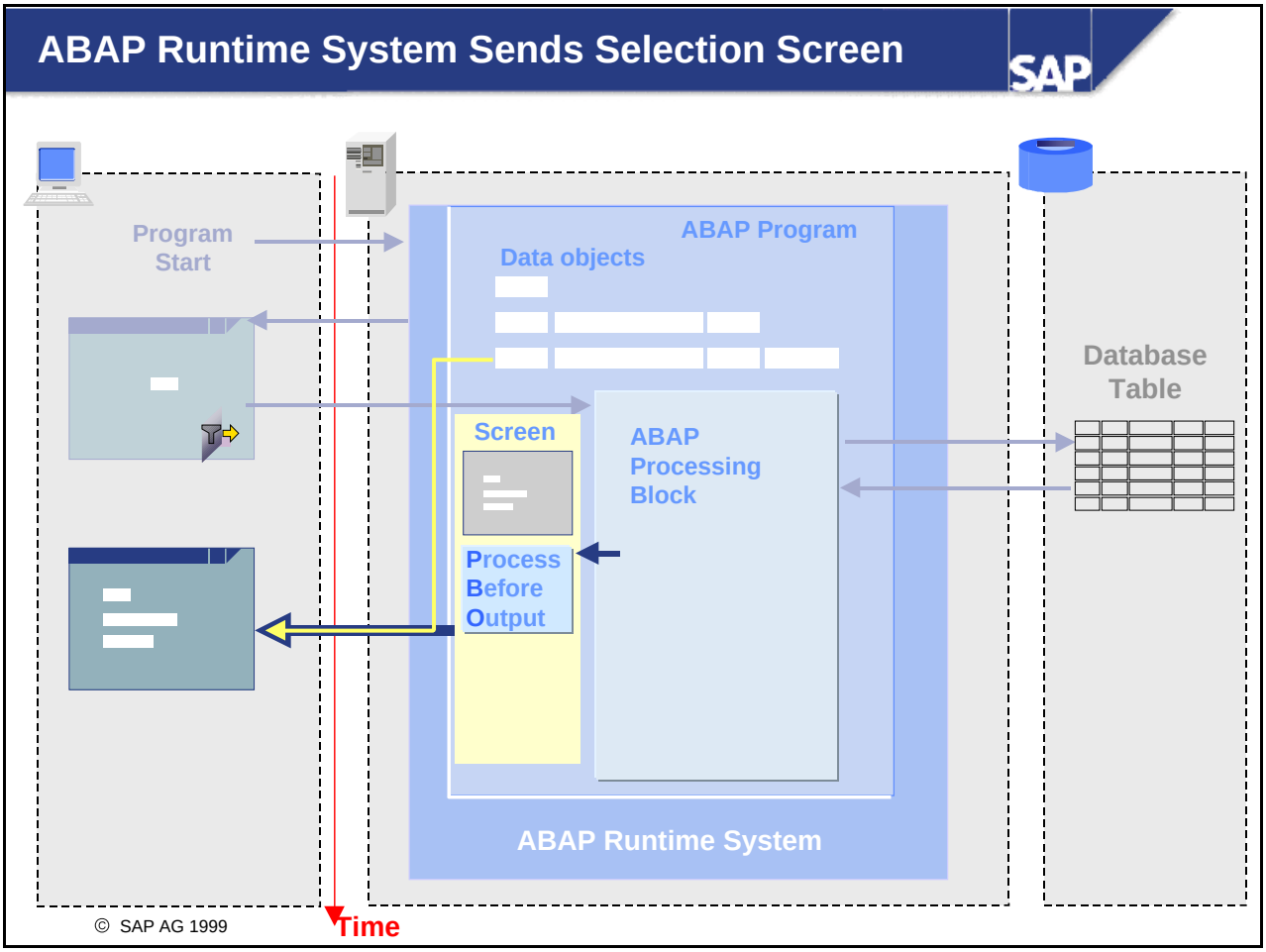

- The ABAP processing block contains a read access to the database that has been programmed into it. The program also passes the database information about which database table to access and which line in the table to read.



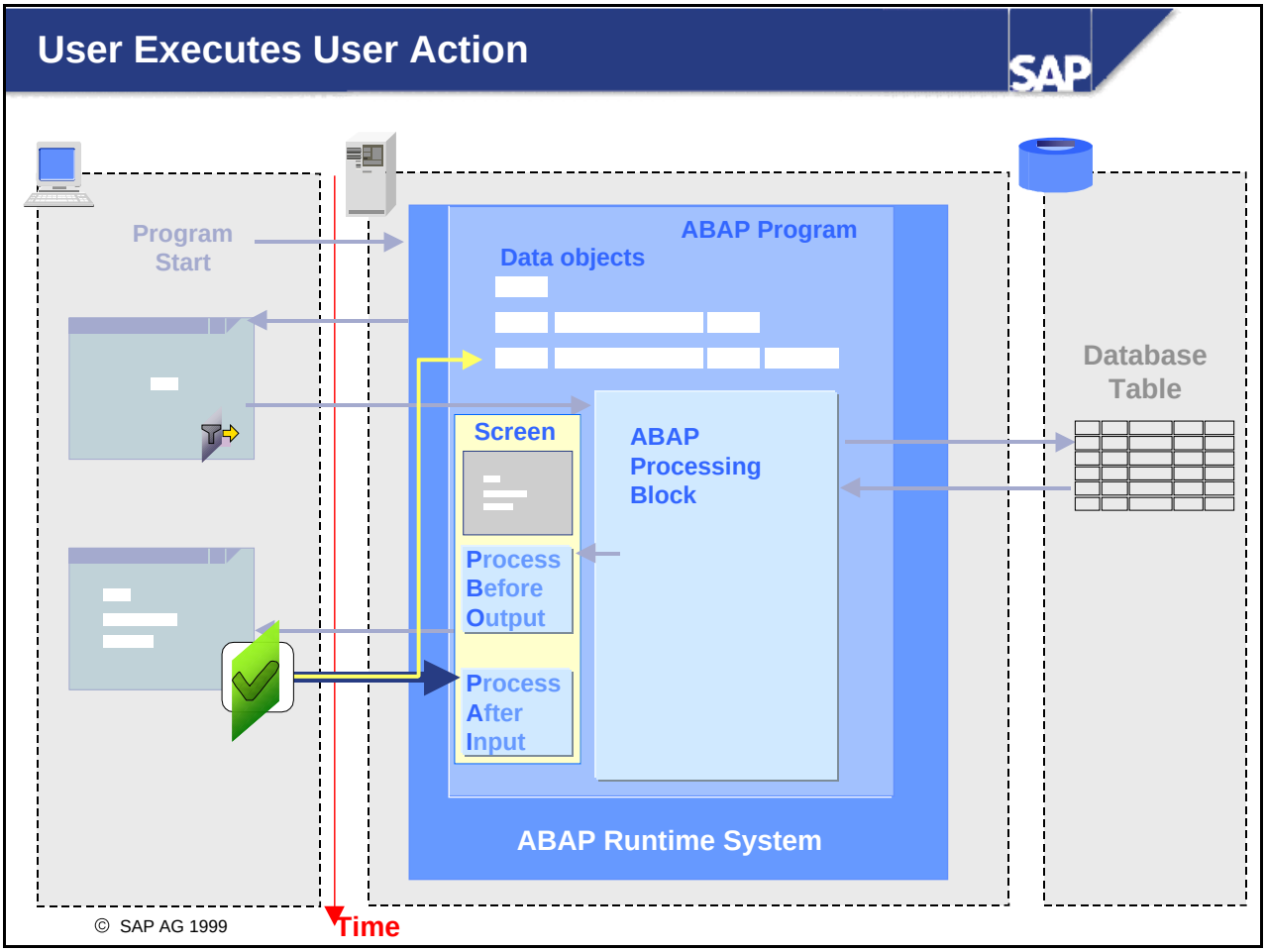

- The database returns the requested data record to the program and the runtime system ensures that this data is stored in the appropriate data objects. Normally a structure is the target field when a single record is accessed. The structure contains variables for all fields requested from the database.



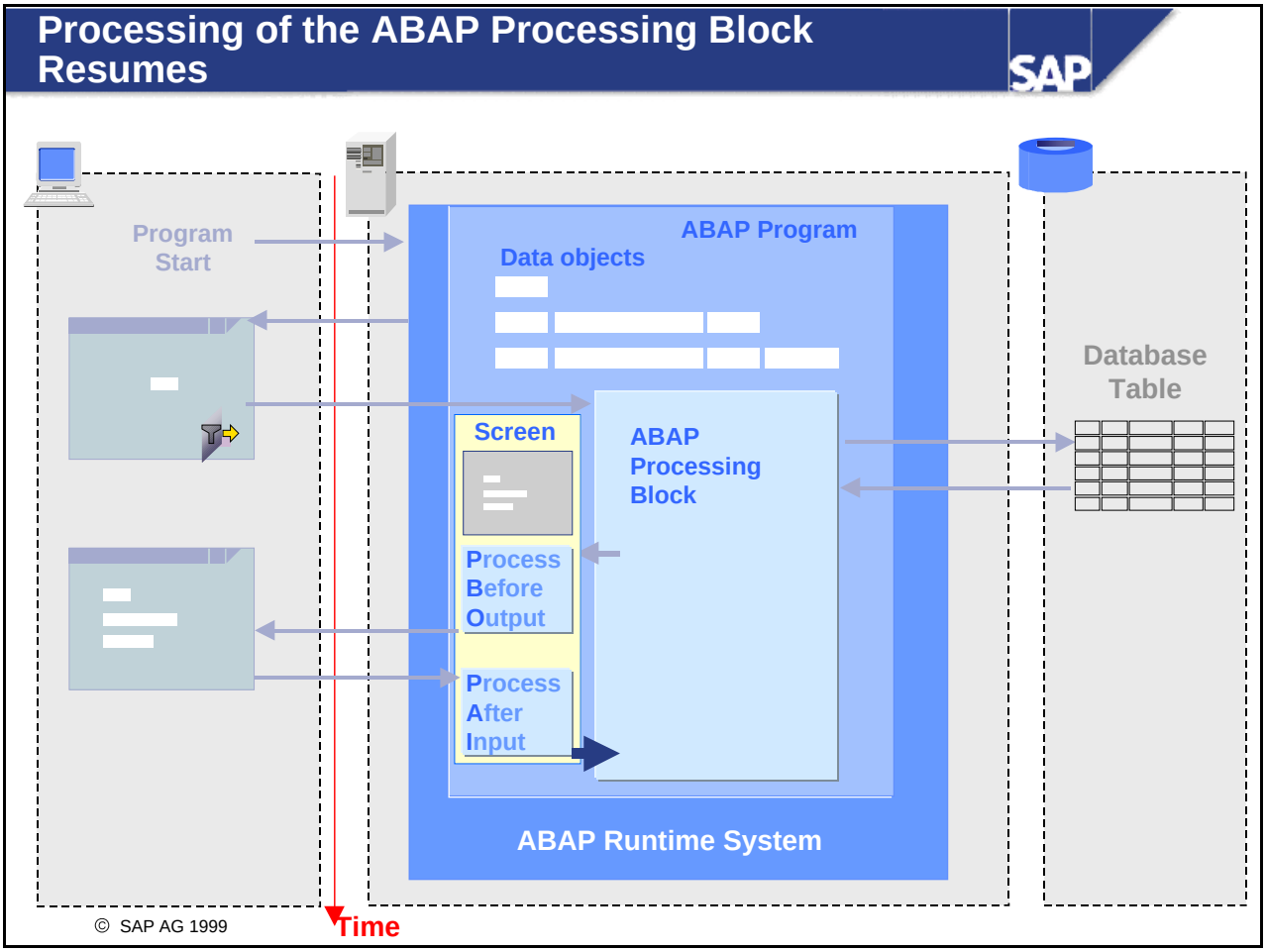

- The ABAP processing block now triggers screen processing. This is often expressed simply by saying 'The program calls the screen'. However, in reality, each screen possesses its own processing block that is sequentially processed before the runtime system sends the screen to the presentation server (**P**rocess **B**efore **O**utput). This allows screens to be used in a very flexible manner.




- After the screen's processing block has been processed, the ABAP runtime system sends the screen to the presentation server. During this process, data is transported into the screen's fields from a structure that serves as an interface for the screen.




- Once the user performs a user action (choosing *Enter,* a function key, a menu function or a pushbutton, for example), control is handed over to the runtime system on the application server again. The screen fields are transported into the structure that serves as the screen's interface and a special processing block belonging to the screen is triggered. This processing block is always processed immediately following a user action (**P**rocess **A**fter **I**nput).




- After the 'Process After Input' processing block has been processed, the sample program continues processing the ABAP processing block that called the screen in the first place.

# Overview

- Client / server architecture
- Sample program with data displayed in list form
- Sample program with data displayed on the screen
- Which ABAP program components are discussed in which units?

## Course Content

| | |
|---|---|
| Unit 1 | **Introduction** |
| Unit 2 | **Program Flow in an ABAP Program** |
| Unit 3 | **Introduction to the ABAP Workbench** |
| Unit 4 | **ABAP Statements and Data Declarations** |
| Unit 5 | **Database Dialogs I (Reading Database Tables)** |
| Unit 6 | **Internal Program Modularization** |
| Unit 7 | **User Dialogs: List** |
| Unit 8 | **User Dialogs: Selection Screen** |
| Unit 9 | **User Dialogs: Screen** |
| Unit 10 | **Interfaces** |

## Course Content

| | |
|---|---|
| Unit 11 | **Reuse Components** |
| Unit 12 | **Database Dialogs II (Making Changes to the Database)** |
| Unit 13 | **Software Logistics and Software Adjustment** |

**Exercises**

**Solutions**

**Appendices**

# Introduction to the ABAP Workbench

## Contents:

- Repository and Workbench
- Analyzing an existing program
- First project: Adjusting a copy of an existing program to fulfill special requirements

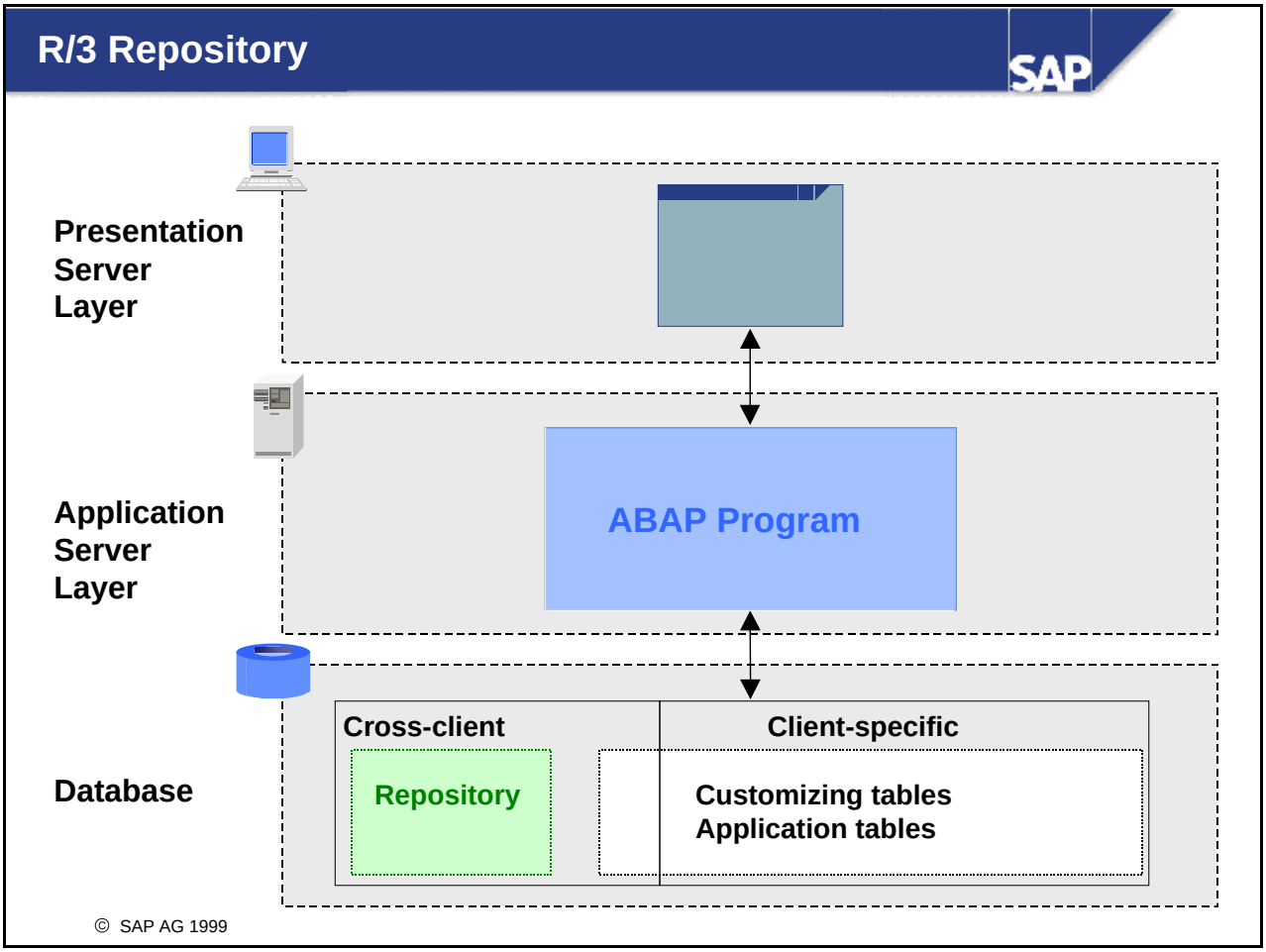
# Introduction to the ABAP Workbench

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

- **Use the different types of navigation available in the ABAP Workbench to reconstruct an existing program**
- **Make simple changes to an existing program's user dialogs using the tools ABAP Editor and Screen Painter**

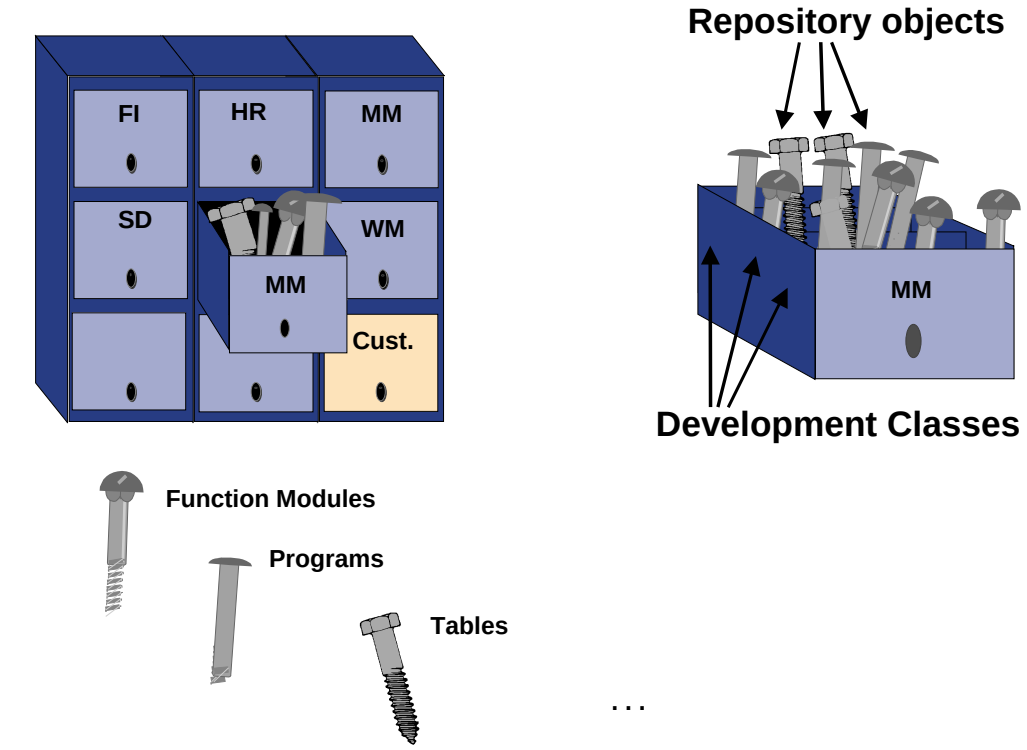
# Overview: Introduction to the ABAP Workbench

- Repository and Workbench
- Analyzing an existing program
- First project: Adapting an existing program to special requirements
- Creating a new program




- Along with the Repository, the database contains application and customizing tables that are usually client-specific.
- The **Repository** contains all development objects, for example, programs, definitions of database tables and global types. Development objects are therefore also known as Repository objects. Repository objects are not client-specific. They can therefore be viewed and used in all clients.

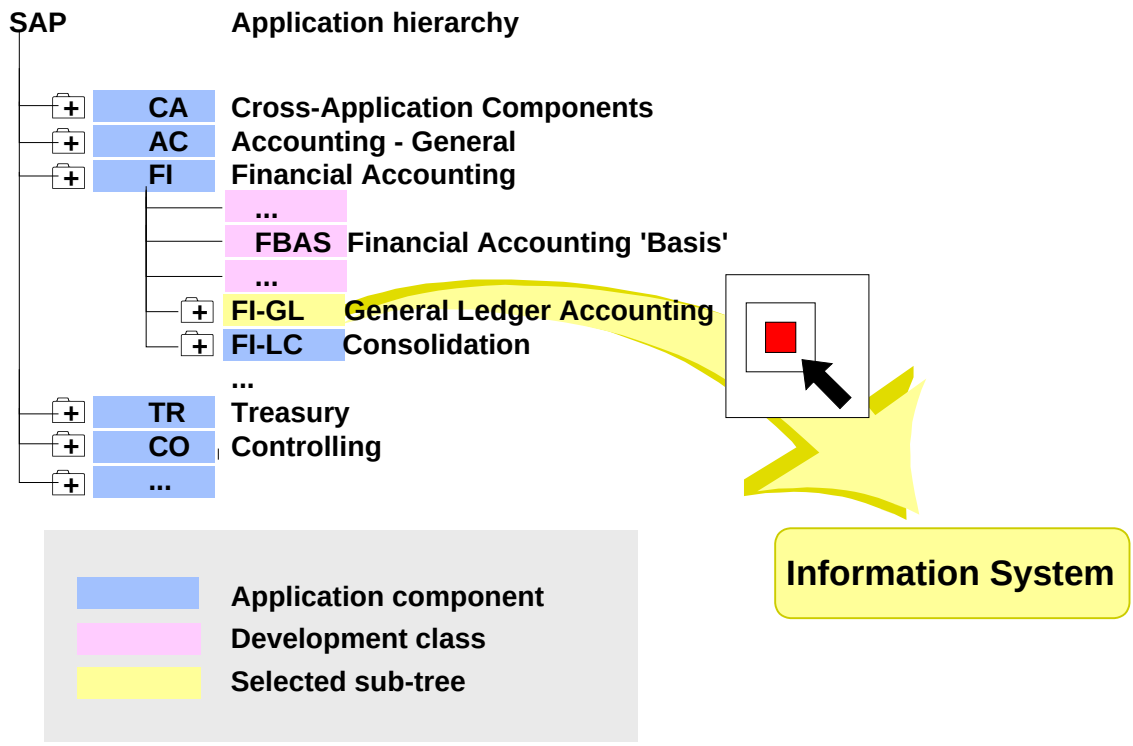
## Repository Structure






- All development objects created with the development tools found in the ABAP Workbench are classified as **Repository objects** and are stored centrally in the **R/3 Repository**.
- The R/3 Repository is a special part of the SAP system's central database.
- The Repository is organized by application. Each application is further divided into logical subdivisions called **development classes**.
- Repository objects are often made up of sub-objects that are themselves Repository objects.
- Each Repository object must be assigned to a development class when it is created.
- You can use the Repository **Information System** to search for **Repository** objects by various criteria.

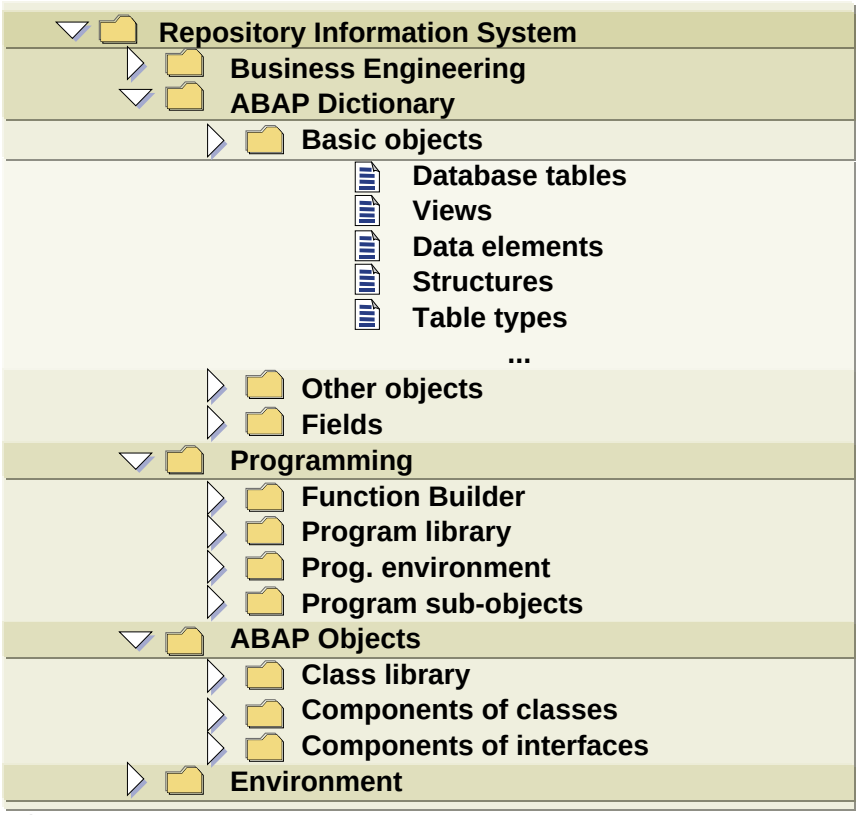
## SAP Application Hierarchy






- You can view the Repository structure in the application hierarchy. You can navigate to the application hierarchy from the initial screen using *Tools -> ABAP Workbench -> Overview -> Application Hierarchy*. (Transaction SE81).
- The application components are displayed in a tree structure in the application hierarchy. Expanding a component displays all the development classes that are assigned to that component.
- You can select a sub-tree and navigate from the application hierarchy to the *Repository Information System*. The system then collects all development classes for the sub-tree selected and passes them to the Information System.

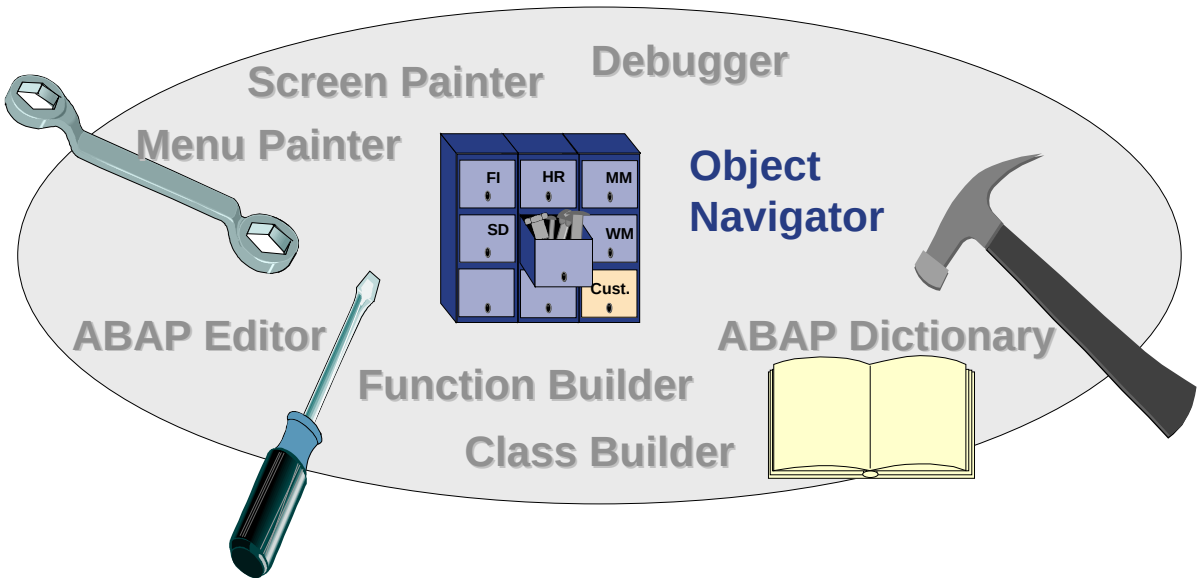
# Repository Information System





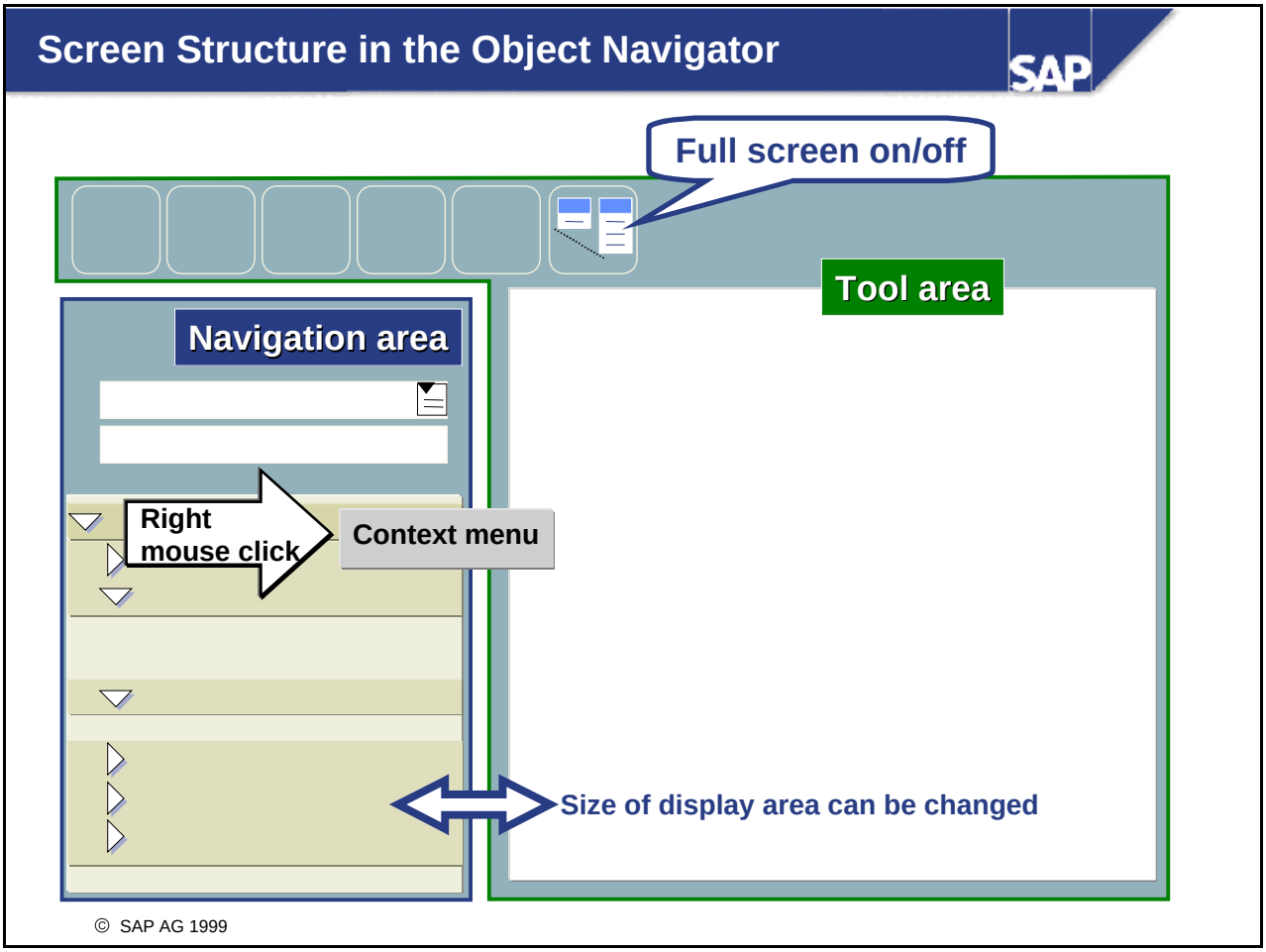

- You can use the *Repository Information System* to search for specific Repository objects. Search criteria are available for the various kinds of Repository objects.
- You can navigate to the Repository Information System using
  - The *Information system* pushbutton in the application hierarchy
  - The menu path *Tools -> ABAP Workbench -> Overview -> Information System*
  - Transaction SE84 in the command field.

## ABAP Workbench Tools

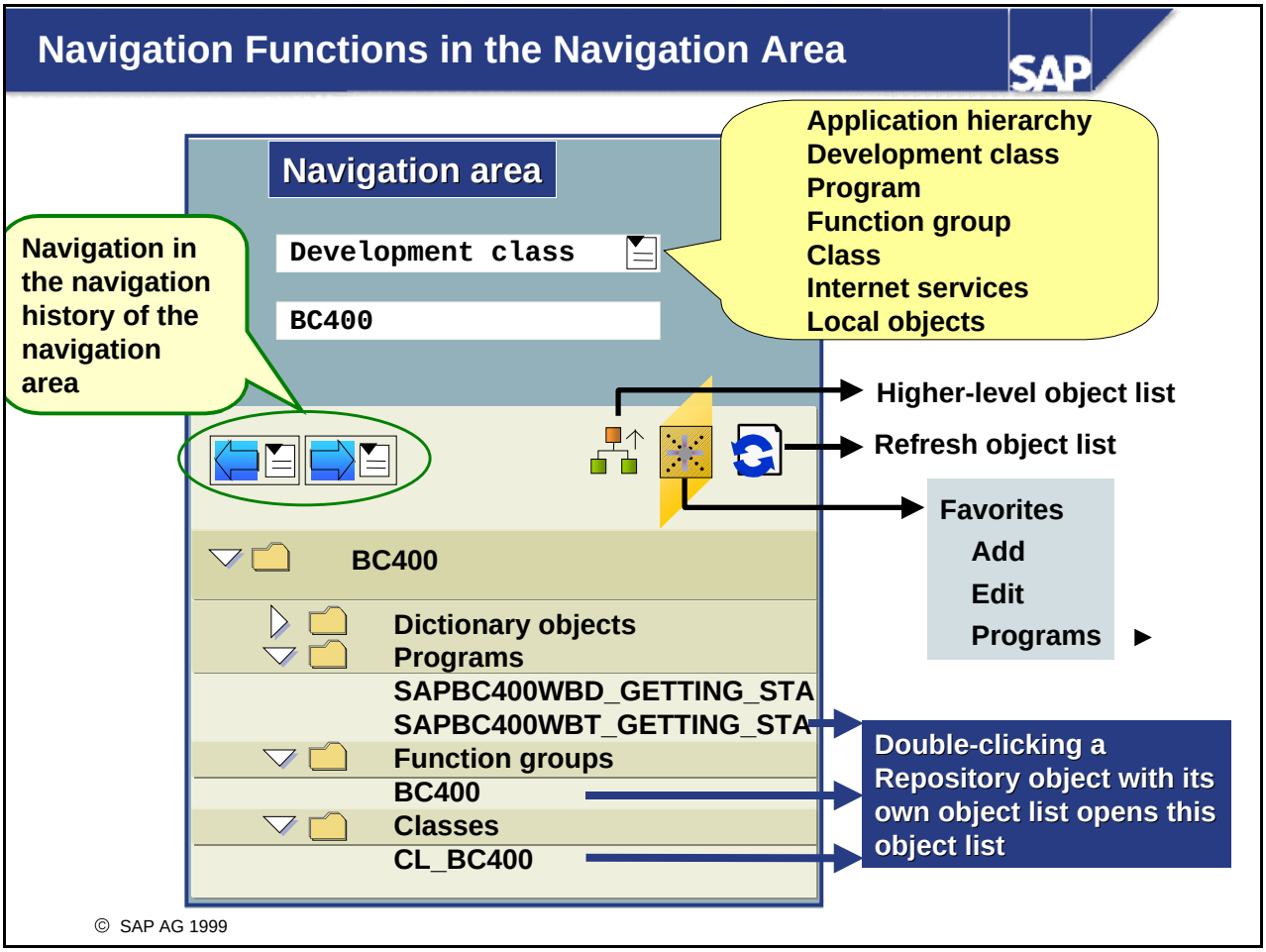




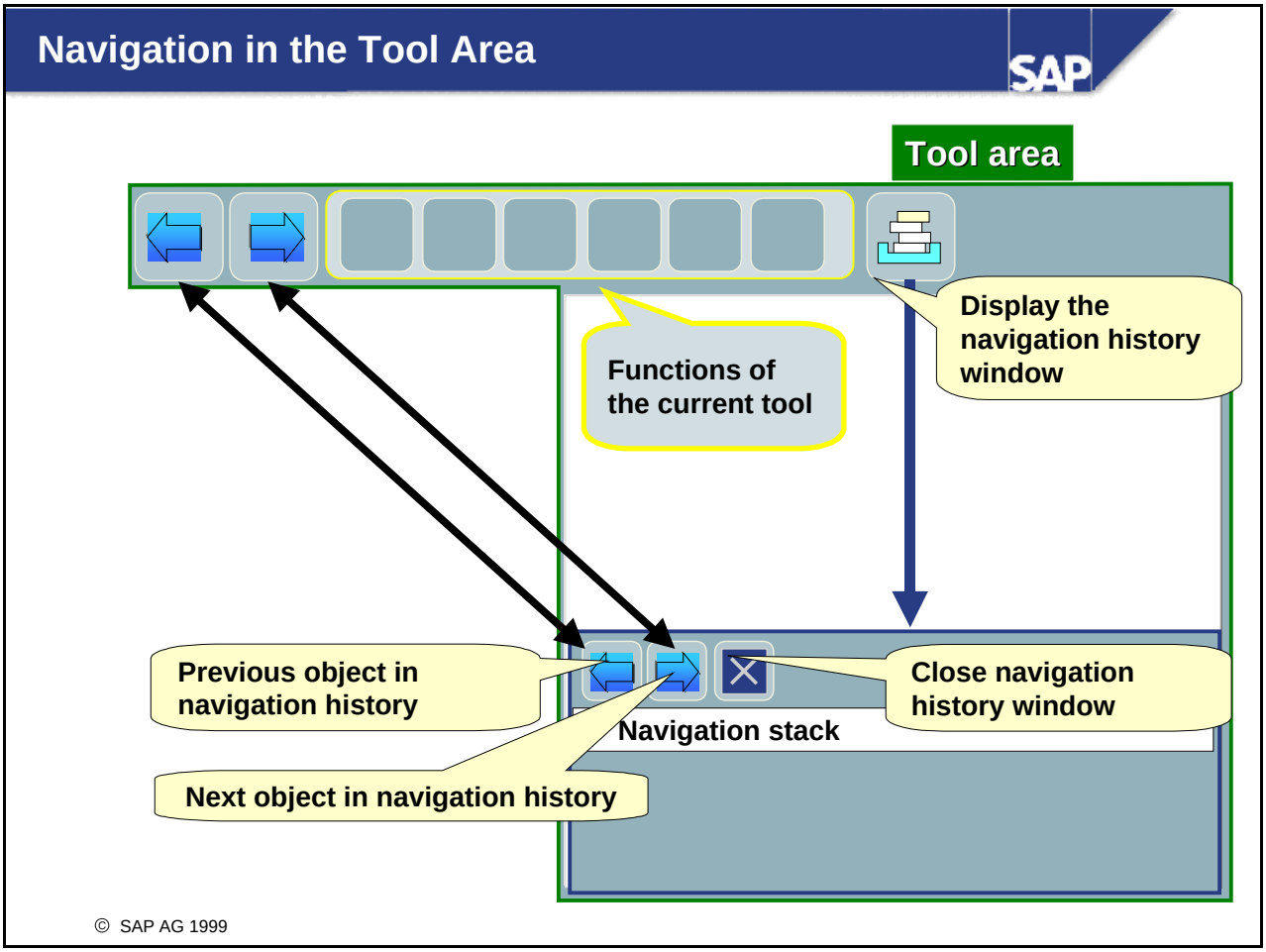

- The ABAP Workbench contains different tools for editing Repository objects. These tools provide you with a wide range of assistance that covers the entire software development cycle.
  The most important tools for creating and editing Repository objects are:
- **ABAP Editor** for writing and editing program code
- **ABAP Dictionary** for processing database tables and retrieving global types
- **Menu Painter** for designing the user interface (menu bar, standard toolbar, application toolbar) (see *Interfaces)*
- **Screen Painter** for designing screens (**dynamic programs**) for user dialogs
- **Function Builder** for displaying and processing function modules (subroutines with defined interfaces that are available throughout the system)
- **Class Builder** for displaying and processing central classes
- There are two different ways to go about using these tools:
  - Either you call each individual tool and edit the corresponding Repository objects.
    You must then call the next tool for the next set of objects...
  - Or you work with the **Object Navigator**: This transaction provides you with a tree-like overview of all objects within a development class or program.




- The Object Navigator screen is divided into two areas:
  - A navigation area for displaying an object list as a hierarchy
  - A tool area, for displaying and editing a development object using the correct tool
- You can hide the navigation area using the *Full screen on/off* pushbutton.
- You can select functions from a context menu in both screen areas. You are only given a choice of those functions that are relevant to displaying or editing the object on which the cursor is positioned. Right-click with the mouse to display the context menu. (Left-click if you have set up your mouse for left-handers).

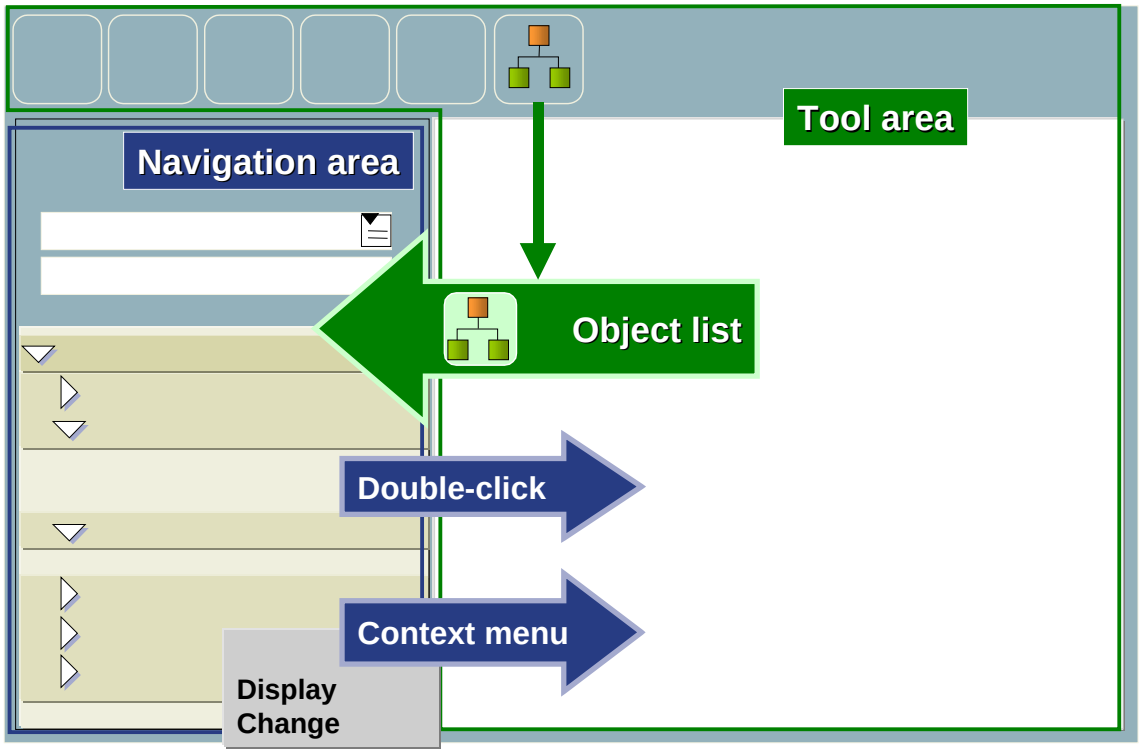



- Repository objects are organized in a hierarchy:
  - Each application component consists of multiple development classes
  - Each development class can contain several different kinds of Repository objects: programs, function groups, ABAP Dictionary objects, ...
  - Each Repository object can consist of different object types:
    - Programs can contain: global data, types, fields, events, ...
    - Function groups can contain: global data, function modules, ...
- You can enter the type of object list and the object name in the upper part of the navigation area. The object list is then displayed in the navigation area.
- Double-clicking on a sub-object in an object list displays the object list for the selected object in the hierarchy area.
- Double-clicking on an object that does not have an object list displays that object in the object window.
- You can use the icons to navigate by history or hierarchy between the object lists.
- You can add object lists that you edit frequently to your favorites.




- You can display a window showing your navigation history. This window displays a list of the objects that you have displayed since you opened the Object Navigator in the tool area.
- You can navigate backwards using the left arrow (blue) in the tool bar in the navigation window, or in the pushbutton bar. The object currently displayed in the tool area is highlighted in a different color.
- You can also navigate "forwards in history" (as in the Internet Explorer) using the blue right arrow in the navigation window tool bar or in the pushbutton bar. This type of navigation is only possible if you have navigated backwards before.

## Synchronizing the Navigation Area and the Tool Area



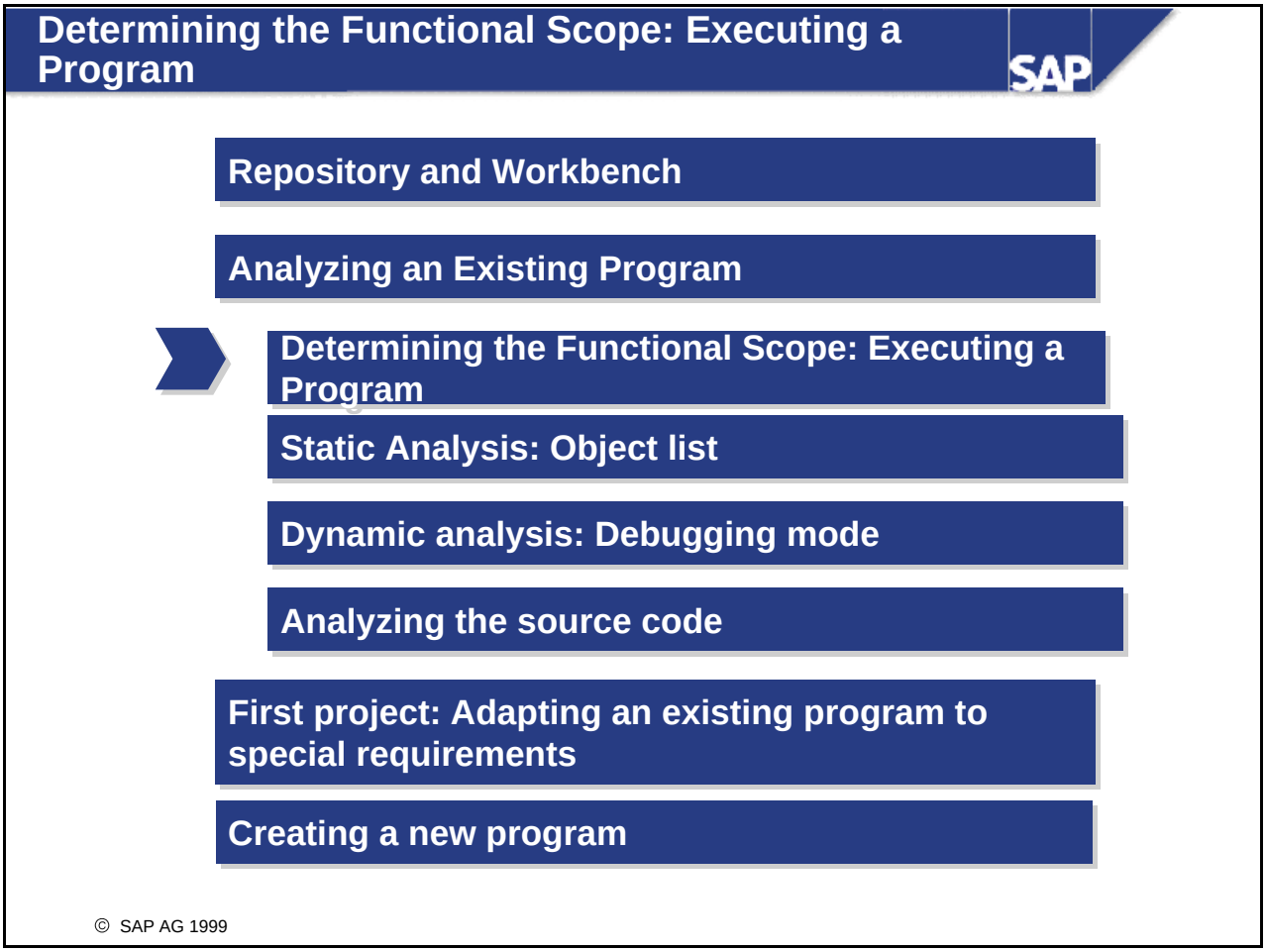



- Navigation in the navigation area is logically independent from navigation in the tools area. This allows you to navigate flexibly. If you want to, you can synchronize the two areas:
  - In the navigation area, if you want to display **the correct object list** for an **object** that is currently being displayed in the tool area, use the *Object list* icon.
  - You can display a **development object from an object list** in the tools area by double-clicking or using the context menu. The system then automatically selects the correct tool for processing the object selected.
- To **create objects** from an object list, you can use the context menu for that object type. If that object type is not available in the object list, you can create any object using the *Edit object* or *Any object*.

# Analyzing an Existing Program

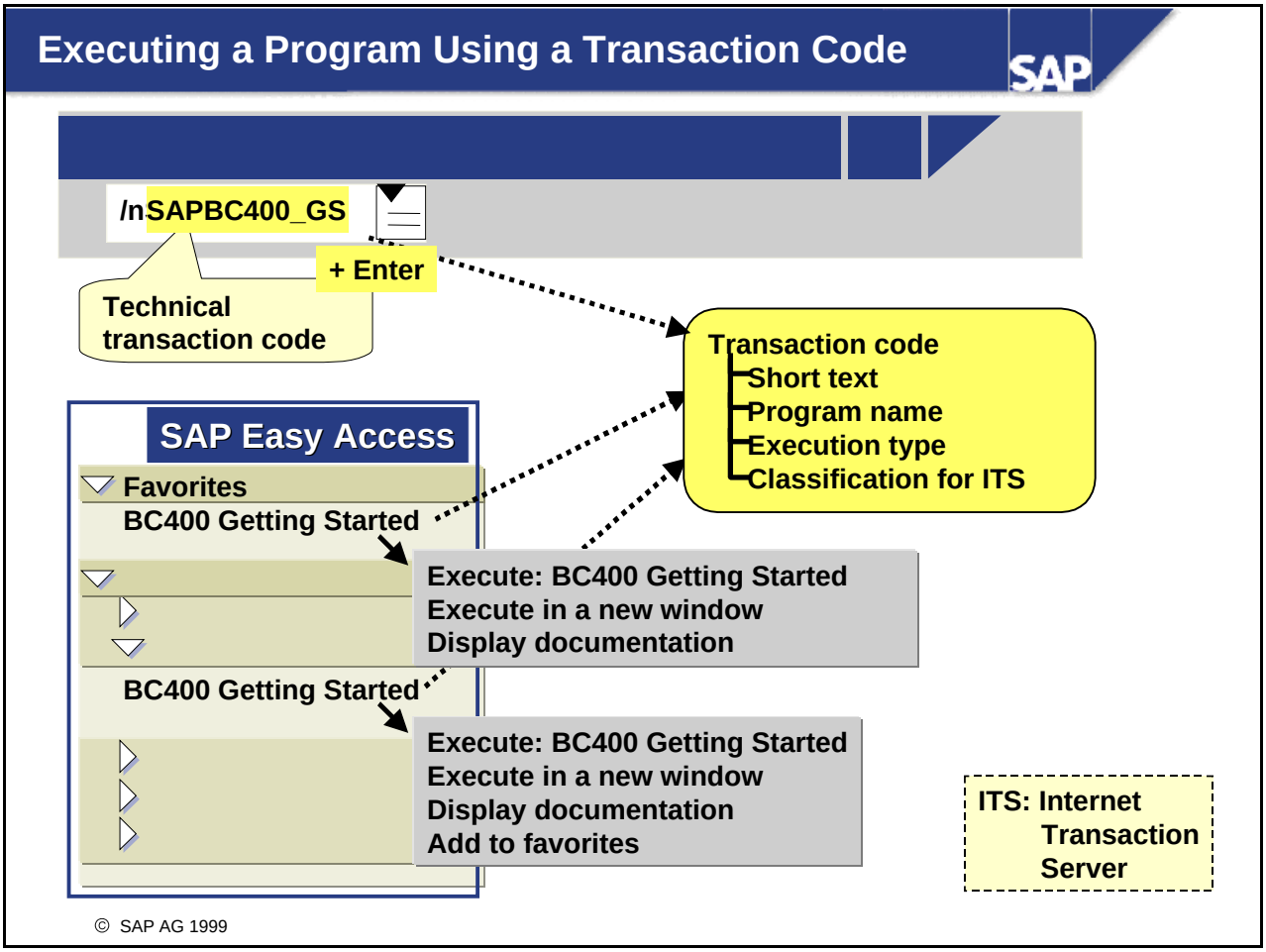
- Repository and Workbench
- Analyzing an Existing Program
  - Determining the Functional Scope: Executing a Program
  - Static Analysis: Object list
  - Dynamic analysis: Debugging mode
  - Analyzing the source code
- First project: Adapting an existing program to special requirements
- Creating a new program

## Determining the Functional Scope: Executing a Program

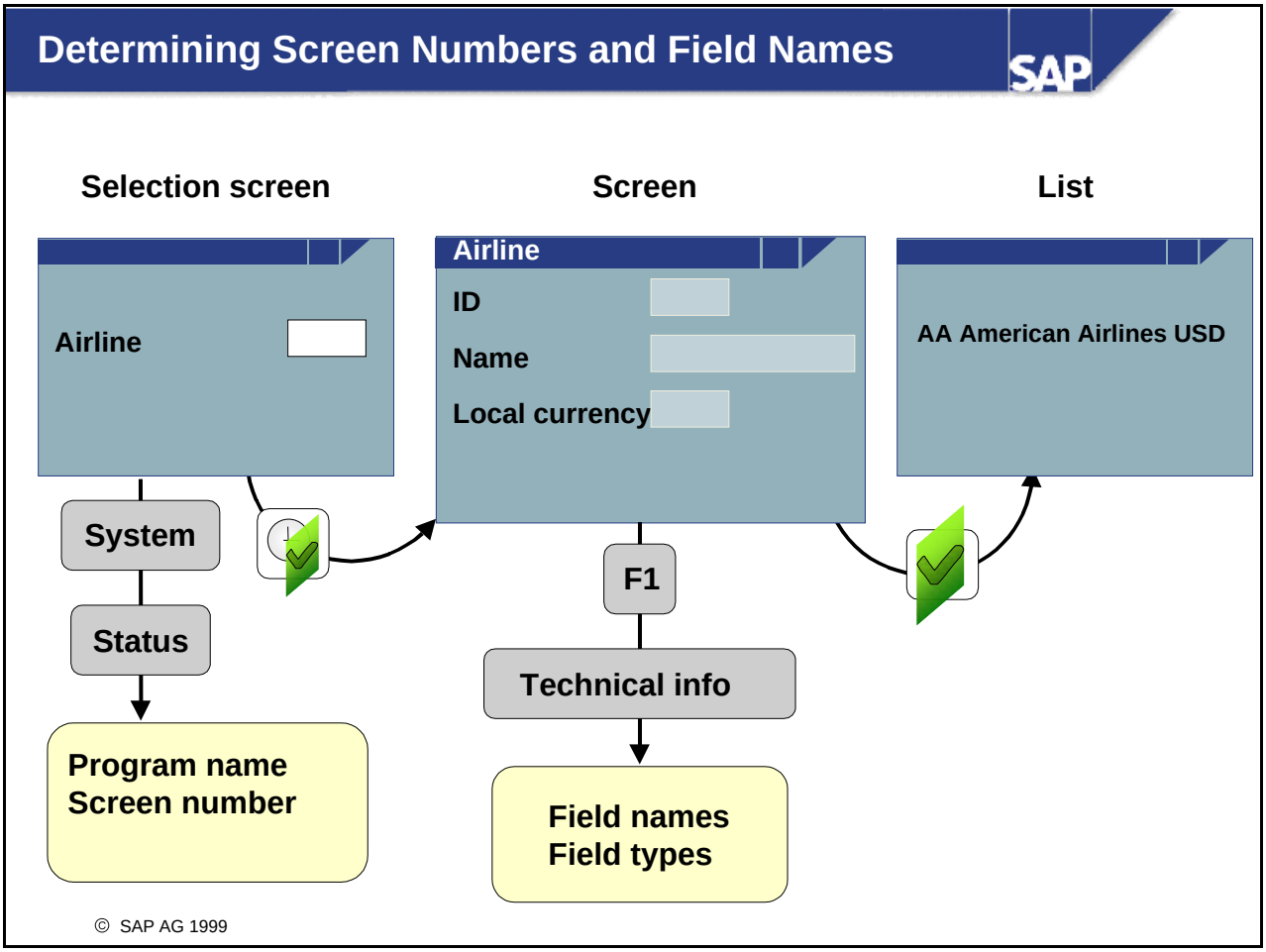





- We will start by answering the question "What information can I find about a program, without using the ABAP Workbench?" This is important whenever you know of a function, which you can start either using the menu or from your role, but you do not know the program name.




- There are various ways of starting a program:
  - From *SAP Easy Access*. As a user, you will have a menu, defined specifically for your role, that allows you to access all the functions and programs you need for your work. You can store programs that you use a lot in your *Favorites*.
    When you choose an entry in the tree, you can start the appropriate program using the context menu. In the background, the system uses a transaction code to achieve this. You can also display the transaction codes in the tree by choosing *Utilities -> Settings* and checking *Show Technical Names*.
  - If you know the transaction code, you can start the program by entering its code in the input field in the toolbar and choosing *Enter.* You may need to stop the current program running first. You can do this by choosing the yellow arrow icon, or by entering '/n' followed by *Enter* in the input field in the toolbar.
- Associated with each transaction code there is a short text, (which appears in your role tree) along with the program that will be started, and the execution type of that program. You can add additional transactions to your favorites, provided you have the necessary authorizations.
- Staring programs by choosing a menu entry follows the same principles. This is often used in the ***Goto*** or ***Environment*** menus. For detailed information on menu bar and tool bar elements, see the ***Interfaces*** chapter.

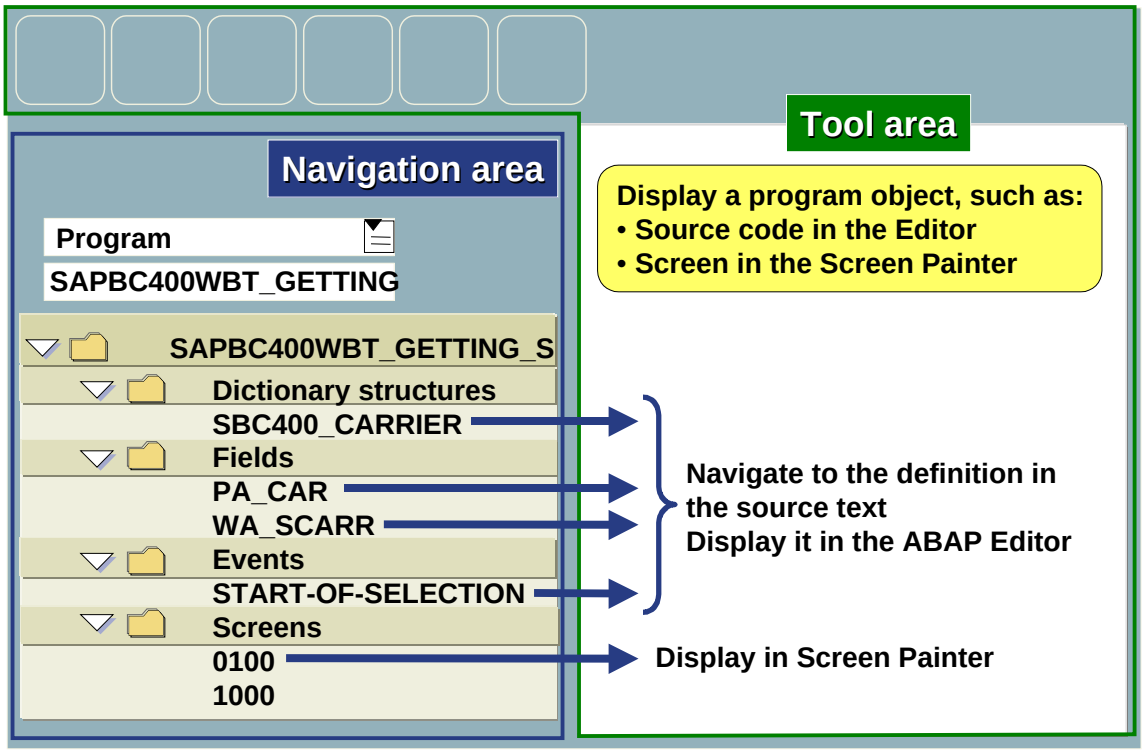



- There are various ways of starting a program:
  - You can start a program from the Object Navigator object list using the context menu or using the 'Test' icon.
  - If the program has a transaction code, then this can be added to a menu. Then all you have to do is click on the menu option with the mouse.
  - You can add programs to the favorites list on the initial screen. Programs can also be made available using the activity groups on the initial screen. Then all you have to do is select the program in the hierarchy on the initial screen.
- You can determine the functional scope by executing the program.
- On any screen, you can access information about the program name and the screen number using *System -> Status.* A standard selection screen has the screen number 1000.
- You can access information on the field name and field type for any field on the screen using *F1 -> Technical Info.*

# Static Analysis: Object list

- Repository and Workbench
- Analyzing an Existing Program
  - Determining the Functional Scope: Executing a Programs
  - **Static Analysis: Object list**
  - Dynamic analysis: Debugging mode
  - Analyzing the source code
- First project: Adapting an existing program to special requirements
- Creating a new program

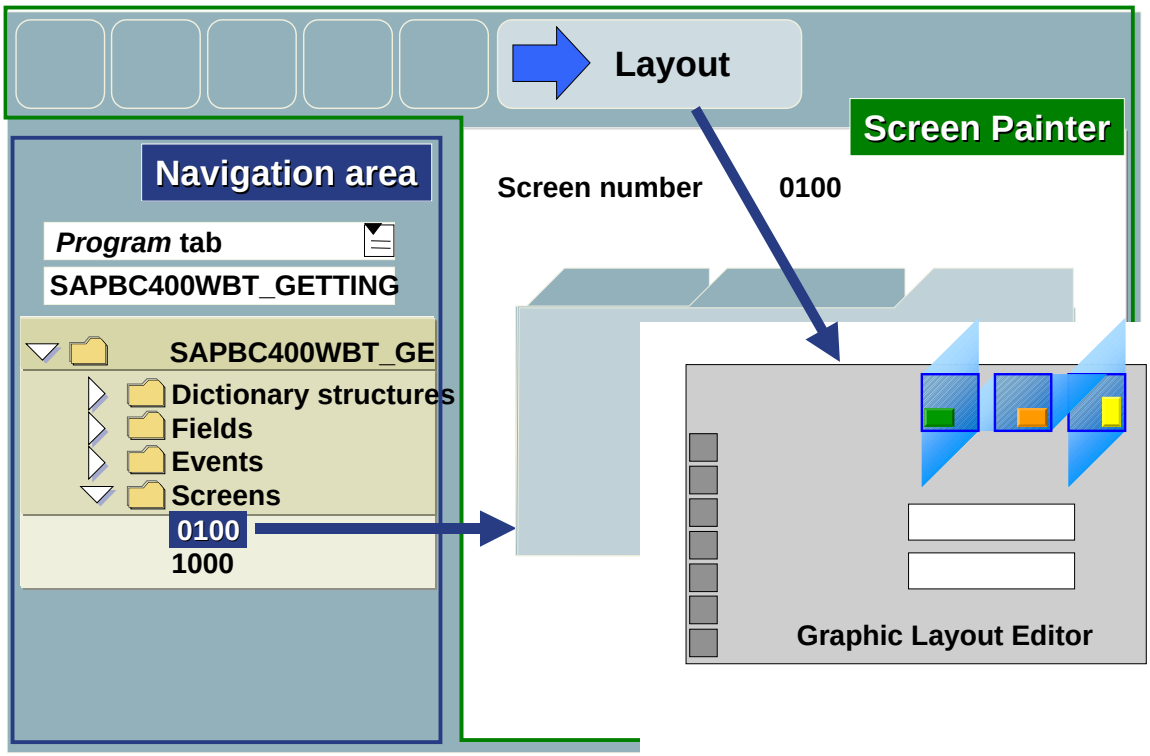
## Static Analysis in the Object Navigator






- You can display an overview of the program objects using the program object list in the Object Navigator. To display the object list for a program, choose *Program* from the first input field; in the second, enter the program name. Then, to display the object list, click the *Display* icon (a pair of spectacles) or choose *Enter.*
- The hierarchy only shows those object types for which objects exist.
- You can display the objects in the Object Navigator details window by double-clicking or using the context menu.
- You can find out about the environment in which each object in the list is used. From the context list for the object, choose *Where-Used List*. The system displays a list showing all the places where the object is used. Double-click an entry in this list to navigate to it.

## Example: Displaying Screen 100 in the Screen Painter





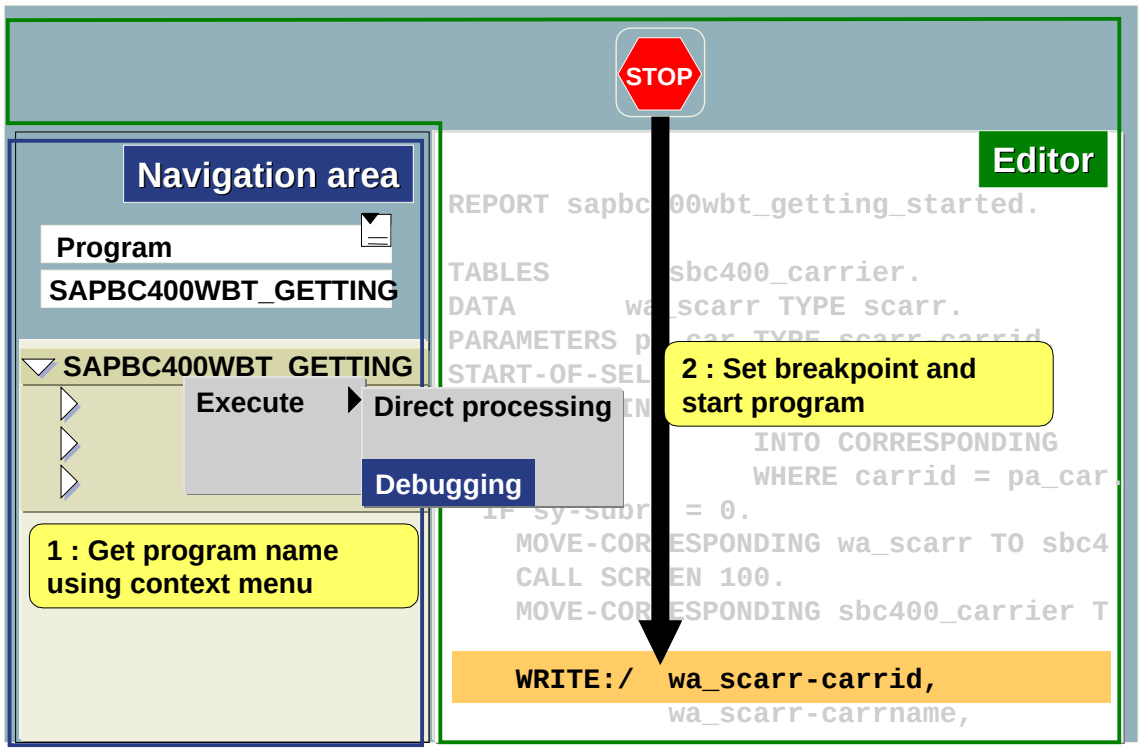

- We will be dealing with the Editor and source code in detail later, so for the moment we shall restrict ourselves to displaying a screen as an example.
- As we have already seen in an example program, a "screen" (or "dynamic program") is a in fact a technique for creating screens for user dialogs. Each screen is identified by a number of up to four digits. They appear in the program object list under the *Screens* node.
- You can display a screen in the tool area either by using the context menu for a particular screen number or by double-clicking that screen number. The tool you use to edit screens is called the Screen Painter. The following information is stored for each screen:
  - **Screen attributes:** such as whether the screen should fill the monitor, or be a modal dialog window.
  - **Layout information:** which screen objects should be displayed on the screen and in which position? An input field is a simple example of a screen object. For each input field, you must enter a type to specify the length of the input field and the value range for automatic type checks.
  - **Flow logic:** allows you to call the ABAP modules that the system executes before and after it displays the screen.
  - For more information, see the *User Dialogs: Screens* dialog.
- You can avoid recording layout information in text form by using the **graphic Layout Editor**. The system displays a portrayal of the layout of the screen. You then generate screen objects by dragging and

dropping special icons. To open the graphic Layout Editor, choose the icon with the blue arrow and the word *Layout*.

# Dynamic analysis: Debugging mode

- Repository and Workbench
- Analyzing an Existing Program
  - Determining the Functional Scope: Executing a Program
  - Static Analysis: Object list
  - **Dynamic analysis: Debugging mode**
  - Analyzing the source code
- First project: Adapting an existing program to special requirements

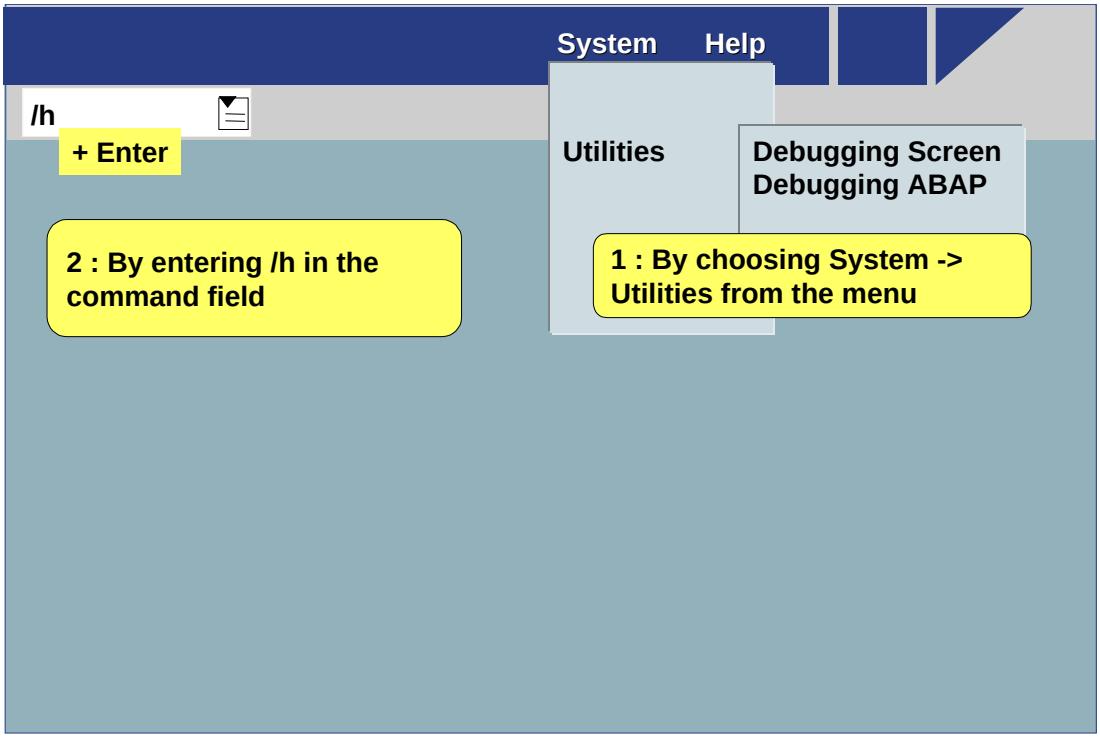
## Starting a Program in Debugging Mode






- There are several ways to start a program in Debugging mode, without having to change the program:
  - Using the context menu for a program name in the program object list, choose *Execute -> Debugging.*
  - Using a breakpoint in the Editor: You can select a point in the program that the runtime system in Debugging mode should switch to. To do this, navigate to the line in the program, select it, and choose the *Breakpoint* icon (that is, the STOP sign). Then start the program, either by choosing *Execute -> Direct processing* or F8.

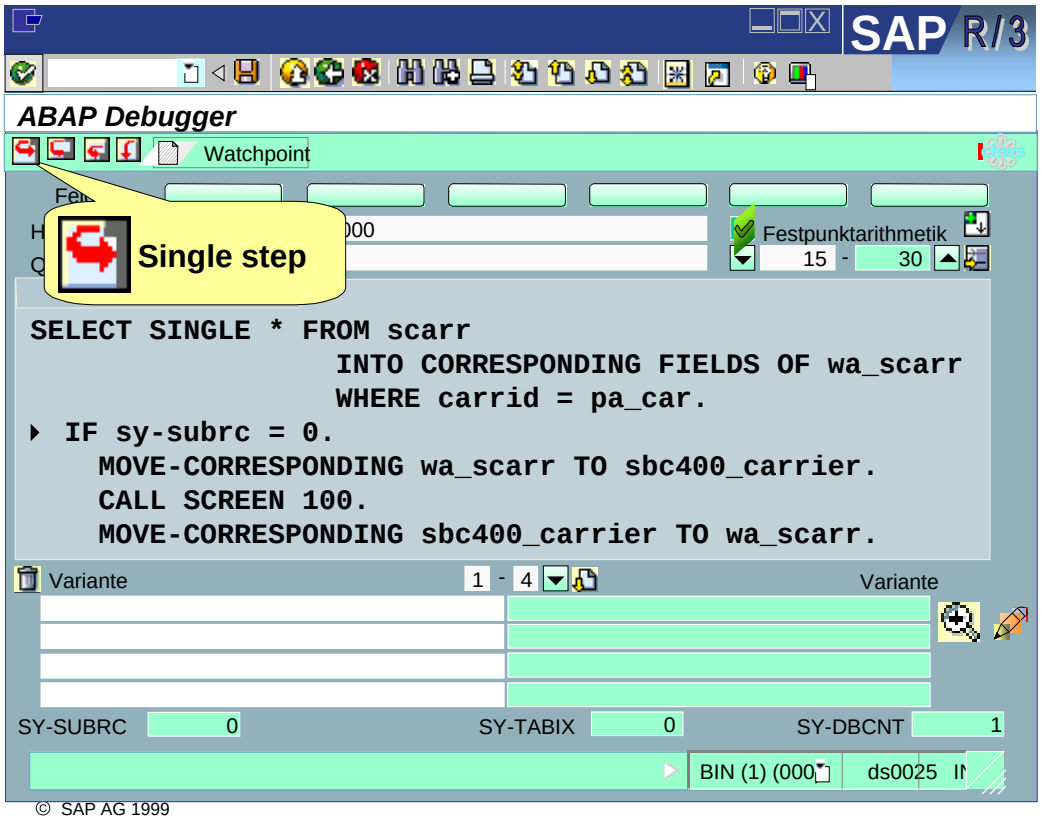
## Switch to Debugging Mode at Runtime





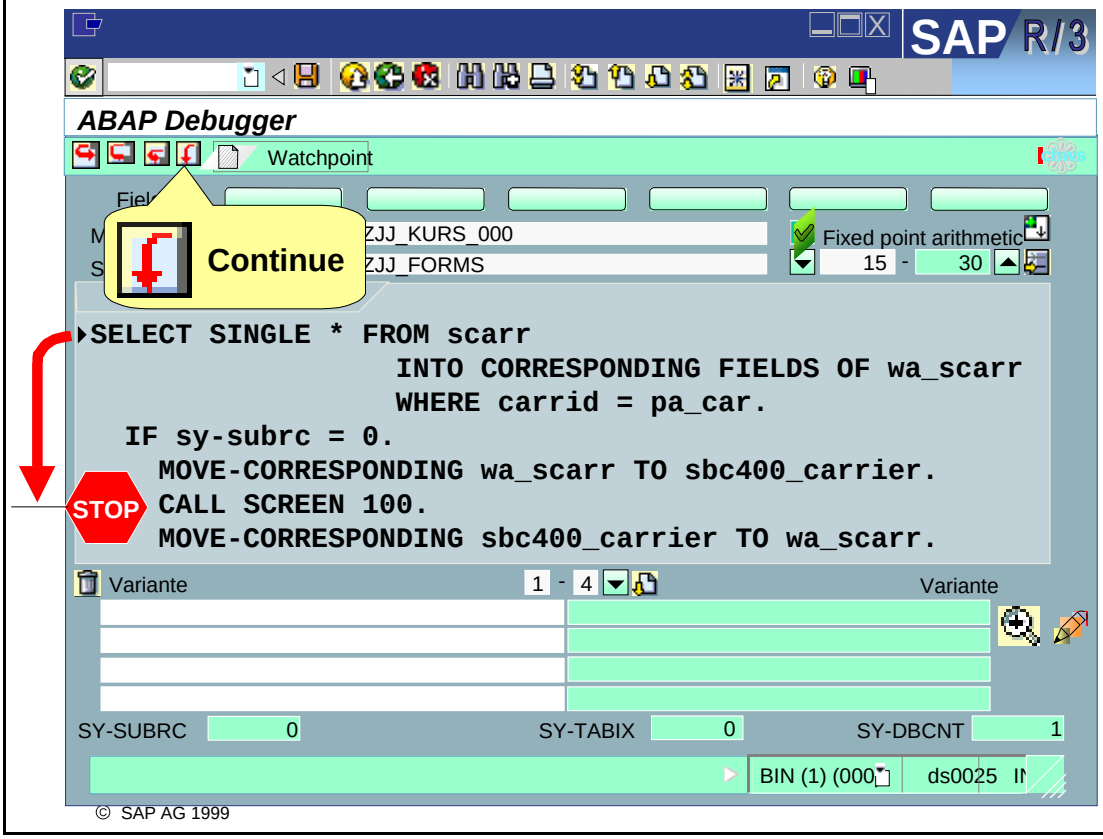

- You can switch to Debugging Mode at runtime:
  - Choose *System -> Utilities -> Debugging Screen* to debug the screen.
  - Choose *System -> Utilities -> Debugging ABAP* to debug the ABAP code.
- You can also switch to Debugging mode by typing /h in the command field in the tool bar, followed by *Enter.*

# Investigating the Behavior of ABAP Programs at Runtime: Breakpoints in the Debugging Mode




- Starting the program in Debugging mode allows you to execute the program line by line using the *Single Step* icon. You can display up to eight variables. To trace the variable values, enter the field names in the left input field. You can also see this entry by double-clicking the field name in the code displayed.

# Breakpoints in the Debugging Mode




- You can set a breakpoint by double-clicking in front of a line of source code in the debugging mode. If you then click on the *Continue* icon, the program will be executed up to the point where the next breakpoint is defined.
- You can find information on content-related breakpoints in the *ABAP Statements and Data Declarations* unit.

# Analyzing the Source Code

- Repository and Workbench
- Analyzing an existing program
  - Determining the functional scope: executing a program
  - Static Analysis: Object list
  - Dynamic analysis: Debugging mode
  - Analyzing the source code
- First project: Adapting an existing program to special requirements
- Creating a new program

## General ABAP Syntax: Key Words

```
TABLES      sbc400_carrier .
DATA       wa_scarr TYPE scarr .
PARAMETERS  pa_car TYPE scarr-carrid .

START-OF-SELECTION .
  SELECT SINGLE * FROM scarr
                  INTO CORRESPONDING FIELDS OF wa_scarr
                  WHERE carrid = pa_car .
  IF sy-subrc = 0 .
    MOVE-CORRESPONDING wa_scarr TO sbc400_carrier.
    CALL SCREEN 100 .
    MOVE-CORRESPONDING sbc400_carrier TO wa_scarr.

    WRITE :  wa_scarr-carrid ,
             wa_scarr-carrname ,
             wa_scarr-currcode .
  ENDIF.
```

Chained statement

ABAP key word

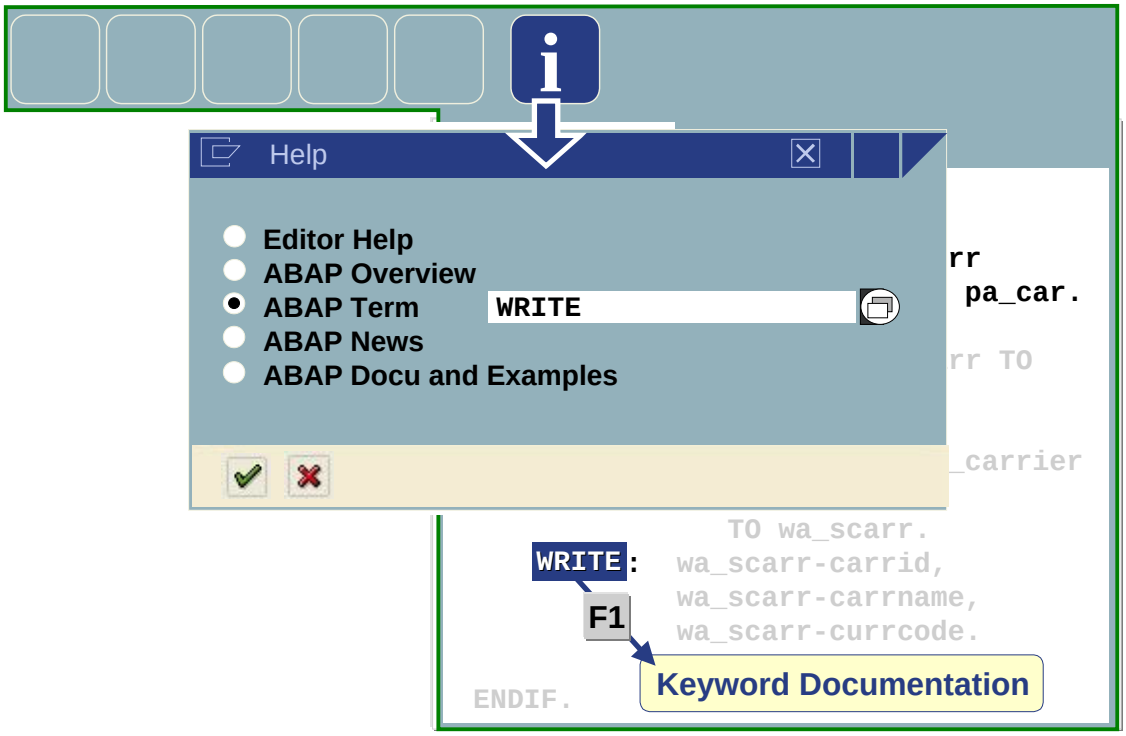
**Additions (depending on keyword)**

Period (ends all ABAP statements)



- ABAP programs are made up of individual statements.
- Each statement ends with a period.
- The first word in a statement is called a keyword.
- Words must always be separated by at least one space.
- Statements can be indented.
- Statements can take up more than one line.
- You may have multiple statements in a single line.
- Consecutive statements **with identical initial keywords** can be condensed into one chained statement:
  - In chained statements, the initial part of the statement containing the keyword must be followed by a colon.
  - Individual elements that come after the colon must always be separated by commas.
  - Blank spaces are allowed before and after all punctuation (colons, commas, periods).
  - Be aware that the system still considers the individual parts of a chained statement to be complete statements that are independent of one another.

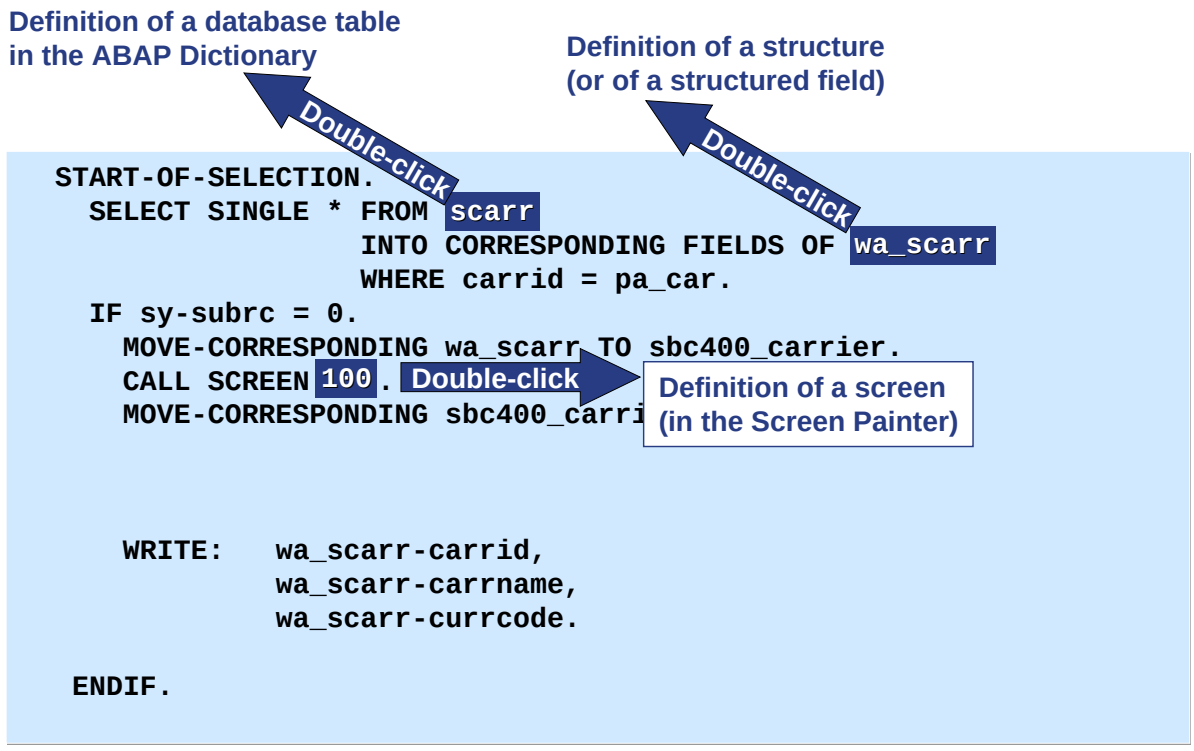
# Keyword Documentation in the Editor






- There are various ways of navigating to keyword documentation for an ABAP statement:
  - ***F1*** on a keyword displays the documentation for the statement on which the cursor is positioned.
  - The ***Information*** icon displays a dialog box offering you various views of the keyword documentation.

## Navigation in the Editor: Double-Clicking






- You can display detailed information on single objects in the Editor by double-clicking:
  - **Double-clicking** the name of a **database table** displays the database table definition using the ABAP Dictionary in the object window of the Object Navigator.
  - **Double-clicking** a **field name** displays the part of the program source code where the data object is defined.
  - **Double-clicking** a **screen number** displays the screen using the Screen Painter in the object window of the Object Navigator.
- Use the *Back* function to get back to the program source code display in the Editor.

## Comments

```
START-OF-SELECTION.
* Read data record from database table SCARR

  SELECT SINGLE * FROM scarr
                  INTO CORRESPONDING FIELDS OF wa_scarr
                  WHERE carrid = pa_car.
  IF sy-subrc = 0.
    MOVE-CORRESPONDING wa_scarr TO sbc400_carrier.
    CALL SCREEN 100.  " Process screen 100
    MOVE-CORRESPONDING sbc400_carrier TO wa_scarr.

    WRITE:   wa_scarr-carrid,
             wa_scarr-carrname,
             wa_scarr-currcode.
  ENDIF.
```

Comment takes up whole line

Comment on rest of line



- There are two ways to insert comments into a program:
  - A star (*) in column 1 allows you to designate the whole line as a comment.
  - Quotation marks (") in the middle of a line designate the remainder of the line as a comment.

## Analyzing the Sample Program Source Code

**Definitions:**

```
TABLES       sbc400_carrier.
DATA        wa_scarr TYPE scarr.
PARAMETERS  pa_car TYPE scarr-carrid.
```

1

**Executable Source Code**

```
START-OF-SELECTION.
  SELECT SINGLE * FROM scarr
                  INTO CORRESPONDING FIELDS OF wa_scarr
                  WHERE carrid = pa_car.
  IF sy-subrc = 0.
    MOVE-CORRESPONDING wa_scarr TO sbc400_carrier.
    CALL SCREEN 100.
    MOVE-CORRESPONDING sbc400_carrier TO wa_scarr.

    WRITE:   wa_scarr-carrid,
             wa_scarr-carrname,
             wa_scarr-currcode.
  ENDIF.
```

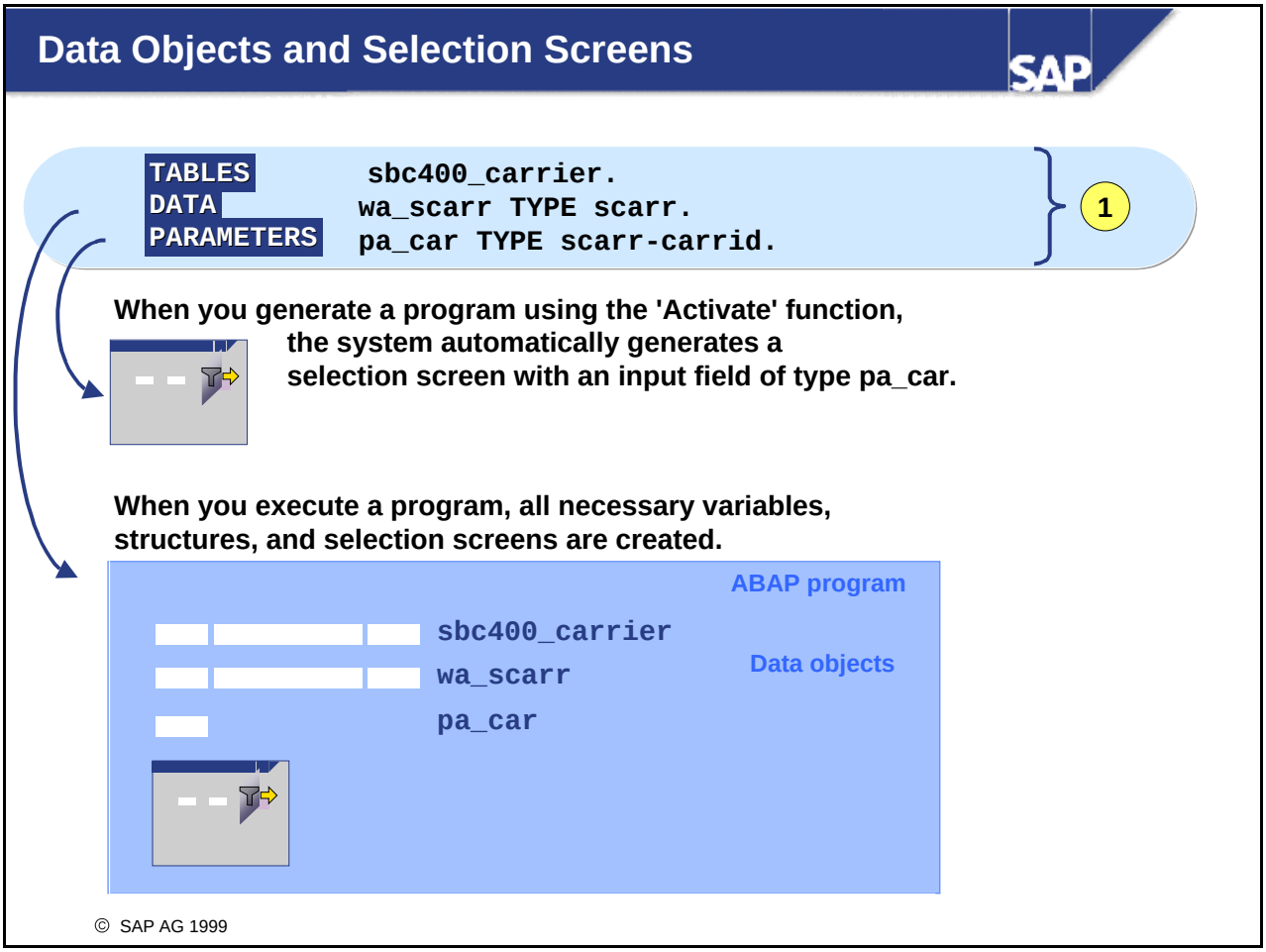
2

3

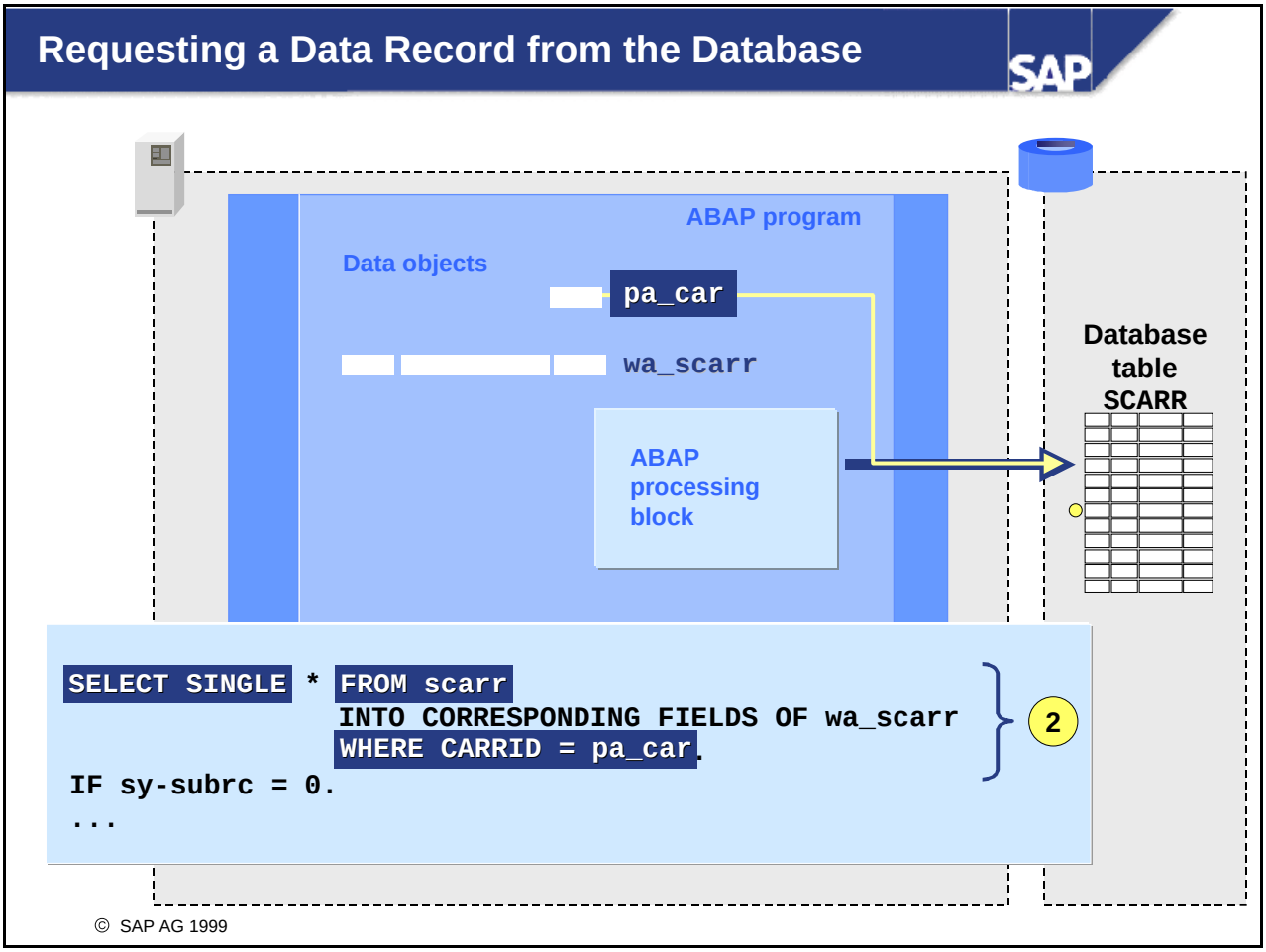
4



- If you need more precise information on parts of the source code, you can analyze it. The following slides explain the most important statements in the sample program.

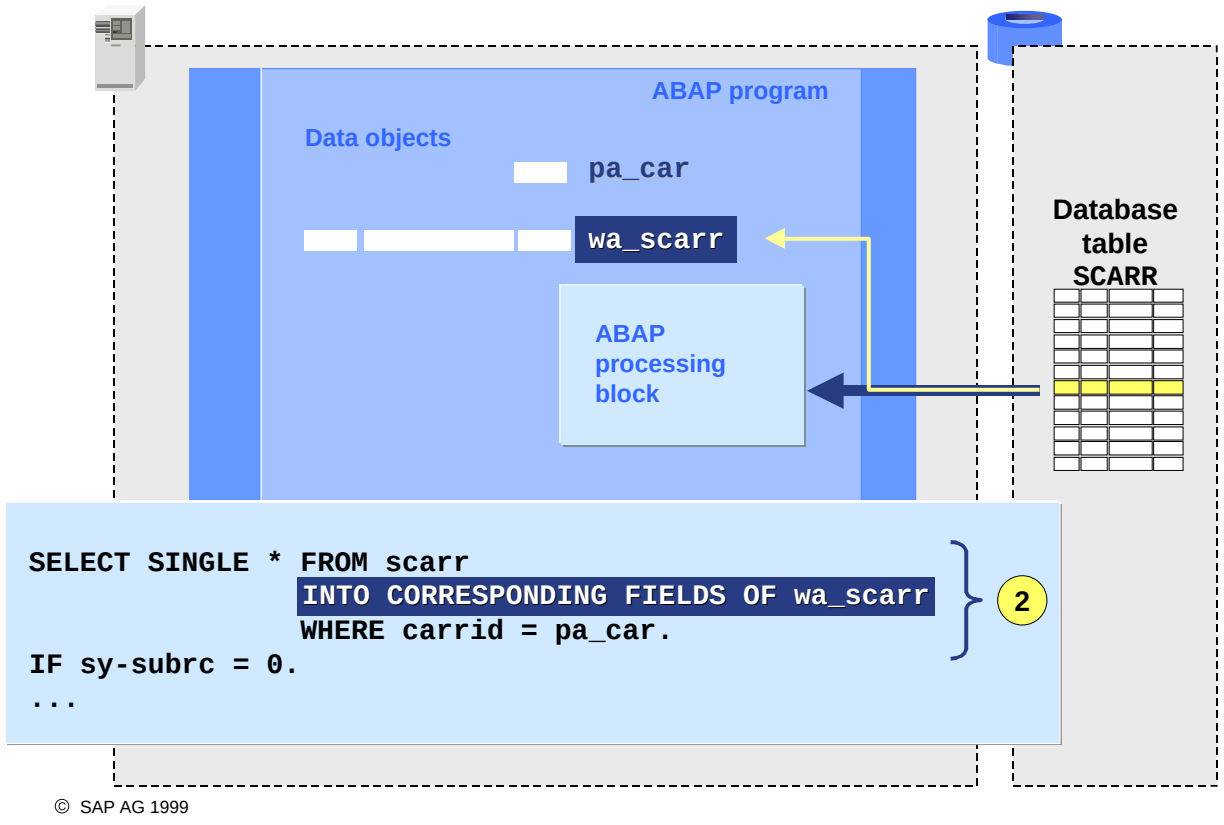



- There are various statements that you can use to define data objects:
  - The **TABLES** statement always refers to the global type of a flat structure that is defined in the ABAP Dictionary. The structure type for the data object in the program is taken from the Dictionary. The data object name is identical to the name of the structure type. **TABLES** structures are stored technically slightly differently to local data objects that are defined using the **DATA** statement. They are normally used as a structure for the interface to the screen.
  - The **DATA** statement is usually used to define local data objects. The data object type is specified using the **TYPE** addition.
  - The **PARAMETERS** statement defines not only an elementary data object, but also an input field on the standard selection screen that is processed at the start of the program.
- When you activate a program, an internal load version is generated. A selection screen is generated from the **PARAMETERS** statement. When the program starts, memory areas are made available for the data objects.
- You can find further information on data objects in the unit entitled *ABAP Statements and Data Declarations*, or in the keyword documentation.



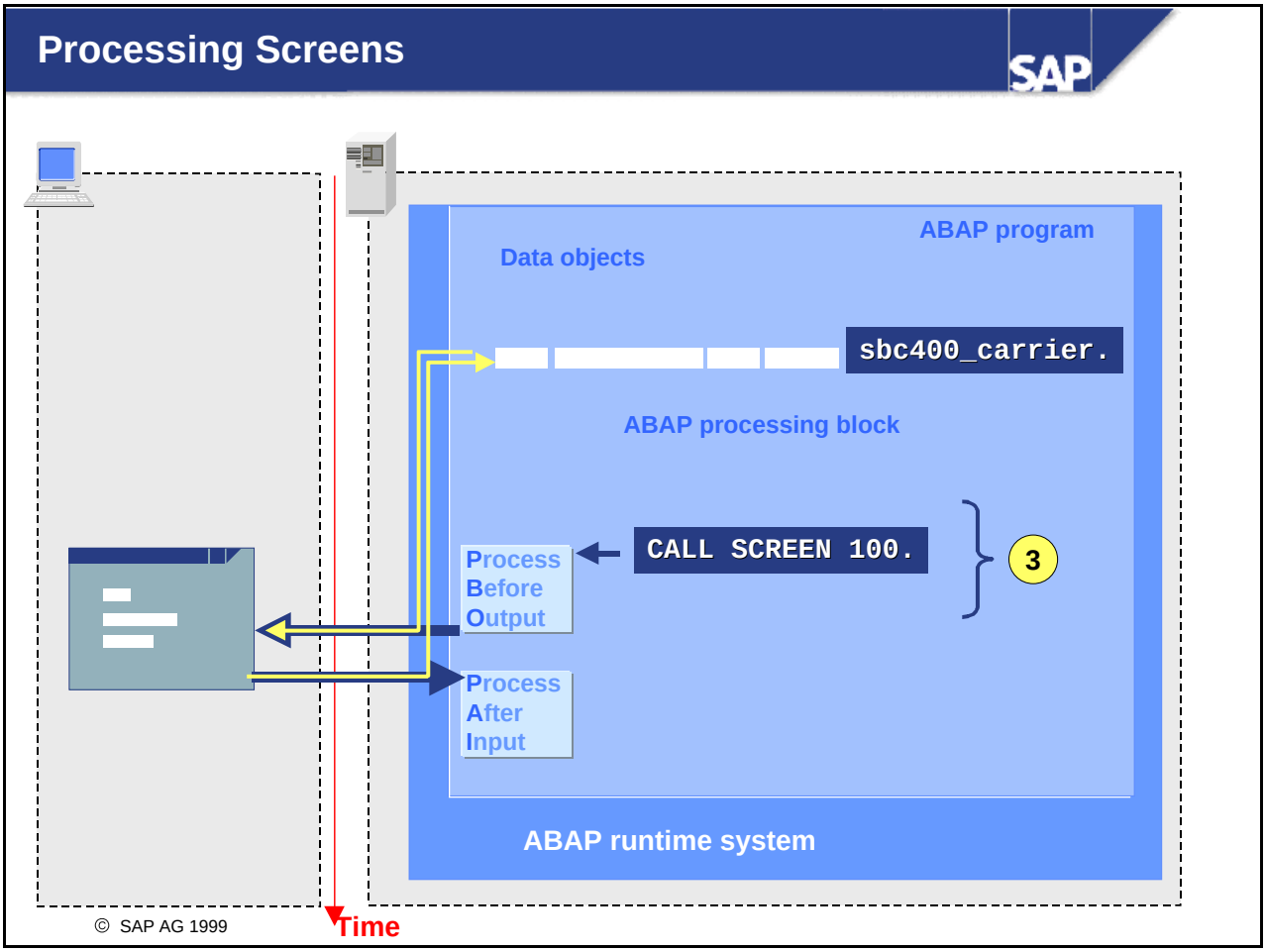

- The **SELECT** statement ensures that data is read from the database. In order to read a record from a database table, the following information must be passed to the database:
  - From which database table is the data read? (**FROM** clause)
  - How many lines are read? The **SINGLE** addition shows that only one line is read.
  - Which line is read? The **WHERE** clause shows which columns of the database table have which values. For a **SELECT SINGLE**, the condition must be formulated so that one line is specified unambiguously.

# Receiving the Results of a Query






- The data supplied by the database is stored in local data objects. The **INTO** clause specifies the data objects into which you want to copy the data. In this example, the data is copied to the components of the same name in structure **wa_scarr**.




- The statement **CALL SCREEN** calls a **screen**.
- A screen must be created using the **Screen Painter** tool.
- A screen is an independent Repository object, but belongs to the program.
- You can define input fields on a screen that refer to the ABAP Dictionary. Screens automatically perform consistency checks on all input and provide any error dialogs that may be needed. Thus, screens are more than just templates for entering data, they are, in fact, **dynamic programs (dynpros)**.
- The statement **TABLES** defines a structure object that serves as an interface for the screen. All data from this structure is automatically inserted into its corresponding screen fields whenever **CALL SCREEN** is called. Data entered by the user on the screen is transferred to its corresponding fields in the program after each user action (after choosing *Enter,* for example).

# Creating Lists in ABAP

```
START-OF-SELECTION.
  SELECT SINGLE * FROM scarr
                  INTO CORRESPONDING FIELDS OF wa_scarr
                  WHERE carrid = pa_car.
  IF sy-subrc = 0.
    MOVE-CORRESPONDING wa_scarr TO sbc400_carrier.
    CALL SCREEN 100.
    MOVE-CORRESPONDING sbc400_carrier TO wa_scarr.

    WRITE:   wa_scarr-carrid,
             wa_scarr-carrname,
             wa_scarr-currcode.
  ENDIF.
```

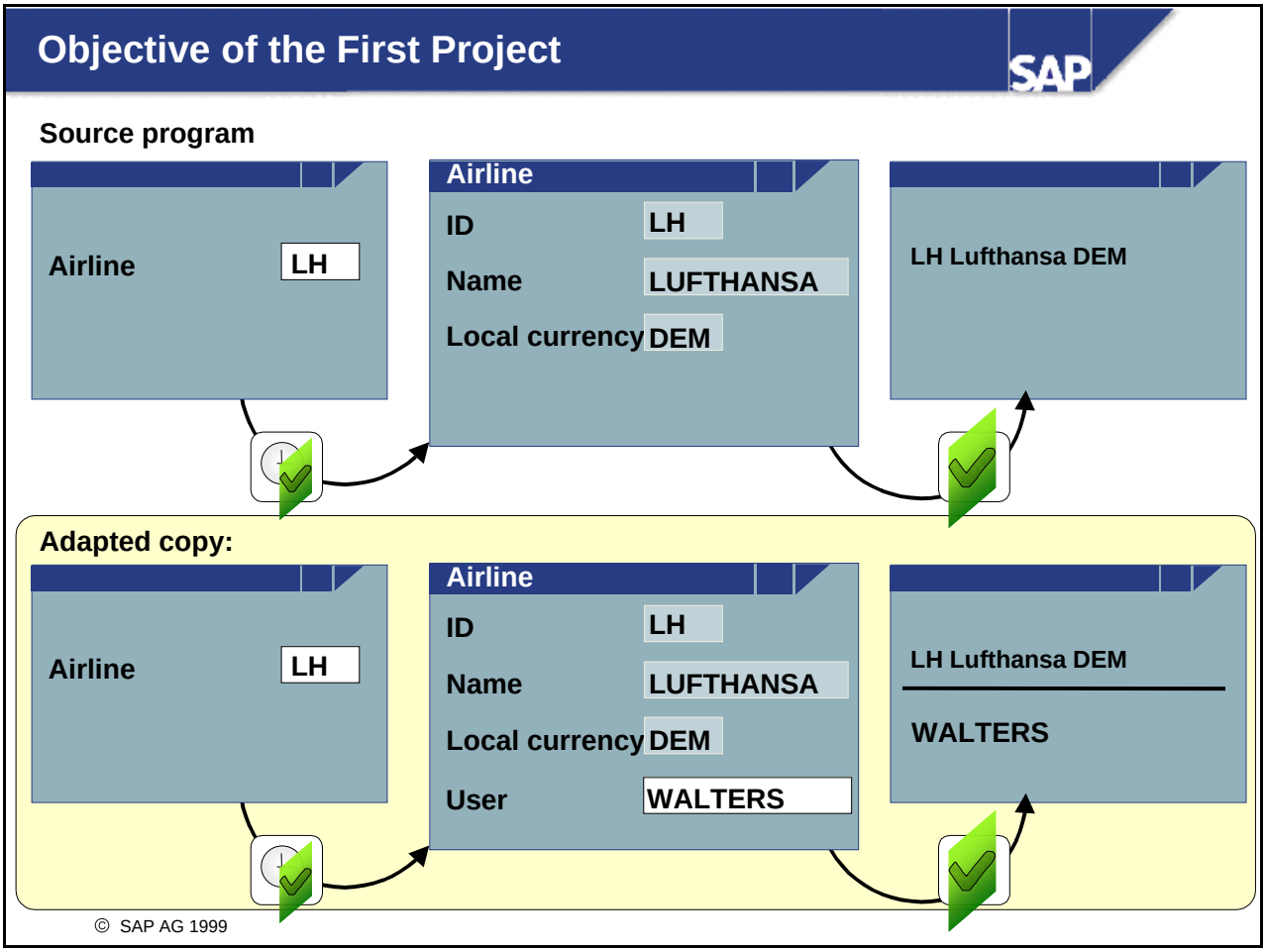
4



- ABAP contains statements (**WRITE**, **SKIP**, **ULINE**) that allow you to create a list.
- **WRITE** statements display field contents, formatted according to their data type, as a **list.**
- Consecutive WRITE statements display output in the same output line. Output continues in the next line when the present one is full.
- You can place a position entry in front of any output value. This allows you to determine carriage feed (*/*), output length (**l**) and where a column begins (**p**). More detailed information about formatting options can be found in the keyword documentation under **WRITE**.
- List output can be displayed in color.
- The **complete** list appears **automatically** at the end of the processing block.

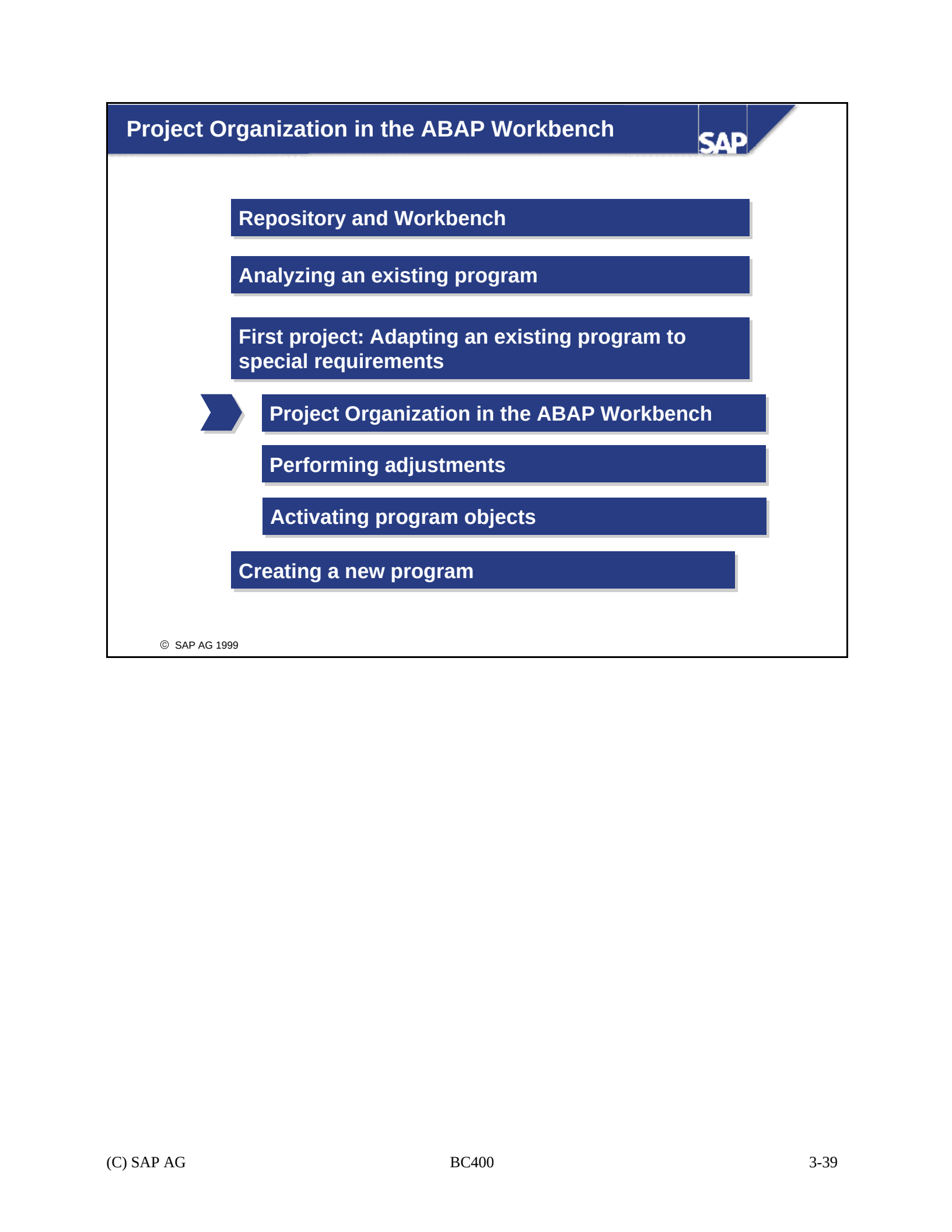
# Overview: Introduction to the ABAP Workbench

- Repository and Workbench
- Analyzing an existing program
- First project: Adapting an existing program to special requirements
  - Project organization in the ABAP Workbench
  - Performing adjustments
  - Activating program objects
- Creating a new program

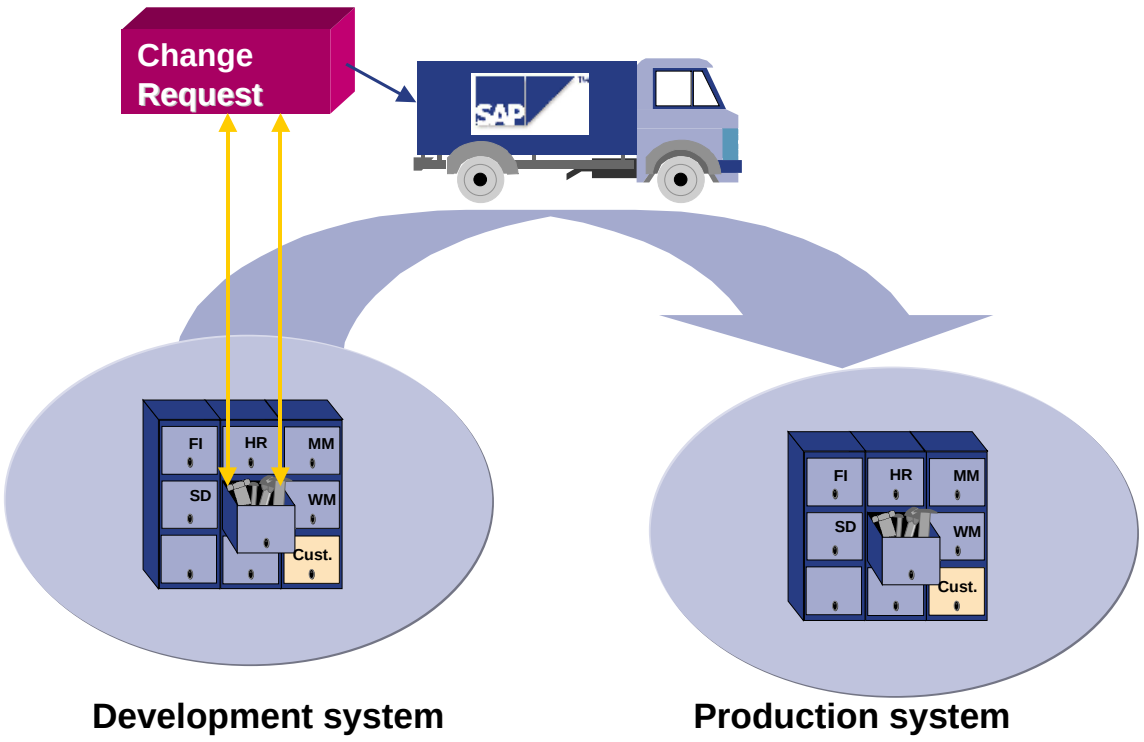



- The first project is to extend an existing program. As no extensions are allowed in the program and modifications are to be avoided, the first step is to copy the program and then change it.
- You must allocate changes to existing programs to a project in the system, just as you would for creating copies of programs or creating new programs. Therefore, the following slides deal first with how a project is represented in the R/3 System.

# Project Organization in the ABAP Workbench

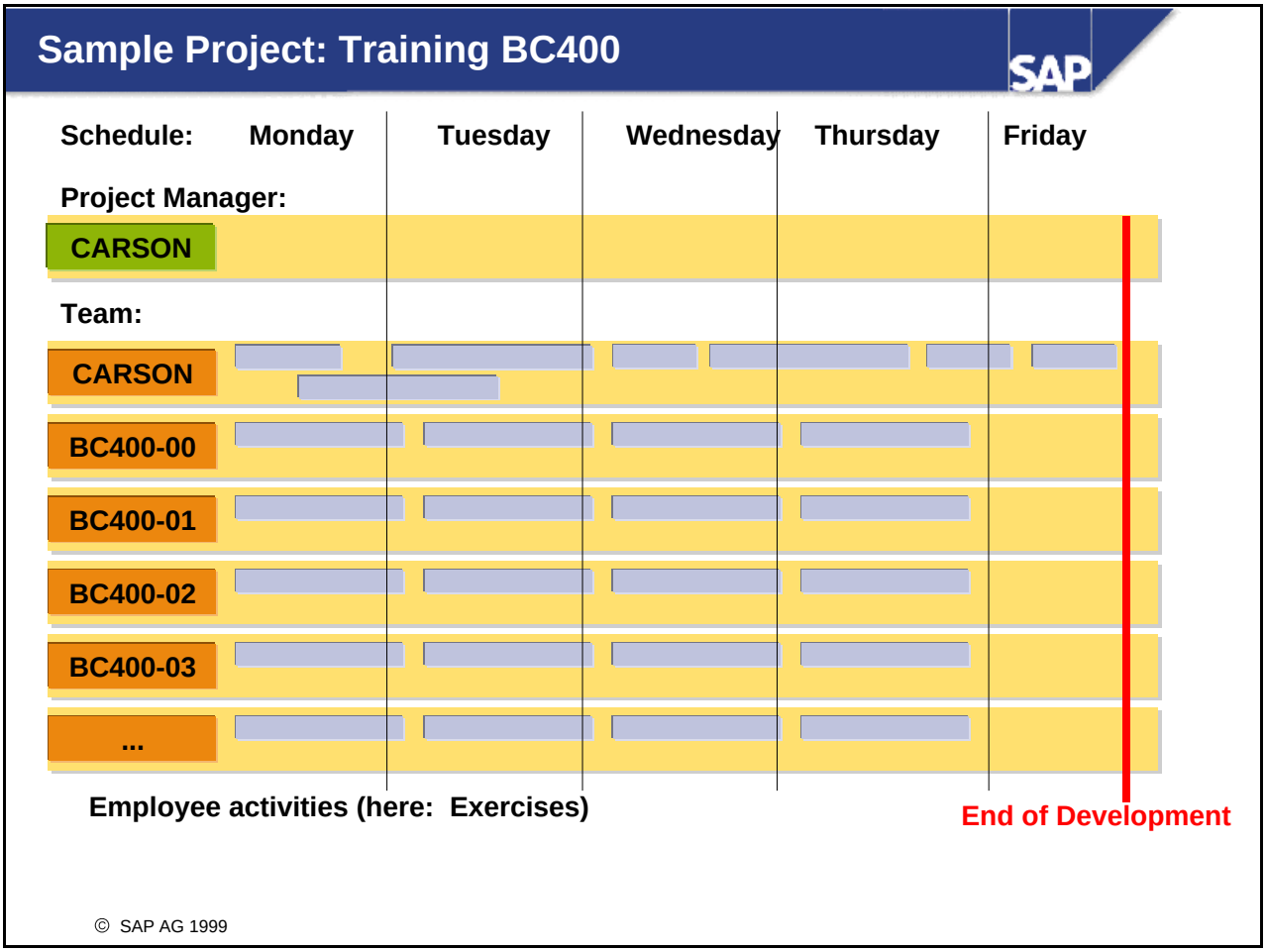


## Transporting Repository Objects



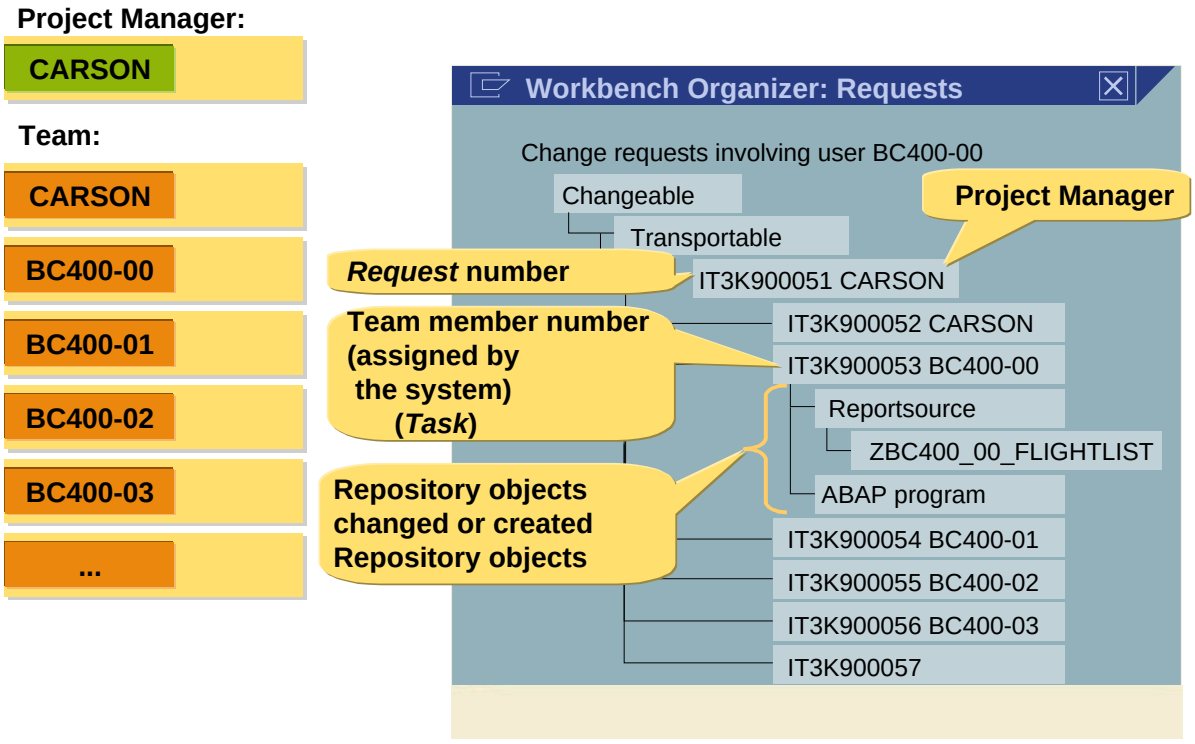



- Projects are always implemented in a development system and then transported to the next system. A decisive criterion for the combination of projects is therefore which Repository objects need to be transported together because of their dependencies. More detailed information on project organization is available in the unit entitled *Software Logistics and Software Adjustment*.
- Repository objects are automatically linked to correction and transport systems when they are assigned to a development class.
- After development has ended, Repository objects are transported into both test systems and production systems by way of certain prescribed pathways.
- The ABAP Workbench tool *Workbench Organizer* (WBO) organizes all development tasks pertaining to Repository objects.



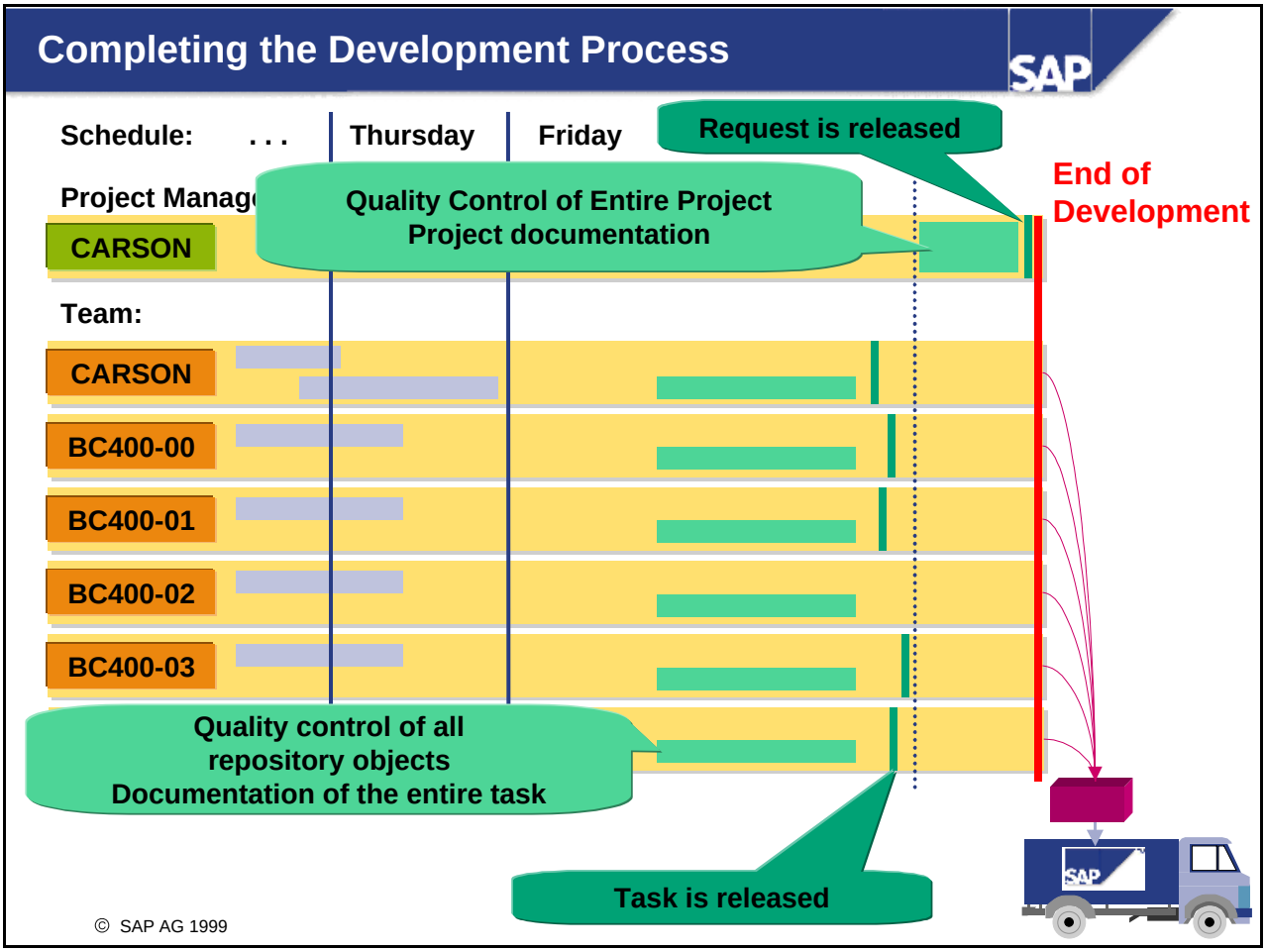

- Each project requires the following information:
  - Name of the Project Manager?
  - What functional scope is to be covered by the object? Which Repository objects are to be changed or created?
  - What is the timeframe for the project?
  - Names of the project participants?
- As an example, we are going to organize Course BC400 as a project.
  - The trainer is the Project Manager.
  - Programs need to be developed for each topic. (These are the trainer's sample programs and the exercise groups' exercises)
  - This project is to be completed by 3:00 p.m. on Friday.
  - The user names of the participants (in this case, the exercise groups) are BC400-XX.

# Project Representation in the Workbench Organizer





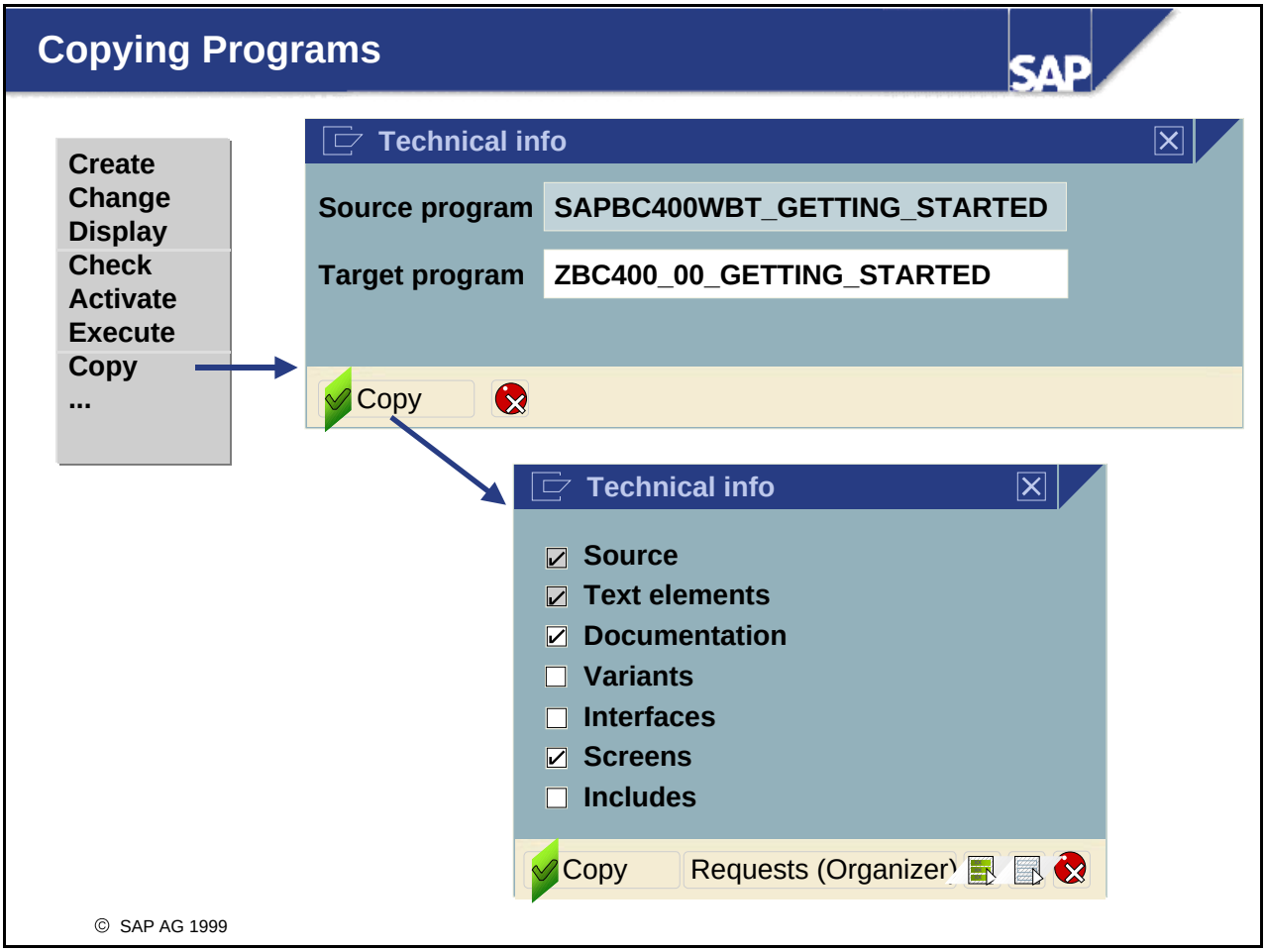

- At the beginning of a development project, the project manager must create a **change request**. The project manager assigns all project team members to the change request. The Workbench Organizer assigns a project number to the change request (<sid>K9<nnnnn>. Example: IT3K900001). <sid> is the system number.
- Next, the Workbench Organizer (WBO) creates a **task** for each employee. From now on whenever an employee allocates a Repository object to that change request, the Repository object will automatically be filed away as that employee's task. Thus all Repository objects that an employee works on during a development program are collected within his or her task folder.
- When changing a Repository object, a developer assigns it to a change request. Unlike the logical functional divisions that separate different development classes, change requests are project-related. Thus, although a program always belongs to only one development class, it can, at different times, belong to different change requests.




- When development is finished, the **developer** carries out a final quality check and releases the **task.** The objects and object locks are passed from the task to the change request. However, all employees assigned to the task can still make changes to the object because the Workbench Organizer will automatically create a new task should the need arise.
- When the project is complete, the Project Manager checks the consistency of the request and the **Project Manager** releases the **change request**. The locks on the objects in the request are released.
- The Repository objects are then exported to the central transport directory.
- The objects are not imported automatically into the target system. Instead, the system administrator uses the transport control program **tp** at operating system level. Afterwards, the developers check the import log.

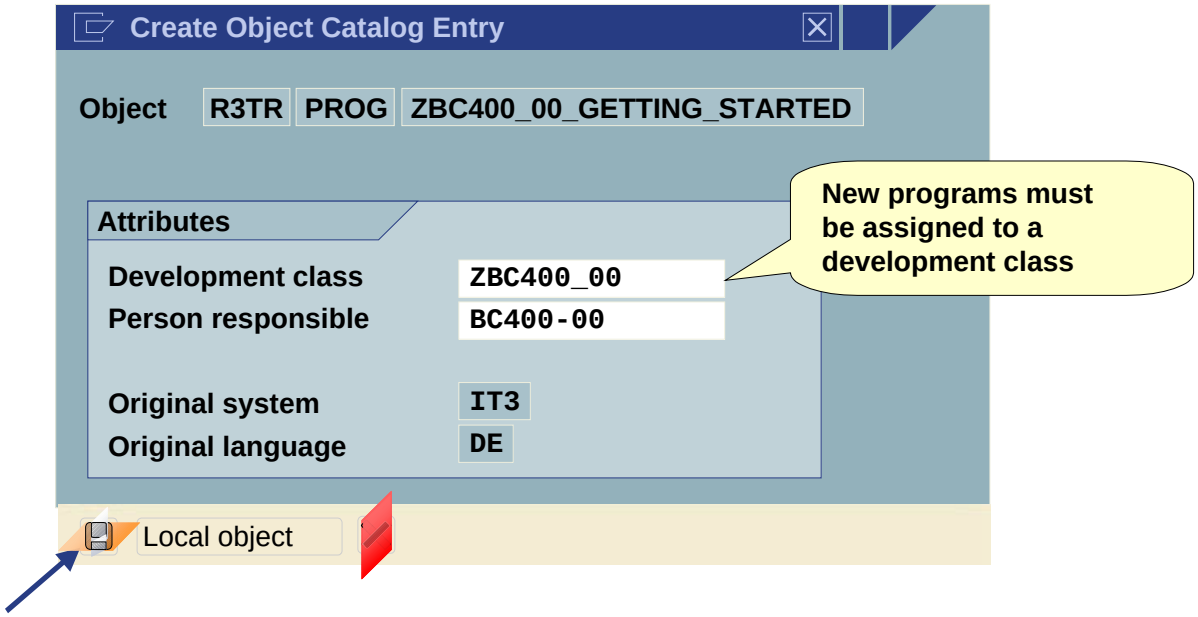
# Performing Adjustments

- Repository and Workbench
- Analyzing an existing program
- First project: Adapting an existing program to special requirements
  - Project Organization in the ABAP Workbench
  - Performing adjustments
  - Activating program objects
- Creating a new program



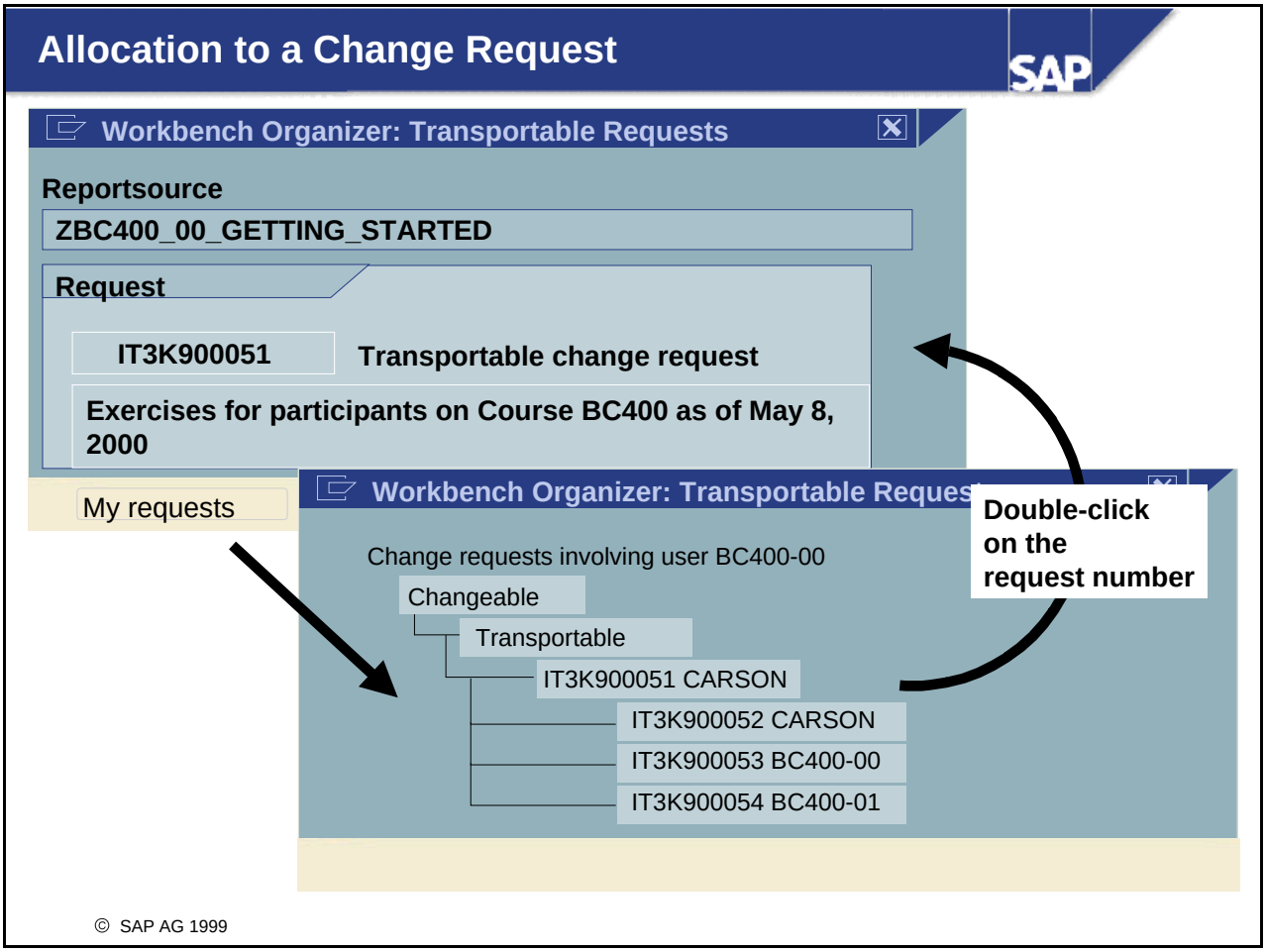

- Program names beginning with Y or Z, or with SAPMZ or SAPMY, are reserved for customer developments. You can also have a namespace reserved for customer developments. Detailed information on customer namespaces for various Repository objects is available in the SAP Library under ***Basis Components -> Change and Transport System(BC-CTS) -> BC Namespaces and Naming Conventions***.
- You can copy a program from the object list of a development class or program. To do so, simply place your cursor on the name of the program you want to copy and click with the right mouse button. Choose *Copy.* The system displays a dialog box where you can enter a new name for your copy. Confirming your entries using the appropriate pushbutton in the application toolbar causes the system to display a dialog box where you can select the sub-objects that you want to copy with the program. Thus, you should decide which sub-objects you want to copy with the program BEFORE you begin the copy procedure. After you confirm these entries, the system displays yet another dialog box where you can save Repository objects.
- If you are copying a program that contains includes, another dialog box is displayed before this one, where you can choose which includes you want to copy and enter new names for them.

# Saving Programs

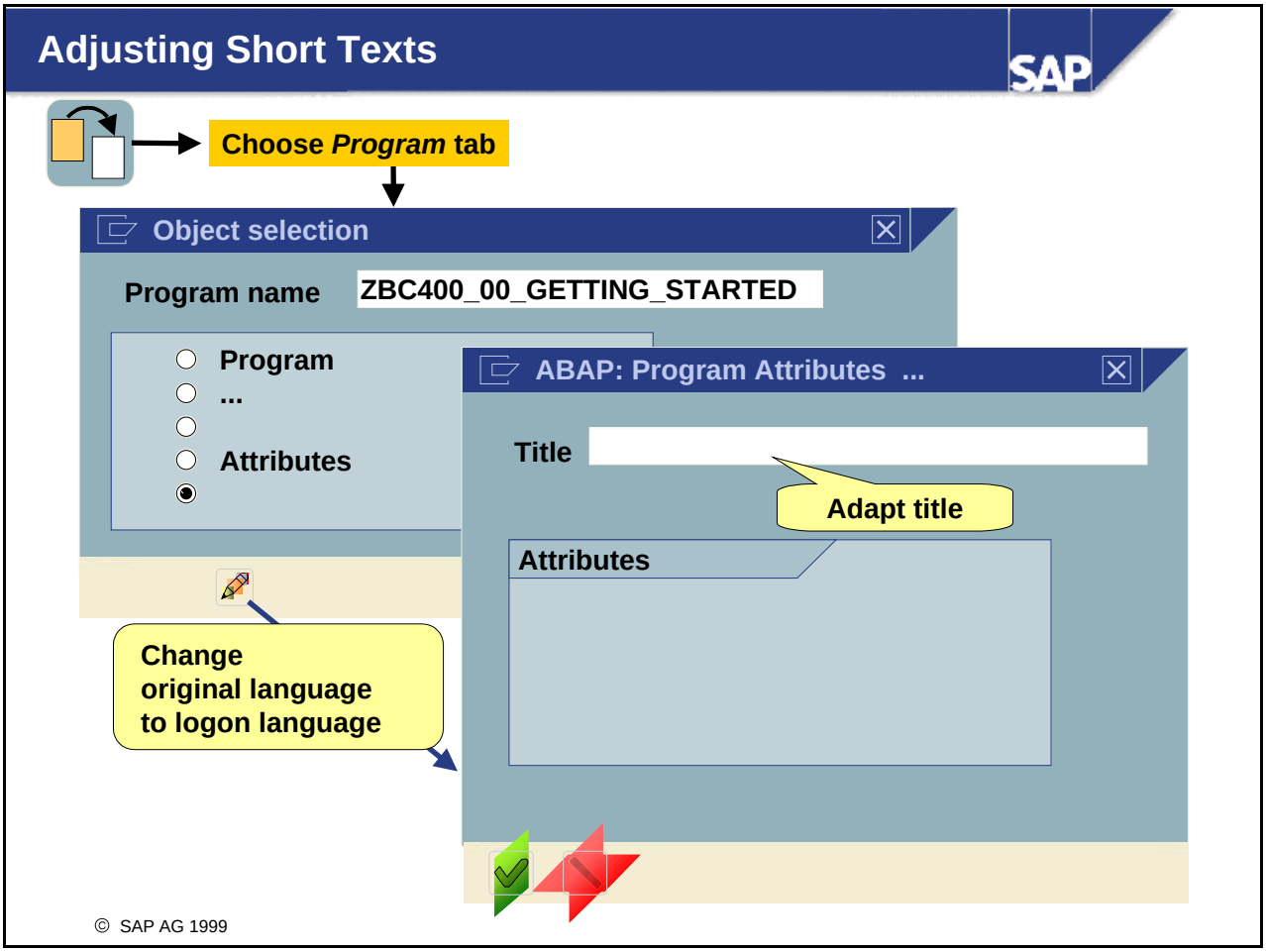





- Assign the program to a development class, so that you can save it. Your name is automatically entered into the system as the person responsible for the new program copy. Check all entries to see if they are correct and then choose *Save*.




- All Repository objects that are created or changed must be assigned to the change request for their corresponding project. For this training course, the trainer has created a change request for the project 'Exercises for Participants on Course BC400 as of May 8, 2000'. Each group has a task within this change request. Save all of your Repository objects (development classes, programs, and so forth) to this change request.
- You can display all change requests in which you have a task using the 'My tasks' pushbutton.
- For more information about project organization from the project management point of view (including creating tasks), refer to the unit on *Software Logistics and Software Adjustment*.




- You can adjust the short text (that is, the title) as follows:
  - Double click on *Program object types* in the Object Navigator object list.
  - Choose attributes.
  - Click on the 'Change' icon.
  - If the original language of the source program is not identical to your logon language, a dialog box appears to ask you whether you want to change the title in the original language or if you want to change the original language.
  - Now you can adapt the title.
- The altered title appears as short text next to the program name in the Object Navigator object list.

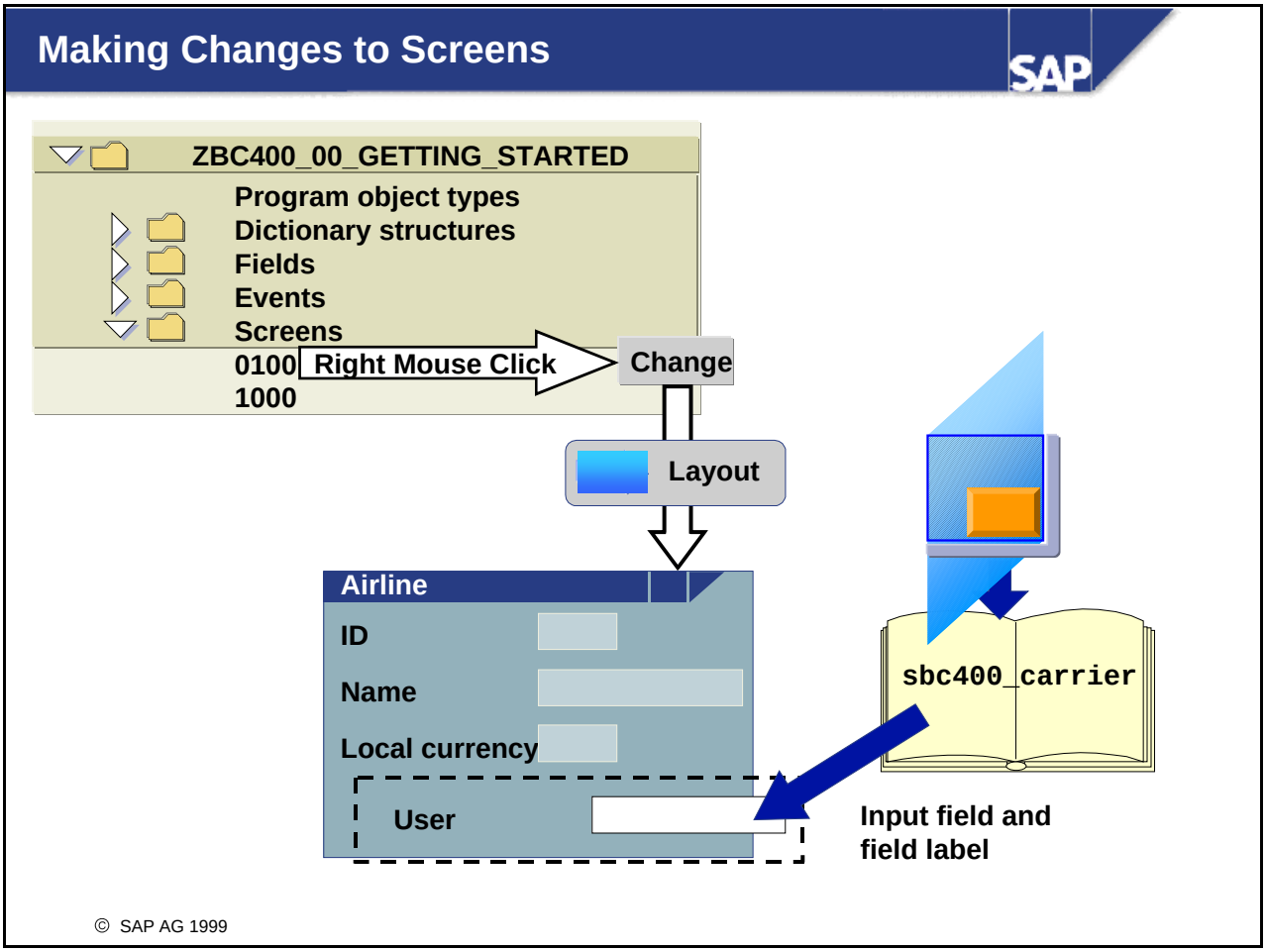
# Adapting Source Code

```
START-OF-SELECTION.
  SELECT SINGLE * FROM scarr
                  INTO CORRESPONDING FIELDS OF wa_scarr
                  WHERE carrid = pa_car.
  IF sy-subrc = 0.
    MOVE-CORRESPONDING wa_scarr TO sbc400_carrier.
    CALL SCREEN 100.
    MOVE-CORRESPONDING sbc400_carrier TO wa_scarr.

    WRITE:   wa_scarr-carrid,
             wa_scarr-carrname,
             wa_scarr-currcode.
    ULINE.
    WRITE sbc400_carrier-uname.
  ENDIF.
```

**Syntax check**



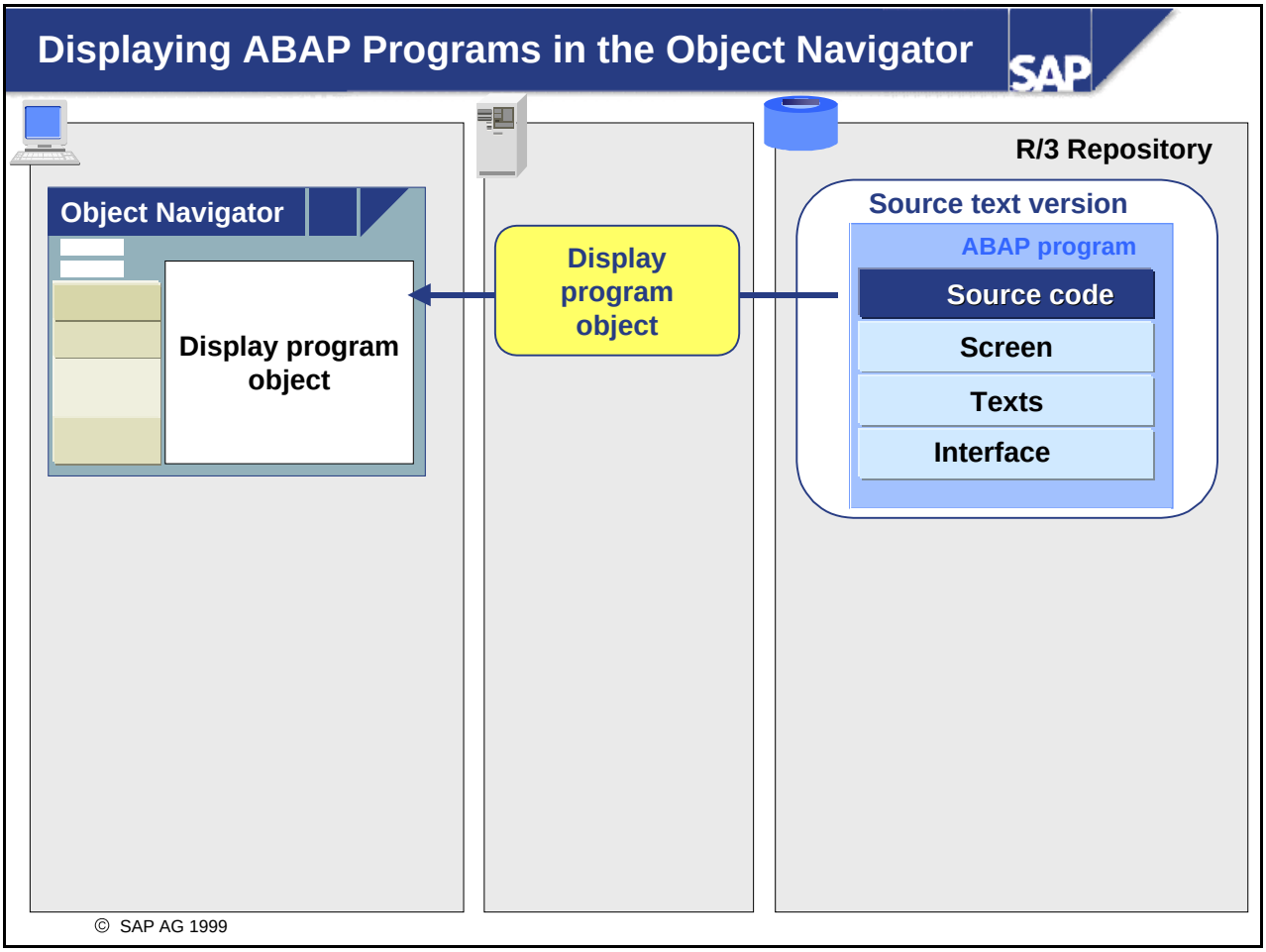
- To adapt the source code, navigate to the Editor (context menu).
- Adapt the list by adding a **ULINE** statement and a **WRITE** statement. You can find further information on these statements in the keyword documentation.
- You can carry out a syntax check after you have changed the source code.



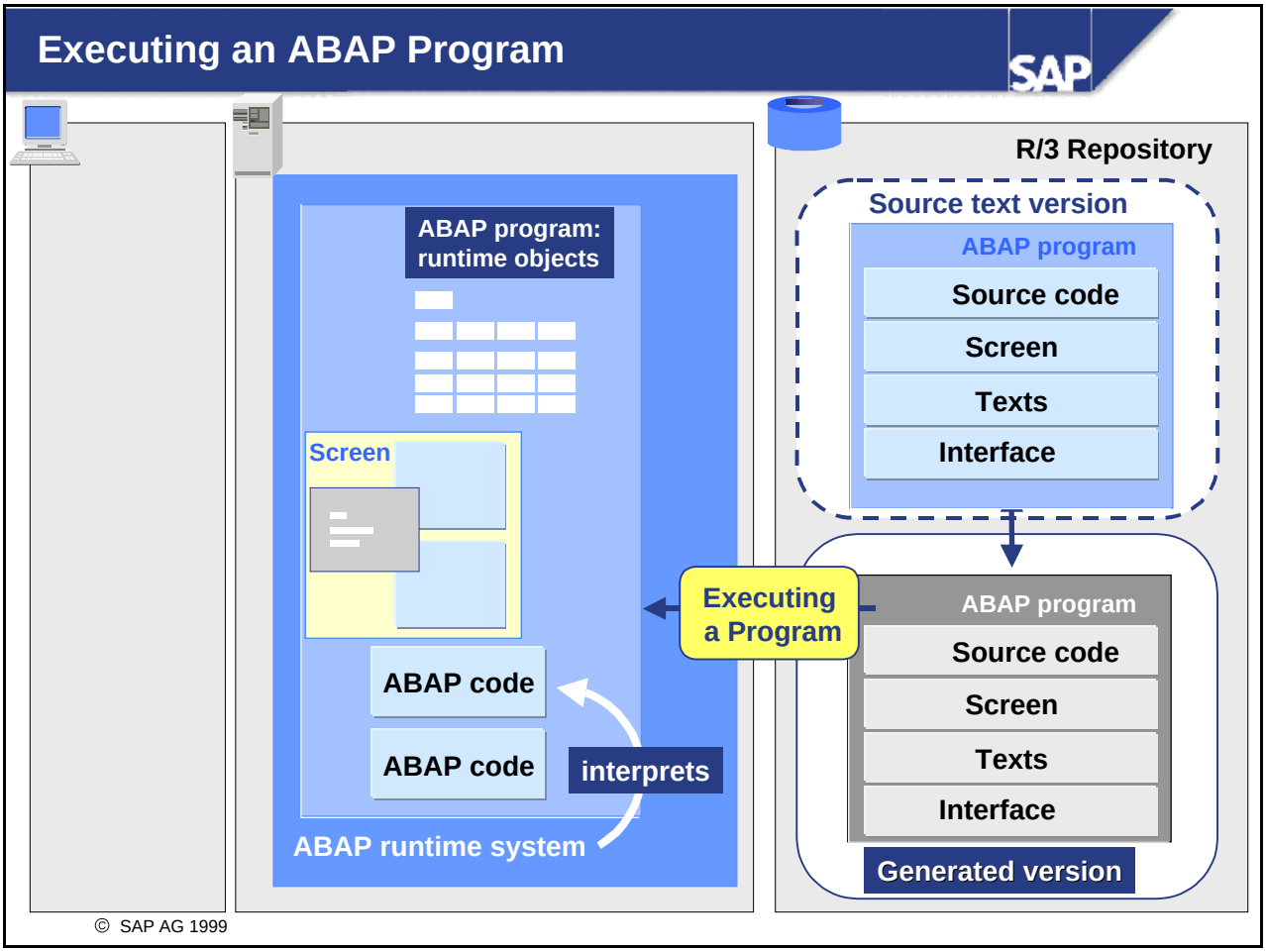

- You can change a screen using the Screen Painter. To change the layout, first use the context menu for the screen in the object list, to navigate to the Screen Painter. From there, use the *Layout* icon to navigate to the graphic *Layout Editor*.
- This contains an icon for creating input/output fields with reference to global types. Enter a structure type. All fields for this structure type are displayed for selection. You cannot select fields that are already contained on the screen. This is shown by a small padlock next to the field.
- The tool for displaying and maintaining global types is called the *ABAP Dictionary*. You can find more detailed information on global types in the *ABAP Statements and Data Declarations* unit.

# Activating Program Objects

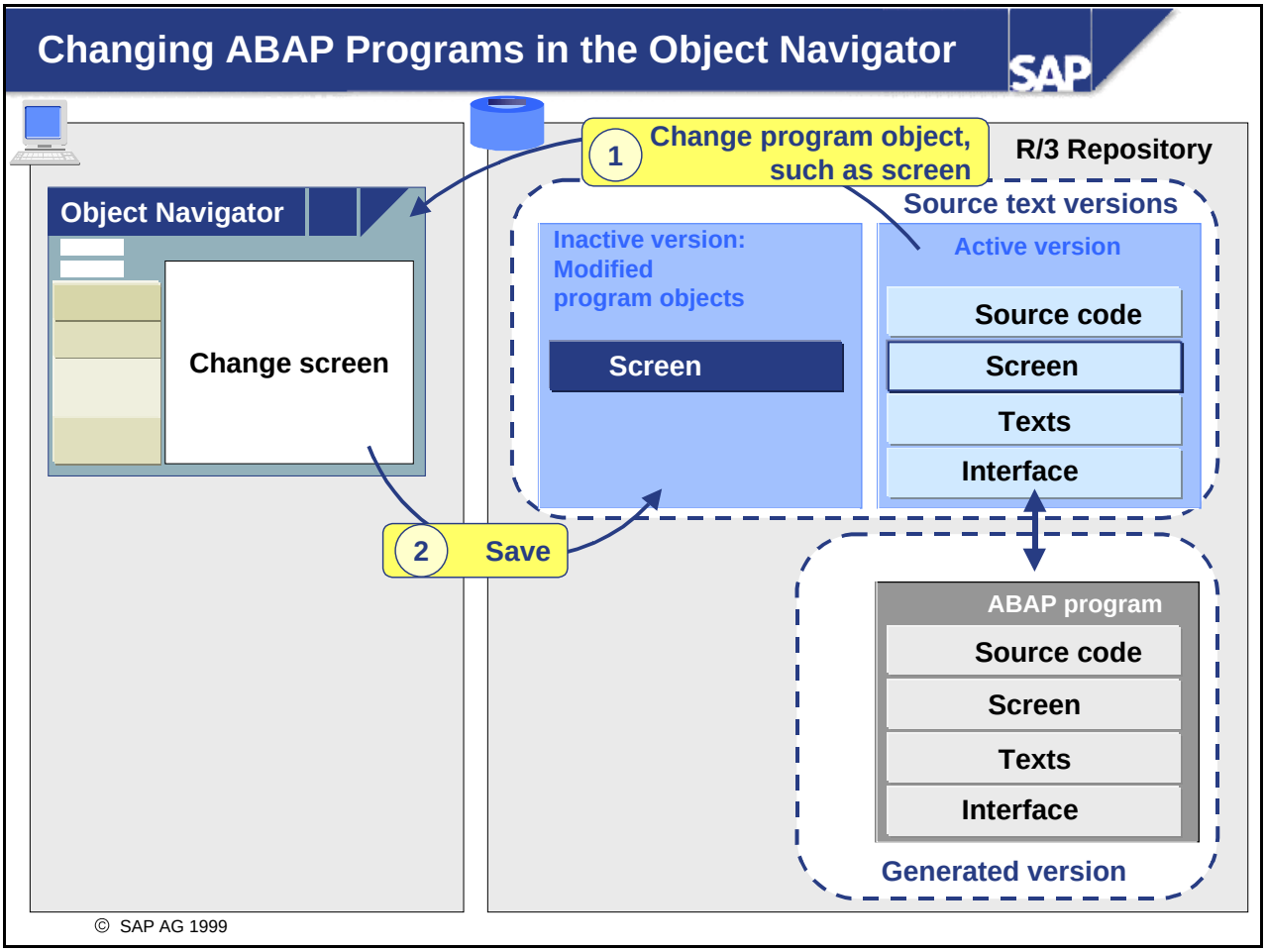
- Repository and Workbench
- Analyzing an Existing Program
- First project: Adapting an existing program to special requirements
  - Project Organization in the ABAP Workbench
  - Performing adjustments
  - Activating program objects
- Creating a new program



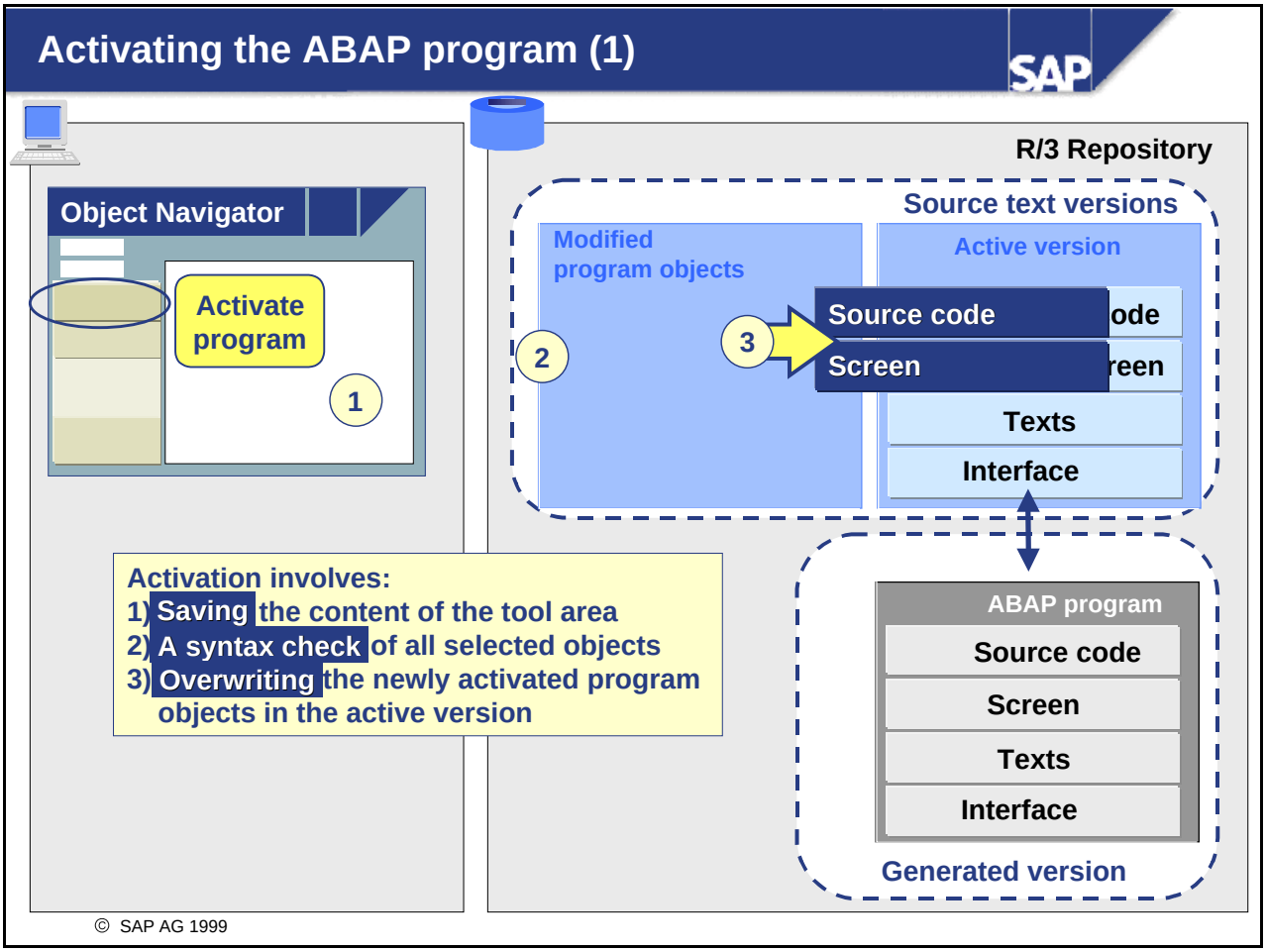

- Each program has different types of program objects associated with it:
  - The **source code** contains all the definitions of data objects and executable ABAP source. You edit ABAP source code in the *ABAP Editor*.
  - A screen must be created using the **Screen Painter** tool.
  - **Texts** contain field descriptions of selection screens, list headers, and text symbols. These texts can be translated and are displayed in the logon language at runtime. You can display texts in the Editor by choosing *Goto -> Text elements*.
  - The **interface** containers the user interface with the menu bars and both toolbars, which are created using the *Menu Painter*.
- When you display a program object in the appropriate tool in the Object Navigator, a source code version is invoked.
- You can display only one program object in a single session. To display several program objects at once, use the ***Display -> In New Window*** in the context menu. Bear in mind that creating a new session is very runtime-intensive. Where possible, use the navigation functions, which allow you to move quickly between displaying different program objects.



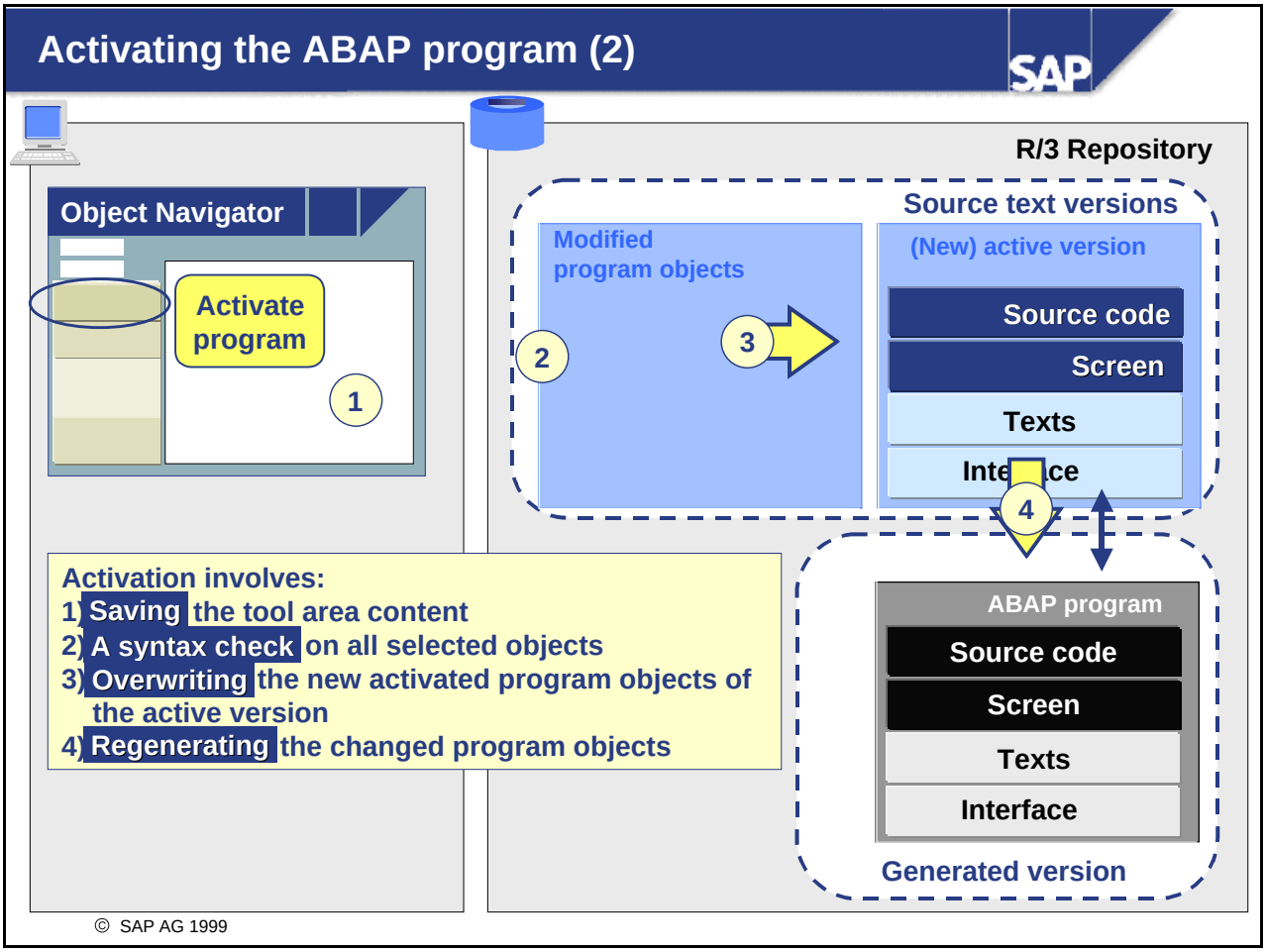

- When you start a program, the system generates runtime objects:
  - Data objects: Memory is made available for each data object. The amount of memory depends on the type. You do not need to concern yourself with memory management when programming in ABAP. You simply define the data objects using ABAP statements and the runtime system does the rest.
  - Screens: The system also has to create a runtime object for each screen while it is being processed. This object can, for example, contain the memory allocated to the screen fields.
  - ABAP code is interpreted block-by-block by an ABAP interpreter. The different types of processing blocks are explained in detail in the *Internal Program Modularization* unit.
- The runtime objects for the ABAP program are created from a generated version of the ABAP program. Each program has no more than one generated version, which is stored in the database in the Repository after having been generated. If there is no current generated version, the system creates one when the program is started.



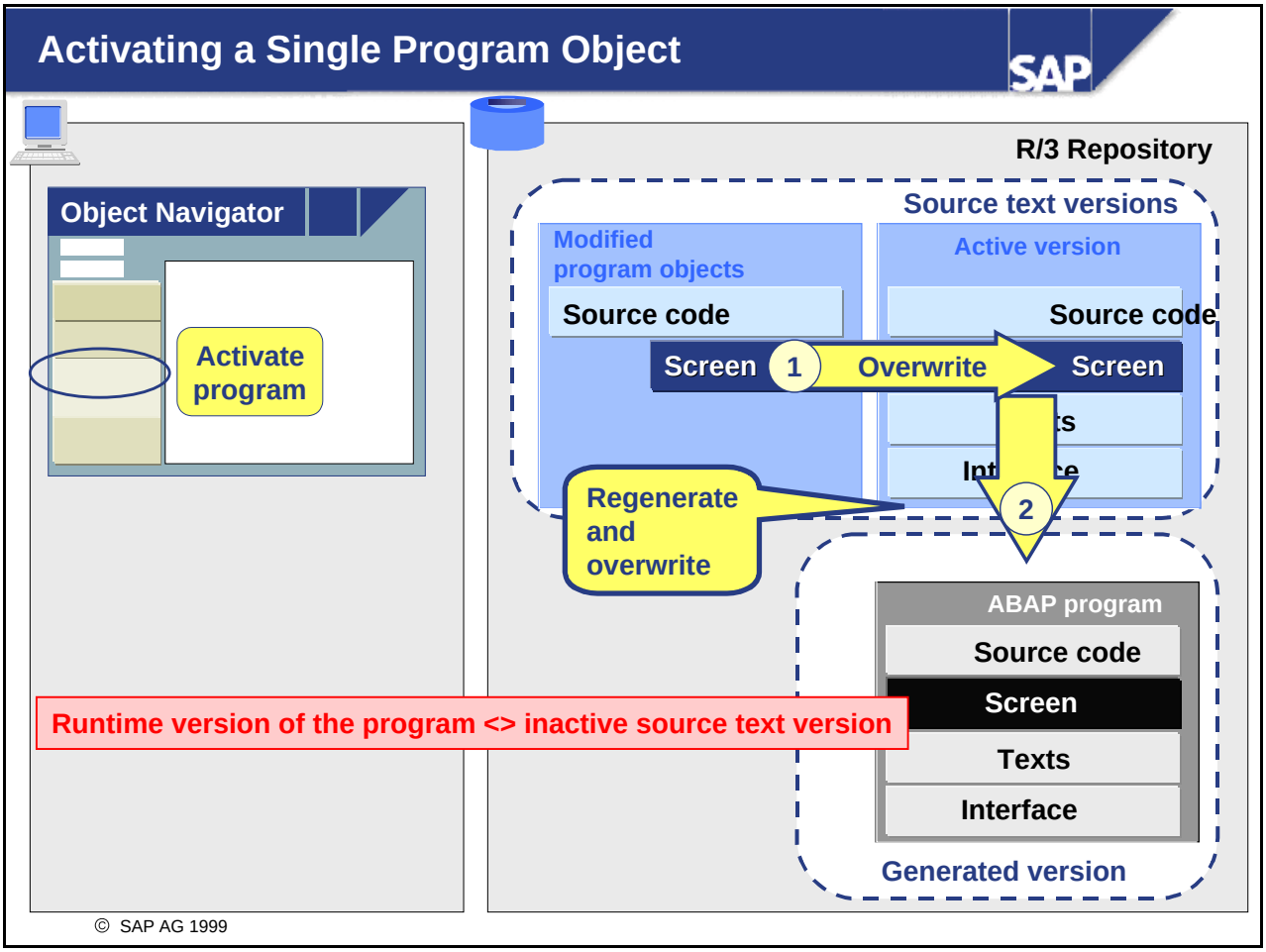

- Once you have edited a program object and stored the changed version, an inactive source text version of the program is created. It contains all the altered program objects. The generated version, however, is not changed. To ensure that the active version of the program is started, run it from the object list context menu.
- Saving the program objects triggers a local save, which does not affect the generated version of the program.
- Note: The *Editor*, *Screen Painter* and *Menu Painter* tools allow you to test program objects locally using a *Test* icon or F8. The system then creates a temporary generated version of the program object. If other parts of the program are required to test the program object, these are supplemented by the generated version. Example: You have changed the source code and a screen in a program and saved them. You choose *Test* from the Editor. The system creates a temporary generated version for the source text, from which the runtime objects for the data objects ad the ABAP code are then generated. For the screen, the generated version of the active program is invoked.



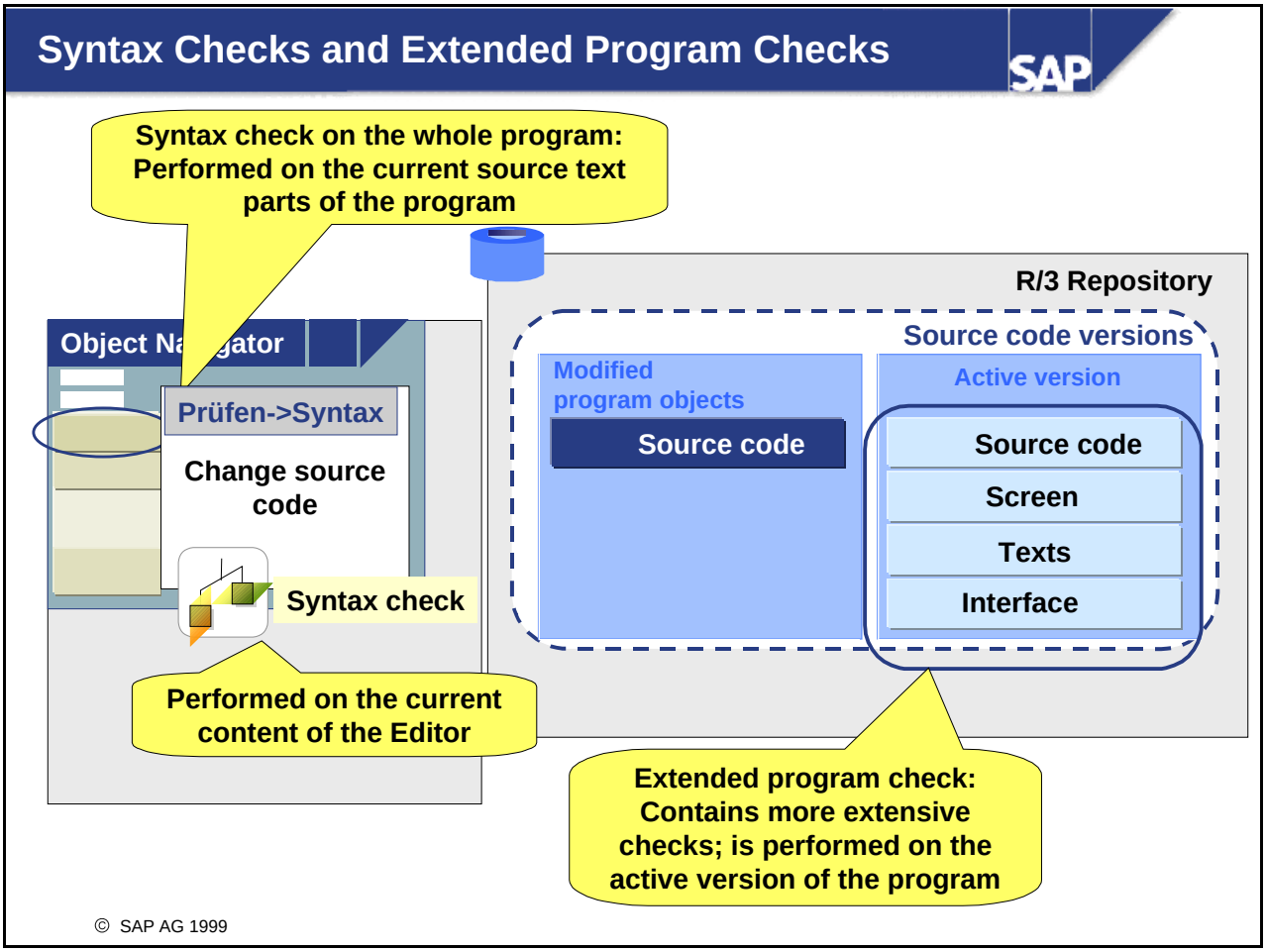

- To convert the modified program objects to an active version, you must activate **all** modified program objects.
- Technically, activating these program objects involves:
  - Saving the content of the tool area
  - Performing a syntax check
  - (Provided there are no syntax errors), overwriting the active source text version and...




- Technically, activating these program objects involves:
  - Saving the content of the tool area
  - Performing a syntax check
  - (Provided there are no syntax errors), overwriting the active source text version
  - Regenerating the modified parts of the program (thus over-writing the generated version).



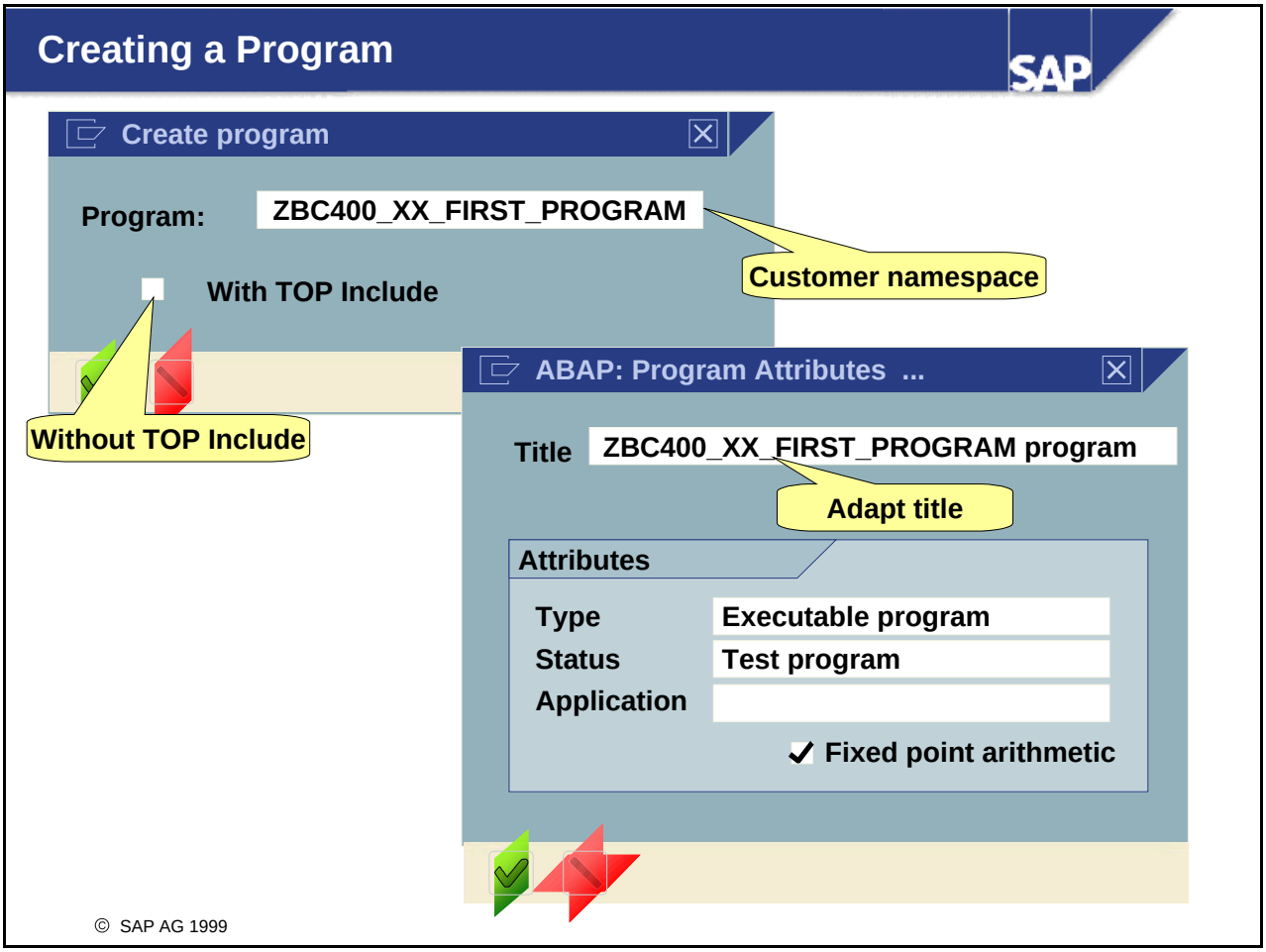

- You can also activate individual program objects. This means that only the objects you selected are regenerated and overwritten in the active version. This can lead to unexpected results, since the runtime version may not be identical to the current, inactive source code version.




- There are several levels of syntax check:
  - **Syntax check on the content of the Editor:** You can check the source code you have worked on in the Editor by choosing the *Syntax check* icon. This check does not take includes into account.
  - **Syntax check on the whole program:** You can trigger a syntax check of all the parts of the program from the context menu of the program name. In the appropriate context menu, choose ***Check->Syntax***. If parts of the program are saved but not activated, the current (that is, inactive) version is invoked for the syntax check.
- To trigger an extended program check, which also performs more extensive consistency checks, activate the program first. The extended program check is performed on the active version of the program In the program's context menu, choose ***Check->Extended program check***. Before you complete a project, perform an extended program check on all the programs it contains.

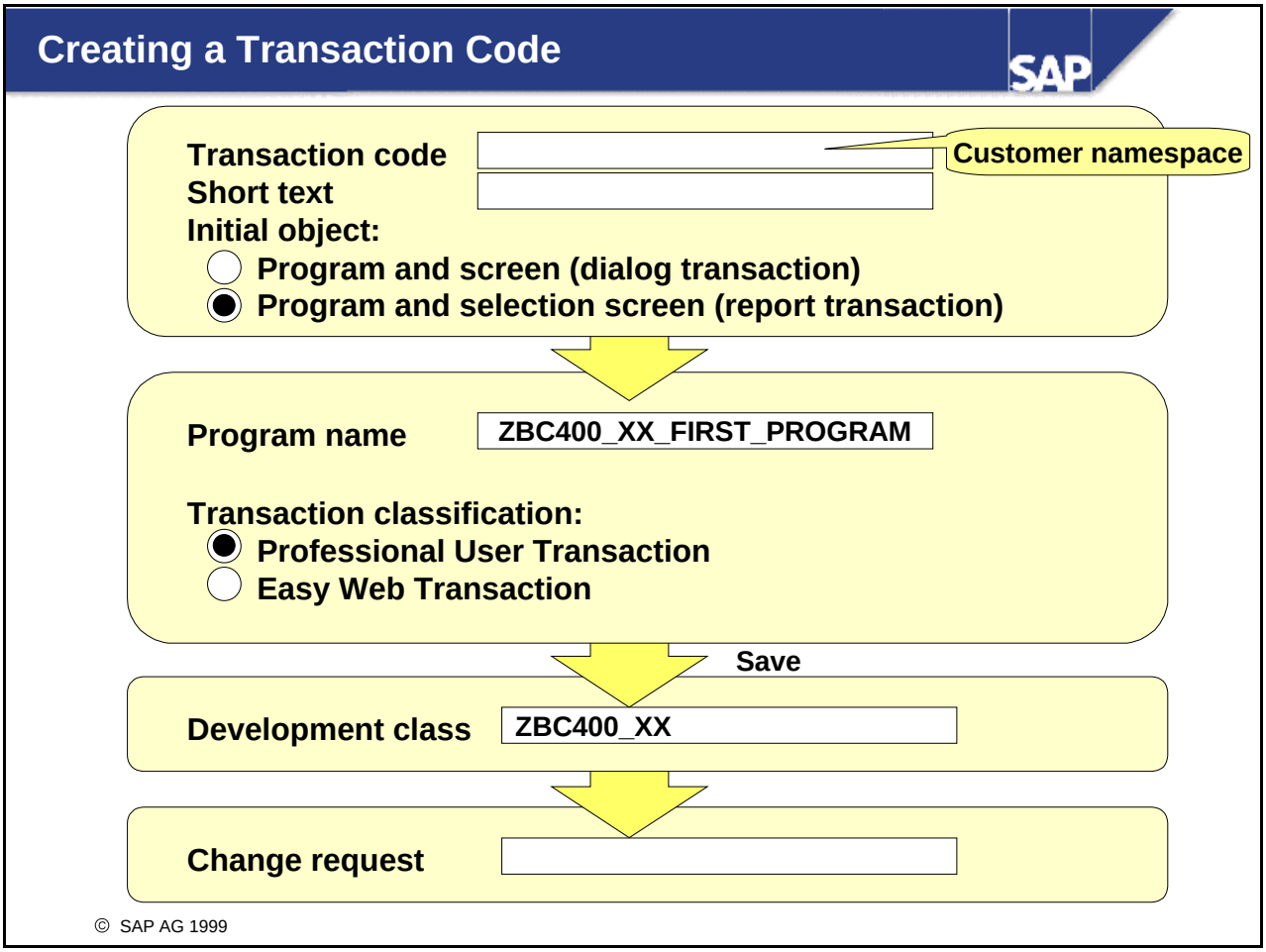
# Creating a New Program

- Repository and Workbench
- Analyzing an Existing Program
- First project: Adapting an existing program to special requirements
- Creating a new program



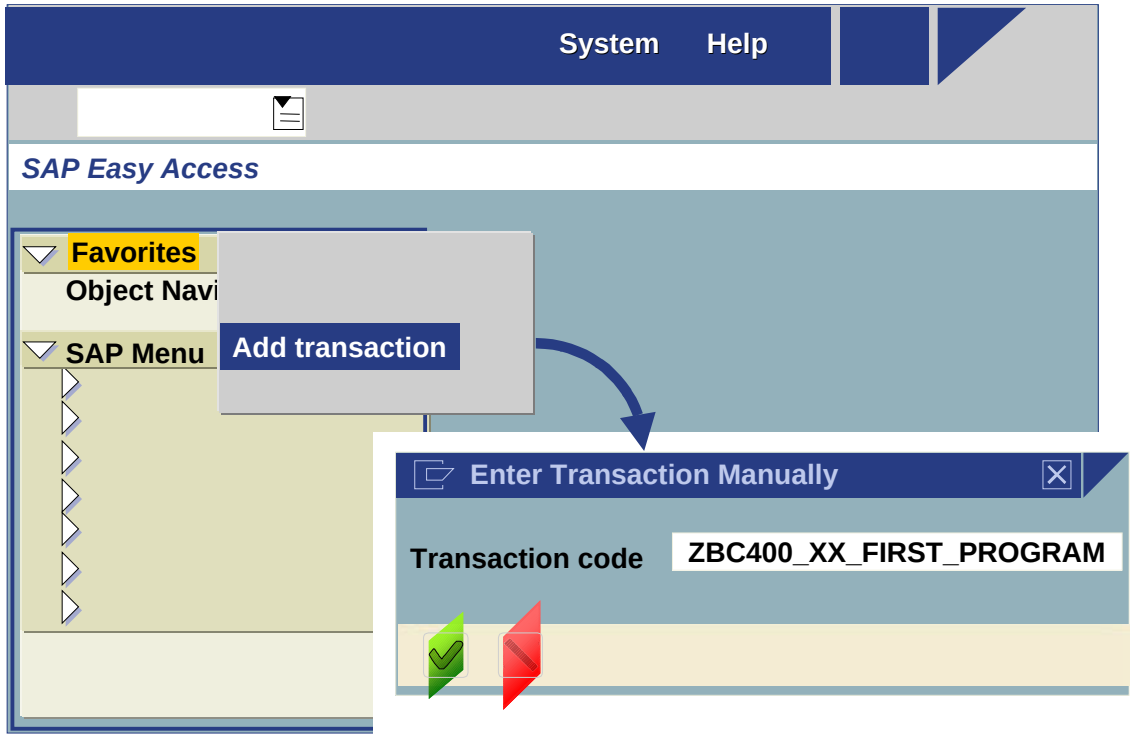

- There are several ways to navigate to the dialog you need to create a program:
  - In the *Object Navigator*, choose the *Program* object type and enter a program name. Comply with the customer namespace conventions. When you choose *Enter*, the system automatically checks to see if your program already exists. If the program is available, the system displays it. If not, the system goes to the dialog sequence that lets you create a program. (Note: this is not possible in 4.6A or 4.6B).
  - In the Object Navigator, display the development classes in which you want to create the program. Trigger the dialog sequence for creating a program using the development class context menu or the *Program* node.
  - Choose the *Other object...* or *Edit object* icon. On the *Program* tab, choose *Program* and enter the name of your program. Choose the *Create* icon.
- In the first dialog box, enter the program name and uncheck the *With TOP Include* field. Choose to confirm.
- Change the title to make it meaningful.
- Choose the *Executable program* program type, so that it can include selection screens and lists if necessary.

- Choose the appropriate program status. If you choose *System program*, the source text is processed as a whole in the normal debugging mode. This means that the program can only be debugged using *System Debugging.*
- You need not assign the program to an application. The program is placed in the part of the application component hierarchy by being part of the development class.




- If you want to include a program in role menus or in the *Favorites* in your SAP Easy Access menu, you must assign a transaction code to it.
- In the Object Navigator, open the object list for your program. In the program context menu, choose ***Create -> Transaction.***
- Enter a transaction code. Comply with the customer namespace conventions. Enter a short text. (This text appears, for example, in any role menus to which the transaction is assigned). If the program is an "executable program", choose *Program and Selection Screen (Report Transaction).*
- On the next screen, enter the name of the program. Choose *Professional User Transaction*. The difference between a Professional User Transaction and an Easy Web Transaction is explained in more detail in the *Developing Internet Applications* unit.
- Choose *Save.*
- On the next screen, enter the name of the development class, to which the program logically belongs.
- On the next screen, assign the change request to which the project (that is, of creating a transaction code) belongs.

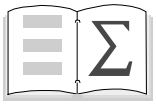
# Including a Transaction Code in *SAP Easy Access*






- To include a transaction code in the your role menu favorites:
  - Display the initial screen (SAP Easy Access).
  - In the favorites context menu, choose *Insert transaction*.
  - In the dialog box that appears, enter a transaction code.
  - Choose *Enter* to confirm.
- The transaction code short text appears under the *Favorites* node. You can start the relevant program from the context menu associated with this short text.

# Introduction to the ABAP Workbench: Unit Summary



**You are now able to:**

- **Use the different types of navigation available in the ABAP Workbench to reconstruct an existing program**
- **Make simple changes to an existing program's user dialogs using the tools ABAP Editor and Screen Painter**

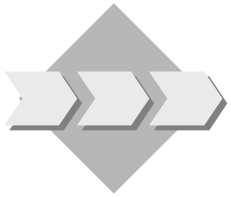
## ABAP Workbench Exercises

**Unit: ABAP Workbench**

**Topic: Analyzing a program**

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

- Use the navigation functions to examine the structure of a program



The program SAPBC400WBT_GETTING_STARTED contains a selection screen that allows the user to enter an airline code. The airline details then appear on a screen. When the user presses Enter, the data is then displayed in a list.

Navigate through the program code and other components to help you understand the structure and flow of the program.

**Program:** SAPBC400WB**T**_GETTING_STARTED

1-1 Open the object list for development class BC400. Find the program **SAPBC400WBT_GETTING_STARTED**, and open its object list. Throughout the exercise, make sure that you remain in **display mode**.

1-2 Run the program to find out how it works. There is an input field on the selection screen.

1-2-1 What information must you pass to the program? (Use the F1 help for the input field)

1-2-2 What values can you enter? (Use the possible entries help F4)

1-2-3 What information does the program provide?

1-2-4 What user dialogs does the program contain? Find out the number of the selection screen and the screen number by choosing *System* → *Status*.

1-2-5 What are the names of the input field on the selection screen and the output fields on the screen? You can display the field names using ***F1*** → ***Technical info***, then see the box with the heading *Field description for batch input***.**

1-3 Use the object list in the Object Navigator to examine the program.

1-3-1 What data objects are there? (Use the program object list) Where are they defined? (Use navigation) Where are they used? (Use the where-used list).

1-3-2 What data object in the ABAP program corresponds to the input field on the selection screen? (Look in the object list for a data object with the same name as the field that you found out in step 1-2-5.)

1-3-3 Which statement processes the screen? (Look in the source code or use a where-used list for the screen number.)

1-3-4 Navigate to the screen, and from there to the graphical layout. Click an output field. Where in the graphical layout editor does the field name appear that you found out in step 1-2-5?

1-4 Navigate to the program source code.

1-4-1 Which statement constructs the list? Open the keyword documentation for this statement.

1-4-2 Which statement is responsible for the database dialog? From which database table is the data read? Navigate to the database table definition. What columns are in the table?

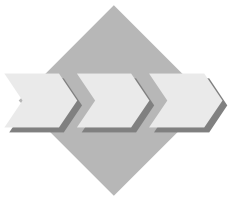
1-4-3 Only one line is read from the database table. In which data object is the information as to which line should be read? When is the variable containing the information about the line of the database to be read filled?

**Unit: ABAP Workbench**

**Topic: Adapting a Program to Special Requirements**

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

- Copy programs and change them using the ABAP Editor and the Screen Painter
- Use the syntax check to identify simple errors



**Program:** ZBC400_##_GETTING_STARTED

**Template:** SAPBC400WB**T**_GETTING_STARTED

**Model solution:** SAPBC400WB**S**_GETTING_STARTED

2-1 Copy the program **SAPBC400WBT_GETTING_STARTED** with **all** of its components to **ZBC400_##_GETTING STARTED** and assign it to your **development class ZBC400_##** and the change request for your project, "BC400…". (## is your group number.)

2-2 Extend the program as follows:

2-2-1 Add the statement "**ULINE**." to the program and do a syntax check. Make a deliberate syntax error and use the syntax check to find it. Activate the program and start it again. What has changed? Run the extended program check.

2-2-2 Change the program so that input fields occur on the screen for the user name, a time, and a date. Navigate to the Screen Painter by double-clicking on the screen number. This takes you to the screen's flow logic. Check that the graphical layout editor is active *Utilities → Settings*). Start the graphical layout editor by choosing the relevant pushbutton in the application toolbar. Check that you are in change mode. Define the additional fields with reference to the ABAP Dictionary. As your reference structure, use **SBC400_CARRIER** and select the fields UNAME, UZEIT, and DATUM. Activate the screen.

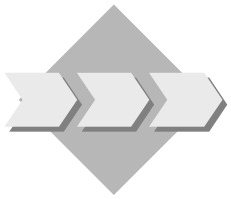
2-3 Display the extra fields in the list. Use the **WRITE** statement. Display the data on a new line, separated from the other fields by an empty space and a horizontal line. To do this, use the ABAP keywords **SKIP** and **ULINE**. Check your program for syntax errors, then activate it, and run it.

**Unit: ABAP Workbench**

**Topic: Creating Transaction Codes**

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

- Create a transaction for your program
- Add a transaction code to the favorites in your role menu



**Program:** ZBC400_##_GETTING_STARTED

3-1 Create the transaction code **ZBC400_##_GS** for your program **ZBC400_##_GETTING_STARTED**. Then assign it to your **development class ZBC400_##** and the change request for your project, “BC400…”. (where ## is your group number.)

3-1-1 From the context menu for the program name in the program object list, choose *Create →Transaction*. Give this transaction the code **ZBC400_##_GS**.

3-1-2 Enter the necessary information in the dialog boxes that appear (using the *Creating a Transaction Code* slide for reference).

3-2 Add this transaction code to the favorites in your role menu in SAP Easy Access.

3-2-1 Navigate to the initial screen, by opening a new session (*System →Open session)*.

3-2-2 Using the *Favorites* context menu, add your transaction code to the favorites.

3-2-3 Start your program from this menu.

**Unit: ABAP Workbench**

**Topic: Analyzing a program**

1-2 Analyzing by executing a program:

1-2-1 You need to add the code for an airline to the program. This information can be displayed from the input field using ***F1***.

1-2-2 The values permitted here depend on the contents of database table **SCARR**. You can display possible entries help from the input field using ***F4***.

1-2-3 The program displays detailed information on the airline company selected. This information is first displayed on the screen and then as a list.

1-2-4 The program contains a selection screen with screen number **1000**, a screen with number **100** and a list.

1-2-5 The field name of the input field on the selection screen is **pa_car** and the names of the output fields on the screen are **sbc400_carrier-carrid**, **sbc400_carrier-carrname** and **sbc400_carrier-currcode**. You can display the field names using ***F1*** → ***technical info***, then see the box with the heading *Field description for batch input***.**

1-3 Analyzing using the program's object list

1-3-1 The program has the structures **sbc400_carrier** and **wa_scarr** and the elementary data object **pa_car**.

1-3-2 The variable pa_car belongs to the input field of the same name.

1-3-3 Screen **100** is processed using the statement **CALL SCREEN 100.**.

1-3-4 The field name appears in an input field above the area for the screen layout.

1-4

1-4-1 The list is structured using the **WRITE** statement.

1-4-2 The **SELECT** statement is responsible for the database dialog. The data is read from the database table **SCARR**. The database table name is specified in the **FROM** clause of the **SELECT** statement. The database table has the **MANDT**, **CARRID**, **CARRNAME**, **CURRCODE** and **URL**.

1-4-3 The information on the line to be read is in data object **pa_car**. This is in the **WHERE** clause of the **SELECT** statement. Data object pa_car is automatically filled with the selection screen input value as soon as the user chooses the Execute function on the *selection screen*.

**Unit: ABAP Workbench**

**Topic: Adapting a Program to Special Requirements**

**Model solution SAPBC400WBS_GETTING_STARTED**

```
*&---------------------------------------------------------------------*
*& Report        SAPBC400WBS_GETTING_STARTED                  *
*&---------------------------------------------------------------------*

REPORT  sapbc400wbs_getting_started   .

TABLES: sbc400_carrier.
DATA: wa_scarr TYPE scarr.
PARAMETERS: pa_car TYPE s_carr_id.


START-OF-SELECTION.
* Select all fields of one dataset from database table SCARR
  SELECT SINGLE * FROM scarr INTO CORRESPONDING FIELDS OF wa_scarr
                WHERE carrid = pa_car.
* At least one record could be selected
  IF sy-subrc = 0.
* Copy fields with corresponding names
    MOVE-CORRESPONDING wa_scarr TO sbc400_carrier.
    CALL SCREEN 100.
* Copy fields with corresponding names back
    MOVE-CORRESPONDING sbc400_carrier TO wa_scarr.

* Write data on list
    WRITE:   wa_scarr-carrid,
             wa_scarr-carrname,
             wa_scarr-currcode.
```

```
* add an empty line
   SKIP.
* add a horizontal line
   ULINE.
* write username, time and date on list
   WRITE:   sbc400_carrier-uname,
            sbc400_carrier -uzeit,
            sbc400_carrier -datum.
 ENDIF.
```

**Screen 100:**

New Fields on screen 100: **SBC400_CARRIER-UNAME**
**SBC400_CARRIER-UZEIT**
**SBC400_CARRIER-DATUM**

**Unit: ABAP Workbench**

**Topic: Creating a Transaction Code**

3-1 Follow the instructions in the *Creating a Transaction Code* slide.

3-2 Follow the instructions in the *Including a Transaction Code in SAP Easy Access* slide.

# ABAP Statements and Data Declarations

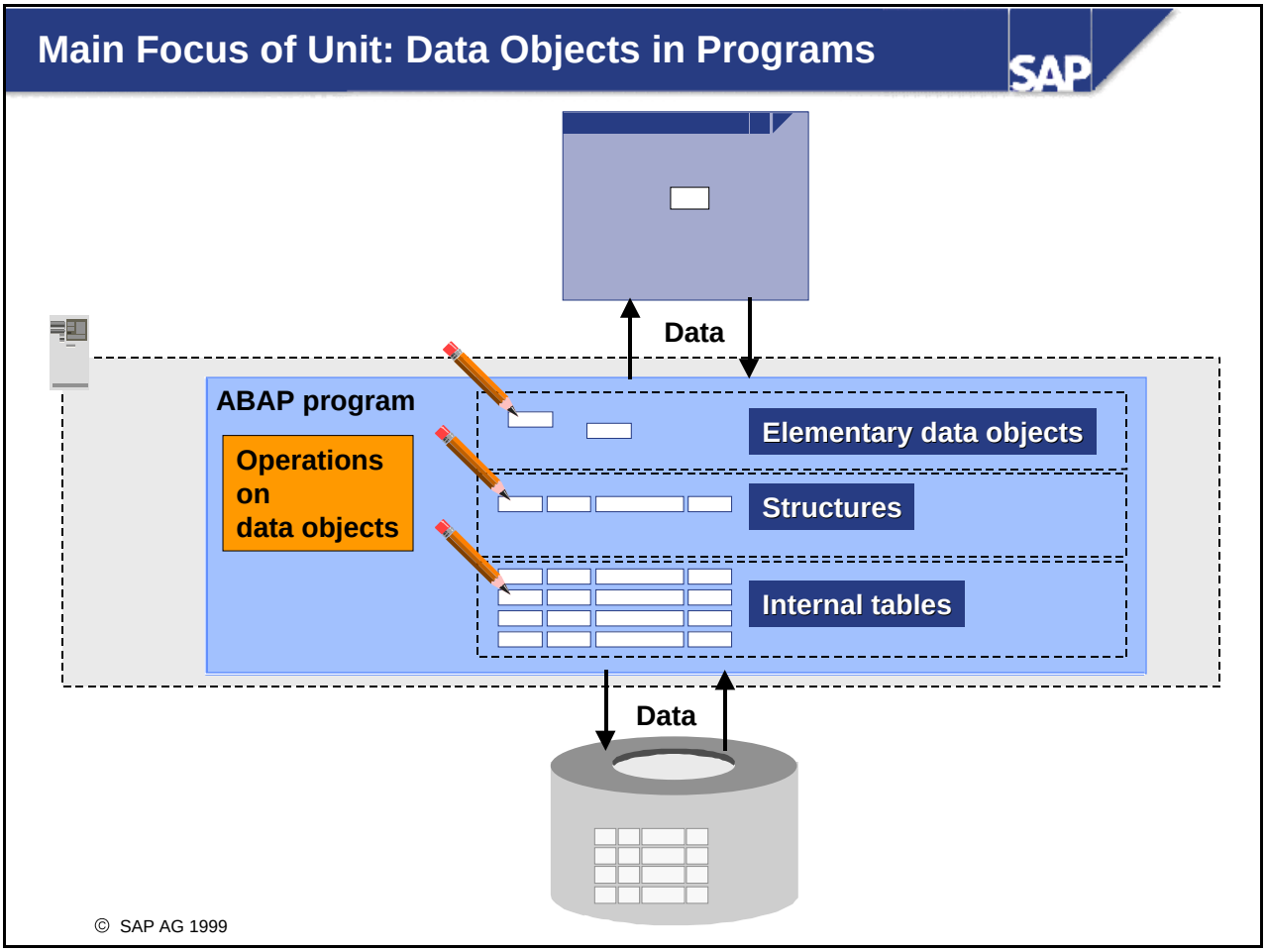
## Contents:

- Types
- Data objects
  - Elementary data objects
  - Structures
  - Internal tables
- Return codes and how to handle them

# ABAP Statements and Data Declarations: Unit Objectives

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

- **Describe the various different data types and their uses**
- **Define elementary data objects, structures, and internal tables**
- **Use Debugging mode to observe how the values of individual data objects change during processing**
- **Program several important operations involving data objects**
- **Find information about the various return codes used by ABAP statements and evaluate these in programs**



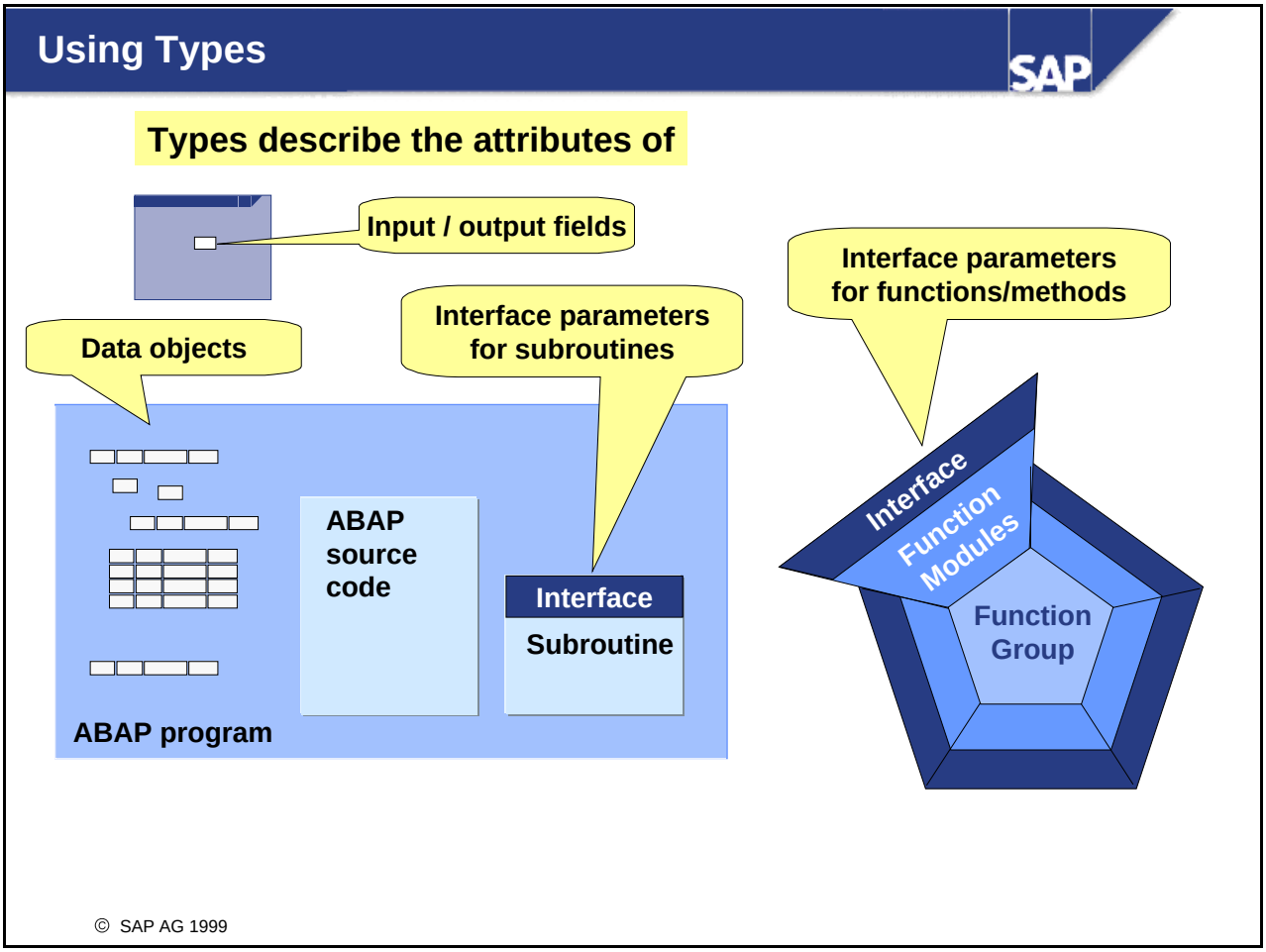

- This unit will focus on the definition of data objects, along with selected ABAP statements.

# Overview: Types

Types

Data objects

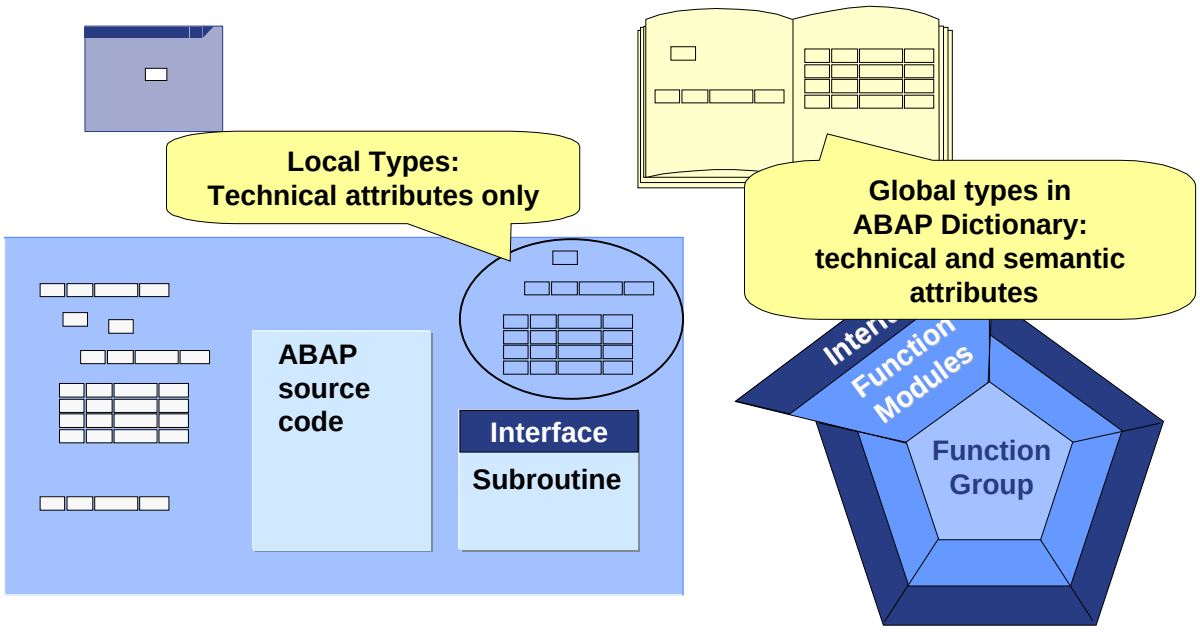
Return codes and how to handle them




- Types describe the attributes of
  - Input and output fields in screens
  - Data objects
  - Interface parameters: Type checks are performed each time a function or subroutine is called according to how the interface parameter is typed. Type conflicts are already identified during the editing process and appropriate syntax errors displayed.

# Attributes of Global and Local Program Types

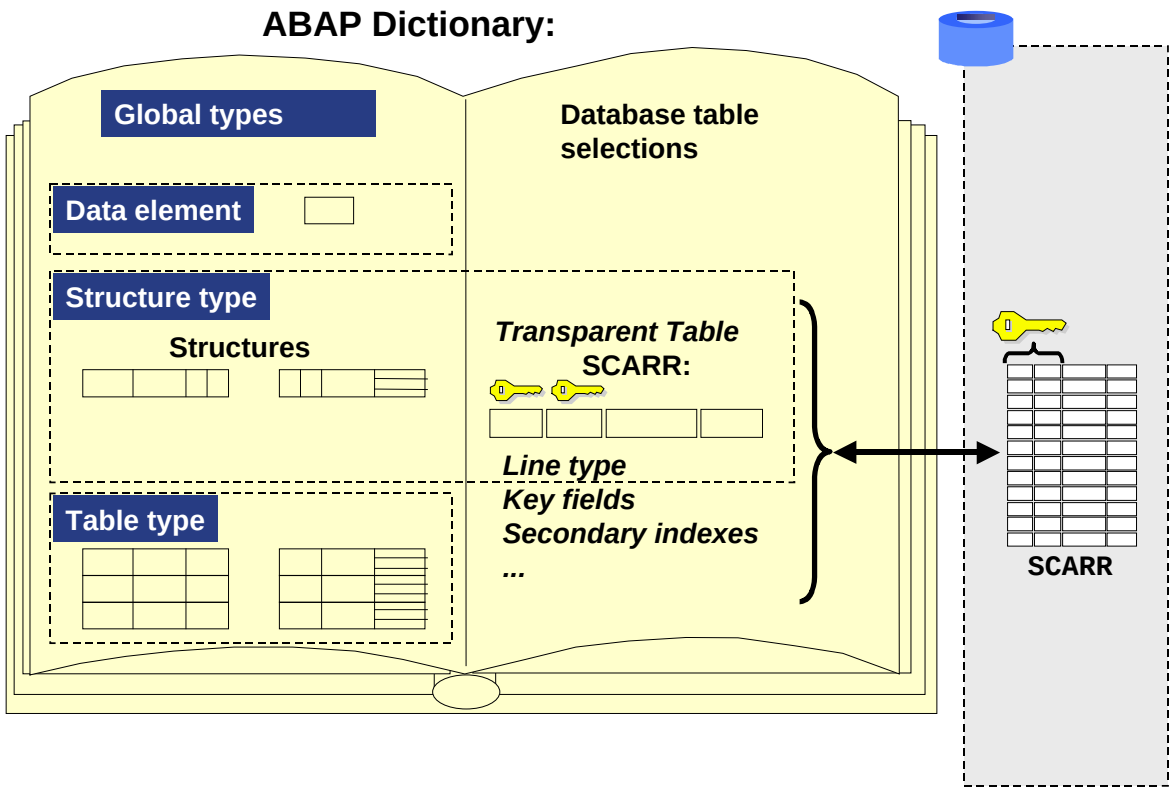
**Types can be defined locally in a program or centrally in the Dictionary**





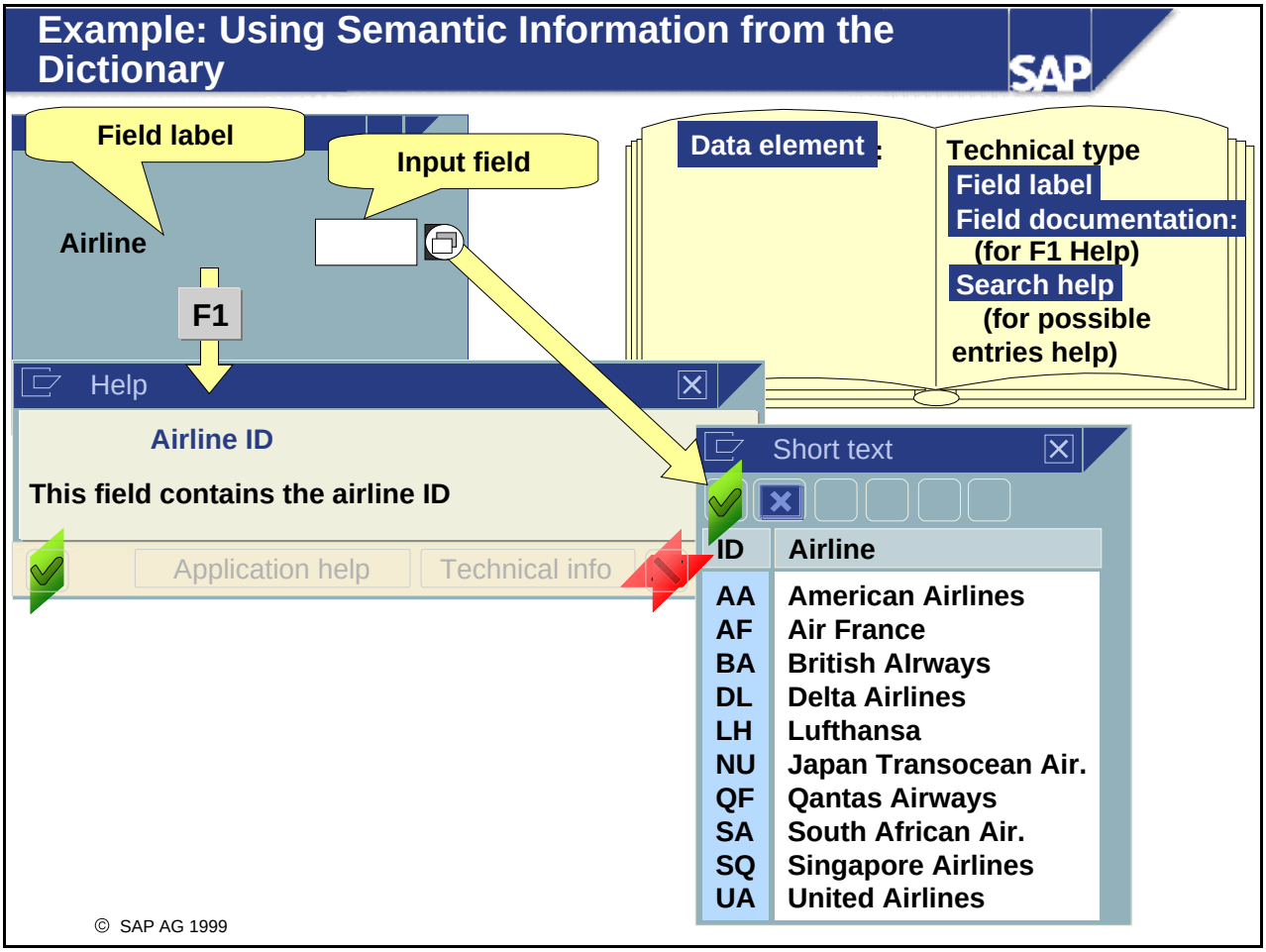

- **Local types** are used in programs
  - If only **technical** attributes are needed, not **semantic** attributes
  - If the types are **only used locally** within a program
- **Global types** ( = ABAP Dictionary types) are used
  - If you intend to use the types **outside of your program** as well
    (for example, for typing the interface parameters of global functions or with those data objects in the program that serve as the interface to the database or the presentation server)
  - If you need semantic information as well (for example, on screens with input and output fields)
- More information on storing semantic information centrally can be found in this unit.

# Global types in the ABAP Dictionary

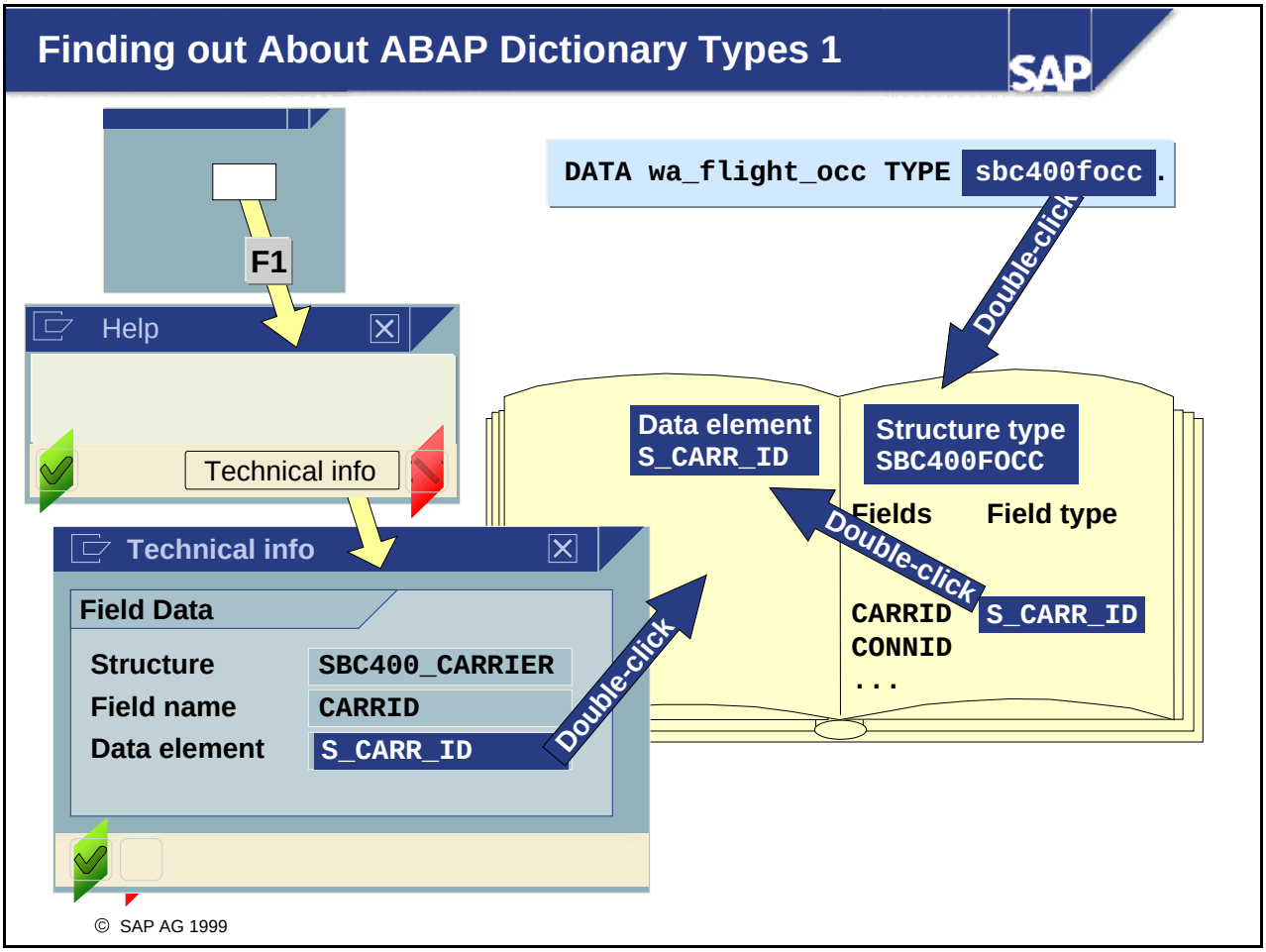





- You define global types and manage the descriptions of database tables in the *ABAP Dictionary*. You have the following options for global types:
  - Elementary types are called **data elements**. They contain a complete description of the technical attributes of an elementary field, along with semantic information.
  - Structure types are usually known as structures. They can consist of elementary components. Alternatively, each component can itself have a structured type. This allows you to construct nested structure types of any depth you want. Note that you can also use the line type of a transparent table (that is one defined as a database table) as a structured type.
  - Table types are types used to define internal tables.

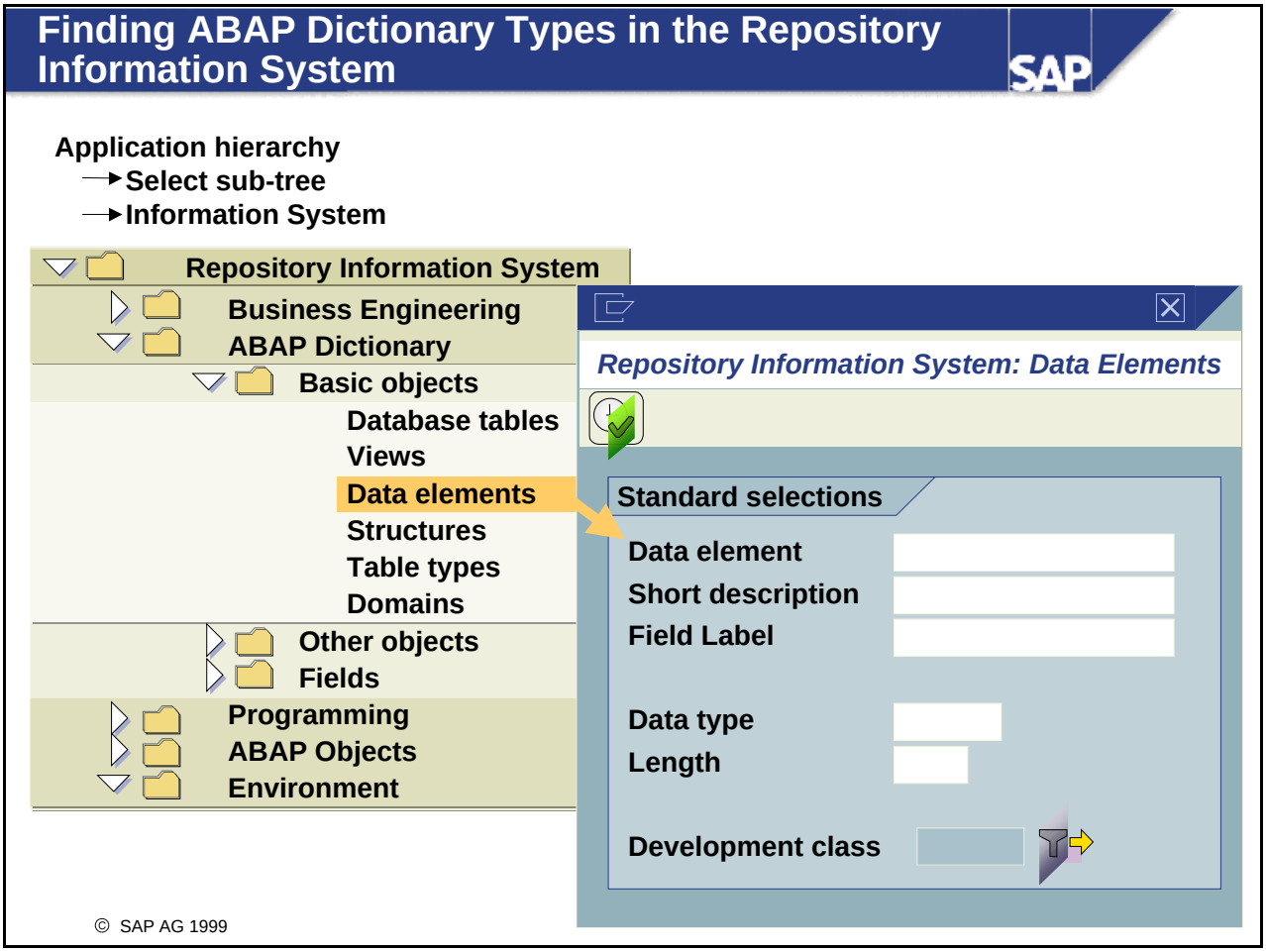



- Elementary Dictionary types are called **data elements**. They contain semantic as well as technical information (technical type, length, number of decimal places).
- A data element can contain the following semantic information:
  - Field Label: This text appears on screens and selection screens to the left next to the input or output fields. A field label can have one of three possible lengths. You must select one of the different field labels when you create a screen.
  - **Field documentation:** The field documentation tells the user what information should be entered in the field. The user gets the field documentation for an input or output field where the cursor is positioned by pressing function key F1.
  - **Search Help:** A data element can be linked to a search help. This search help defines the value help provided by function key F4 or the corresponding icon.



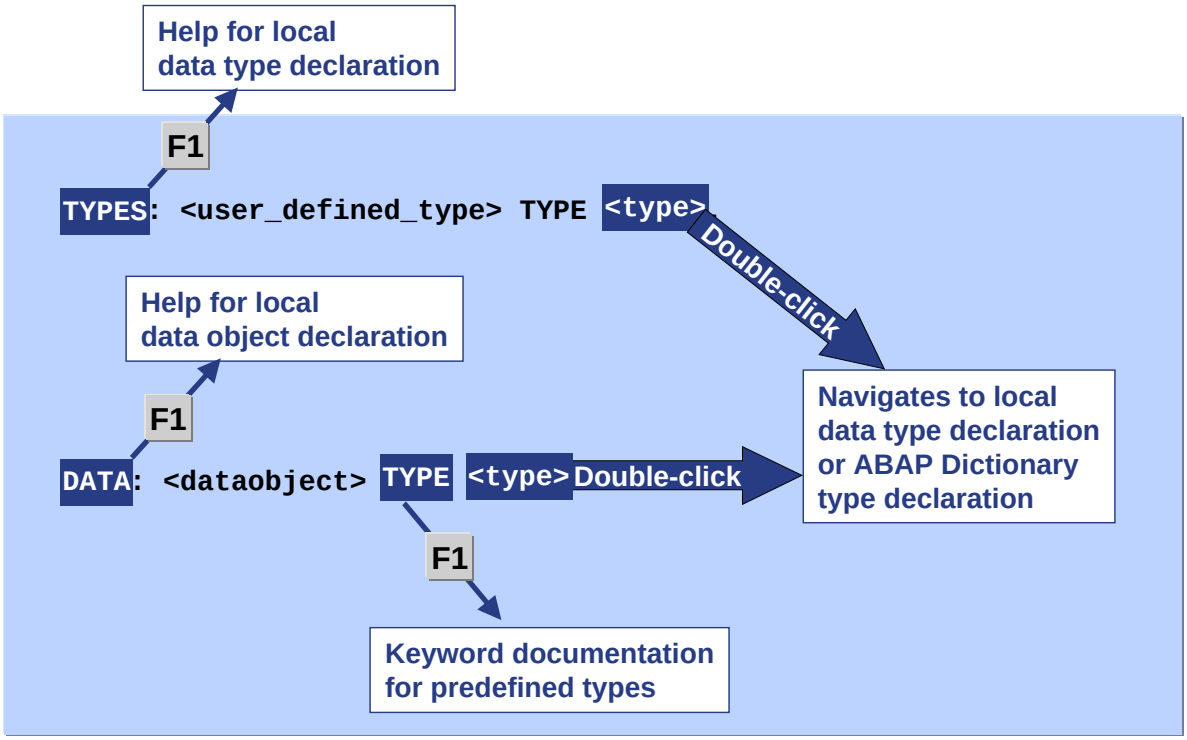

- You can find more information on elementary ABAP Dictionary types:
  - For **screen fields**: Using F1 -> *Technical info.* or by double-clicking the output field next to the data element
  - For **local types** in programs or **data objects**: By double-clicking the type
- Technical types and technical domains may be directly assigned to data elements. If you want more information on other data elements found within the same domain, you can navigate to the domain from the data element by double-clicking and then executing the function *Where-used list*.

# Finding ABAP Dictionary Types in the Repository Information System




- You can search for data elements by using the application hierarchy and the R/3 Repository Information System.
  - In the application hierarchy, select the components that you want to search.
  - Go to the R/3 Information System.
  - Choose *ABAP Dictionary --> Basic objects --> Data elements* and restrict the search.
- If you go to the R/3 Information System from the application hierarchy, the development classes of the selected application components are automatically entered.
- You can also go directly to the R/3 Information System. If you do not select a development class, the entire Repository is scanned.

## Local Data Types in Programs



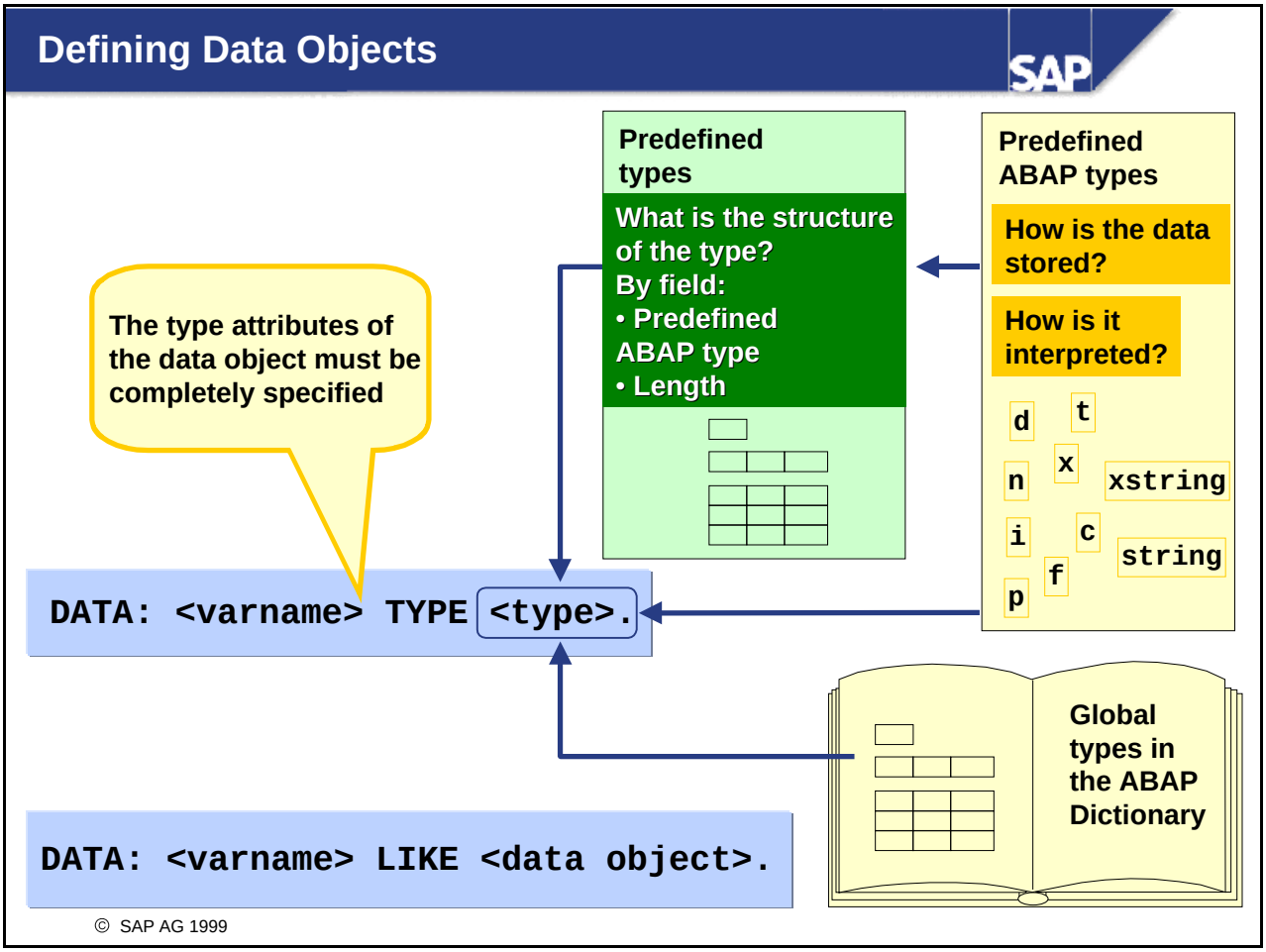



- Generally, data objects are typed using either a complete local program type or a complete global type. Double-click the name of the type to display its definition. For local program types this means: Navigate to the line in the source codes where the type has been defined. For global program types this means: Navigate to the Dictionary and display the global type.
- You can use the complete predefined ABAP types directly, to provide a type for variables. If you do so, double-clicking on the type after the **TYPE** statement has no effect. For more information on pre-defined types, refer to the keyword documentation on **TYPES** or **DATA.**
- The following predefined ABAP types are complete:
  - **d** **D**ate: (YYYYMMDD)
  - **t** **T**ime: HHMMSS)
  - **i** **I**nteger
  - **f** **F**loating Point Number
  - **string** character **S**tring (string, of variable length)
  - **xstring** byte sequence (he**X**adecimal string, of variable length)
- You must define the length for these types.
  - **c** **C**haracter

- **n** Numeric text (**N**umeric Character)
- **x** Byte (he**X**adecimal)
- **p** **P**acked number (= Binary Coded Decimals). You must enter the number of decimal places.

# Overview: Data objects

- Types
- Data objects
  - Elementary data objects
  - Structures
  - Internal tables
- Return codes and how to handle them

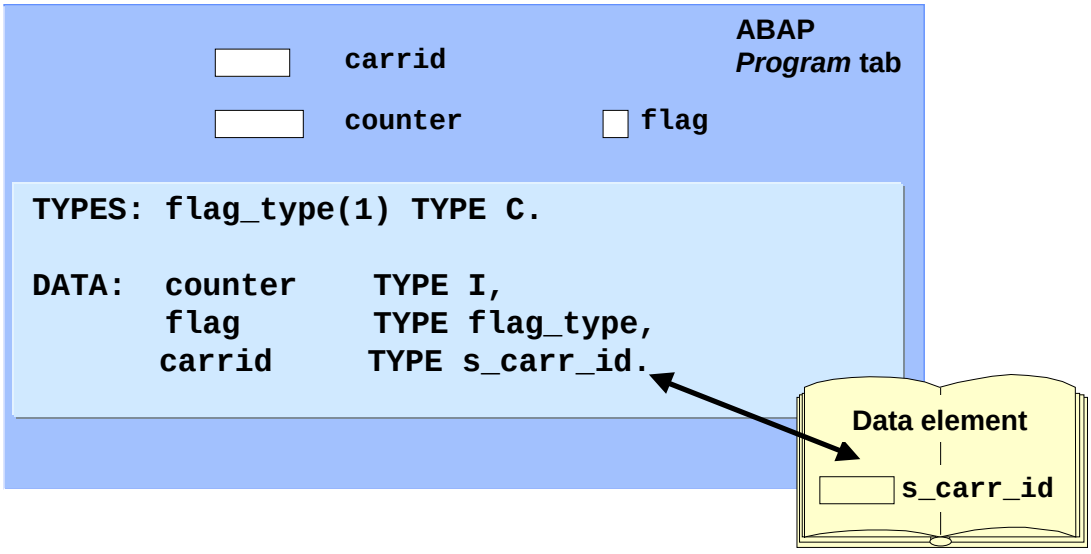



- Assign data object types by referring your object to either a built-in ABAP type, a user defined type, or an ABAP Dictionary object.
- If a variable **v2** refers to variable **v1** using the addition **LIKE (DATA v2 LIKE v1)**, then **v2** inherits its type from **v1**.
- Up to Release 4.0B, you could only refer to Dictionary types using **LIKE.** Only structure fields could be used as elementary types up to that point. Only flat structures were provided as structure types.

# Overview: Elementary Data Objects

- Types
- Data objects
  - Elementary data objects
  - Structures
  - Internal tables
- Return codes and how to handle them

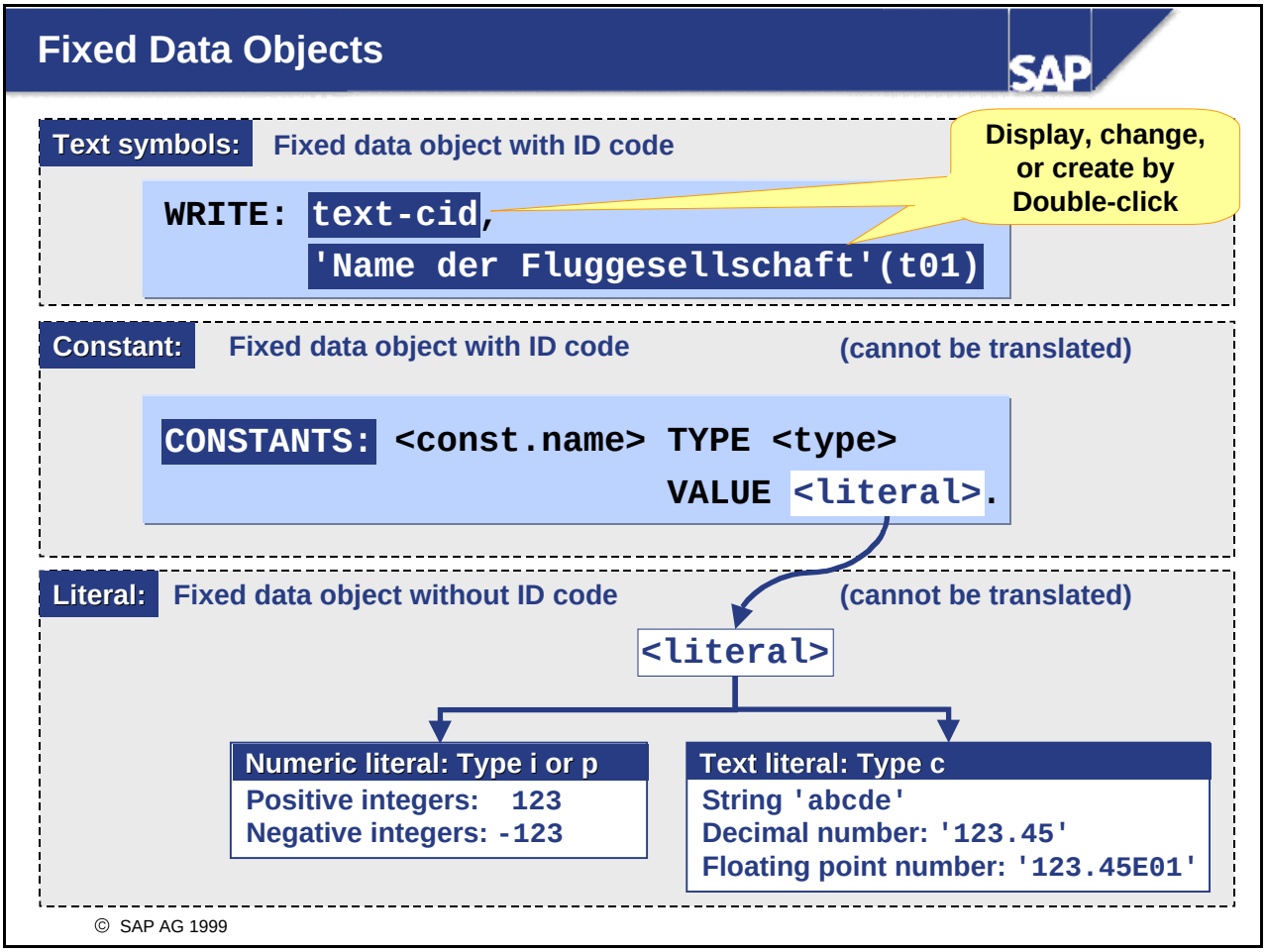
# Syntax Example: Defining Elementary Data Objects





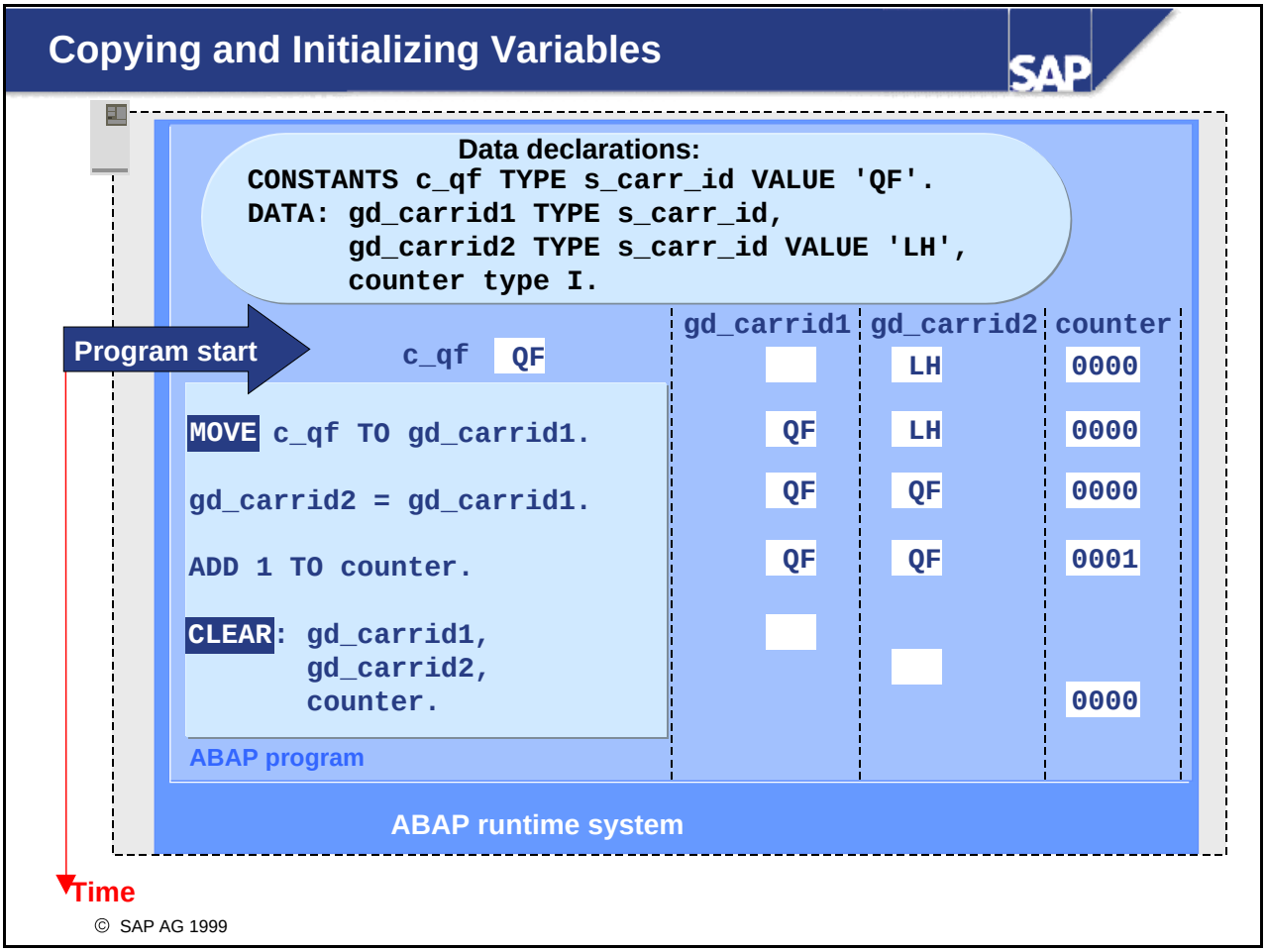

- You can define an elementary data object as follows:
  - **DATA <do name> TYPE <predefined ABAP Type>.**
    if you want to define an object with the type **d**,**t**,**i**,**f**,**string** or **xstring**. These predefined ABAP types are complete.
  - **TYPES <type name>(<length>) TYPE <predefined ABAP Type>. DATA <do name> TYPE <type name>.**
    if you want to defined an object with the type **c**, **n**, **p**, or **x**. You must define the length for these types. For type **p** objects, you can also define the number of decimal places using the **DECIMALS <nn>** addition.
  - **DATA <do name> TYPE <global type>.**
    if there is a suitable global type defined in the Dictionary.
- For more information, see the keyword documentation for the **DATA** statement.
- For compatibility reasons, you can still construct data objects in the **DATA** statement without first having to define the type locally in the program with a **TYPES** statement. Default values are also defined in addition to the type information for the following generic types:
  - With data types **p**, **n**, **c**, and **X** you can enter a length in parentheses after the type name. If no length is entered, the default length for this data type is used. You can find the standard lengths in the keyword documentation for **TYPES** and **DATA.**

- With data type **P** you can use the **DECIMALS** addition to determine the number of decimal places that should be used (up to a maximum of 14). If this addition is missing, the number of decimal places is set to zero.
- If you do not specify a type, then the field is automatically type C.




- You assign a value to each fixed data object in the source code. These cannot be changed at runtime.
- You can create **translatable text literals,** or ***text symbols***, for all ABAP programs. A three-digit code, **<xxx>**, is assigned to each text symbol. In the program, you can address it using **text-<xxx>**. **Example: WRITE text-cid.**
  Users then see the text in lists, in their logon language. In the program source code, you can navigate to the current version of the text by double-clicking. To open the maintenance environment for text symbols in the Editor, choose the menu entry *Goto-> Text elements.*
- You define **constants** using the ABAP keyword **CONSTANTS**. Their type is defined similarly to the type of a variable in the **DATA**.statement. You can assign a value using a literal, with the **VALUE** addition.
- ABAP recognizes two types of literals: **number literals** and **text literals**. The latter are always enclosed in inverted commas (').
  - Whole numbers (with preceding minus sign if they are negative) are stored as number literals of type **i** (provide they fall within the value range of four-byte integers). Otherwise, they are stored internally as packed numbers.
  - All other literals (character, numbers with decimal places, floating point numbers) are stored as text literals with data type C. If a literal is assigned to a variable that does not have type C, then a type conversion is carried out. The conversion rules are described in the keyword documentation about **MOVE**.

- If you want to include an inverted comma (') in a text literal, you must enter it twice.




- When a program is started, the program context is loaded into a storage area of the application server and made available for all the data objects.
- Each elementary field comes as standard with an initial value appropriate to its type. You can set an start value for an elementary field yourself using the **VALUE** addition. After **VALUES** you may only specify a fixed data object.
- You can copy the field contents of a data object to another data object with the **MOVE** statement. If the two data objects have different types, the type is automatically converted if there is a conversion rule. If you want to copy the field contents of variable **var1** to a second variable **var2**, you can choose one of two syntax variants:
  - **MOVE var1 TO var2.**
  - **var2 = var1.**
- You can find detailed information about copying and about the conversion rules in the keyword documentation for **MOVE** or in the *BC402* training course.
- The **CLEAR** statement resets the field contents of a variable to the initial value for the particular type. You can find detailed information about the initial values for a particular type in the keyword documentation about **CLEAR**.

## Performing Calculations

**COMPUTE** performs calculations (keyword is optional)

**Data declarations:**

```
DATA: gd_max TYPE sbc400focc-seatsmax,
      gd_occ TYPE sbc400focc-seatsocc,
      gd_percentage TYPE sbc400focc-percentage.
```

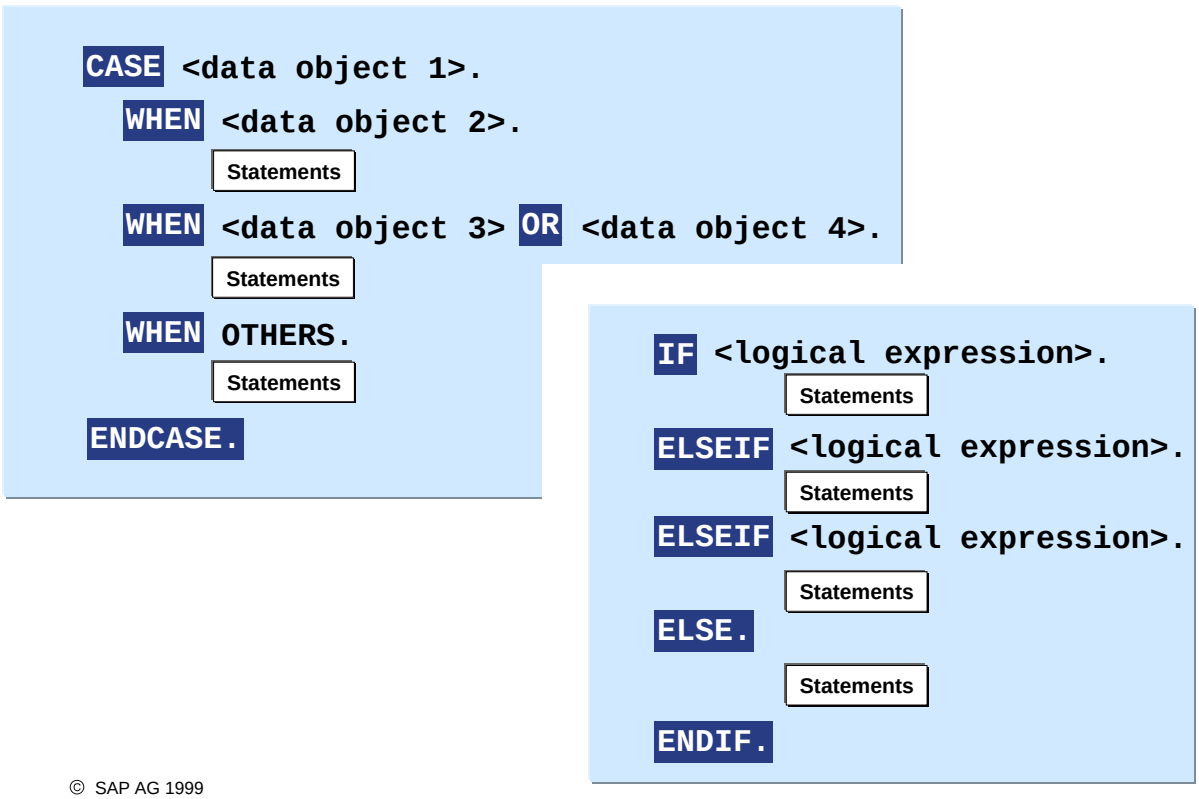
```
COMPUTE  gd_percentage = gd_occ * 100 / gd_max.

* Second possibility with same calculation


gd_percentage = gd_occ * 100 / gd_max.
```

ABAP program



- You can precede calculations with the **COMPUTE** statement. (This statement is optional). You can use either of the following two syntax variants to calculate percentage occupancy using the variable **v_occupancy** for 'current occupancy', **v_maximum** for 'maximum occupancy', and **v_percentage** for 'percentage occupancy':
  - **COMPUTE v_percentage = v_occupancy * 100 / v_maximum.**
  - **v_percentage = v_occupancy * 100 / v_maximum.**
- You can find detailed information on the operations and functions available in the keyword documentation on **COMPUTE**.

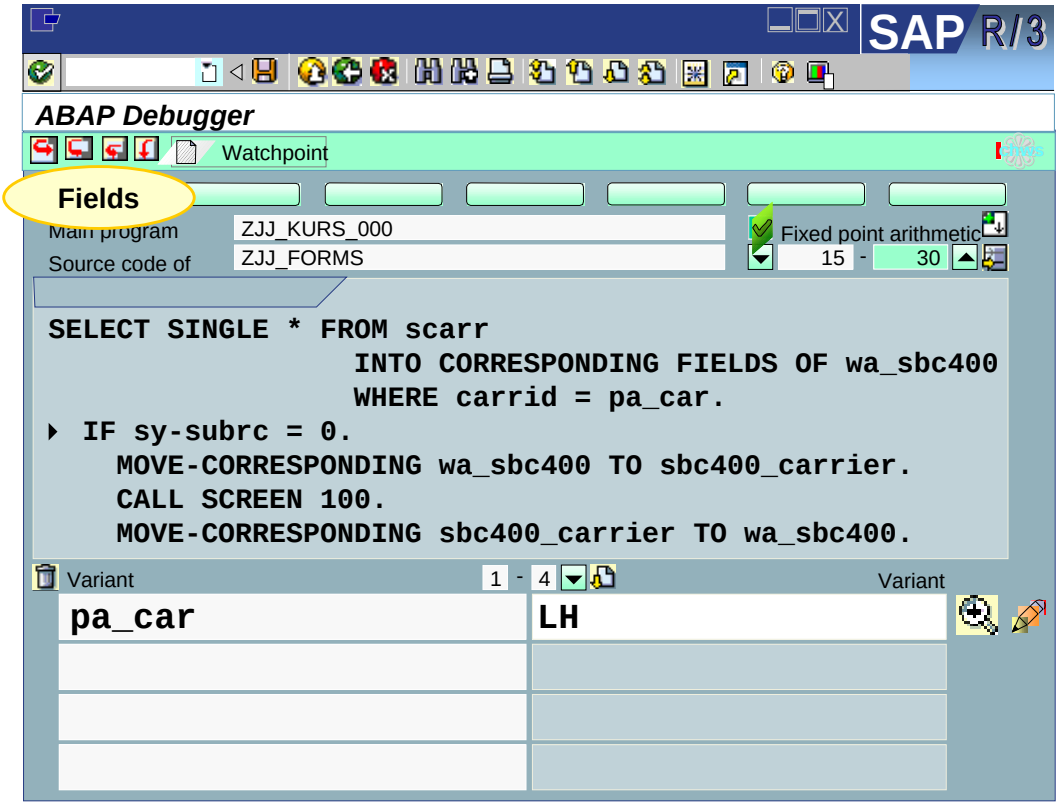
## Evaluating Field Contents






- **IF** and **CASE** statements allow you to make case distinctions:
- **CASE ... ENDCASE:**
  - Only one of the sequences of statements is executed.
  - The **WHEN OTHERS** statement is optional.
- **IF ... ENDIF**:
  - The logical expressions that are supported are described in the keyword documentation for the **IF** statement.
  - The **ELSE** and **ELSEIF** statements are optional.
  - If the logical expression is fulfilled, the following sequence of statements is executed.
  - If the logical expression is not fulfilled, the **ELSE** or **ELSEIF** section is processed. If there is no **ELSE** or no further **ELSEIF** statement, the program continues after the **ENDIF** statement.
  - You can include any number of **ELSEIF** statements between **IF** and **ENDIF**. A maximum of one of the sequences of statements will be executed.

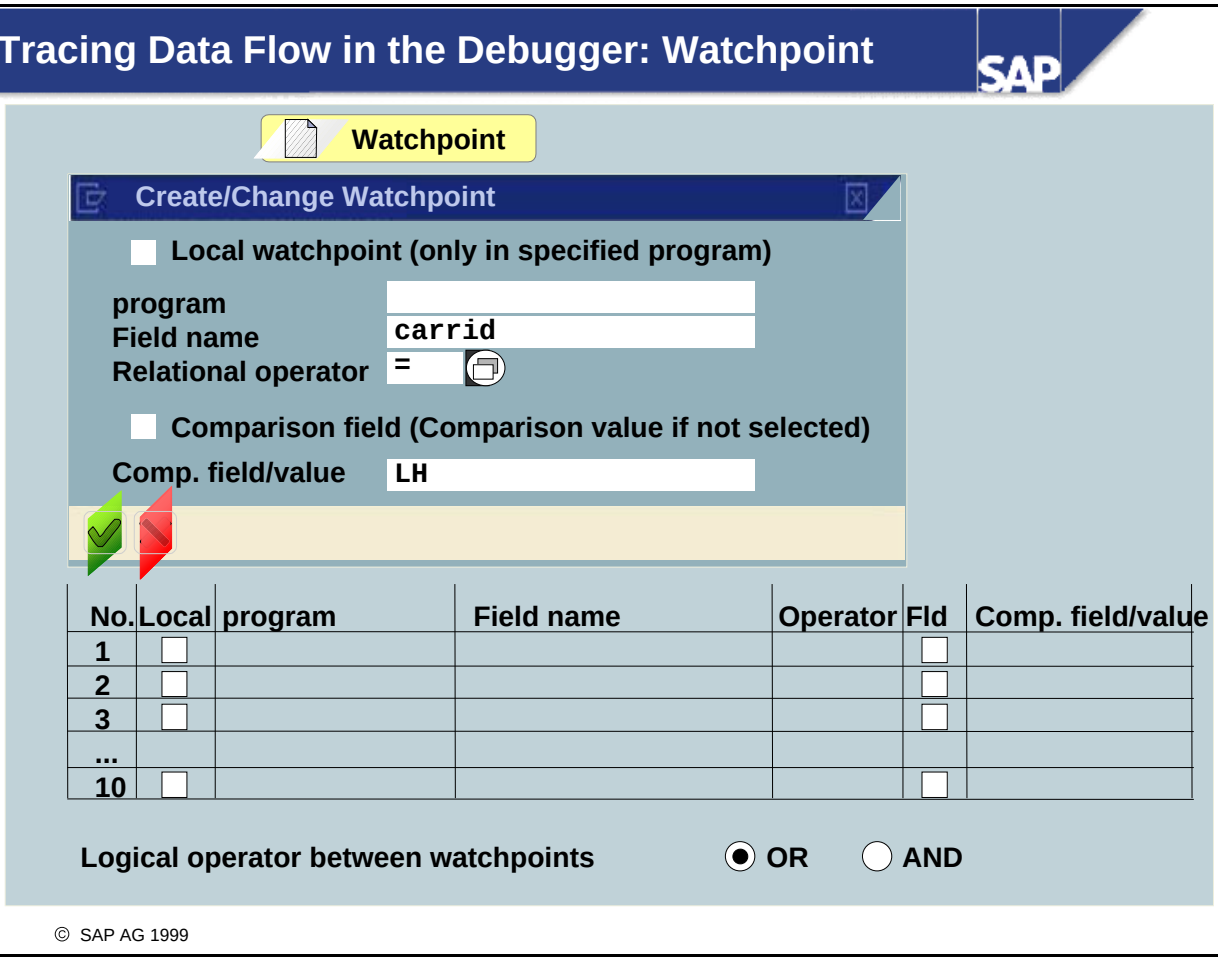
# Tracing Data Flow in the Debugger: Field View






- You can trace the field contents of up to eight data objects in debugging mode by entering the field names on the left side or by creating the entry by double-clicking on a field name.
- You can change field values at runtime by overwriting the current value and pressing the *Change* icon.

# Tracing Data Flow in the Debugger: Watchpoint



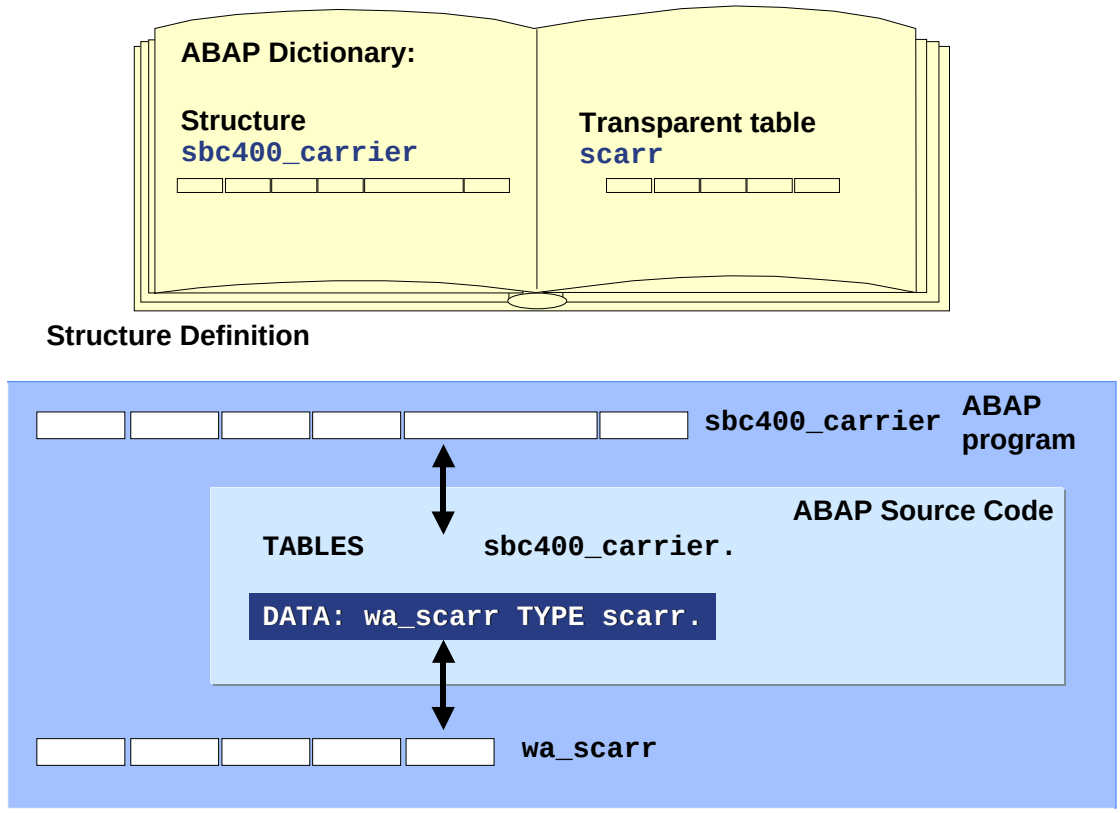



- From Release 4.6, you are allowed to set up to 10 watchpoints and link them using the logical operators **AND** and **OR**. Watchpoints are breakpoints that are field-specific. You can create the following types of watchpoints:
  - **Variable <operator> value:** The system stops processing once the logical condition is fulfilled. The 'Comparison field' flag is not selected and the value is inserted at 'Comp. field/value'.
  - **Variable1 <operator> variable2:** The system stops processing once the logical condition is fulfilled. The 'Comparison field' flag is selected and variable2 is inserted at 'Comp. field/value'.
  - **Variable:** The system stops processing each time the variable's value changes.
- You can also set a breakpoint for a specific ABAP statement. To do this, choose *Breakpoints-> Breakpoint at ...-> Statement*

# Overview: Structures

- Types
- Data objects
  - Elementary data objects
  - Structures
  - Internal tables
- Return codes and how to handle them

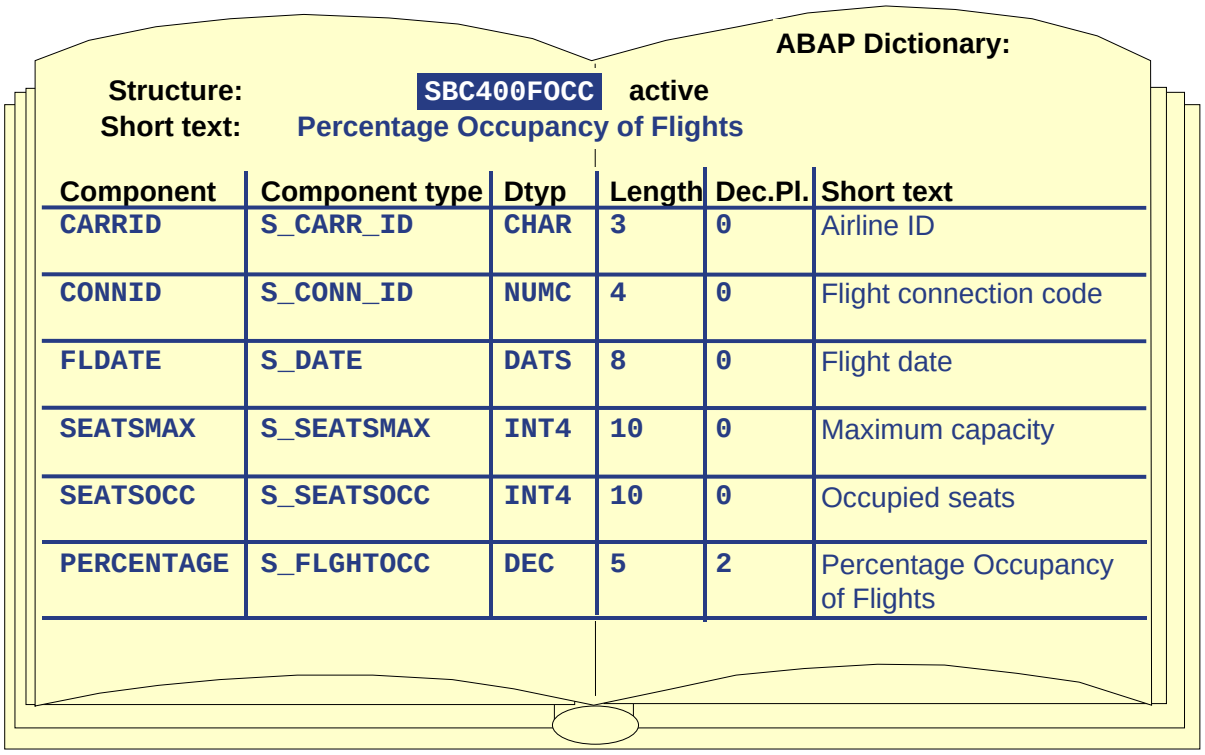
## Defining Structures with a Dictionary Type Reference






- You can define structured data objects (also called structures) in ABAP. This allows you to combine variables that belong together into one object. Structures can be nested. This means that other structures or even tables can be sub-objects of your original structure.
- There are two different kinds of structures in ABAP programs:
  - Structures defined using
    **DATA <name> TYPE <structure_type>.**
    These kinds of structures serve as the target fields for database accesses or for calculations performed locally within the program. You can declare these types of structures either in the ABAP Dictionary or locally within your program. For more information on how to declare local structures, refer to the keyword documentation on **TYPES**.
  - Structures defined using
    **TABLES <ABAP-Dictionary-Structure>.**
    These types of structures are technically administered in their own area. From Release 4.0, **TABLES** structures only serve as interface structures for screens. This statement may only be used for tables whose non-key fields are all numeric.

## Example: Dictionary Structure Type SBC400FOCC



**ABAP Dictionary:**

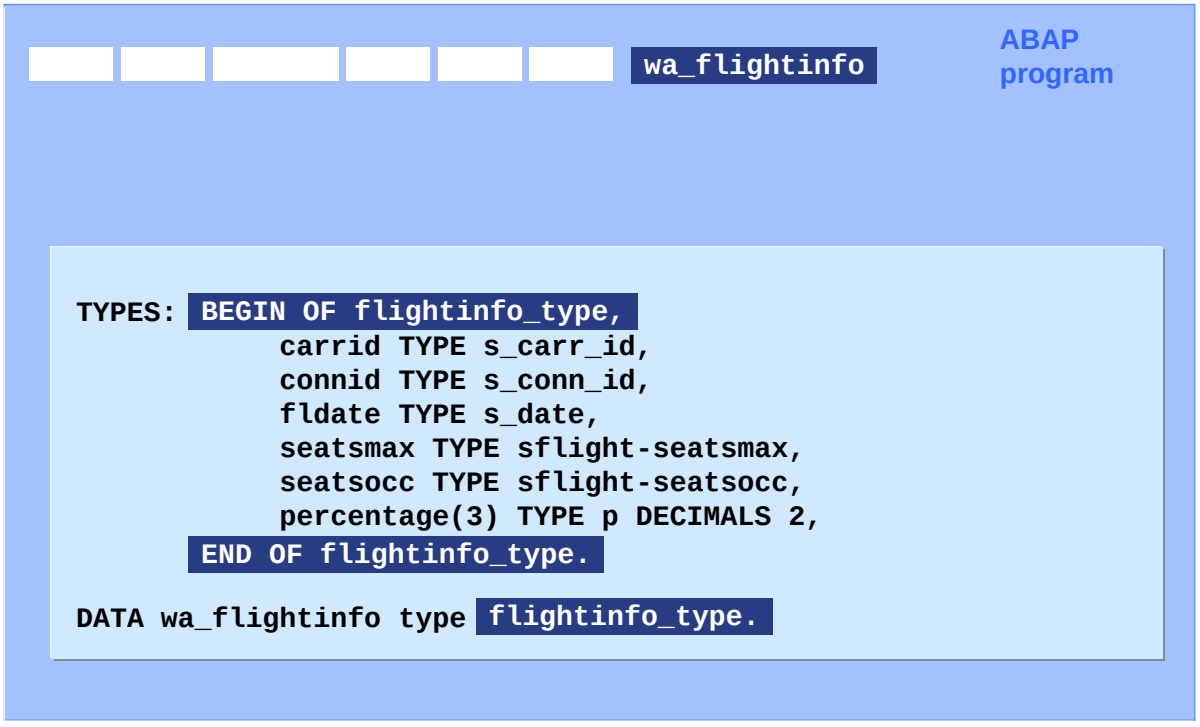
**Structure:** SBC400FOCC **active**
**Short text:** Percentage Occupancy of Flights

| Component | Component type | Dtyp | Length | Dec.Pl. | Short text |
|---|---|---|---|---|---|
| CARRID | S_CARR_ID | CHAR | 3 | 0 | Airline ID |
| CONNID | S_CONN_ID | NUMC | 4 | 0 | Flight connection code |
| FLDATE | S_DATE | DATS | 8 | 0 | Flight date |
| SEATSMAX | S_SEATSMAX | INT4 | 10 | 0 | Maximum capacity |
| SEATSOCC | S_SEATSOCC | INT4 | 10 | 0 | Occupied seats |
| PERCENTAGE | S_FLGHTOCC | DEC | 5 | 2 | Percentage Occupancy of Flights |



- Each component is assigned to a structure type. The following information is stored for each component
  - Component name: You can choose any name you want; there are no naming conventions. If the component is given the type of a data element that has a default name assigned to it, you should use this default value.
  - Component type: Generally, a data element is used to provide the type. If so, the component inherits all the type attributes of this data element. The type and length are displayed in the structure definition.
  - Short text: describes the component.
- Keep the following in mind, if you want to create a structure in the Dictionary:
  - The name of the structure must be in the customer namespace.
  - Currency fields usually have the type **CURR**. You must enter the associated currency field (type **CUKY**) as a reference field.
  - Lengths, weights, and other sizes measured in units generally have the type **QUAN**. You must enter the associated unit field (type **UNIT**) as a reference field.
- To find out the rules on mapping predefined Dictionary types and predefined ABAP types, see the SAP Library, under *Basis-> ABAP Workbench-> BC-ABAP Dictionary-> Data types in the ABAP Dictionary-> Mapping of ABAP data types*

## Syntax Example: Local Program Structure Types

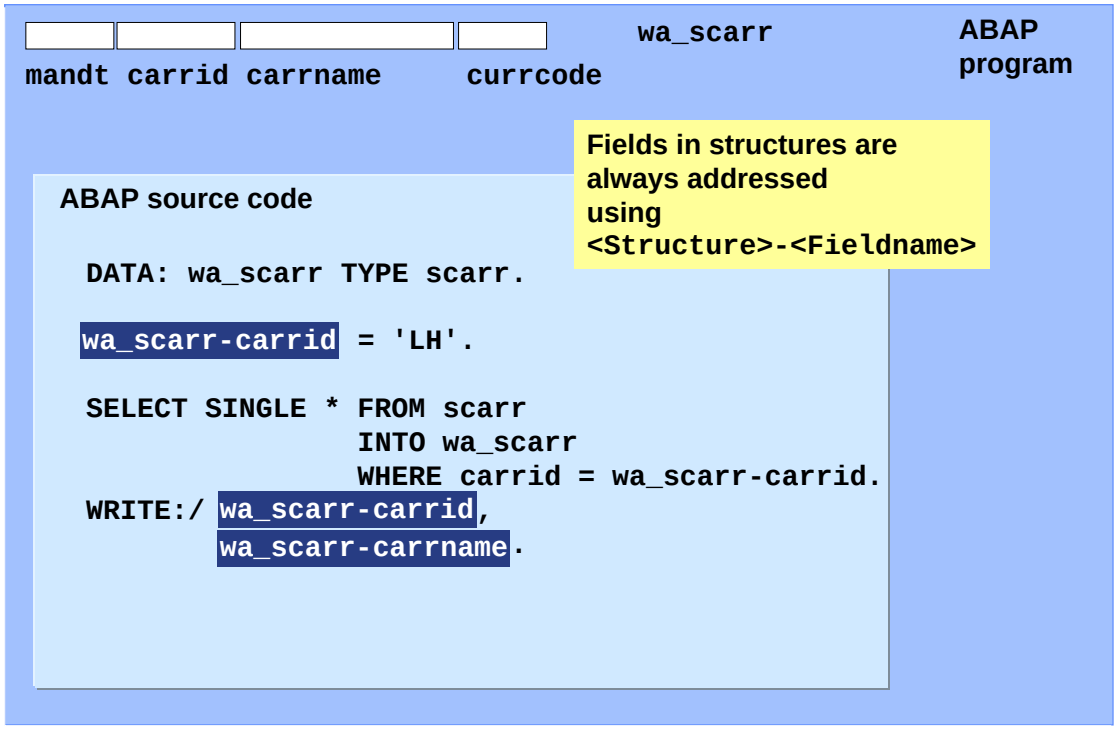





- You can also define structure types locally in programs using the **TYPE** statement. The components are enclosed within the statements
  **BEGIN OF <structure type.**
  **...**
  and

  **END OF <structure type>.**
- You must assign a name and type to each component.
- For more information, see the keyword documentation for the **SORT** statement.You can then create a structure using
  **DATA <structure name> TYPE <structure type name>.**

## Addressing Fields in Structures



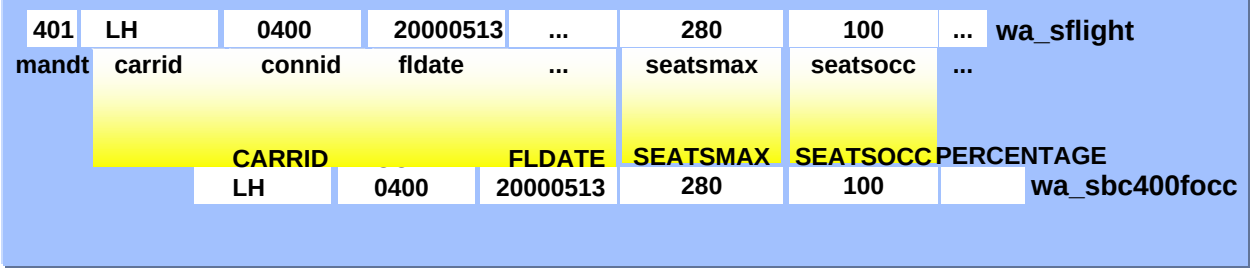



- Fields of a structure are always addressed using **<Structure>-<Field_name>**.
- This means that you should never use a hyphen in a variable name. The hyphen character is reserved for separating the structure name and the field name.

## Copying Identically-Named Fields Between Structures

```
MOVE-CORRESPONDING <rec1> TO <rec2>.
```

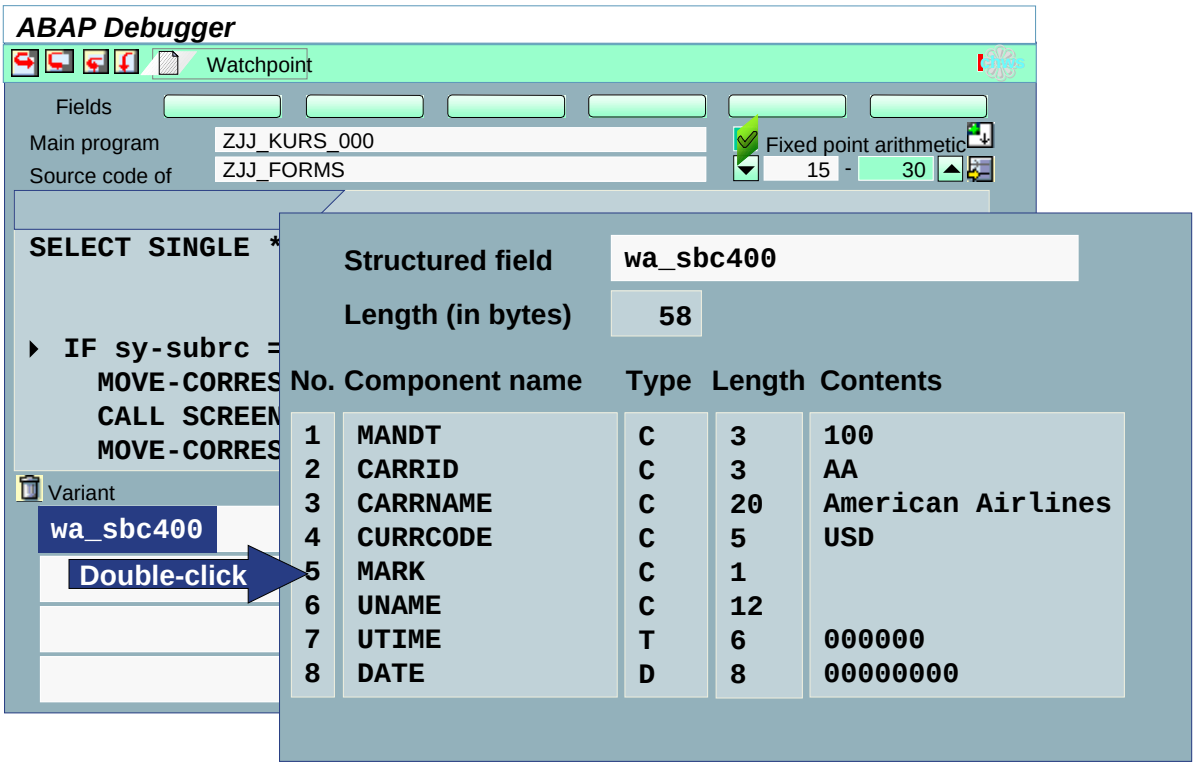
```
DATA: wa_sflight TYPE sflight,
      wa_sbc400focc TYPE sbc400focc.

MOVE-CORRESPONDING wa_sflight TO wa_sbc400focc.
```






- The **MOVE-CORRESPONDING <rec1> TO <rec2>** statements transports values field by field between structures <rec1> and <rec2>. This only works if the components have identical names.
- The system looks for all fields in **<rec1>** whose name also occurs in **<rec2>** and transports field **<rec1>-<field name>** to **<rec2>-<field name>** in all cases where it finds a match. All other fields remain unchanged.

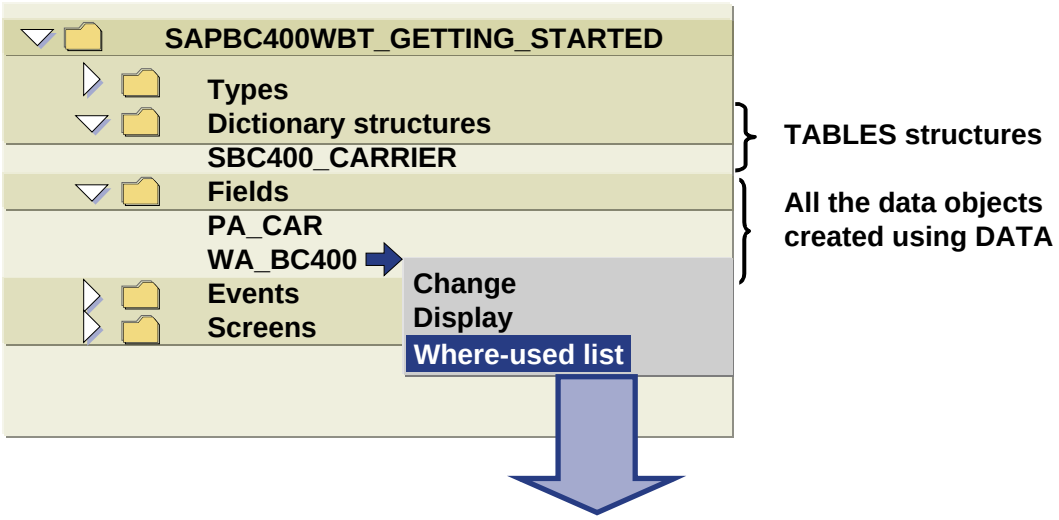
# Structures in the Debugger






- You can trace the field contents of a structure by entering the name of the structure in the left column. The field view of the structure is displayed if you double-click on this entry.

# Data Objects in a Program's Object List and in the Where-Used List

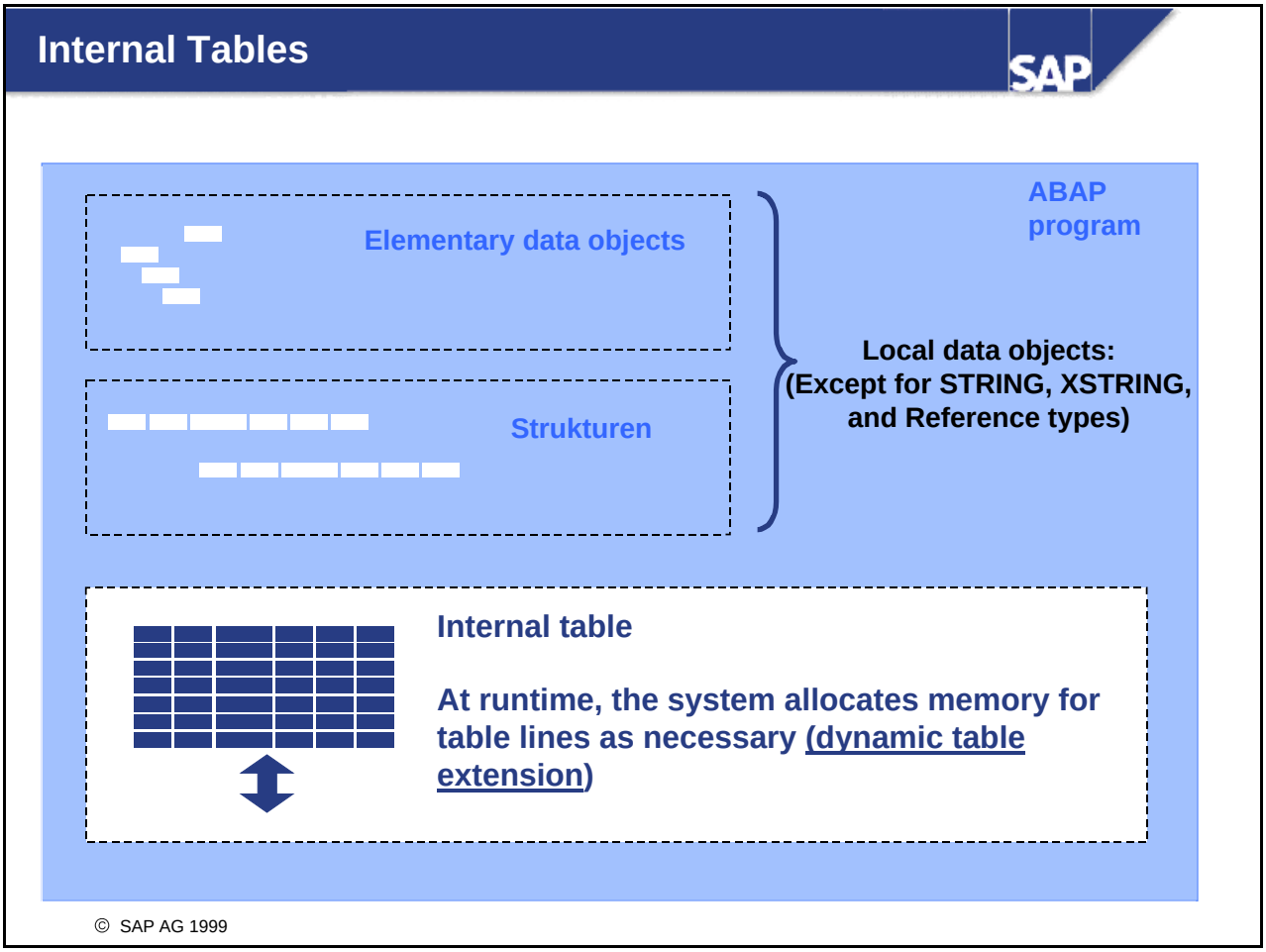





- Elementary data objects appear in the program object list under the *Fields* node.
- From the object list, you can use the right mouse button to ***navigate*** to the part of the source code where the data object is defined.
- You can use the ***Where-used list*** function to display all lines of source code where the data object is used.
- Structures created using the **TABLES** statement are a special case. They are stored under the object type **Dictionary structures**, for historical reasons.

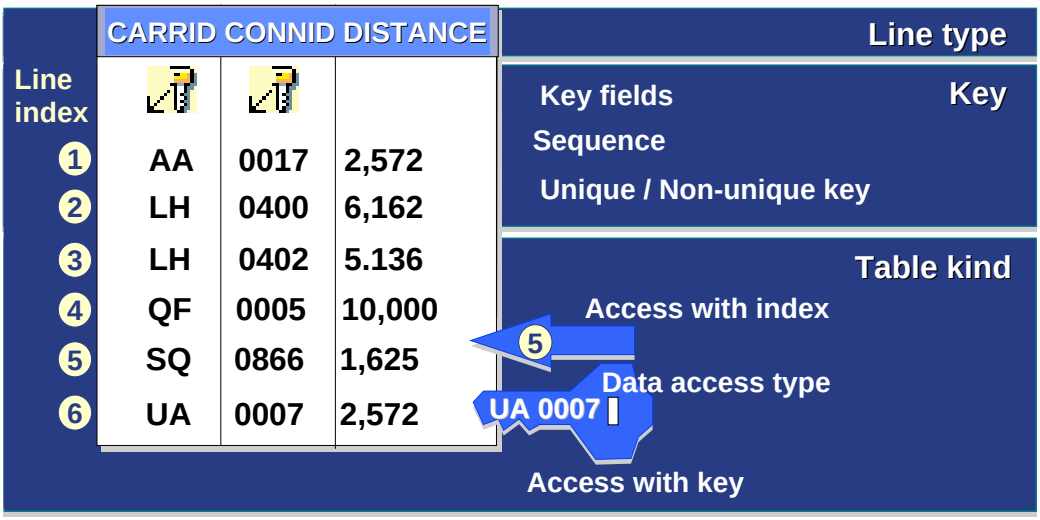
# Overview: Internal Tables

- Types
- Data objects
  - Elementary data objects
  - Structures
  - Internal tables
- Return codes and how to handle them




- Internal tables are data objects that allow you to retain data records in memory that are of one structure type, but several lines long. For this reason, you can refer to each component in a line as a column in an internal table.
- Internal tables are **dynamic data objects** made up of any number of lines that are all of the same type. The number of lines in an internal table is limited only be the capacity constraints of each specific system installation.
- Any ABAP data type can be used to form the line type of the internal table - elementary, structured, or another internal table.
- Each line in an internal table is known as a table row or table entry.

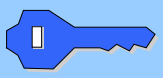
## Attributes of Internal Tables






- You must define the following information in order to specify a table type fully:
  - **Line type:** You can ensure the information about the required columns, their names and types, by defining a structure type as line type.
  - **Key:** A fully specified key must define: Which columns should be key columns? In what order? Should the key uniquely specify a record of the internal table (unique key)? Unique keys cannot be defined for all the table types.
  - **Table kind:** There are three kinds of table: standard tables, sorted tables and hashed tables. The estimated **access type** is mainly used to choose the table type.
- The **access type** defines how the runtime system accesses individual table entries. There are two different types of data access in ABAP, access using the index and access using a key.
- **Access using the index** involves using the data record index that the system maintains to access data.
  - **Example**: Read access to a data record with index 5 delivers the fifth data record of your internal table (Access quantity: one single data record).
- **Access using a key** involves using a search term, usually either the table key or the generic table key, to access data.
  - **Example**: Read access using the search term '**UA 0007**' to an internal table with the unique key **CARRID CONNID** and the data pictured above delivers exactly one data record.

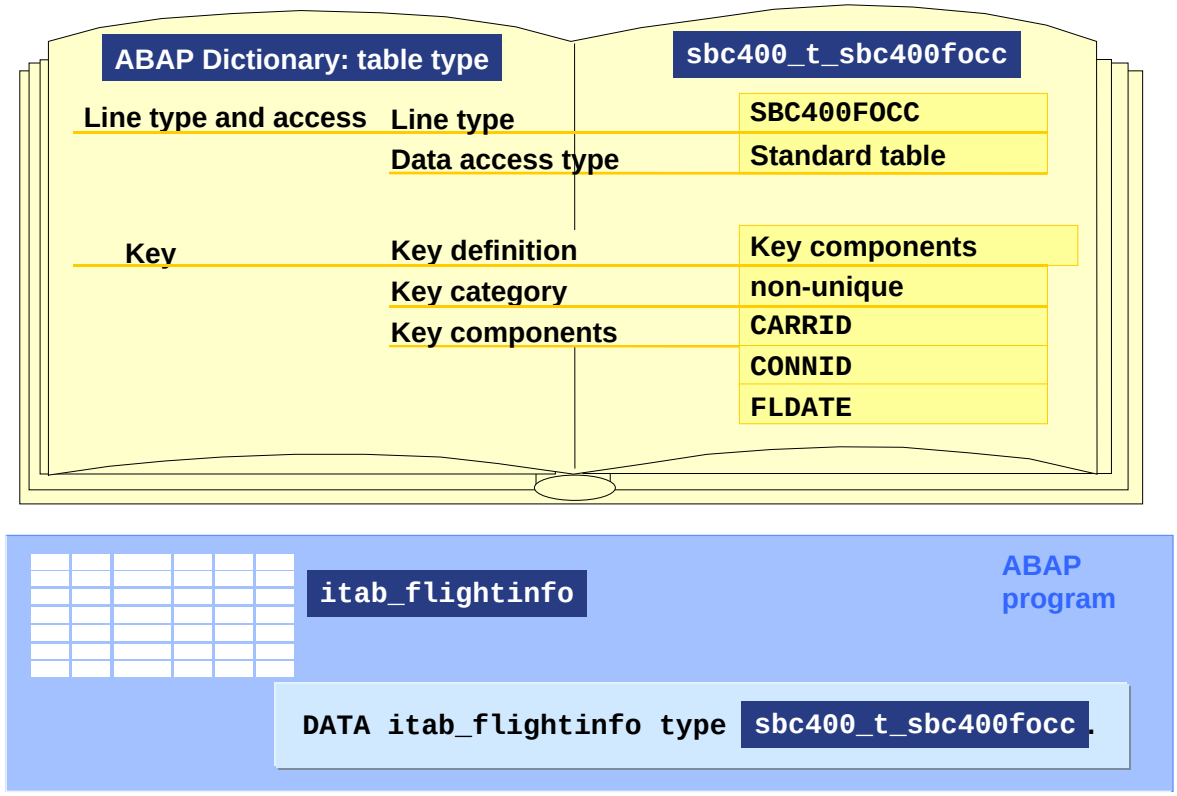
## The Relationship Between the Table Kind and the Access Type

| | Index tables | | Hash table |
|---|---|---|---|
| **Table kind** | STANDARD TABLE | SORTED TABLE | HASHED TABLE |
| **Access with index**  | ✓ | ✓ | |
| **Access with key**  | ✓ | ✓ | ✓ |
| **Unique / Non-unique key** | NON- UNIQUE | UNIQUE \| NON-UNIQUE | UNIQUE |
| **Access using** | Mainly index | Mainly keys | Keys only |



- Another internal table attribute is the table kind. Internal tables can be divided into three table kinds according to the way they access data:
  - **Standard tables** maintain a linear index internally. This kind of table can also be accessed using either the table index or keys.
  - **Sorted tables** are sorted according to key and saved. Here too, a index is maintained internally. This kind of table can also be accessed using either the table index or keys.
  - **Hashed tables** do not maintain an index internally. Hashed tables can only be accessed using keys.
- Which table type you use depends on how that table's entries are normally going to be accessed. Use standard tables when entries will normally be accessed using the index, use a sorted table when entries will normally be made using keys, and use hashed tables when entries will exclusively be made with keys.
- In this course, we will discuss the syntax of **standard tables only**.

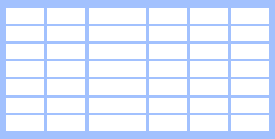
## Declaring Internal Tables with a Dictionary Type Reference






- Table types can be defined locally in a program or centrally in the ABAP Dictionary
- To define a table-type data object or an internal table, specify the type as a central/global table type or a local table type.
- For detailed information on the definition of global table types in the ABAP Dictionary, see the SAP Library under *Basis-> ABAP Workbench-> BC-ABAP Dictionary->Types->Table types.*

## Syntax Example: Local Table Types in Programs



**itab_flightinfo**

**ABAP program**

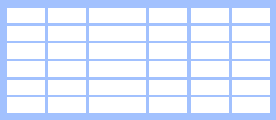
```
TYPES: flightinfo_type
            TYPE STANDARD TABLE OF sbc400focc
            WITH NON-UNIQUE KEY carrid connid fldate.


DATA itab_flightinfo type flightinfo_type.
```



- You can also define internal table types locally in programs using the **TYPE** statement. Enter the table kind after the **TYPE** statement, the line type after the **OF** addition, and the key after the **WITH** addition.
- You can find detailed information on declaring table types in the keyword documentation on the **TYPES** statement, or in *BC402 ABAP Programming Techniques*.

## Example: Filling Internal Tables Line by Line

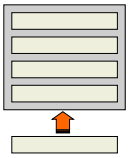
```
* Declaration of internal table and workarea
DATA: itab_flightinfo TYPE sbc400_t_sbc400focc,
      wa_flightinfo   LIKE LINE OF itab_flightinfo.
```



itab_flightinfo

wa_flightinfo

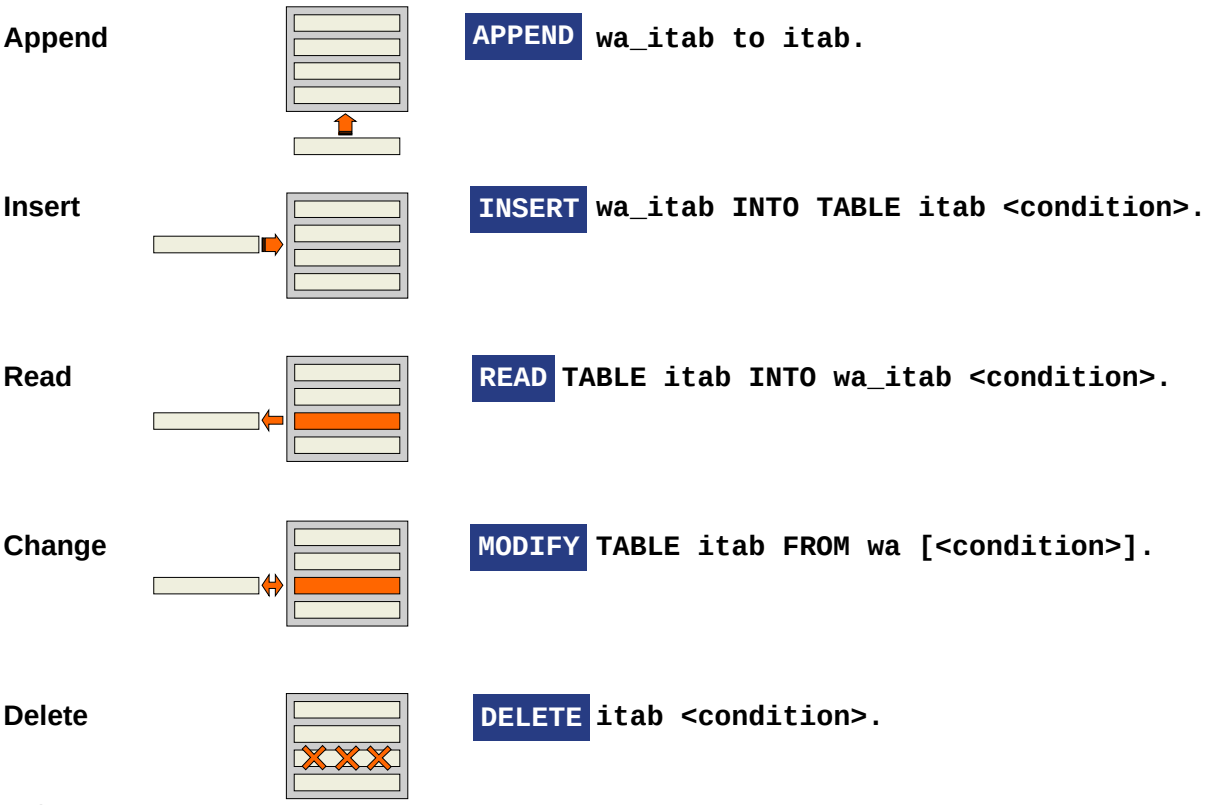
```
* Struktur wa_flightinfo mit Werten füllen
...
* Struktur wa_flightinfo in Interne Tabelle
* einfügen
INSERT wa_flightinfo INTO TABLE itab_flightinfo.
```





- You can add lines to a standard table by first filling a structure with the required values and then adding it to the internal table with the **INSERT** statement. For standard tables, this means that the line is appended to the end of the table. For sorted tables and hash tables, the system inserts the line after referring to the key.
- Note: There is a special statement, **APPEND**, that you can use with standard tables. This statement appends the line to the end of the standard table. This statement often occurs in programs, since sorted and hash tables were only introduced in Release 4.0. Both statements have the same effect on standard tables.

## Overview: Accessing Single Records

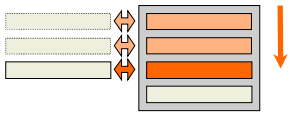





- The following single-record operations are provided for internal tables:
  - **APPEND** adds the contents of a structure (that has the same type as the line type) to the end of an internal table. This operation can only be used with standard tables.
  - **INSERT** inserts the contents of a structure that has the same type as the line type in an internal table. This causes a standard table to be appended and a sorted table to be inserted in the right place; a hashed table is inserted according to the hashing algorithm.
  - **READ** copies the contents of a line in the internal table to a structure that has the same type as the line type.
  - **MODIFY** overwrites a line of the internal table with the contents of a structure that has the same type as the line type.
  - **DELETE** deletes a line in the internal table.
  - **COLLECT** inserts the contents of a structure having the same type as the row in an internal table into an internal table in compressed form.. This statement may only be used for tables whose non-key fields are all numeric. The numeric values are added to the same keys.
- You can find detailed information about the ABAP statements described here in the keyword documentation for the relevant ABAP keywords.

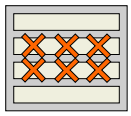
## Overview: Processing Sets of Records

**Using a loop:**
**for all operations**



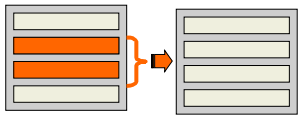
```
LOOP AT itab INTO wa_itab.

ENDLOOP .
```

**Delete**



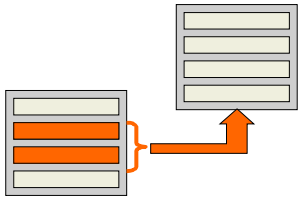
```
DELETE itab <condition>.
```

**Insert**
**multiple lines**
**from another**
**internal table**



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

**Append**
**multiple lines**
**from another**
**internal table**



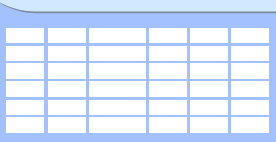
```
APPEND itab2 <condition2>
  FROM itab1 <condition1>.
```



- The following operations on sets are provided for internal tables:
  - **LOOP ... ENDLOOP** The **LOOP** places the rows of the internal table in the structure specified in the **INTO** clause one-by-one. The structure must have the same type as the row of the internal table. All single-record operations can be executed within the loop. In this case the system provides the information about the row to be edited for the single-record operations.
  - **DELETE** deletes the rows of the internal table that satisfy the condition <condition>.
  - **INSERT** copies the contents of several rows of an internal table to another internal table.
  - **APPEND** appends the contents of several rows of an internal table to another standard table.
- You can find detailed information about the ABAP statements described here in the keyword documentation for the relevant ABAP keywords.

## Example: Reading Internal Table Contents Using a Loop

```
* Declaration of internal table and workarea
DATA: itab_flightinfo TYPE sbc400_t_sbc400focc,
      wa_flightinfo   LIKE LINE OF itab_flightinfo.
```

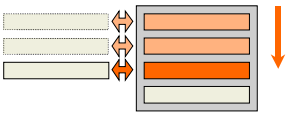


itab_flightinfo



wa_flightinfo

```
LOOP AT itab_flightinfo INTO wa_flightinfo.

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

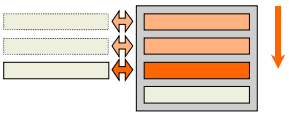




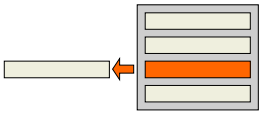
- You can read and edit the contents of an internal table using a **LOOP** statement. In this example, one line is copied from internal table **it_flightinfo** to structure **wa_flightinfo**. The fields of the structure can then be edited. You can then create a list from the fields using a **WRITE** statement.
- If you want to change the contents of the internal table, first change the value of the structure fields within the loop and then overwrite the line of the internal table with the **MODIFY** statement.

## Example: Reading Internal Tables Using the Index

```
LOOP AT itab_flightinfo INTO wa_flightinfo
                        FROM 1 TO 5.
  WRITE: /  wa_flightinfo-carrid,
            wa_flightinfo-connid,
            wa_flightinfo-fldate,
            wa_flightinfo-seatsmax,
            wa_flightinfo-seatsocc,
            wa_flightinfo-percentage,
            '%'.

ENDLOOP.
```



```
READ TABLE itab_flightinfo INTO wa_flightinfo
                           INDEX 3.
  WRITE: /  wa_flightinfo-carrid,
            wa_flightinfo-connid,
            wa_flightinfo-fldate,
            wa_flightinfo-seatsmax,
            wa_flightinfo-seatsocc,
            wa_flightinfo-percentage,
            '%'.
```

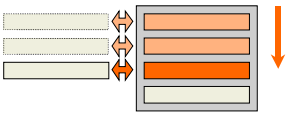




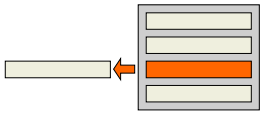
- You can restrict access to certain line numbers using the **INDEX** addition. You may only perform index operations on index tables. Both standard and sorted tables are supported here.
- The above example shows the syntax for loop editing that only scans the first five lines of the internal table.
- The example below shows the syntax for reading the third line of the internal table.

## Example: Reading Internal Tables Using Keys

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

ENDLOOP.
```



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

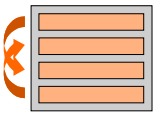




- You can restrict access to lines with certain values in key fields using the **WHERE** addition . Key operations are supported for all table types. Key access to sorted or hashed tables is more efficient than key access to standard tables.
- The above example shows the syntax for loop processing, which only scans the lines of the internal table whose '**carrid**' field has the value 'LH'. The sorted table is most suitable for this type of editing. Loop editing with the **WHERE** addition is supported for sorted and standard tables.
- The example below shows the syntax for reading a line in the internal table with a fully specified key. The return code sy-subrc is set to zero if the internal table contains this line. The hashed table is most suitable for single-record access by key. This type of access is supported for all table types. Note that all the key fields must be defined in key accesses with the **WITH TABLE KEY** addition. It is easy to confuse this addition with the **WITH KEY** addition, which permitted key access to standard tables prior to Release 4.0, when it was not yet possible to define key columns explicitly.

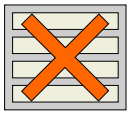
## Operations on the Whole Internal Table

**Sort**



```
SORT itab <conditions>
```

**Set the content of the internal table to initial**

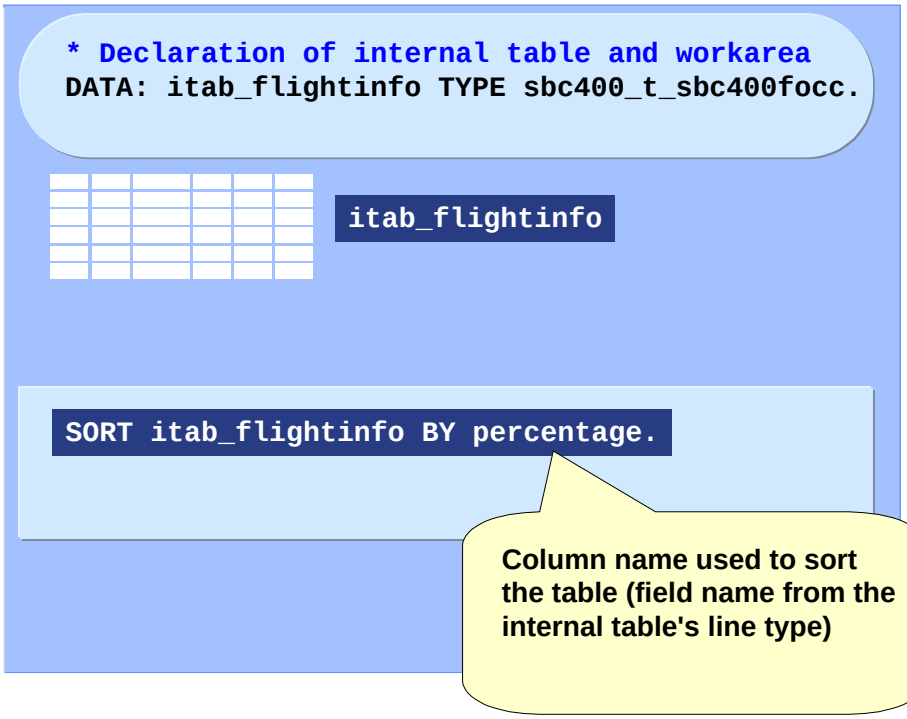


```
CLEAR itab.
```



- The following operations can be executed on internal tables:
  - **SORT** You can sort tables in increasing or decreasing order of any column. However, sorted tables cannot be resorted.
  - **CLEAR** Sets the contents of the internal table to the right initial value for the column type.
  - **REFRESH** works like **CLEAR**.
  - **FREE** Deletes the internal table and releases the memory.

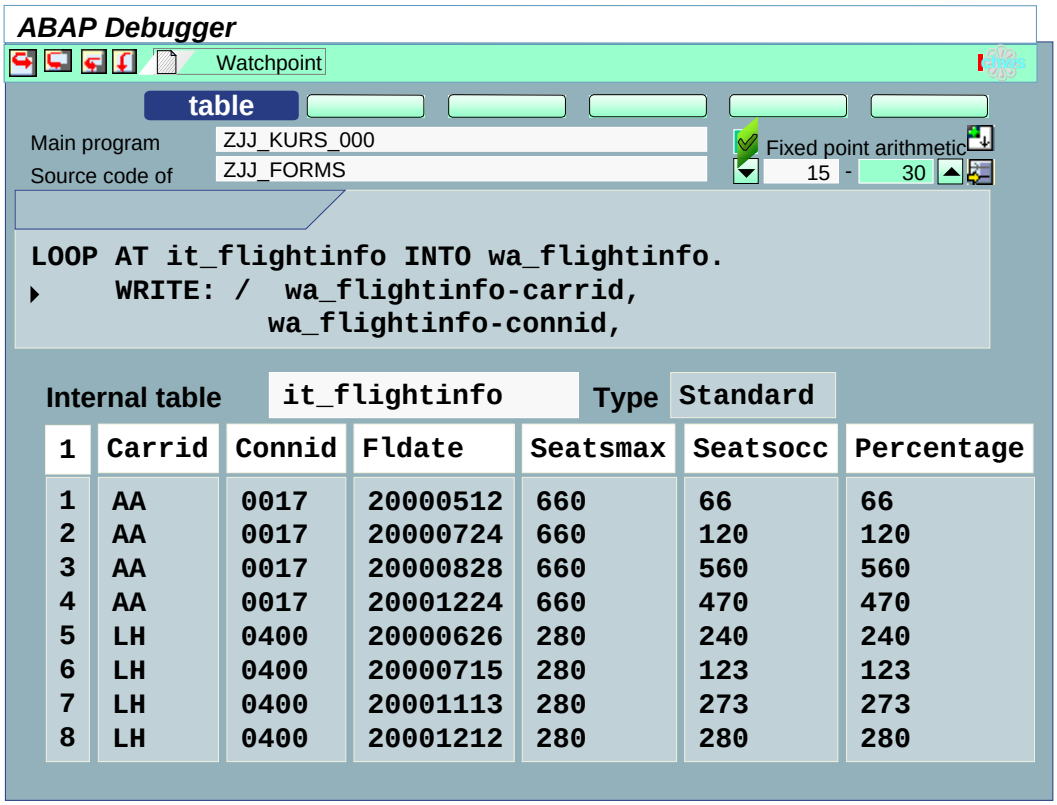
## Syntax Example: Sorting a Standard Table






- You can sort standard tables by any column, simply by entering this column name after a **BY** addition to the **SORT** statement.
- For more information, see the keyword documentation for the **SORT** statement.

# Internal Tables in Debugging Mode

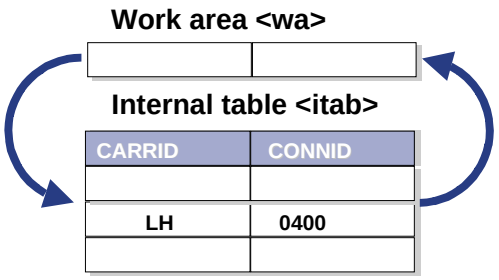


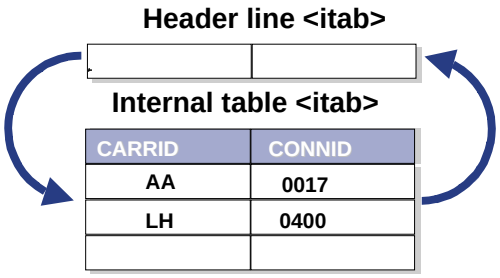



- You can trace the contents of an internal table in debugging mode by choosing *Table* and entering the name of the internal table.

## Internal Tables with Headers

```
DATA    <itab> TYPE <itabtype> [WITH HEADER LINE].
```




```
APPEND <wa> TO <itab>.
MODIFY <itab> INDEX <n> FROM <wa>.

LOOP AT <itab> INTO <wa>.
 WRITE <wa>-<feld>.
ENDLOOP.

READ TABLE <itab> INDEX <n> INTO <wa>.
WRITE <wa>-<feld>.
```

```
APPEND <itab>.
MODIFY <itab> INDEX <n>.

LOOP AT <itab>.
 WRITE <itab>-<feld>.
ENDLOOP.

READ TABLE <itab> INDEX <n>.
WRITE <itab>-<feld>.
```

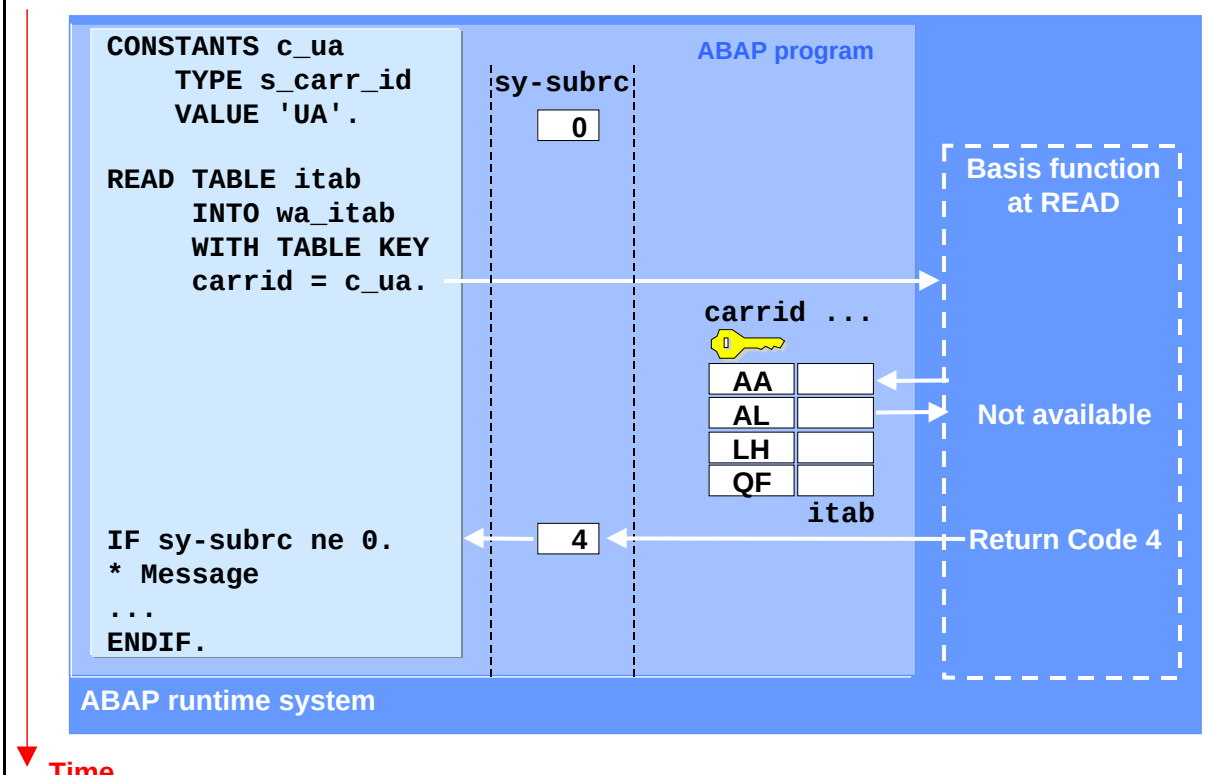


- Internal tables can be defined with or without a header line. An **internal table with header line** consists of a work area (the header line) and the actual body of table, both of which are addressed with the **same name**. How this name is interpreted depends on the context in which it is used. For example: at **MOVE** the name is interpreted to mean the header line, while at **SORT** it is interpreted as the table body.
- You can declare an internal table with a header line using the addition **WITH HEADER LINE.**
- In order to prevent mistakes, it is recommended that you create internal tables without header lines. However, in internal tables with header lines you can often use a shortened syntax for certain operations.

# Overview: ABAP Statement Attributes

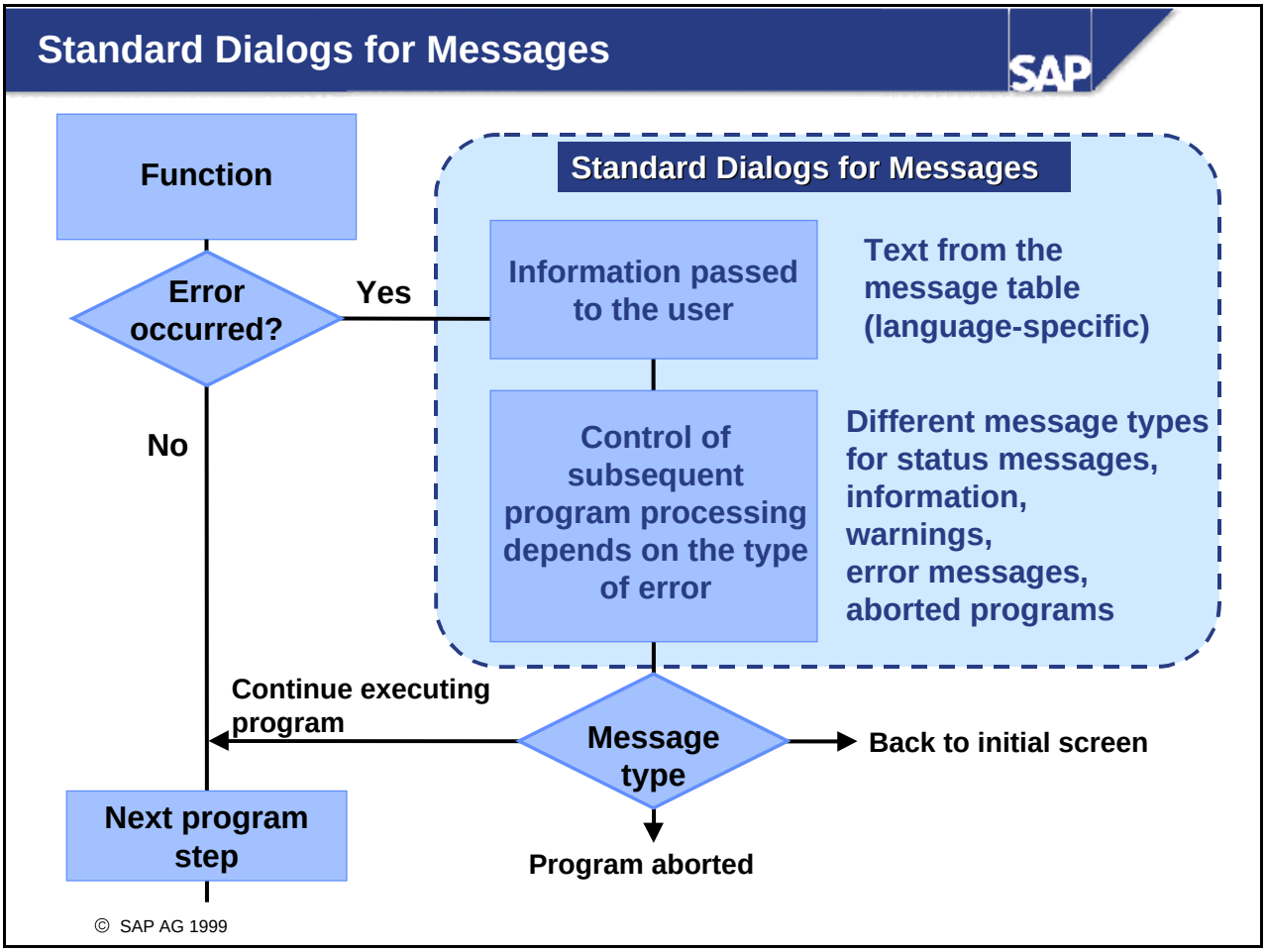
- Types
- Data objects
- Return codes and how to handle them

# ABAP Statement Return Codes






- Many ABAP statements support a return code. Depending on the statement, the system catches various exceptions. If such an exception occurs, a value is stored in field **sy-subrc** and the function for the statement is terminated. The keyword documentation for the particular statement describes the exceptions that are supported and their values. When you start a program, a structure named **sy** is automatically provided as data object. This structure contains various fields that are filled by the system.. You can access these fields from the program. **subrc** is one of the fields in this structure. There is thus no need to create a data object for the return code.
- In this example, the system should read a line from internal table **itab** with key access. There is no line with the required key at runtime. The Basis function for the **READ** statement is therefore terminated and the value 4 is placed in field **sy-subrc**. Field **sy-subrc** is queried in the program immediately after the **READ** statement.




- In many cases, the user of a program needs to be told that an error has occurred. Standardized message dialogs are available for this purpose. Texts are stored in one location for these standard dialogs. Several types of dialog are available. They can then be triggered by an ABAP statement, while the program is running. The following slides explain this concept in more detail.

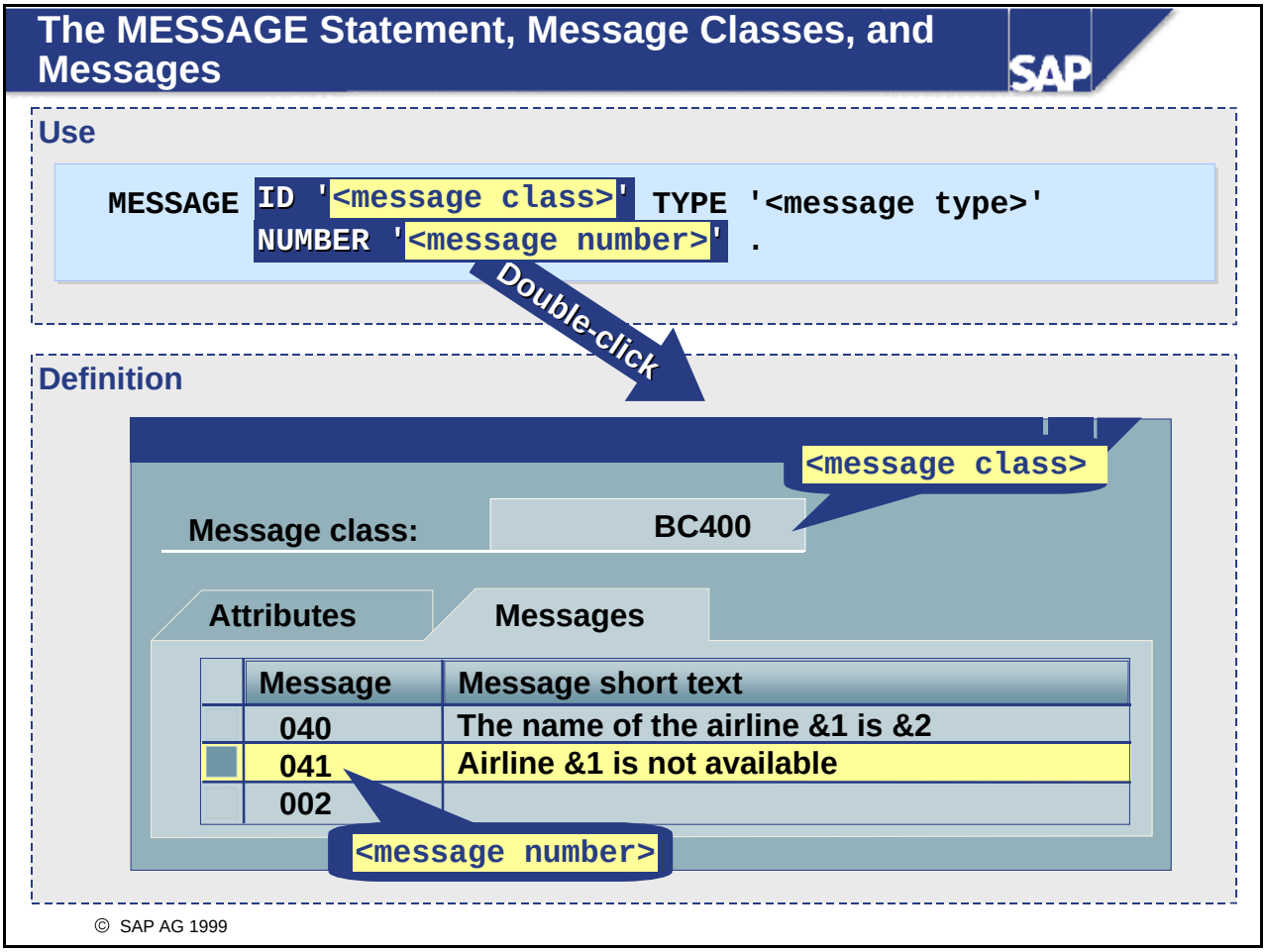
## Syntax Example: MESSAGE Statements

```
CONSTANTS c_ua TYPE s_carr_id VALUE 'UA'.

...
READ TABLE itab INTO wa_itab
                WITH TABLE KEY carrid = c_ua.

IF sy-subrc ne 0.
* Message
  MESSAGE ID 'BC400' TYPE 'I'
          NUMBER '041' WITH wa_itab-carrid.
ELSE.
  MESSAGE ID 'BC400' TYPE 'I'
          NUMBER '040' WITH wa_itab-carrid wa_itab-carrname.

ENDIF.
```

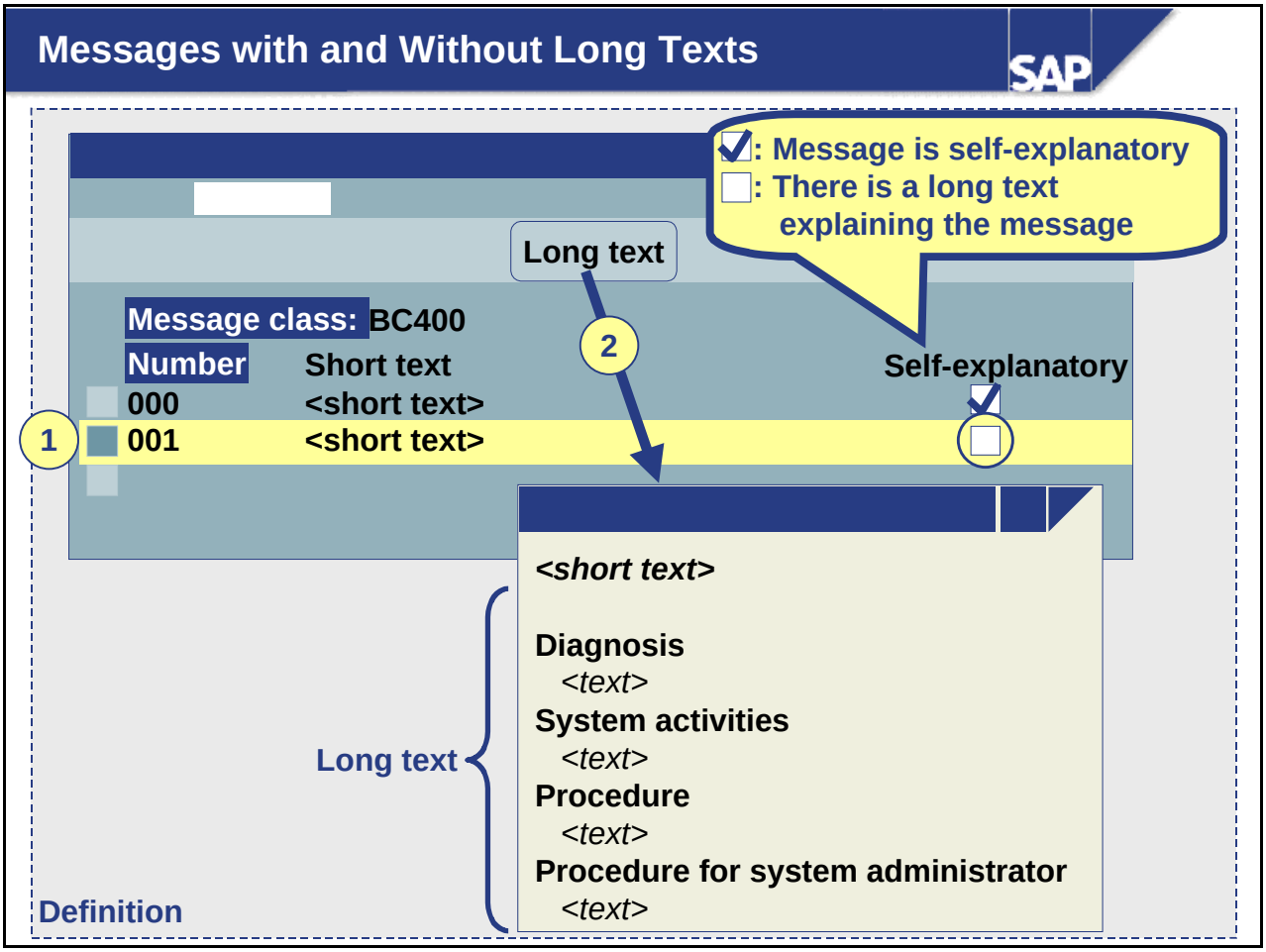
**Message class BC400**
**Message:**
**040: The name of the airline &1 is &2**
**041: Airline &1 is not available**



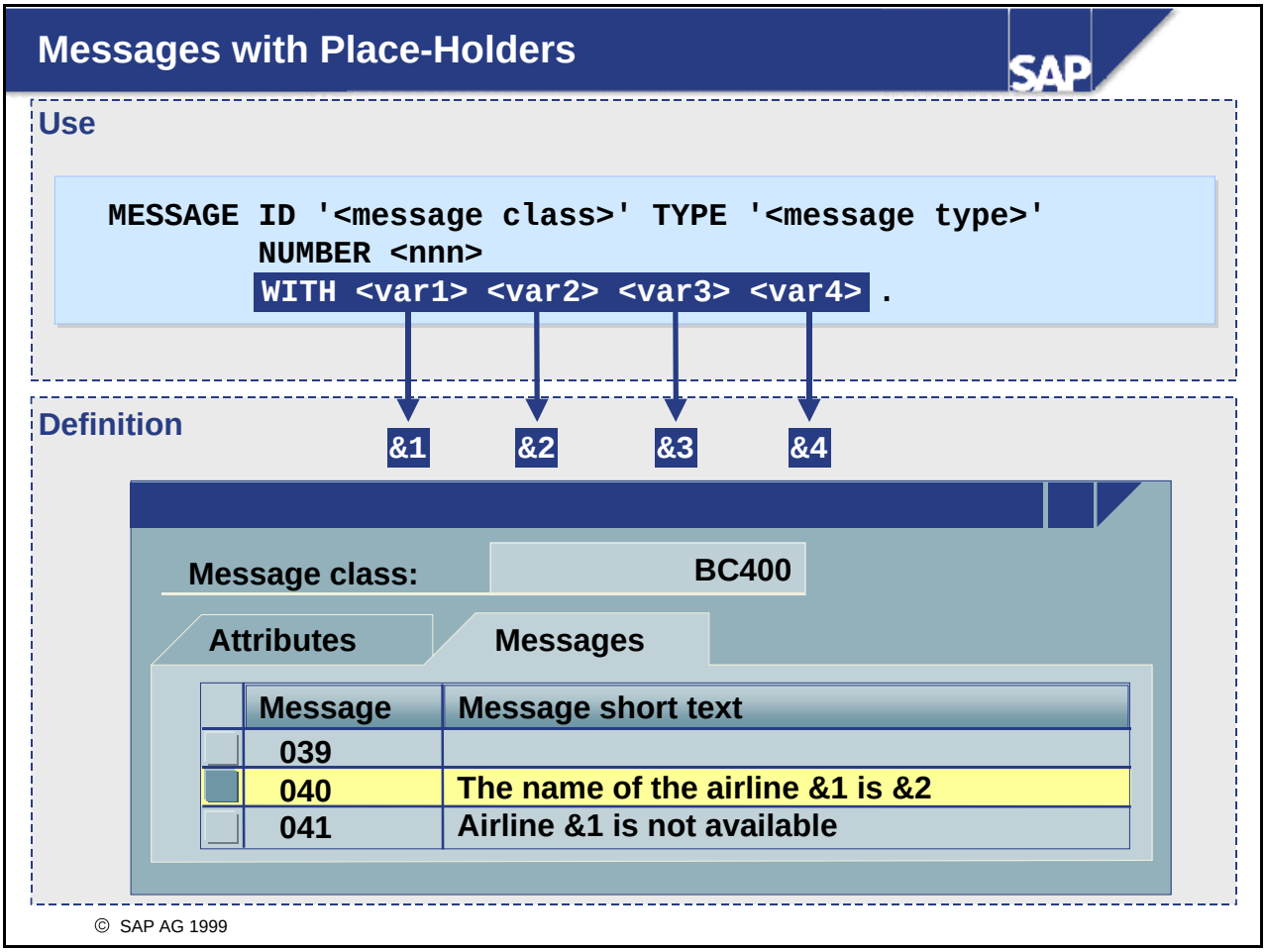
- The ABAP statement **MESSAGE** triggers a standard message dialog. The following slides deal with the additions to the **MESSAGE** statement in more detail.
- Message classes are Repository objects that contain the texts for message dialogs. These texts can be translated and are displayed in the logon language. Each message text is assigned to a message class and given a three-digit identifier.




- To trigger a message dialog in a program, enter the **MESSAGE** statement with the following additions:
  - **ID '<message class>'** - the message class
  - **NUMBER '<number>'** the message number.
- To display the message text for a **MESSAGE** statement in a program's source code, double-click the message number to navigate the associated message class texts.
- For information on the other syntax variants available for the **MESSAGE** statement, see the keyword documentation.




- If a message short text contains all the information the user needs, the message is described as **self-explanatory**. **Example:** "The program has been saved."
  Self-explanatory messages are flagged as such in the message class.
- If you want to provide more detailed information for the user, you can do so by storing a long text with the message. In the *Message Maintenance* screen, the *Self-Explanatory* flag is unchecked, if there is a long text stored for a given message. To display the long text, select the message line and choose the *Long Text* pushbutton. The system then displays the maintenance environment. You can display the formatted text by choosing *Screen Output.* You usually create the long text from a template, which contains the headings: *Diagnosis, System Activities, Procedure,* and *Procedure for the System Administrator.* The system does not display the heading to the user if there is no text stored under it.




- You can include up to four place-holders in a message. (**&1**, **&2**, **&3** and **&4**). You can then assign current parameters to them in the **MESSAGE** statement using the **WITH** addition. You can use literals, text symbols, or variables. You must include a space between each one. The current parameters are assigned to the place-holders **&1, &2, &3,** and **&4** in order.
- In the long text, the place-holders are given the names **&V1&**, **&V2&**, **&V3&,** and **&V4&** and replaced at runtime in order, in a similar way. To insert a place-holder in the long text:
  - Place your cursor in the text where you want to insert the place-holder.
  - Choose *Edit->Command-> Insert command.* The system displays a dialog box. In *Symbols*, enter **&V1&** (or **&V1&, &V1&**, or **&V1&** as appropriate).
  - Choose *Enter* to confirm the settings in the dialog box.

## The Dialog Behavior of Messages: Message Types

```
MESSAGE ID '<message class>' TYPE '<message type>'
        NUMBER <message number>
        WITH <var1> <var2> <var3> <var4> .
```

| Type | Meaning | Behavior | Message appears in |
|---|---|---|---|
| S | Status message | Program continues without interruption | Status line in next screen |
| I | Information | Program interrupted when dialog box is displayed | Modal dialog box |
| W | Warning | Depends on context | Status line (and modal dialog box) |
| I | Error | Depends on context | Status line (and modal dialog box) |
| A | Termination | Program aborted | Modal dialog box |
| X | Short dump | Runtime error `MESSAGE_TYPE_X` | Integrated in short dump |



- You administer the dialog behavior of the message using the **TYPE** of the **MESSAGE** statement.
- There are six different types of message: **A**, **X**, **E**, **I**, **S** and **W**. The runtime behavior of each dialog message is context-specific. The letters have the following meaning:

  **A** Termination Processing is terminated; the user must restart the transaction

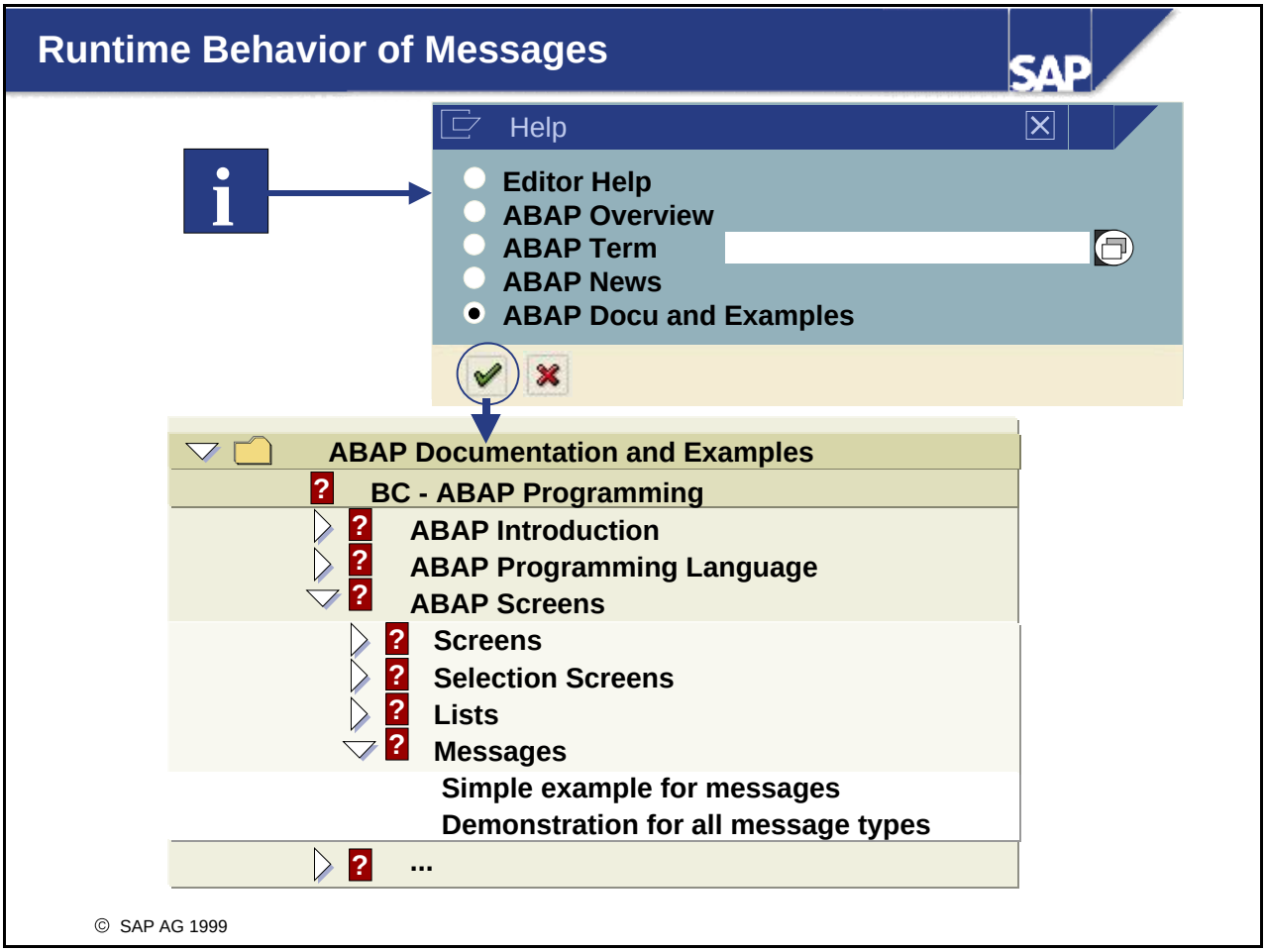
  **X** Exit Like a termination message, but with short dump MESSAGE_TYPE_X

  **E** Error Runtime behavior depends on context

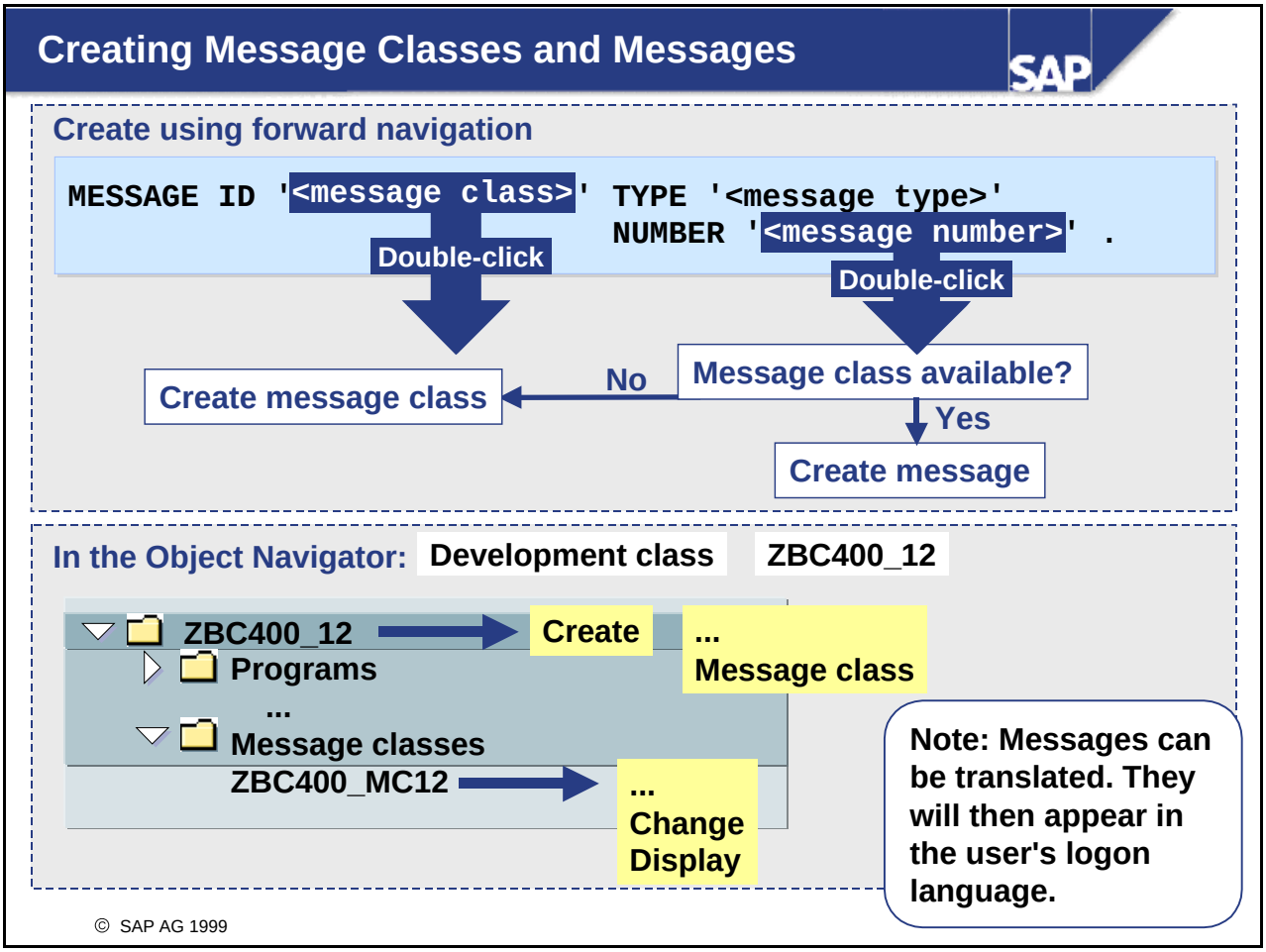
  **W** Warning Runtime behavior depends on context

  **I** Information Processing is interrupted, the message is displayed in a dialog box and the program continues when the message has been confirmed with ENTER.

  **S** Set The message appears in the status bar on the next screen.

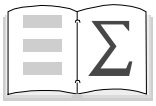



- You can find a program for testing the runtime behavior in the sample programs of the documentation. You can find the sample programs with transaction code ABAPDOCU or in the Editor with the '*Information*' icon and radio button *ABAP Docu and Examples.*




- To create your own message class, give it a name in the customer namespace. (That is, a name beginning with Y or Z, or with the namespace prefix).
- To create a message, assign a three-digit number and a message class to it.
- You can create both the message class and the message itself using forward navigation from the **MESSAGE** statement.
- In the Object Navigator, you can create and edit a message class in any of the following ways:
  - From the context menu belonging to the root node in the object list of the appropriate development class.
  - From the context menu belonging to the *Message class* node in the object list of the appropriate development class.
  - From the *Other object...* icon. A dialog box containing several tabs appears. In the *Other*, enter a message class, or a message with its message class, and then display, create, or change it by choosing the appropriate icon.

# ABAP Statements and Data Declarations: Unit Summary



**You are now able to:**

- **Describe the various different data types and their uses**
- **Define elementary data objects, structures, and internal tables**
- **Use Debugging mode to observe how the values of individual data objects change during processing**
- **Program several important operations involving data objects**
- **Find information about the various return codes used by ABAP statements and evaluate these in programs**

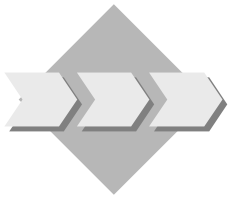
Exercises

**Unit: Data Objects and Statements**

**Topic: Structures and Assigning Values**

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

- Use the Debugger to understand how a program works and how data is transferred between objects in the program
- Use the **MOVE-CORRESPONDING** statement to assign values between structures.



Debug the program that you wrote in the exercises to the last unit (or the corresponding model solution).

**Program:** **ZBC400_##_GETTING_STARTED**

**Model solution:** **SAPBC400WBS_GETTING_STARTED**

1-1 Start the program **ZBC400_##_GETTING_STARTED**. On the selection screen, enter the airline code '**LH**'. In the command field, enter '**/h**', then choose *Execute*. You are now in Debugging mode.

1-2 Check that all of the data objects are initial. Put all of the data objects declared in the program into the field view. Find out the structure and data types of the individual data objects (if you double-click a structured data object, the Debugger displays the components).

1-3 Step through the program using the Single step (F5) function. Which fields of the structure **WA_SCARR** does the **SELECT** statement fill? What is the value of system field **SY-SUBRC** after the statement?

1-4 Now observe how fields are copied from **WA_SCARR** to **SBC400_CARRIER**. Which field values are copied?

1-5 The statement **CALL SCREEN 100** processes screen 100. On the screen, enter appropriate values for the user name, date, and time, and continue with the program. Now observe how fields are copied from **SBC400_CARRIER** to **WA_SCARR**.

1-6 Finally, observe how the **WRITE** statement constructs the list. Note especially that an extra button appears in the application toolbar after the first **WRITE** statement, which allows you to display the current contents of the list buffer at any time.

1-7 Restart the program in Debugging mode. Set a breakpoint at the **MOVE-CORRESPONDING** statement. Before the screen is processed, assign a name to the structure component **SBC400_CARRIER-UNAME** from the Debugger. (Next to the input/output field is an icon that you can use to change a field value at runtime.)

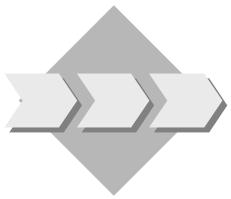
1-8 Repeat step 1-1. Now set a breakpoint (*Breakpoint → Breakpoint at…*) for the **CALL SCREEN** statement, and a watchpoint for whenever the value of a component of structure **WA_SCARR** changes. Each time the program stops, use the 'Continue' (F8) function in the Debugger to carry on processing.

**Unit: Data Objects and Statements**

**Topic: Internal tables**

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

- Declare internal tables with reference to a table type defined in the ABAP Dictionary
- Use the **LOOP…ENDLOOP** statements to process data buffered in an internal table



Create a program to display all of the flights stored in the system on a list.
To do this, read the contents of database table **SPFLI** into an internal table.
Then use a **LOOP … ENDLOOP** structure to display the data records in a list.

**Program:** **ZBC400_##_ITAB_LOOP**

**Model solution:** **SAPBC400TSS_ITAB_LOOP**

2-1 Create a program with the name **ZBC400_##_ITAB_LOOP** and **no TOP include**. Assign your program to **development class ZBC400_##** and the change request for your project "BC400…" (## is your group number).

2-2 In your program, create an internal table with the line structure of table **SPFLI**. Do this by referring to a suitable table type defined in the ABAP Dictionary (use the where-used list function to display all table types that use the definition of database table **SPFLI**). You also need to create a structure with reference to the definition of database table **SPFLI**.

2-3 To read the data from the database table **SPFLI** and place it in the internal table, use the following ABAP statement in your program:
**SELECT * FROM SPFLI INTO TABLE <itab>.**
<itab> is the name of the internal table.

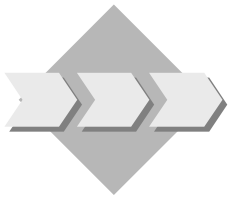
2-4 Display the data from the internal table in a list. Use the **LOOP … ENDLOOP** statements.

**Unit: Data Objects and Statements**

**Topic: Message Classes and Messages**

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

- Create a message class
- Create messages and use them in the program



Extend your internal tables program. The program should read a line, using single-record access. If it finds a record, it should display an information message.

**Program:** **ZBC400_##_ITAB_LOOP**

**Model solution:** **SAPBC400TSS_MESSAGE**

3-1 Create a message class with the name **ZBC400_##**. Assign your program to **development class ZBC400_##** and the change request for your project “BC400…” (where ## is your group number).

3-2 Create a message (message number 001) with the following text: *The internal table has an record in line &1*. Flag the message as *self-explanatory.*

3-3 In the keyword documentation on the ABAP statement **READ**, look for the return values possible when reading a line of an internal table. Which value does the runtime system place in the field **sy-subrc** if it finds a record? Which value does the runtime system place in the field **sy-subrc** if it does not find a record?

3-4 Extend your program, **ZBC400_##_ITAB_LOOP**. After the **SELECT** statement, read the fifth record in the internal table. Use the **INDEX 5** addition to the **READ** statement. Find out more about this addition from the keyword documentation on the **READ** statement (used to read a line from an internal table). Get the value stored in the field **sy-subrc**. If a record has been read, the program should display your message (number **001**), with the message type **I**. Fill the placeholder with the line number using the **WITH** addition to the **MESSAGE** statement. Display the data from the line you have read, in a list. Separate the display of this single line from the display of the rest of the table using an underscore (ABAP statement: **ULINE**). Test your program.

3-5 Test the program in debugging mode and observe how many lines the internal table has after it has been filled. Change the program so that, if you access a single record in the internal table, you will not find any records. Test your program again. The message should no longer appear.

3-6 **Optional:** Create a message that appears if single-record access fails to find any records. Create a long text for this message. Extend your program, so that this message appears as information, when the runtime system fails to find a record. Test the program and the display of the long text.

**Unit: Data Objects and Statements**

**Topic: Structures and Assigning Values**

1-3 Which components of the structure are filled in the statement?

**MANDT, CARRID, CARRNAME, CURRCODE, URL**
What value does **SY-SUBRC** have after the **SELECT** statement?
**SY-SUBRC has the value 0, because the airline LH (Lufthansa) is maintained in the SCARR table.**

1-4 Which field values are copied?

**MANDT, CARRID, CARRNAME, CURRCODE**

**Unit: Data Objects and Statements**

**Topic: Internal tables**

**Model Solution SAPBC400TSS_ITAB_LOOP**

```
*&---------------------------------------------------------------------*
*& Report        SAPBC400TSS_ITAB_LOOP                        *
*&                                                             *
*&---------------------------------------------------------------------*

REPORT  sapbc400tss_itab_loop        .

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


START-OF-SELECTION.
* read all fields of all datasets from the database table SPFLI into
* the internal table it_spfli.
  SELECT * FROM spfli INTO TABLE it_spfli.
* at least one dataset selected
  IF sy-subrc = 0.
* move each single dataset from internal table to structure WA_SPFLI
* in order to write data on list
    LOOP AT it_spfli INTO wa_spfli.
      WRITE: / wa_spfli-carrid,
               wa_spfli-connid,
               wa_spfli-cityfrom,
               wa_spfli-cityto,
               wa_spfli-deptime,
               wa_spfli-arrtime.
    ENDLOOP.
```

```
ENDIF.
```

**Unit: Data Objects and Statements**

**Topic: Message Classes and Messages**

3-1, 3-2 Follow the instructions in the slides.

3-3 Does the runtime system place the value **0** in the field **sy-subrc** if it finds a record? The runtime system place the value **4** in the field **sy-subrc** if it finds a record.

3-4 See the model solution.

**Model Solution SAPBC400TSS_MESSAGE**

Note: The messages of the message class BC400 are used in the model solution.

```
REPORT  sapbc400tss_message        .

DATA: it_spfli TYPE sbc400_t_spfli.
DATA: wa_spfli TYPE spfli.
CONSTANTS c_index TYPE i VALUE '5'.

START-OF-SELECTION.
  SELECT * FROM spfli INTO TABLE it_spfli.
* at least one dataset selected
  READ TABLE it_spfli INTO wa_spfli INDEX c_index.
  IF sy-subrc = 0.
    MESSAGE ID 'BC400' TYPE 'I' NUMBER '001' WITH c_index.
    WRITE: / wa_spfli-carrid,
             wa_spfli-connid,
             wa_spfli-cityfrom,
             wa_spfli-cityto,
             wa_spfli-deptime,
             wa_spfli-arrtime.
    ULINE.
  ELSE.
    MESSAGE ID 'BC400' TYPE 'I' NUMBER '002' WITH c_index.
  ENDIF.
  IF sy-subrc = 0.
```

```
* move each single dataset from internal table to structure WA_SPFLI
* in order to write data on list
  LOOP AT it_spfli INTO wa_spfli.
    WRITE: / wa_spfli-carrid,
             wa_spfli-connid,
             wa_spfli-cityfrom,
             wa_spfli-cityto,
             wa_spfli-deptime,
             wa_spfli-arrtime.
  ENDLOOP.
```

# Reading Database Tables

## Contents:

- Information on Database Tables in R/3
- Reading Database Tables
- Authorization Checks
- Outlook: Reading Multiple Database Tables

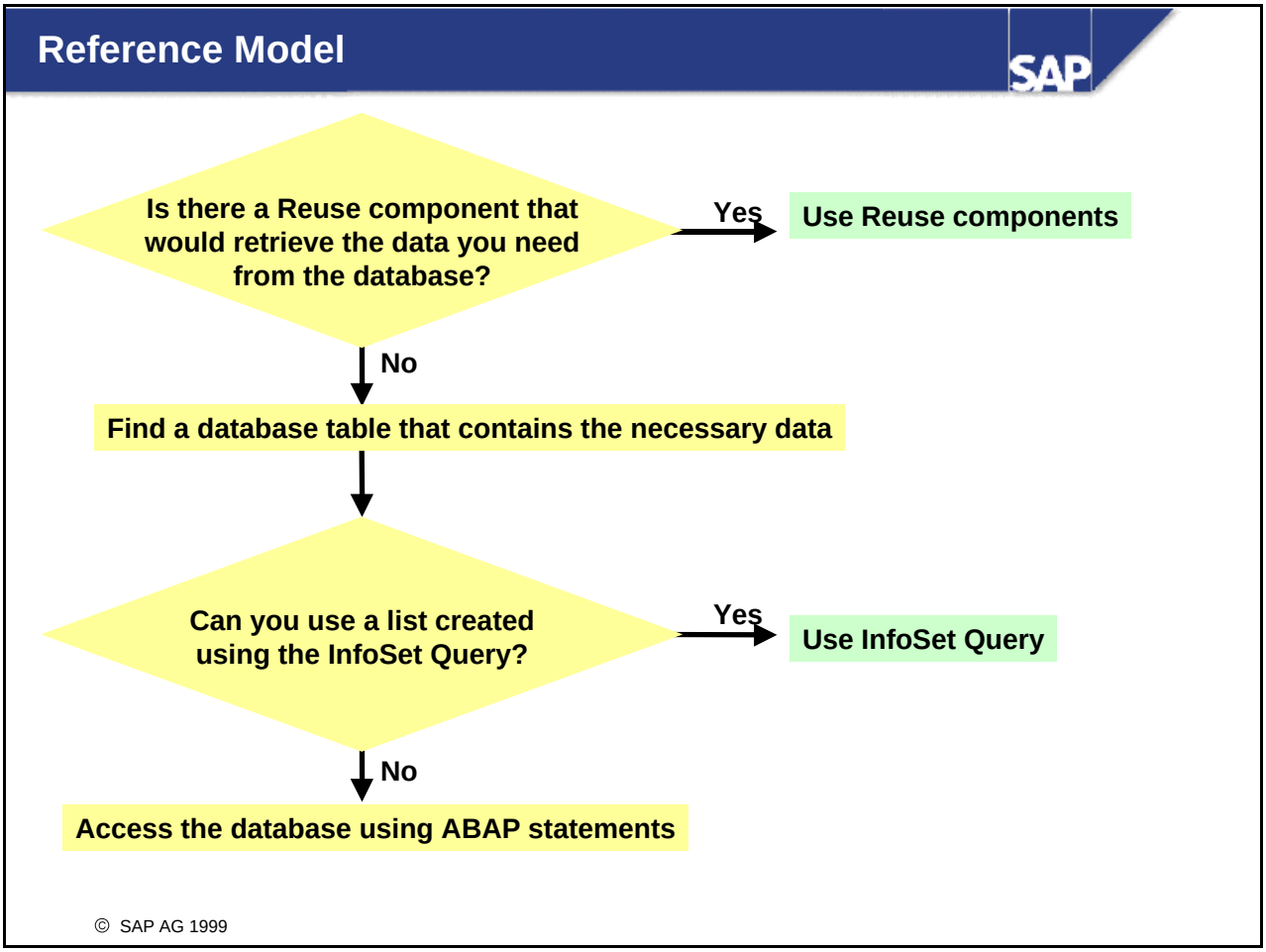
# Reading Database Tables: Unit Objectives

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

- **Extract information about database tables from the ABAP Dictionary**
- **List various ways of finding database tables**
- **Program read access to specific columns and lines within a particular database table**
- **Implement authorization checks**
- **List the different kinds of read access possibilities for database tables**

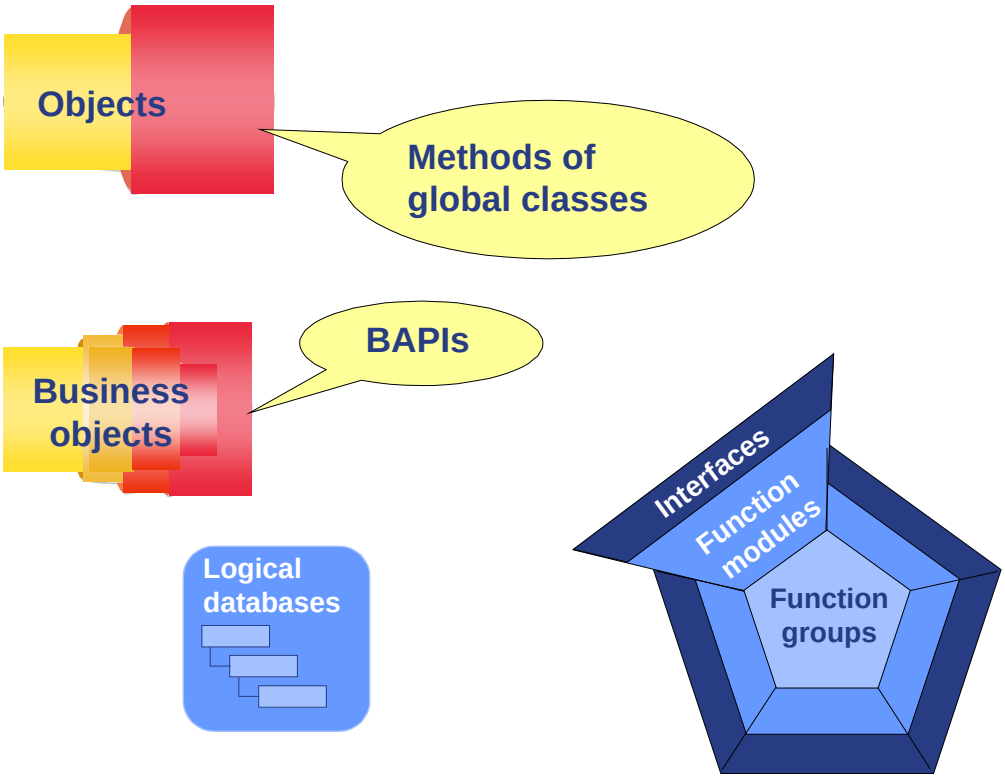
# Overview: Using Reuse Components

- Using Reuse components
- Information on database tables in R/3
- Reading database tables
- Authorization checks
- Outlook: Reading multiple database tables




- There are several benefits to using a Reuse component to retrieve the data you need from the database:
  - You do not need to find out the details of the database tables in which the data is stored.
  - You do not need to change your program if changes are made to the data model.
  - You can find information on the main techniques in the *Reuse Components* unit.
- If there are no Reuse components available, you must start by finding out which database tables contain the data.
- You can use InfoSet Query to create a list. The ABAP code is generated automatically in the background. For detailed information on the InfoSet Query, refer to the SAP Library under ***Basis->ABAP Workbench->SAP Query->InfoSet Query***.
- If the InfoSet Query does not have enough functions, you can implement a database access using ABAP statements.

## Overview: Available Reuse Techniques

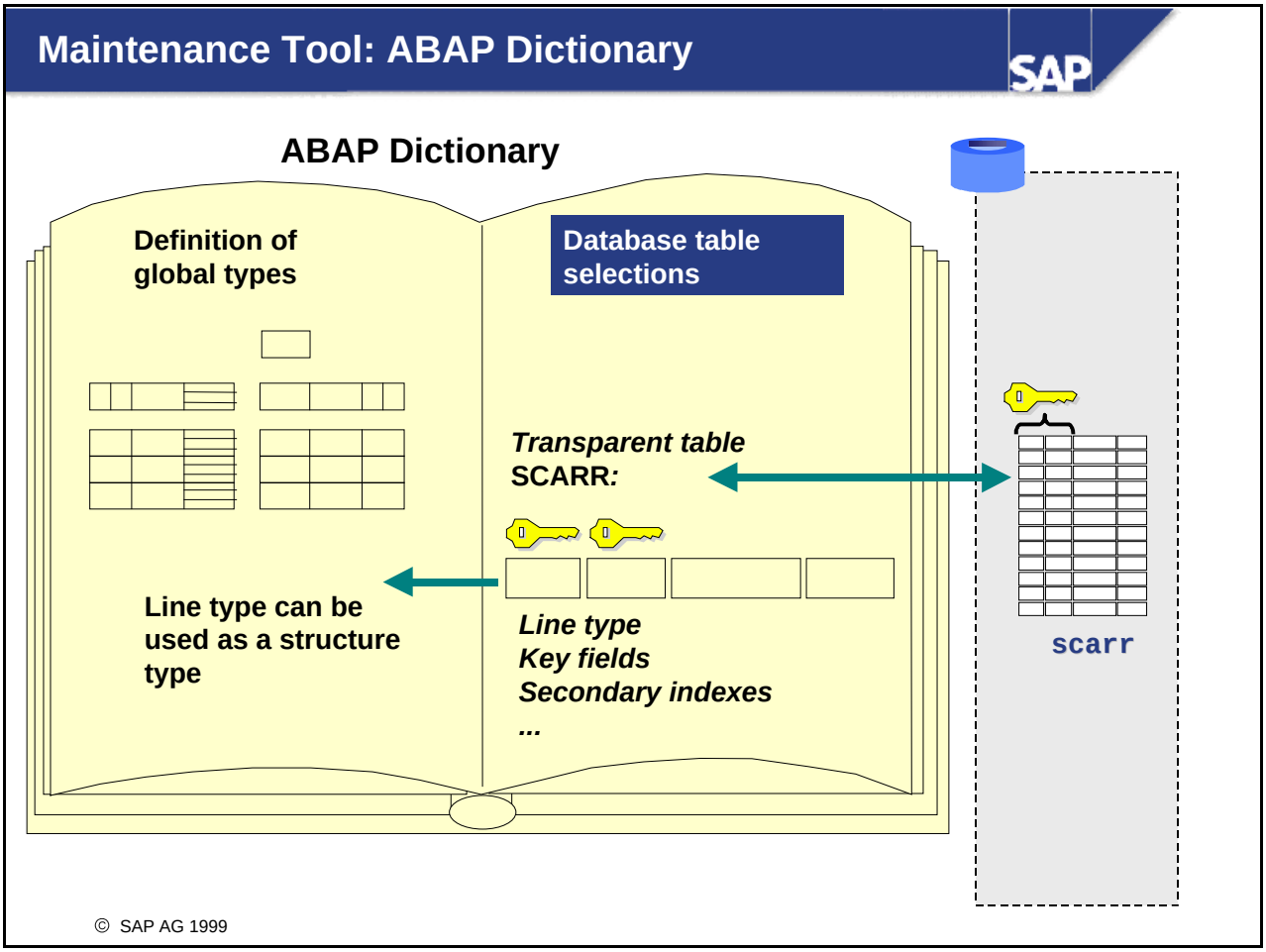





- If reusable components that encapsulate complex data retrieval are available, then you must use them. There are four techniques available for doing this:
  - Methods of global classes
  - Methods of business objects
  - Function modules
  - Logical databases - data retrieval programs delivered by SAP that return data in a hierarchically logical sequence.
- You can find information on the various techniques in the *Reuse Components* unit.
- For more information on logical databases, see the SAP Library under *Basis->ABAP Programming and Runtime Environment->ABAP Database Access®Logical Databases*

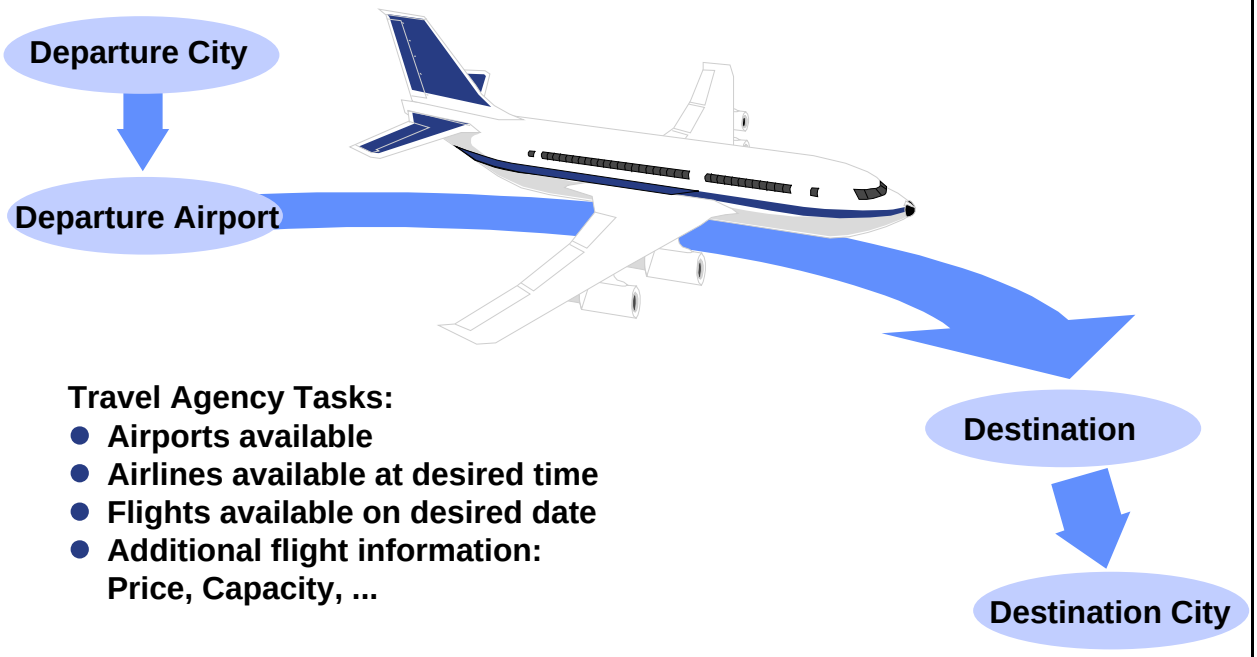
# Information on Database Tables in R/3

- Information on Database Tables in R/3
- Reading database tables
- Authorization checks
- Outlook: Reading multiple database tables

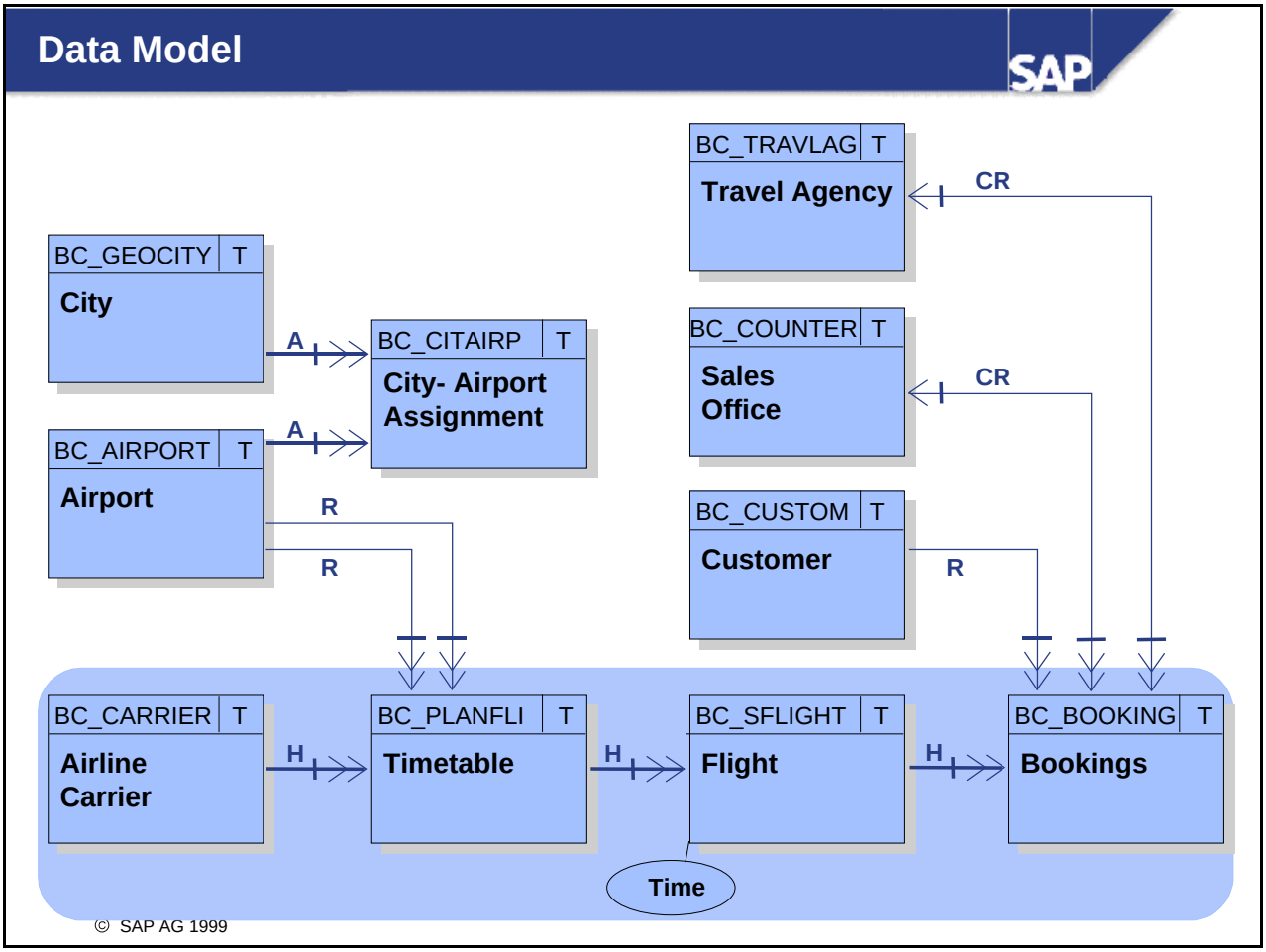



- **Database tables** in the R/3 System are administered in the **ABAP Dictionary**. There you can find current information about a database table's technical attributes. Database tables that have been created in the database using the same line type and name are called **transparent tables** in the ABAP Dictionary.
- There are a two ways to navigate to transparent tables in the ABAP Dictionary:
  - Choose *Tools-> ABAP Workbench-> Development-> Dictionary* to call the ABAP Dictionary directly and insert the name of the transparent table in the appropriate input field.
  - Navigate directly to the ABAP Dictionary from the ABAP Editor while editing the program: by double-clicking on the name of the transparent table in the **FROM** clause of the **SELECT** statement.

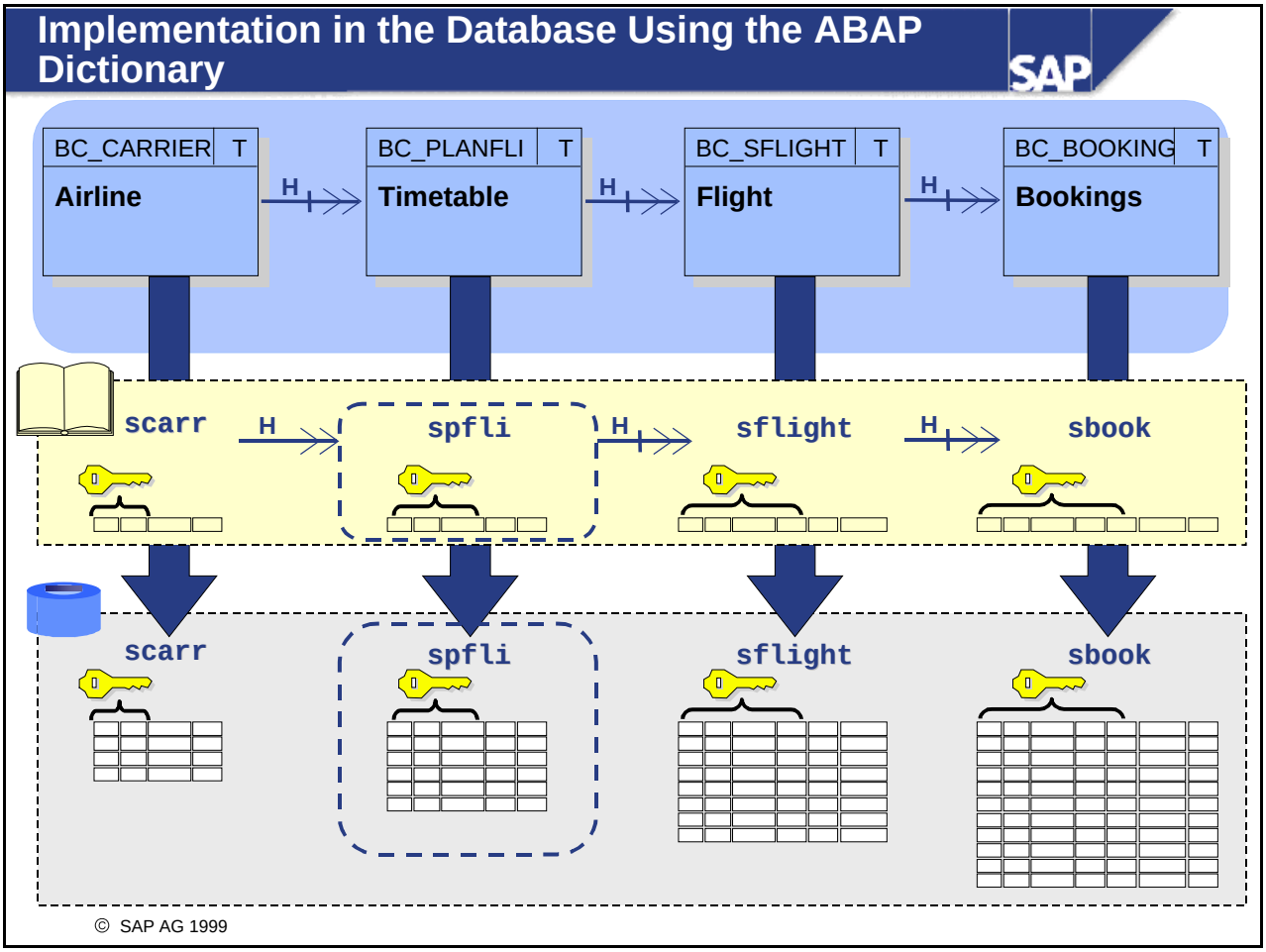
## Flight Data Model for ABAP Training Courses




- ABAP training courses all use the same flight data model. At this time, a simple cross-section of the flight data model will be presented; you can get more detailed information about it at any time.
- As a traveler trying to get from one place to another, you expect your travel agency to be able to provide you with the following information:
  - What connection offers me the best and most direct flight?
  - At what times are flights offered on the date that I want to travel?
  - How can I optimize the conditions under which I am travelling, that is, what is the cheapest flight, the fastest connection, the connection that gets me there nearest the time I want to arrive, ...?
- A **travel agency's** point of view is a bit different: all necessary technical flight data in a data model is organized and stored in tables in a central database according to the database's structure. The amount of data stored is far greater than that which a travel agency wants or needs. They are primarily interested in which one of their customers has booked which flight, when the booking was made, how much the customer paid, and so on. These different views and their corresponding demands on the data model demonstrate the necessity of using programs to organize data in a manner that fulfills all of the different demands that users make.




- A separate entity exists for all pieces of information that are logically dependent on each other: In the ABAP flight data model there are individual entities for:
  - All cities
  - All airports
  - All airline carriers
  - All flight routes
  - All flights.
- These entities all relate to each other in certain ways:
  - Flight routes all depart from and arrive at an airport.
  - A flight route is characterized by airline, departure airport, destination airport, and departure time.
  - Flights for a particular flight route can be offered on many different days in a given year, but the flight route must exist before a flight can be created.
  - Cities must have all airports in their vicinity assigned to them.
- This data model manages all the data you need without unnecessary redundancy and makes it possible for a travel agency to access data for a customer's point of view.




- ABAP training course examples and exercises, as well as ABAP documentation, all use SAP's flight data model. All flight data model Repository objects are located in the development class **SAPBC_DATAMODEL**.
- The following is a list of the flight data model tables most frequently used in ABAP training courses:
  - **SCARR**: Table of airlines
  - **SPFLI**: Flight connections table
  - **SFLIGHT**: Flights table
  - **SBOOK**: Bookings table

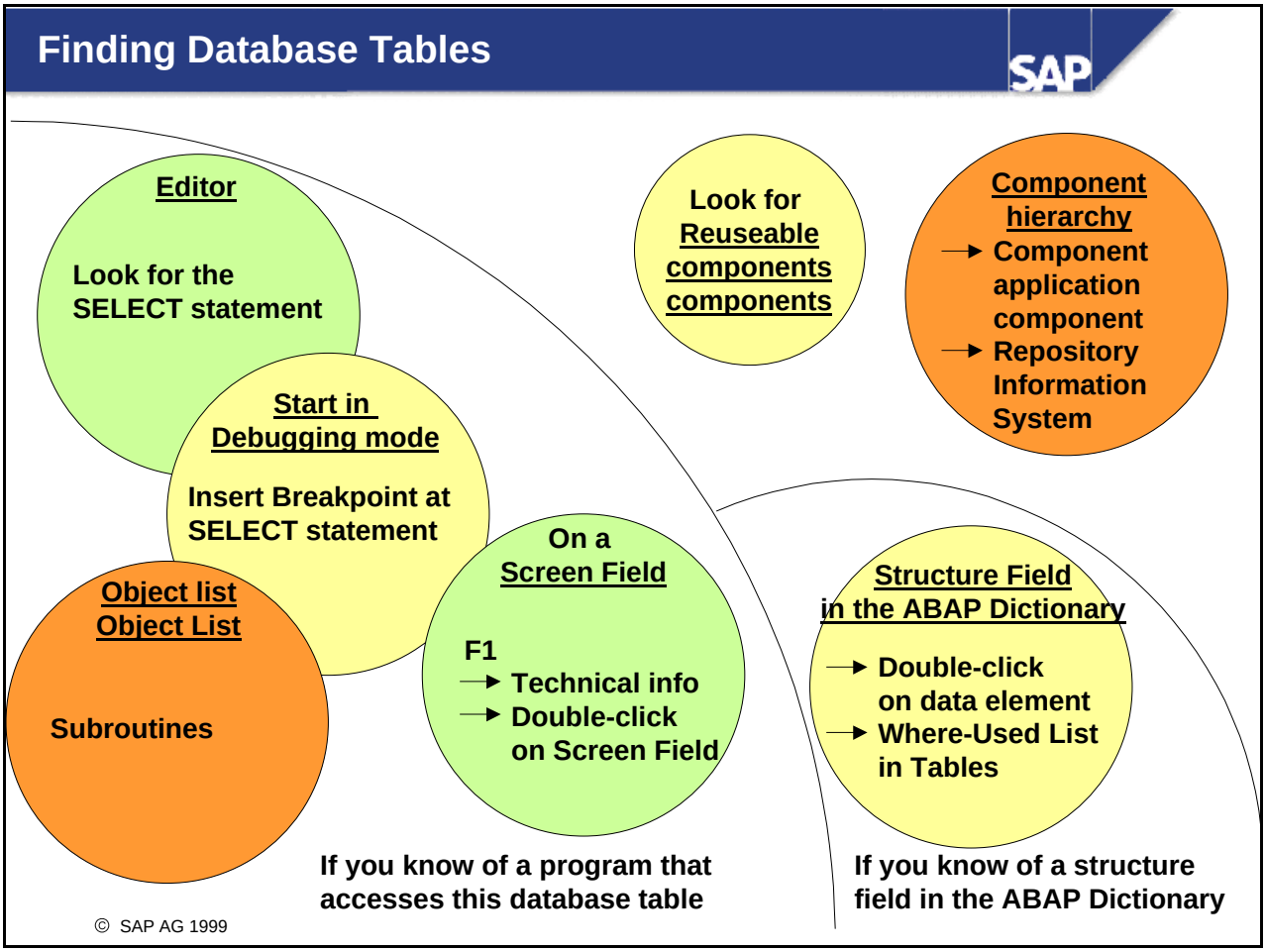
## Finding Fields, Key Fields, and Secondary Indexes in the ABAP Dictionary

**Transparent table** `spfli` **Indexes**

| Fields | Key | ... | Field type | ... | Short description | ... |
|---|---|---|---|---|---|---|
| MANDT | ☑ | | S_MANDT | | ... | |
| CARRID | ☑ | | S_CARR_ID | | ... | |
| CONNID | ☑ | | S_CONN_ID | | ... | |
| COUNTRYFR | ☐ | | LAND1 | | ... | |
| CITYFROM | ☐ | | S_FROM_CIT | | ... | |
| AIRPFROM | ☐ | | S_FROMAIRP | | ... | |
| COUNTRYTO | ☐ | | LAND1 | | ... | |
| CITYTO | ☐ | | S_TO_CITY | | ... | |
| AIRPTO | ☐ | | S_TOAIRP | | ... | |
| FLTIME | ☐ | | S_FLTIME | | ... | |
| DEPTIME | ☐ | | S_DEP_TIME | | ... | |
| ARRTIME | ☐ | | S_ARR_TIME | | ... | |
| DISTANCE | ☐ | | S_DISTANCE | | ... | |
| DISTID | ☐ | | S_DISTID | | ... | |
| FLTYPE | ☐ | | S_FLTYPE | | ... | |



- As soon as you navigate to the definition of a database table in the ABAP Dictionary, information about all of the table's technical attributes is available.
- To optimize the performance of database accesses, bear the following in mind:
  - **Key fields:** If the lines requested from the database are being retrieved according to key fields, the Databank Optimizer can perform access using a primary index. Checkboxes are provides for all key fields.
  - **Secondary indexes:** You may also use secondary indexes to select specific lines. These are displayed in a dialog box whenever you choose the *Indexes* pushbutton. You can choose an index from the dialog box by simply double-clicking on it. The system then displays a screen with additional information about that index.

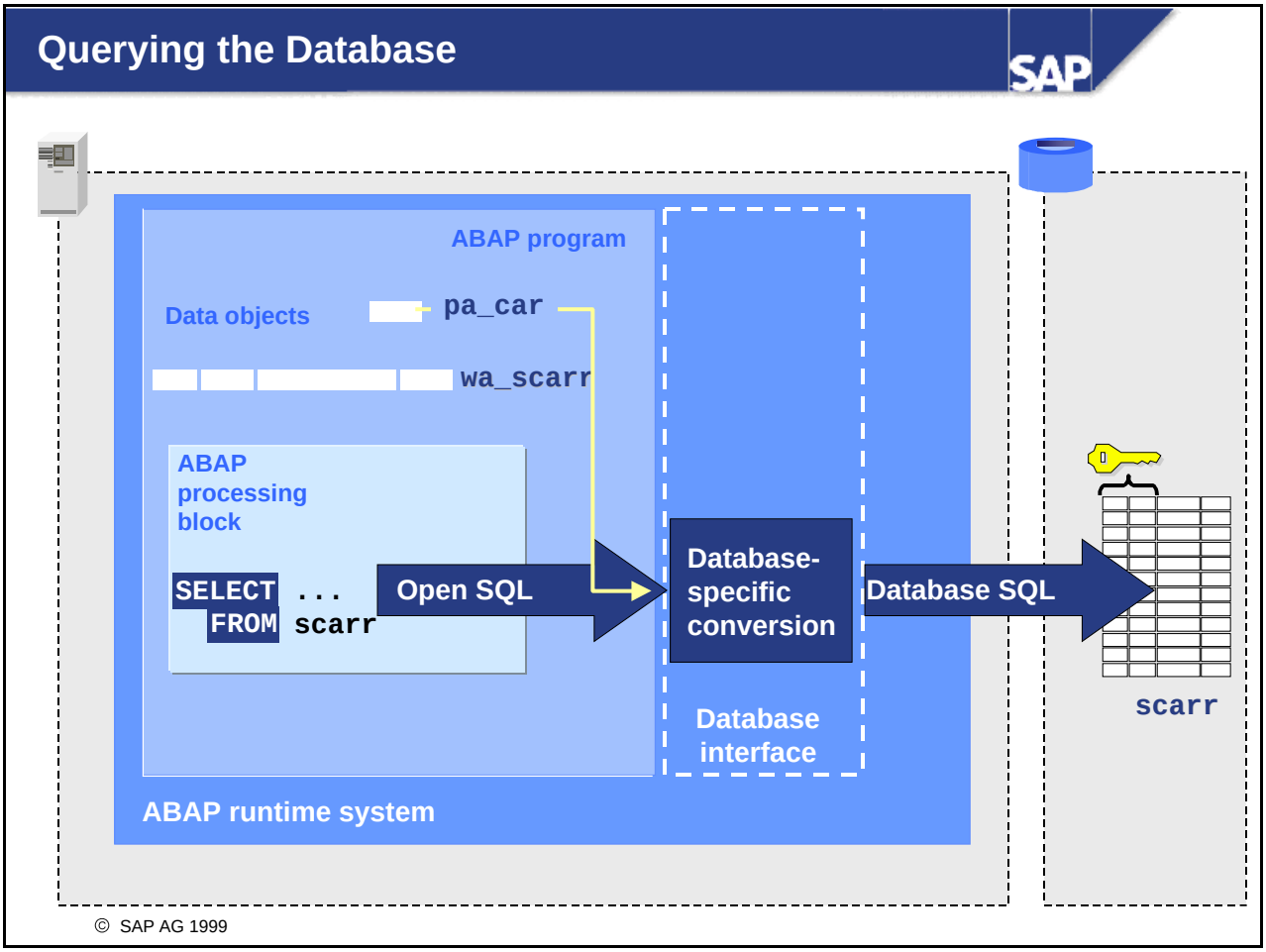



- You can search for database tables in several different ways:
  - **Application hierarchy** and the **Repository Information System:** You may choose application components from the application hierarchy and go directly to the information system. There you can search for database tables according to their short texts (among other criteria).
- If you have the name of a program that accesses the database table:
  - Input field on a screen; If you know of a program that contains a screen with input fields connected to the table you are looking for, choose *F1-> Technical info*. Then navigate to the ABAP Dictionary by double-clicking the technical name of the screen field. This is often a field in a structure. Double-click on the **data element** and then use the **where-used list** function to search for transparent tables according to the field type.
  - **Debugger:** If you know the name of a program that accesses the database table that you are looking for, you can start this program in debugging mode and set a **breakpoint** at the **SELECT statement**.
  - **Editor:** Look for the SELECT statement
  - **Object list in the Object Navigator:** Pick out the subroutines that encapsulate the database accesses.
- If you know of a structure field in the ABAP Dictionary.

- Double-click the **data element** and then use the **where-used list** function to search for transparent tables according to the field type.

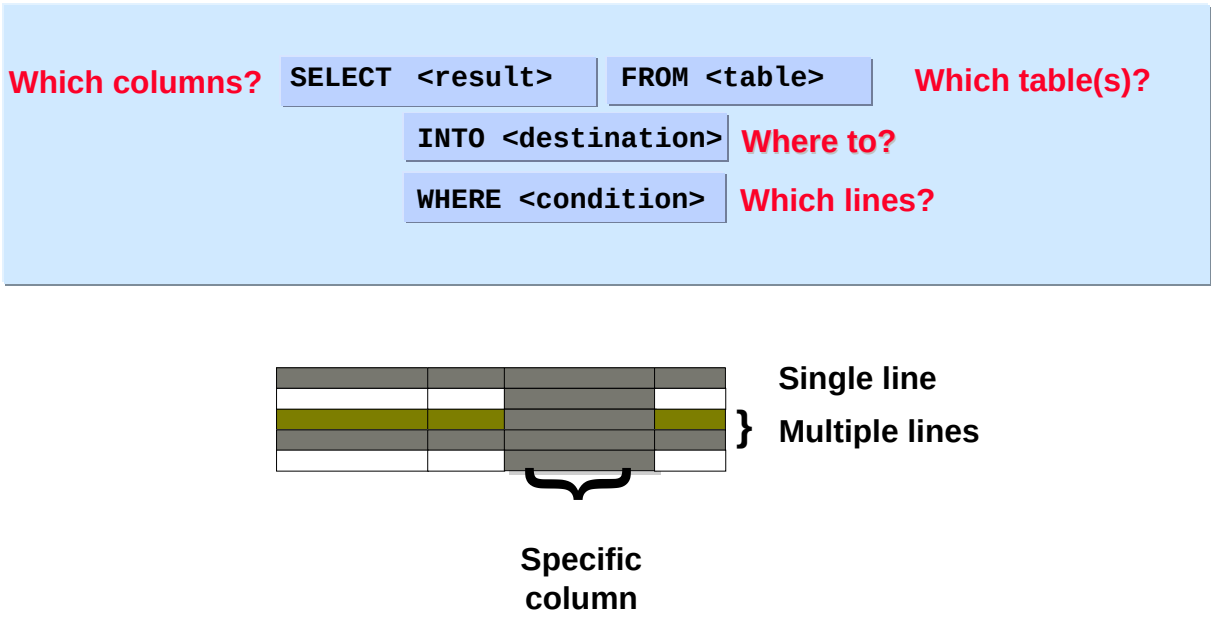
# Reading Database Tables

- Information on Database Tables in R/3
- Reading Database Tables
- Authorization checks
- Outlook: Reading multiple database tables




- **Open SQL** statements are a subset of **Standard SQL** that is fully integrated in the ABAP language. They allow you to access the database in a uniform way from your programs, regardless of the database system being used. Open SQL statements are converted into database-specific Standard SQL statements by the **database interface**.

## SELECT Overview



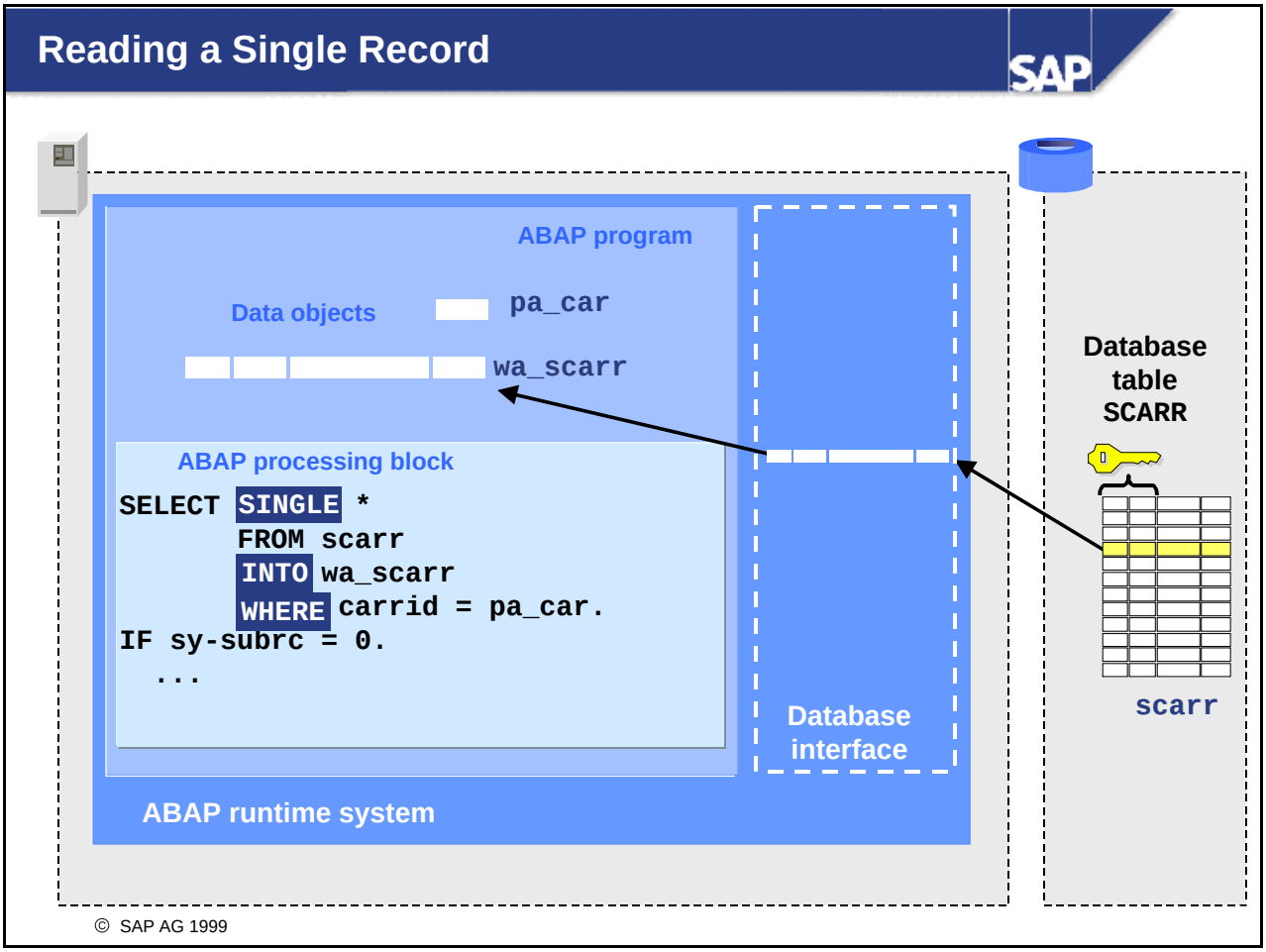



- You use the Open SQL statement **SELECT** to read data from the database.
- Underlying the **SELECT** statement is a complex logic that allows you to access many different types of database table.
- The statement contains a series of clauses, each of which has a different task:
  - The **SELECT clause** specifies

    Whether the result of the selection is to be a single line or several lines.

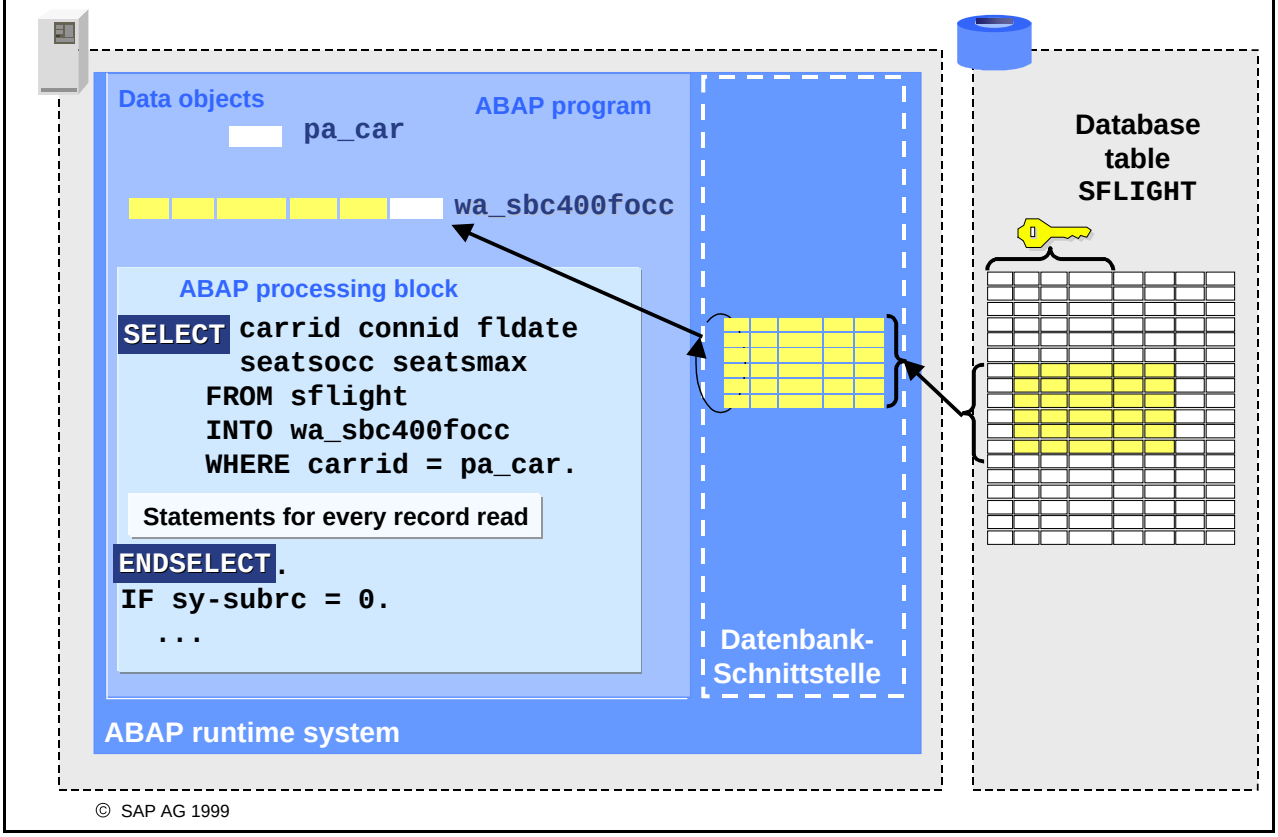
    Which fields should be included in the result.

    Whether the result may contain two or more identical lines.
  - The **INTO clause** specifies the internal data object in the program into which you want to place the selected data.
  - The **FROM clause** specifies the source of the data (database table or view).
  - The **WHERE** clause specifies conditions that selection results must fulfill. Thus, it actually determines what lines are included in the results table.
  - For information about other clauses, refer to the keyword documentation in the ABAP Editor for the **SELECT** statement.




- The **SELECT SINGLE *** statement allows you to read **a single record** from a database table. To ensure that the entry you read is unique, all the key fields must be filled by the **WHERE** clause. The ***** informs the database interface it should read all columns in that line of the database table. If you only want to read certain columns, you can insert a field list instead.
- The name of a structure to which you want the database interface to copy a data record is inserted after the **INTO** clause. The structure should have a structure identical to the columns of the database table being read and be left-justified.
- If you use the **CORRESPONDING FIELDS OF** addition in the **INTO** clause, you can fill the target work area component by component. The system only fills those components that have identical names to columns in the database table. If you do not use this addition, the system fills the work area from the left-hand end without any regard for its structure.
- If the system finds a table entry matching your conditions, **SY-SUBRC** has the value 0.
- The **SINGLE** addition tells the database that only one line needs to be read. The database can then terminate the search as soon as it has found that line. Therefore, **SELECT SINGLE** produces better performance for single-record access than a **SELECT** loop if you supply values for all key fields.

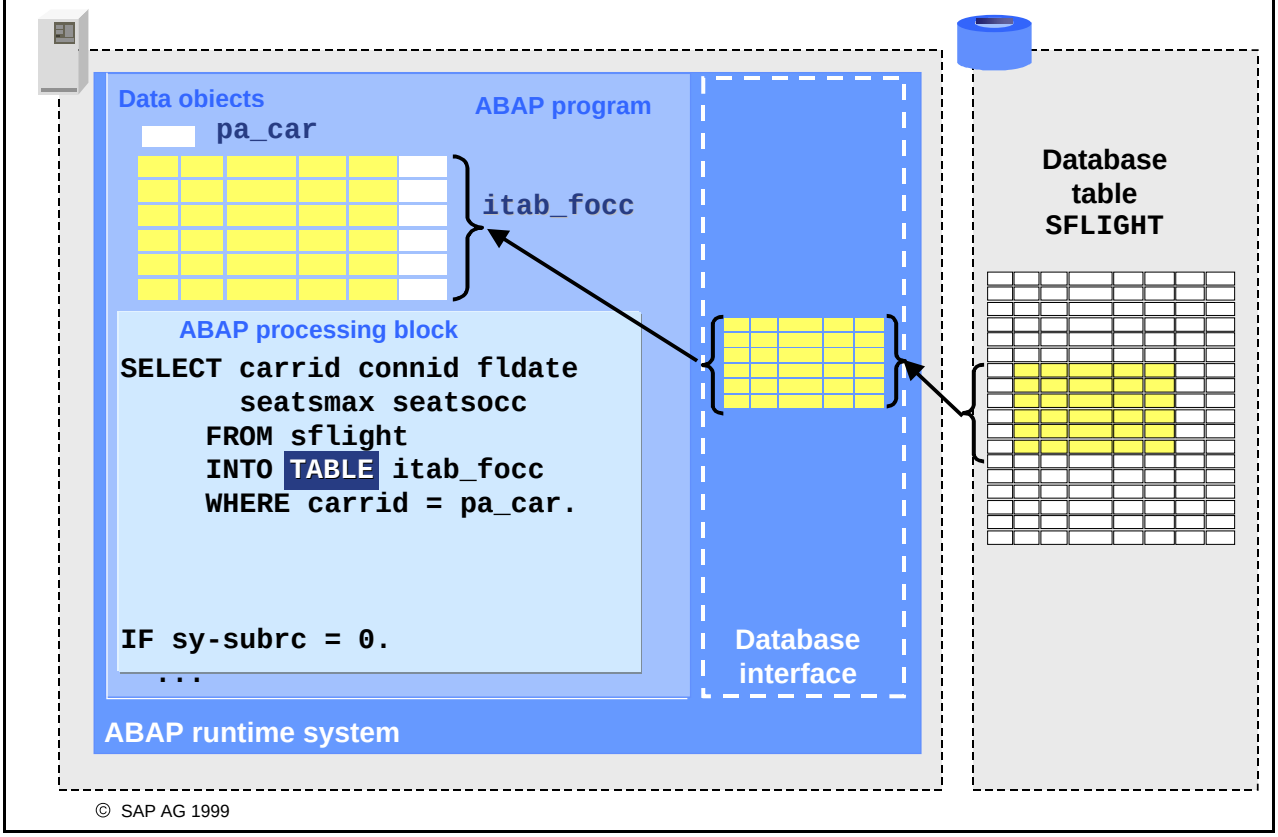
## Reading Several Records Using a SELECT Loop





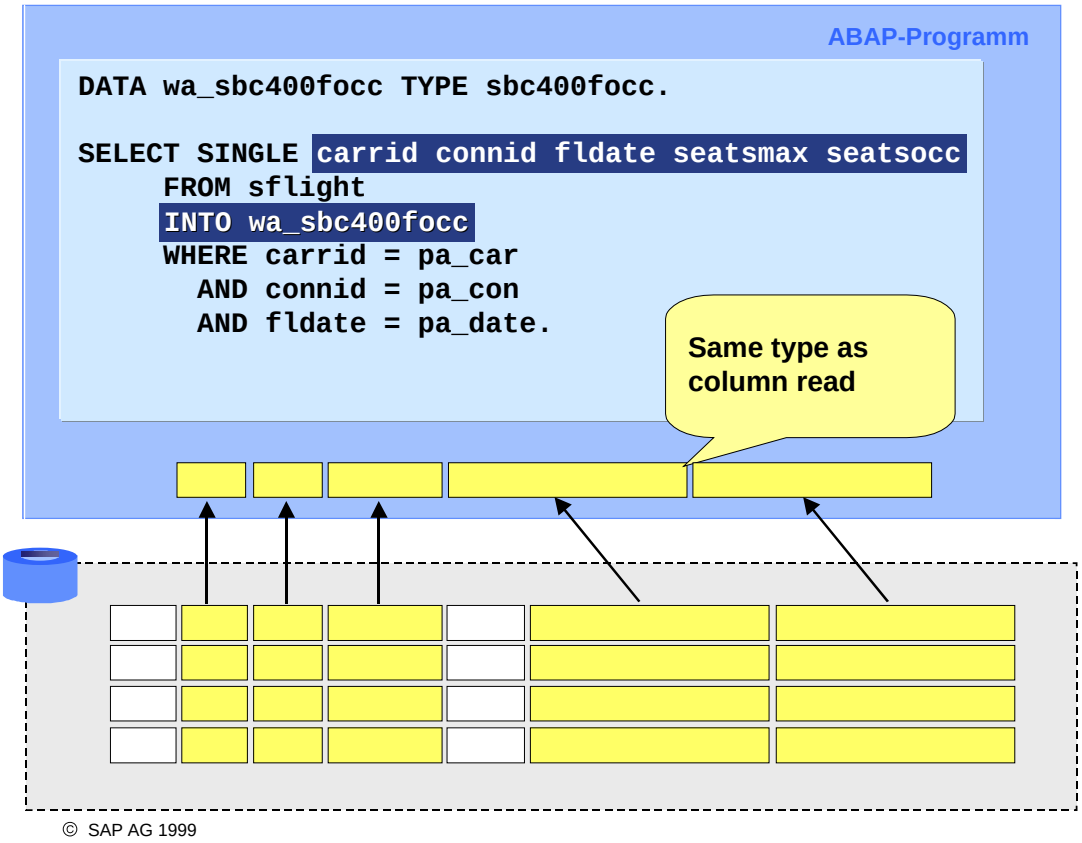

- If you do not use the **SINGLE** addition with the **SELECT** statement, the system reads multiple records from the database. The field list determines the **columns** whose data is to be read from the database.
- You should restrict the number of **lines** to be read by using a **WHERE** clause using either the database table's key fields or a secondary index. For more information on key fields and secondary indexes, see the ABAP Dictionary. For example, double-clicking the database table included in the **FROM** clause will take you directly to the Dictionary.
- The **WHERE** clause specifies only the fields that are to be read. The name of the database table you want to access is specified in the **FROM** clause. (Example of a correct statement: **SELECT** ... **FROM spfli WHERE carrid =** ... , Example of an incorrect statement: **SELECT** ...**FROM spfli WHERE spfli-carrid** = ... )
- Multiple logical conditions can be added to the **WHERE** clause using **AND** or **OR**.
- The database delivers data to the database interface in packets. The ABAP runtime system copies the data records to the target area line by line using a loop. It also provides for the sequential processing of all of the statements between **SELECT** and **ENDSELECT**.
- After the **ENDSELECT** statement, you can query the return value for the **SELECT** loop. **sy-subrc** = 0 if the system was able to select at least one entry. After the **SELECT** statement is executed in each loop pass, the system field **sy-dbcnt** contains the number of lines read.

# Reading Several Records Using an Array Fetch




- The addition **INTO TABLE <itab>** causes the ABAP runtime system to copy the contents of the database interface directly to the internal table **itab**. This is known as an **array fetch**.
- Since an array fetch is not logically a loop, you do not use an **ENDSELECT** statement.
- If you want add lines to the end of an internal table already filled (rather than over-writing it) use the **APPENDING TABLE <itab>** addition.
- **SY-SUBRC** = 0, provided the system was able to read at least one table entry.
- For further information about array fetch and internal tables, refer to the *Internal Tables* unit in this course.

## The Field List and Appropriate Target Structure: The INTO Clause

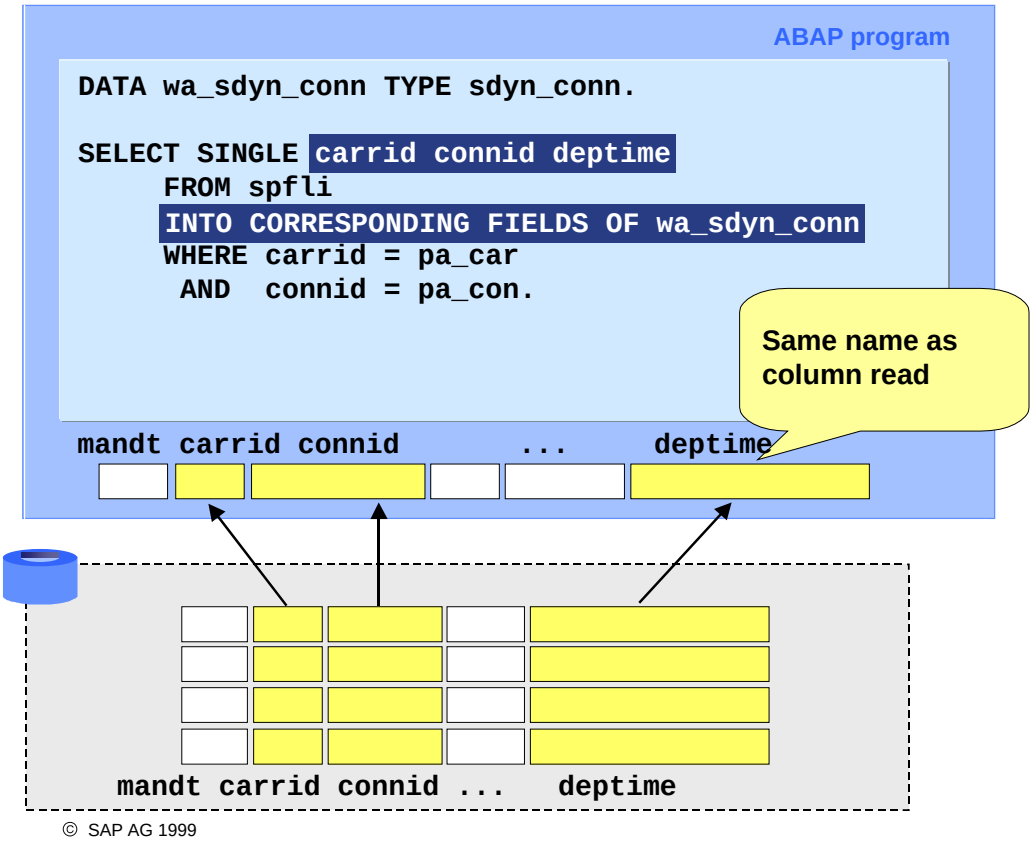





- For each column required from a database table, the program must contain a data object with a suitable type . For program maintenance reasons, you must use the corresponding Dictionary objects to assign types to the data objects. The **INTO** clause specifies the data object into which you want to place the data from the database table. There are two different ways to do this:
  - **Flat structure:** You define a structure in your program that has the fields in the same sequence as the field list in the **SELECT** clause. Then you enter the structure name in the **INTO** clause. The contents are copied by position. The system disregards the structure's field names.
  - **Single data objects:** You enter a set of data objects in the **INTO** clause. Example:

```
DATA:   gd_carrid TYPE sflight-carrid,
        gd_connid TYPE sflight-connid,
        gd_fldate  TYPE sflight-fldate,
        gd_seatsmax TYPE sflight-seatsmax,
        gd_seatsocc TYPE sflight-seatsocc.
START-OF-SELECTION.
SELECT carrid connid fldate seatsmax seatsocc
 FROM sflight
 INTO (gd_carrid, gd_connid, gd_fldate, gd_seatsmax, gd_seatsocc)
 WHERE ...
```

# Target Structures with Identically-Named Fields for All Columns Specified

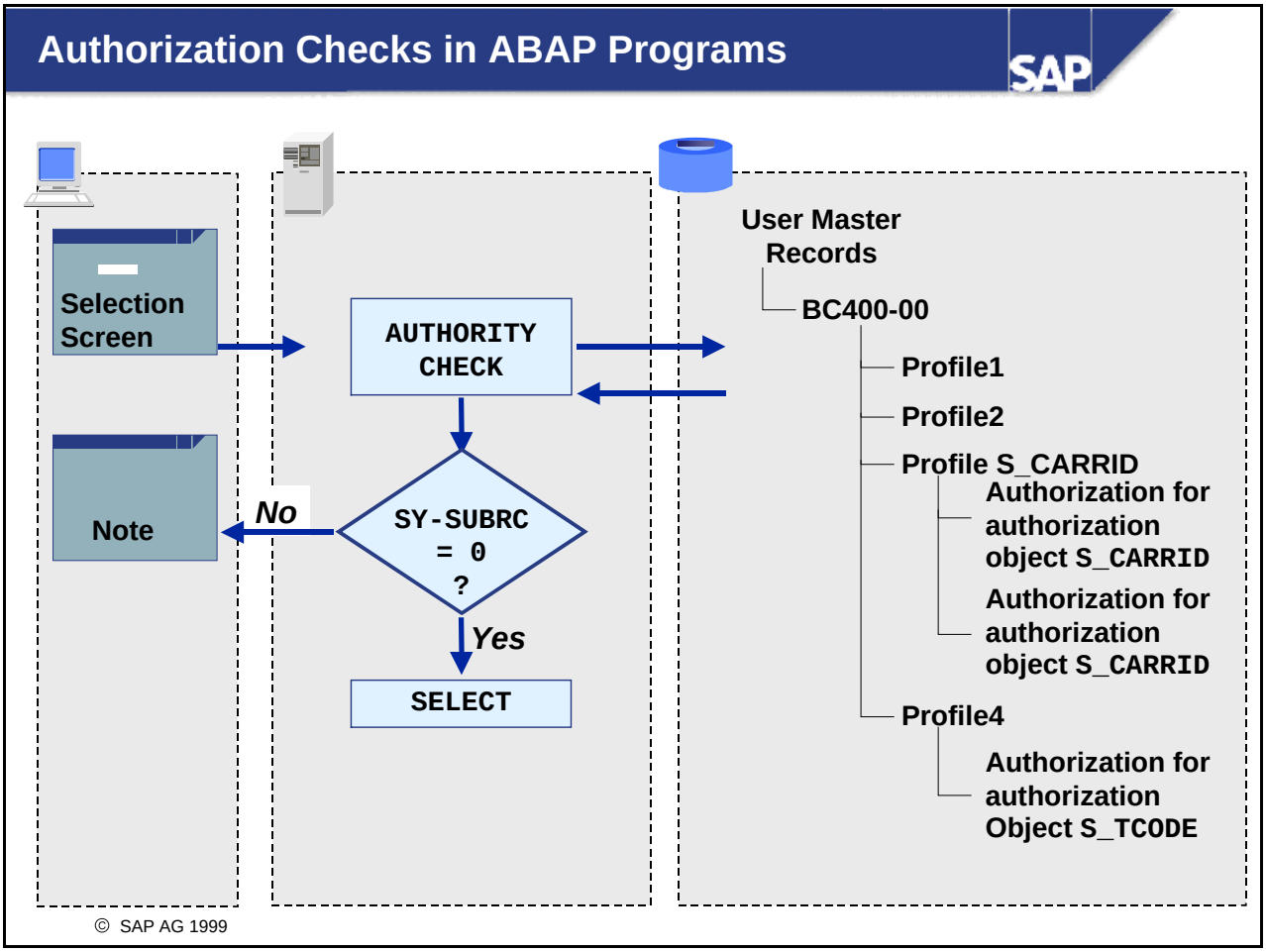





- If you use the **INTO CORRESPONDING FIELDS** clause, the data is placed in the structure fields that have the same name.
- Advantages of this construction:
  - The structure does not have to be structured in the same way as the field list
  - This construction is easy to maintain, since extending the field list does not require other changes to be made to the program, as long as there is a field in the structure that has the same name and type.
- Disadvantages of this construction:
  - **INTO CORRESPONDING FIELDS** is more runtime-intensive than **INTO**.
- If you want to place data into internal table columns of the same name using an array fetch, use **INTO CORRESPONDING FIELDS OF TABLE <itab>**.

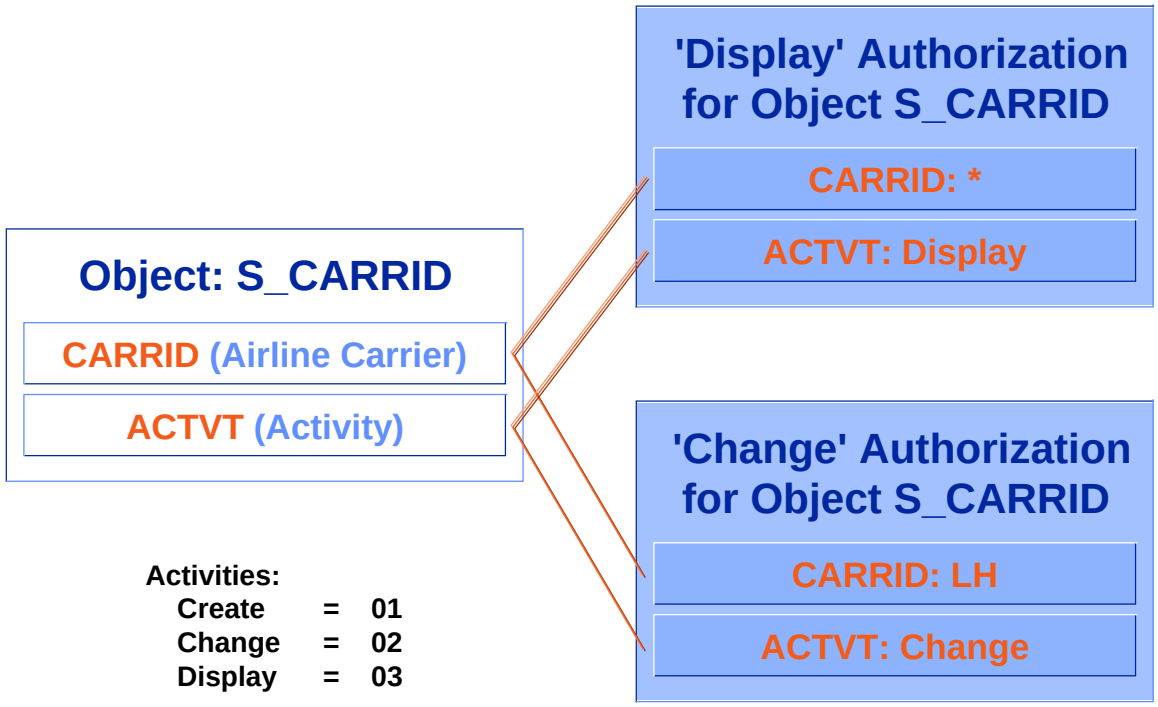
# Authorization Checks

- Information on database tables in R/3
- Reading database tables
- Authorization checks
- Outlook: Reading multiple database tables



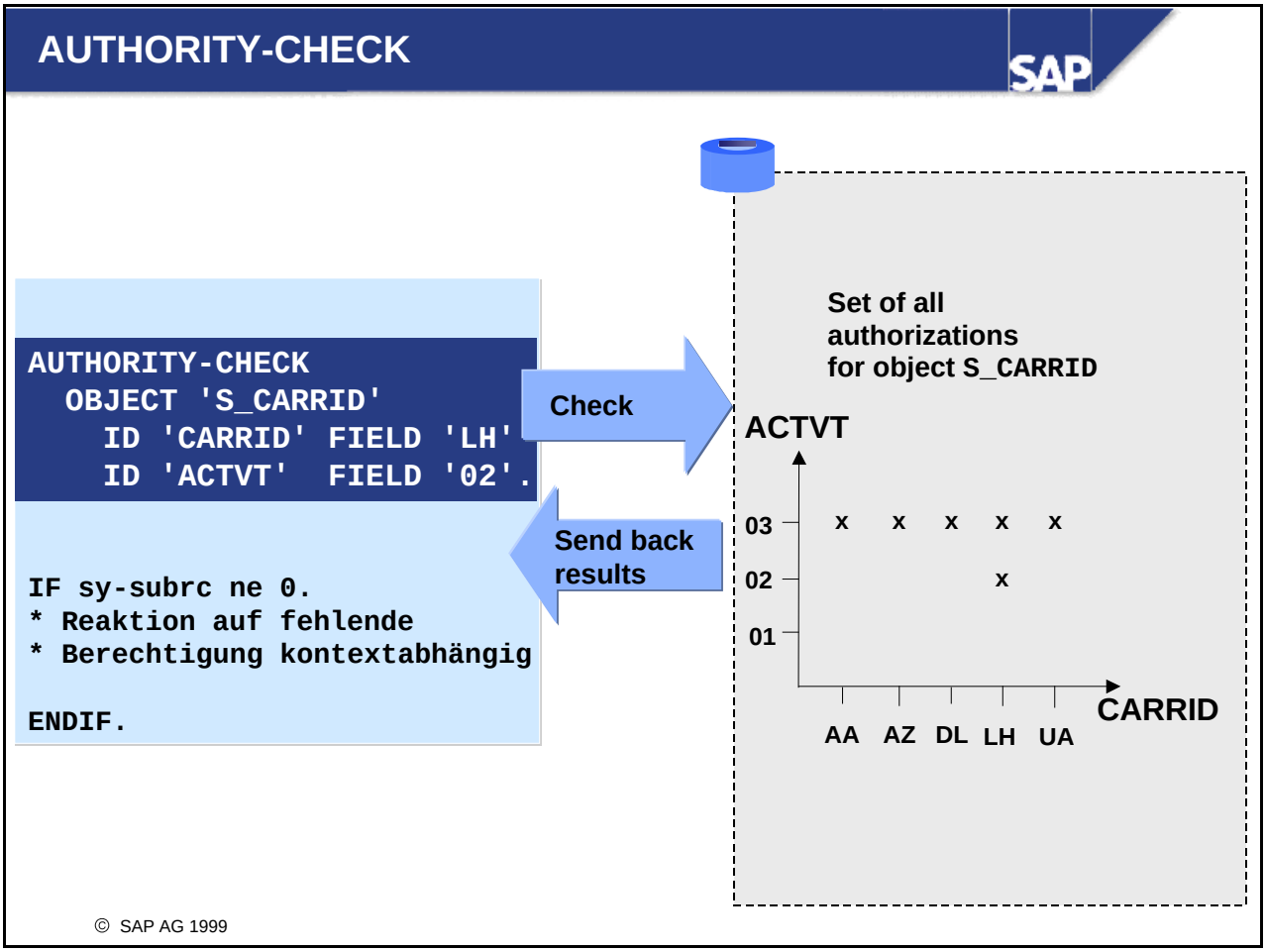

- You should carry out an authorization check before accessing the database. The **AUTHORITY-CHECK** statement first checks whether the user has the authorization containing all the required values. You then read the code value in the system field **SY-SUBRC**. If this value is 0, the user has the required authorization and the program can continue. If the value is not 0, the user does not possess the required authorization and the system outputs an appropriate message.
- Later in this course, you will learn how to make fields on the selection screen ready for input again if you perform the authorization check right after the selection screen, and how to output a message if the user does not have the required authorization.

## Authorization Objects and Authorizations

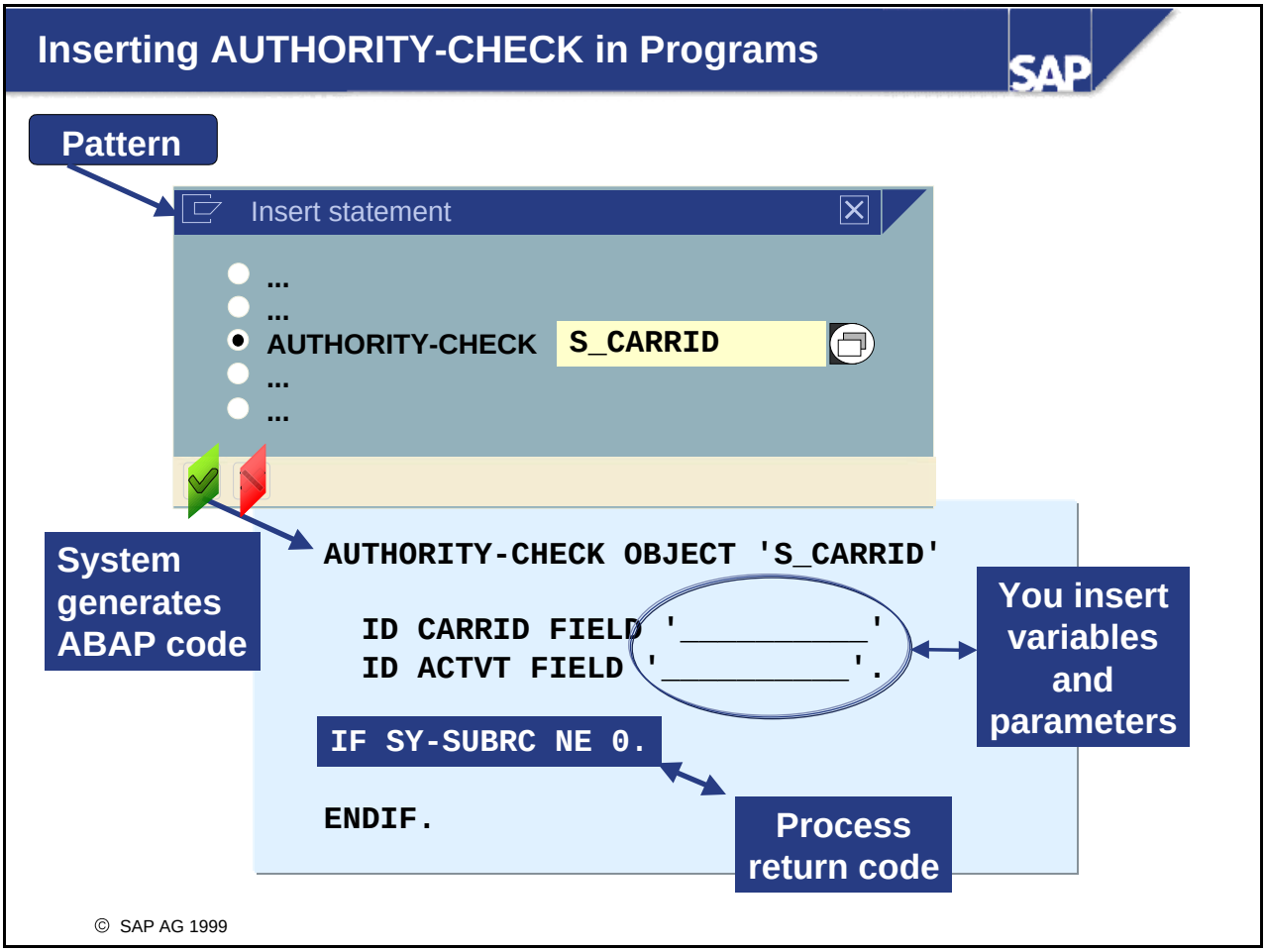





- All data in the SAP system must be protected from unauthorized access by users who do not explicitly have permission to access it.
- The system administrator assigns user authorization when maintaining user master data. During this process, you should determine exactly **which data** users are allowed to access and **what kind of access** should be allowed. For example, you might want to allow users to display data for all airline carriers, but only allow them to change data for certain selected ones. In this case, the system must look for a combination of the fields 'activity' and 'airline carrier' each time it performs an authorization check. Both fields must be filled with values during authorization creation as well (in this example, activity 'Change' and airline carrier 'LH' or activity 'Display' and airline carrier '*'). This is carried out by an authorization object composed of the fields 'Activity' and 'Airline carrier' that has to be addressed both during the authorization assignment process and whenever your program performs an authorization check.
- Authorization objects simply define the combination of fields that need to be addressed simultaneously and serve as templates for both authorizations and authorization checks. They are organized into object classes in order to make it easier to find and administer them; one object class or several may exist in each application. You call the authorization object maintenance transaction from the 'Development' menu in the ABAP Workbench. A complete list of all development objects, sorted according to class and including their corresponding fields and documentation, is part of this transaction.




- When making authorization checks in programs, you specify the object and values the user needs in an authorization to be able to access the object. You do not have to specify the name of the authorization.
- The above example checks whether or not the user is authorized for the object **S_CARRID,** which has the value **'LH'** in the field **CARRID** (airline) and the value **'02'** for 'Change' in the field **ACTVT** (activity). The abbreviations for the activities are documented in the tables **TACT** and **TACTZ** and also in the appropriate objects.
- **Important:** The Authority-Check statement performs the authority check and returns an appropriate return code value. When reading this return code, you can specify yourself the consequences of a missing authorization (for example, program terminates or skips some input lines).




- You must specify **all** fields of the object in an **AUTHORITY-CHECK.** Otherwise you receive a return code not equal to zero. If you do not want to carry out a check for a particular field, enter **DUMMY** after the field name.
  Example: When calling a transaction to change flight data, you should check whether or not the user is authorized to change the entries for a particular airline carrier: **AUTHORITY-CHECK OBJECT 'S_CARRID' ID 'ACTVT' FIELD '02' ID 'CARRID' DUMMY.**
- The most important return codes for **AUTHORITY-CHECK** are:
  - 0: The user has an authorization containing the required values.
  - 4: The user does not have the required authorization.
  - 8: The check could not successfully be carried out since not all fields of the object were specified.
- For a complete list of return codes, refer to the keyword documentation for the **AUTHORITY-CHECK** statement.
- You can only specify a single field after the **FIELD** addition, not a selection table. There are function modules which carry out the **AUTHORITY-CHECK** for all values in the selection table.

# Outlook: Reading Multiple Database Tables

- Information on database tables in R/3
- Reading database tables
- Authorization checks
- Outlook: Reading multiple database tables

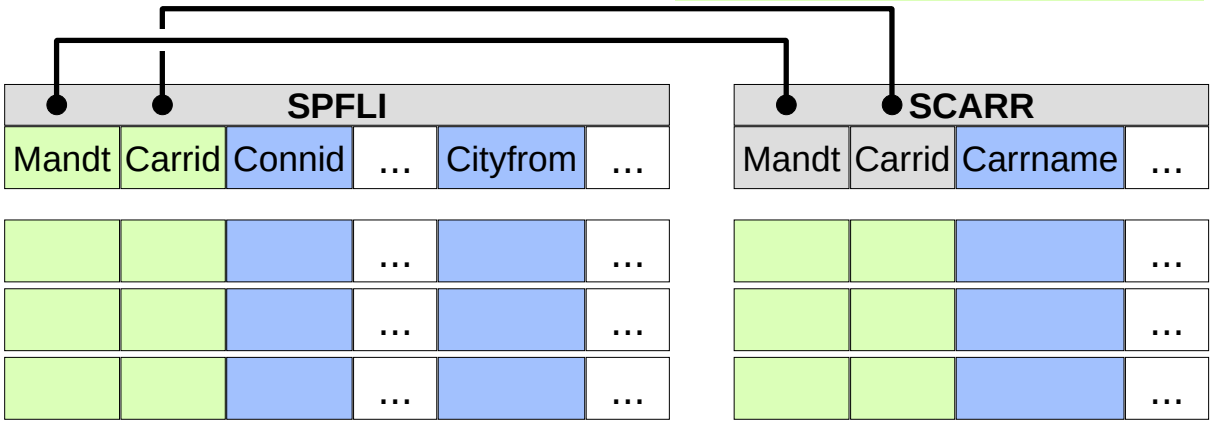
## Reading Multiple Database Tables

**Which database tables** should be taken into account?

```
spfli und scarr
```

**Prerequisite for linking** A common line can be created, provided:

```
spfli-mandt = scarr-mandt
spfli-carrid = scarr-carrid
```




**Which columns** are to be read, and from which database table?

```
spfli-mandt
spfli-carrid
```

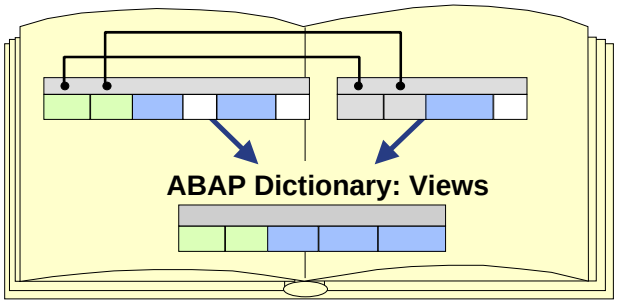
```
spfli-connid.
spfli-cityfrom
scarr-carrname
```



- You can access **several** database tables with **one** database query, provided they are logically related. To do so, you must enter the following information:
  - Which database tables should be accessed?
  - How should the link condition appear? In this condition, columns from the database tables are linked. A record is placed in a common line in the results table if all the field values of the linked columns match.
  - Which columns are to be read? If a column appears in more than one database table, you must specify the table from which this column is to be read.

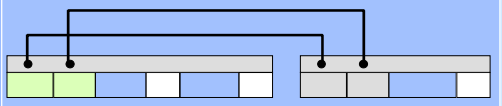
## ABAP Join and Dictionary Views

**Static link**




**Dynamic link**

ABAP program



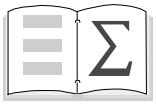
Link in the ABAP program using ABAP Join

```
SELECT ...
    FROM spfli INNER JOIN scarr
     ON  spfli~carrid = scarr~carrid
     AND spfli~connid = scarr~connid
    WHERE ...
```



- You can define the link conditions either statically or dynamically.
  - You can define the **static link** in the ABAP Dictionary. It is known as a view. There are several kinds of views: For detailed information, refer to *Basis->ABAP Workbench->BC ABAP Dictionary->Views.*
  - You can implement a **dynamic link** using ABAP statements. It is known as an ABAP join. An appropriate database query to the database used is generated in the database interface at runtime. For more information, see the keyword documentation for the **SELECT** statement (**FROM** clause).

# Reading Database Tables: Unit Summary



**You are now able to:**

- **Extract information about database tables from the ABAP Dictionary**
- **List various ways of finding database tables**
- **Program read access to specific columns and lines within a particular database table**
- **Implement authorization checks**
- **List the different kinds of read access possibilities for database tables**

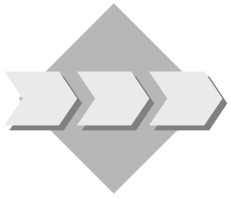
Exercises

**Unit: Reading Database Tables**

**Topic: SELECT Loops**

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

- Use the ABAP construction **SELECT**...**ENDSELECT** to read data from a database table into your program.



Create a program that (in this exercise) displays selected information on all flights, in list form. In the program, you should also calculate the percentage occupancy of each flight and display this as well.

You should extend the program so that, when the user chooses an airline on a selection screen, only the flight data for this airline is read from the database and displayed in a list.
The flight data is contained in the database table **SFLIGHT**.

**Program:** **ZBC400_##_SELECT_SFLIGHT**

**Model solution:** **SAPBC400DDS_SELECT_SFLIGHT**

1-1 Create the program **ZBC400_##_SELECT_SFLIGHT without a TOP include**. Assign your program to **development class ZBC400_##** and the change request for your project "BC400…" (## is your group number).

1-2 Create a structure with reference to the structure **SBC400FOCC**, which is defined in the ABAP Dictionary. To find out the components of the structure, look at its definition in the ABAP Dictionary.

1-3 To find out the fields in the database table **SFLIGHT**, look at its definition in the ABAP Dictionary. Read all the flights from table **SFLIGHT**. Use a **SELECT … ENDSELECT** block. Place the data line by line into the structure that you created in exercise 1-2. Make sure that you only read fields from the database table for which there is also a target field in the structure.

1-4 Within the **SELECT** loop, calculate the percentage occupancy using the corresponding field of the work area. Assign the result to the **PERCENTAGE** field in your structure.

1-5 Create a list displaying the information you read from the database and the percentage occupancy of each flight.

1-6 Adapt the program so that the system reads data for one airline only from the database. The user should be able to input this airline on a selection screen.

1-6-1 Program a selection screen with a field for the airline. Use the **PARAMETERS** statement. Bear in mind that the field should have the same type as the column in the database table **SFLIGHT**, which contains the airline ID.
Note: You can use the short text of the data element as a descriptive text for the input field on the selection screen. Activate your program. Navigate from the Editor to the tool for maintaining selection texts, by choosing ***Goto →Text elements → Selection texts***. Make sure you are in *Change* mode and check the *Dictionary reference* field. Save your changes and test the program.

1-6-2 Limit the choice of lines using the **CARRID** field in a **WHERE** clause.

1-6-3 Why is the program with the **WHERE** clause less runtime-intensive than the program you extended in parts 1-1 to 1-5 of this exercise?

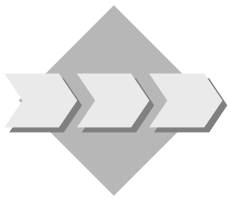
1-6-4 Compare the **WHERE** clause with the key fields of the database table **SFLIGHT**. What do you notice?

**Unit: Reading from the Database**

**Topic: SELECT Loops and Filling Internal Tables**

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

- Use the ABAP construction **SELECT … ENDSELECT** to read data from a database table into your program and fill an internal table.



The task is the same as in exercise 1. Display the data on the list sorted by the percentage occupancy. To do this, you must fill an internal table with the required data and then sort it by the occupancy field.

**Program:** **ZBC400_##_SELECT_SFLIGHT_ITAB**

**Model solution:** **SAPBC400DDS_SELECT_SFLIGHT_TAB**

2-1 Copy your program **ZBC400_##_SELECT_SFLIGHT** or the model solution **SAPBC400DDS_SELECT_SFLIGHT** and rename it **ZBC400_##_SELECT_SFLIGHT_ITAB**. Assign your program to **development class ZBC400_##** and the change request for your project “BC400…” (## is your group number).

2-2 In addition to your structure that refers to the ABAP Dictionary type **SBC400FOCC**, create an internal table with the line type **SBC400FOCC**. Use the where-used list for the ABAP Dictionary line type SBC400FOCC to find a suitable table type in the Dictionary.

2-3 Fill the internal table line by line by using an **APPEND** statement in the **SELECT** loop.

2-4 Sort the internal table according to occupancy.

2-5 Display the sorted contents of the internal table in a list. Use a **LOOP … ENDLOOP** structure to do this.

**OPTIONAL:**

**Model solution: SAPBC400DDS_SELECT_ARRAY_FETCH**

2-6 Copy the program **ZBC400_##_SELECT_SFLIGHT_ITAB** to program **ZBC400_##_ARRAY_FETCH_SFLIGHT**.

2-7 Replace the SELECT loop with an array fetch and fill the internal table with the relevant data from the database table SFLIGHT. The column for the percentage occupancy only contains initial values.

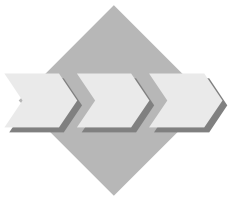
2-8 Calculate the percentage occupancy for each line of the internal table using a loop, and change the line using a MODIFY statement. To find out how to use MODIFY within a loop, refer to the keyword documentation.

**Unit: Reading from the Database**

**Topic: Authorization checks**

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

- Perform authorization checks



Change your programs **ZBC400_##_SELECT_SFLIGHT** and **ZBC400_##_SELECT_SFLIGHT_ITAB** so that the data can only be read from the database and displayed in the list if the user has read authorization for the required airline.

**Program:** **ZBC400_##_AUTHORITY_CHECK**

**Model solutions:** **SAPBC400DDS_AUTHORITY_CHECK**,
**SAPBC400DDS_AUTHORITY_CHECK_2** and
**SAPBC400DDS_AUTHORITY_CHECK_3**

3-1 Change your programs **ZBC400_##_SELECT_SFLIGHT** and **ZBC400_##_SELECT_SFLIGHT_ITAB** as follows:

Add an authorization check that checks against the object **S_CARRID**. Make sure that the database is not accessed if the user does not have authorization for the airline that he or she entered on the selection screen. Instead, ensure that the program displays an appropriate error message.

3-2 Restart your program. On the selection screen, try entering **AA** for the airline, then **UA**.

Solutions

**Unit: Reading from the Database**

**Topic: SELECT Loops**

1-2 The structure type **SBC400FOCC** (maintained in the ABAP Dictionary) contains the following fields: **CARRID** (type **S_CARR_ID**), **CONNID** (type **S_CONN_ID**), **FLDATE** (type **S_DATE**) **SEATSMAX** (type **S_SEATSMAX**), **SEATSOCC** (type **S_SEATSOCC**), and **PERCENTAGE** (type **S_FLGHTOCC**).

1-3 The **CARRID**, **CONNID**, **FLDATE**, **SEATSMAX** and **SEATSOCC** fields, in the database table **SFLIGHT** have the same types as their identically-named counterparts in the structure type **SBC400FOCC**. The database table **SFLIGHT** does not contain a *percentage occupancy* field.

**Model solution: Program SAPBC400DDS_SELECT_SFLIGHT**

```
*&---------------------------------------------------------------------*
*& Report        SAPBC400DDS_SELECT_SFLIGHT                  *
*&---------------------------------------------------------------------*
REPORT  sapbc400dds_select_sflight    .

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

START-OF-SELECTION.
* Select all datasets from database table SFLIGHT corresponding
* to carrier PA_CAR
  SELECT carrid connid fldate seatsmax seatsocc FROM sflight
        INTO CORRESPONDING FIELDS OF wa_flight
        WHERE carrid = pa_car.
* Calculate occupation of each flight
   wa_flight-percentage =
   100 * wa_flight-seatsocc / wa_flight-seatsmax.
* Create List
   WRITE: / wa_flight-carrid,
            wa_flight-connid,
            wa_flight-fldate,
            wa_flight-seatsocc,
```

```
        wa_flight-seatsmax,
        wa_flight-percentage,'%'.
ENDSELECT.
```

1-6-1 There are two different ways to do this:
**PARAMETERS pa_car TYPE s_carr_id.** or
**PARAMETERS pa_car TYPE sflight-carrid.**

1-6-2 SELECT .... FROM SFLIGHT **WHERE CARRID = pa_car.**

1-6-3 Firstly, the quantity of data is smaller, since the system does not have to transport all the records from the database to the application server. Secondly, the records can be chosen using the primary key. (see 1-6-4).

1-6-4 The **MANDT** field of the **CLNT** Dictionary type, which contains the client is flagged as a key field. The field does not appear in the **WHERE** condition, even though all key fields should usually be included in this clause.
**Reason:** When the runtime system accesses a client-specific table, it assumes that data should only be taken into account if it belongs to the client in which the user is logged. Thus by default, the database interface automatically assumes the **WHERE** condition to be **WHERE mandt = sy-mandt**. In the above example, this allows the database to use the primary index to select lines from the table. If you specified only the flight number instead of the airline ID, then the **CARRID** field would be missing from the key sequence. This would mean that the database could not use the primary index. This usually leads to longer runtimes. You can measure the runtimes of the different programs you have created by choosing ***Execute →Runtime analysis***. In the next screen, choose *Execute*, followed by *Evaluate.* The system displays a screen with three columns, which shows clearly what proportion of the program runtime is taken up by ABAP processing, database accesses, and by general load on the R/3 System.

**Unit: Reading from the Database**

**Topic: SELECT Loops and Filling Internal Tables**

**Model solution: Program SAPBC400DDS_SELECT_SFLIGHT_TAB**

```
*&---------------------------------------------------------------------*
*& Report        SAPBC400DDS_SELECT_SFLIGHT_TAB              *
*&                                                           *
*&---------------------------------------------------------------------*

REPORT  sapbc400dds_select_sflight_tab    .

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

START-OF-SELECTION.

* Select all datasets from database table SFLIGHT corresponding
* to carrier PA_CAR
  SELECT carrid connid fldate seatsmax seatsocc FROM sflight
         INTO CORRESPONDING FIELDS OF wa_flight
         WHERE carrid = pa_car.
* Calculate occupation of each flight
    wa_flight-percentage =
    100 * wa_flight-seatsocc / wa_flight-seatsmax.
* Build up internal table
* Insert single line into internal table
    APPEND wa_flight TO it_flight.
ENDSELECT.
* sort internal table
SORT it_flight BY percentage.
```

*** Create List from sorted internal table**

```
LOOP AT it_flight into wa_flight.

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

**Unit: Reading from the Database**

**Topic: Array Fetch (optional)**

**OPTIONAL:**

**Model solution: Program SAPBC400DDS_SELECT_ARRAY_FETCH**

```
*&---------------------------------------------------------------------*
*& Report        SAPBC400DDS_SELECT_ARRAY_FETCH                *
*&                                                             *
*&---------------------------------------------------------------------*

REPORT  sapbc400dds_select_array_fetch    .

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

START-OF-SELECTION.
*---------------------------------------------------------------------
* Optional:
* Array Fetch to fill the first 5 comumns of internal table,
*---------------------------------------------------------------------
  SELECT carrid connid fldate seatsmax seatsocc FROM sflight
         INTO CORRESPONDING FIELDS OF TABLE it_flight
         WHERE carrid = pa_car.

* At least one dataset is selected
  IF sy-subrc = 0.
*Calculate percentage in a loop and modify internal table to fill
* 6th column of internal table
    LOOP AT it_flight INTO wa_flight.
      wa_flight-percentage =
      100 * wa_flight-seatsocc / wa_flight-seatsmax.
      MODIFY it_flight FROM wa_flight
```

```
                INDEX sy-tabix
                TRANSPORTING percentage.
  ENDLOOP.

  SORT it_flight BY percentage.

* Loop over internal table to write content of datasets on list
  LOOP AT it_flight INTO wa_flight.

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

**Unit: Database Dialogs 1**

**Topic: Authorization Check**

**Model solution:**

**Programs SAPBC400DDS_AUTHORITY_CHECK,**
**SAPBC400DDS_AUTHORITY_CHECK_2 and**
**SAPBC400DDS_AUTHORITY_CHECK_3**

```
*&---------------------------------------------------------------------*
*& Report  SAPBC400DDS_AUTHORITY_CHECK                                 *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  sapbc400dds_authority_check_#.
CONSTANTS  actvt_display TYPE activ_auth value '03'.
DATA: wa_flight TYPE sbc400focc,
      ...
PARAMETERS: pa_car TYPE s_carr_id.

START-OF-SELECTION.
* Authority-Check: Is user authorized to read data for carrier
* PA_CAR?
  AUTHORITY-CHECK OBJECT 'S_CARRID'
    ID 'CARRID' FIELD pa_car
    ID 'ACTVT'  FIELD actvt_display.

  CASE sy-subrc.
* User is authorized
    WHEN 0.

*       SELECT loop or Array Fetch ...

* User is not authorized or other error of authority-check
    WHEN OTHERS.
```

```
* The message in this program is a simplified version only.
* Error messages for selection screens will be explained in
* detail in unit 'User Dialog Selection Screen'.

    MESSAGE ID 'BC400' TYPE 'I' NUMBER '045' WITH pa_car.

  ENDCASE.
```

# Internal Program Modularization

## Contents:

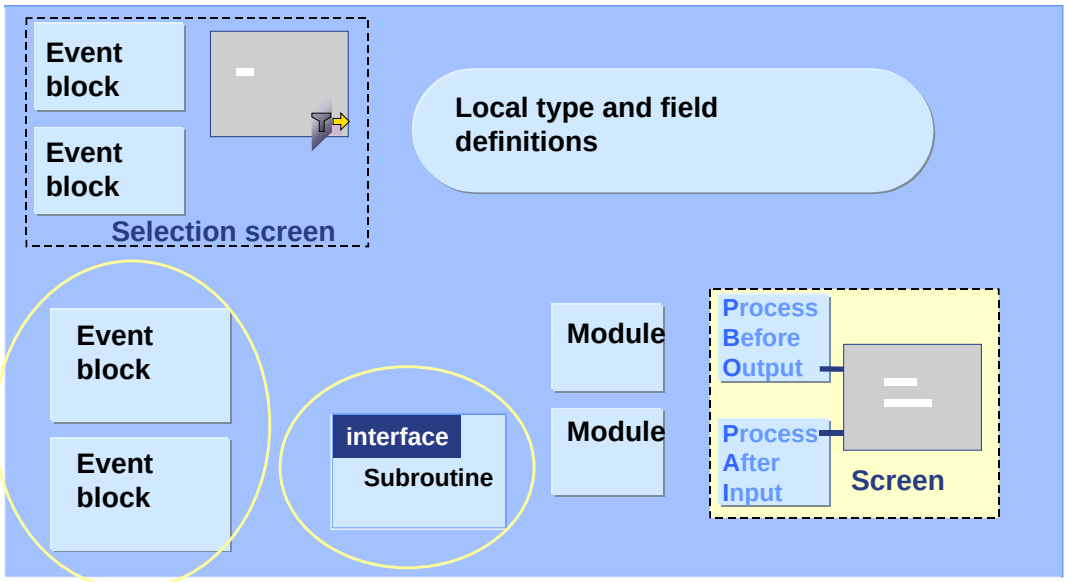
- ABAP event blocks
- Subroutines

# Internal Program Modularization: Unit Objectives

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

- **Explain how a program containing event blocks functions at runtime using `INITIALIZATION` and `START-OF-SELECTION` as examples**
- **Encapsulate functions in a simple subroutine with an interface**

## Possible Elements in an ABAP Program





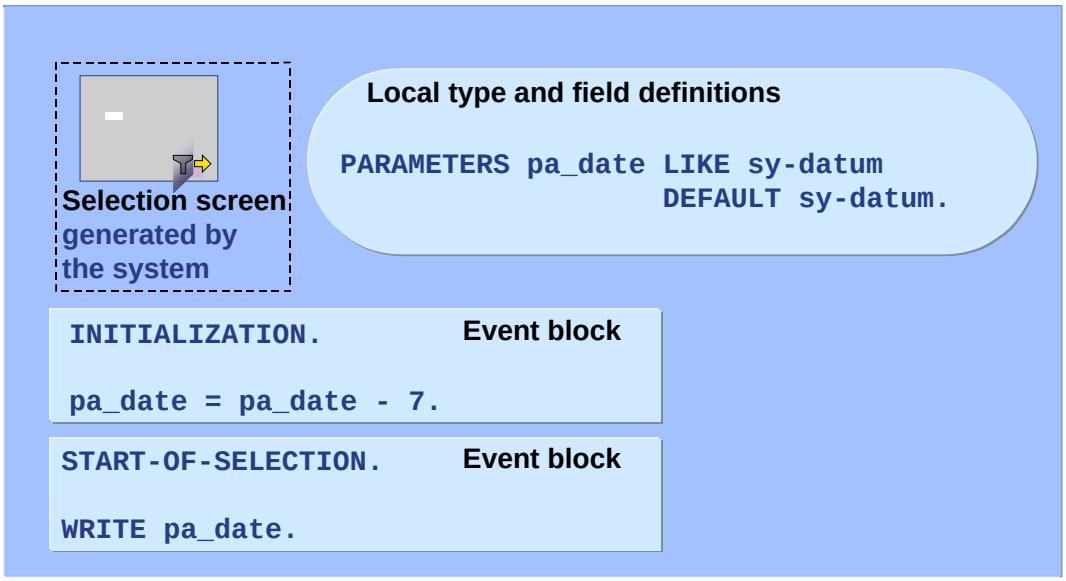

- An ABAP program is a collection of processing blocks. A processing block is a passive section of program code that is processed sequentially when called.
- Processing blocks are the smallest units in ABAP. They cannot be split, which also means that they cannot be nested.
- There are various kinds of ABAP processing blocks:
  - **Event blocks** are ABAP processing blocks that are called by the runtime system. Event blocks can logically belong to the executable program, to the selection screen, to the list or to the screen. This unit deals with event blocks that belong to the executable program. You can find information on event blocks that belong to the selection screen, the list or the screen in the units on user dialogs.
  - Subroutine processing is triggered by an ABAP statement. Parameters can be passed to subroutines using an interface and subroutines can contain local variables.
  - **Modules** are special ABAP processing blocks for processing screens. Therefore modules are dealt with in the *User Dialogs: Screens* unit.
- Memory areas are made available for all a program's global data objects when that program is started. Declarative ABAP statements are therefore not components of ABAP processing blocks but are collected from the overall source code using a search when the program is generated. The exceptions to this are local data objects in subroutines.

# Event Blocks

Event Blocks

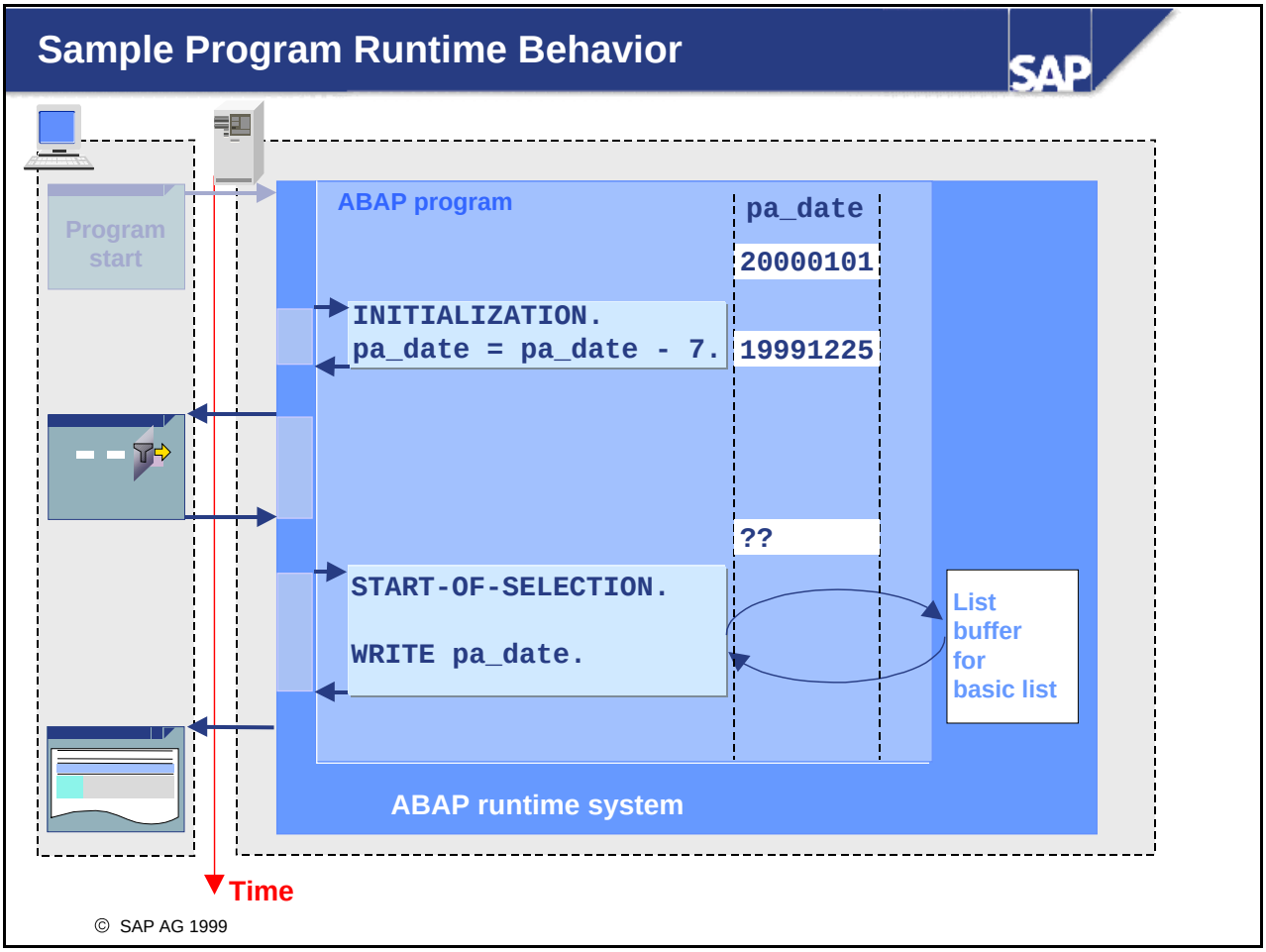
Subroutines

## Example: ABAP Program with Event Blocks and a Selection Screen



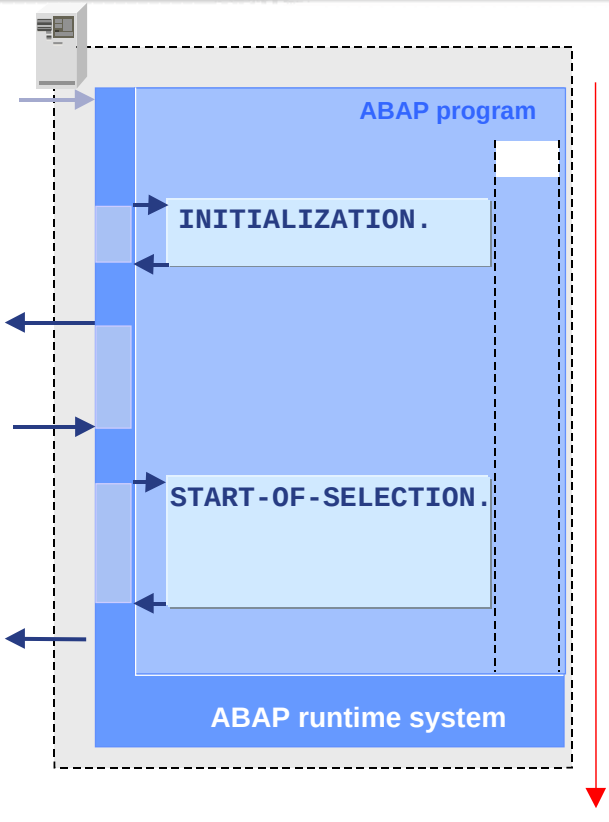



- In all of the programs that we have seen so far in this course, there has only been one processing block in addition to the data declaration. In this case, there is no need to declare the processing block explicitly. However, in more complex programs, we will require several different processing blocks and will need to specify the type and name.
- The program shown above is an example of event blocks. It contains an input value for a date on a selection screen. The default value is the date from the week before. This cannot be realized by a default value from the **PARAMETERS** statement, since a calculation is required. The **DEFAULT** addition to the **PARAMETERS** statement ensures that the data object is filled with the default value at the start of the program. Default values can be literals or fields from the **sy** structure. The runtime system fills the **sy-datum** field with the current date at the start of the program. You can use the **INITIALIZATION** event block to change variables at runtime but before the standard selection screen is sent. **START-OF-SELECTION** is an event block for creating lists.
- All global declarations are recognized as such by the system by the declarative ABAP key words that they use, and these form a logical processing block (regardless of where they are placed in the program code). When you generate the program, the system searches the entire program code for declarative statements. However, for the sake of clarity, you should place all declarative statements together at the beginning of your programs. The **PARAMETERS** statement is one of the declarative language elements. When the program is generated, a selection screen is also generated along with the information on the elementary data object of the type specified.




- The easiest events to understand are those for an executable program (type 1).
- The ABAP runtime system calls event blocks in a sequence designed for generating and processing lists:
  - First, the **INITIALIZATION** event block is called
  - Then a selection screen is sent to the presentation server
  - After the user leaves the selection screen, **START-OF-SELECTION** is called
  - If the **START-OF-SELECTION** event block contains the ABAP statements **WRITE**, **SKIP** or **ULINE**, a list buffer is filled.
  - The list buffer is subsequently sent to the presentation server as a list.

## Event Blocks in Executable Programs

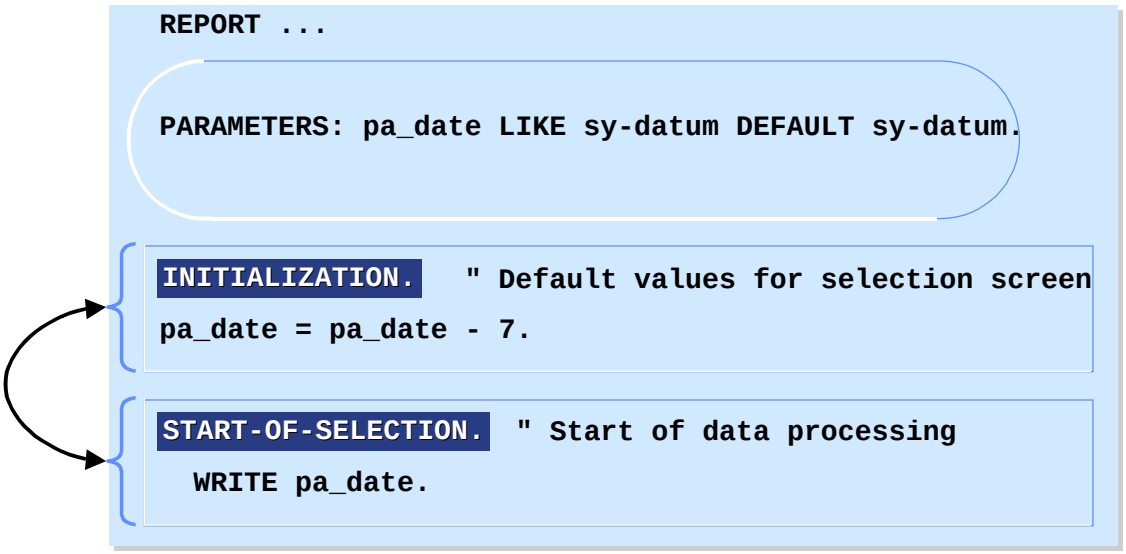



- Introduced with an event keyword
- Delimited by the start of the next processing block
- Different event blocks have different tasks
- The sequence in which the event blocks are processed is determined by the runtime system
- Default event block: `START-OF-SELECTION`



- Event blocks are processing blocks that are called by the ABAP runtime system. The sequence if which they are processed is determined by the runtime system.
- In executable programs, there are different event blocks for the various tasks involved in creating lists.

## Syntax: Event Blocks




**The sequence of event blocks in the source code has no effect on the sequence in which they are called by the ABAP runtime system. call**

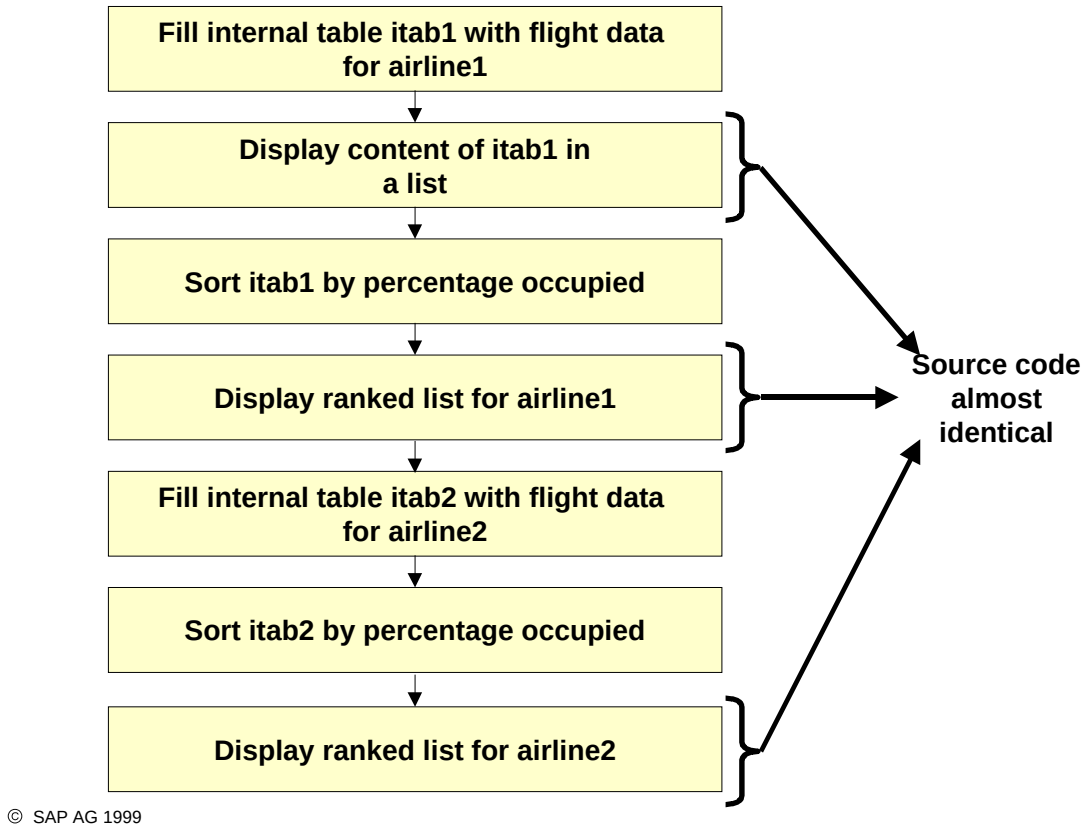


- In an ABAP program, an event block is introduced with an **event key word**. It ends when the next processing block starts. There is no ABAP statement that explicitly concludes an event block.
- Event blocks are called by the ABAP runtime system. The order in which you arrange the event blocks in your program is irrelevant - the system always calls them in a particular order.
- **START-OF-SELECTION** is the first event for processing data and generating a list. It is called by the ABAP runtime system as soon as you have left the standard selection screen.
- **INITIALIZATION** is an event that you can use if you need to set a large number of default values. This event block allows you to set default values that can only be determined at runtime. In the above example, the date 'A week ago' is calculated and placed in data object pa_date. The ABAP runtime system then sends a selection screen to the presentation server containing the calculated value as a default. The value can, of course, still be changed.

# Subroutines

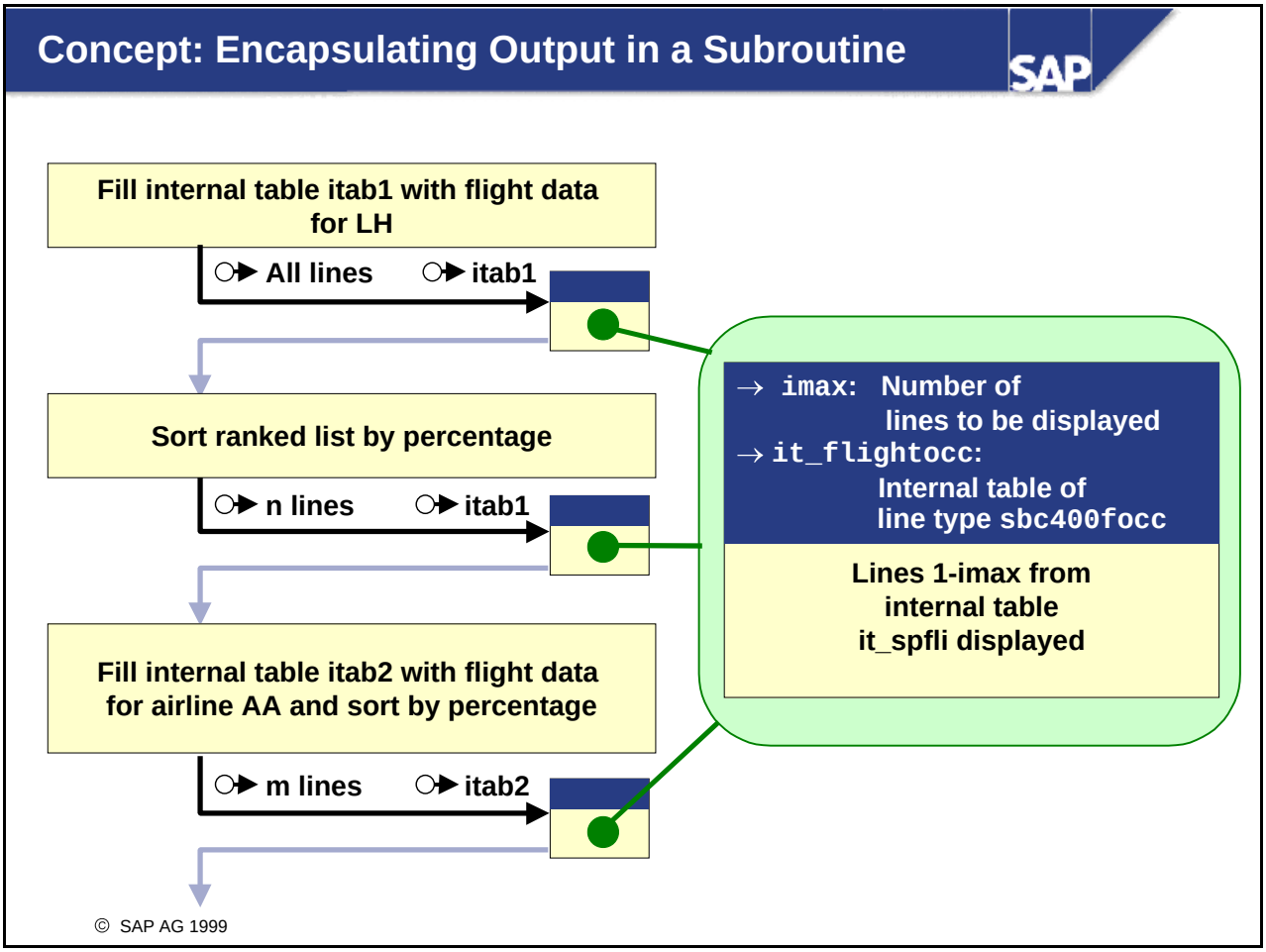
- Events
- Subroutines

## Example: Flow Chart

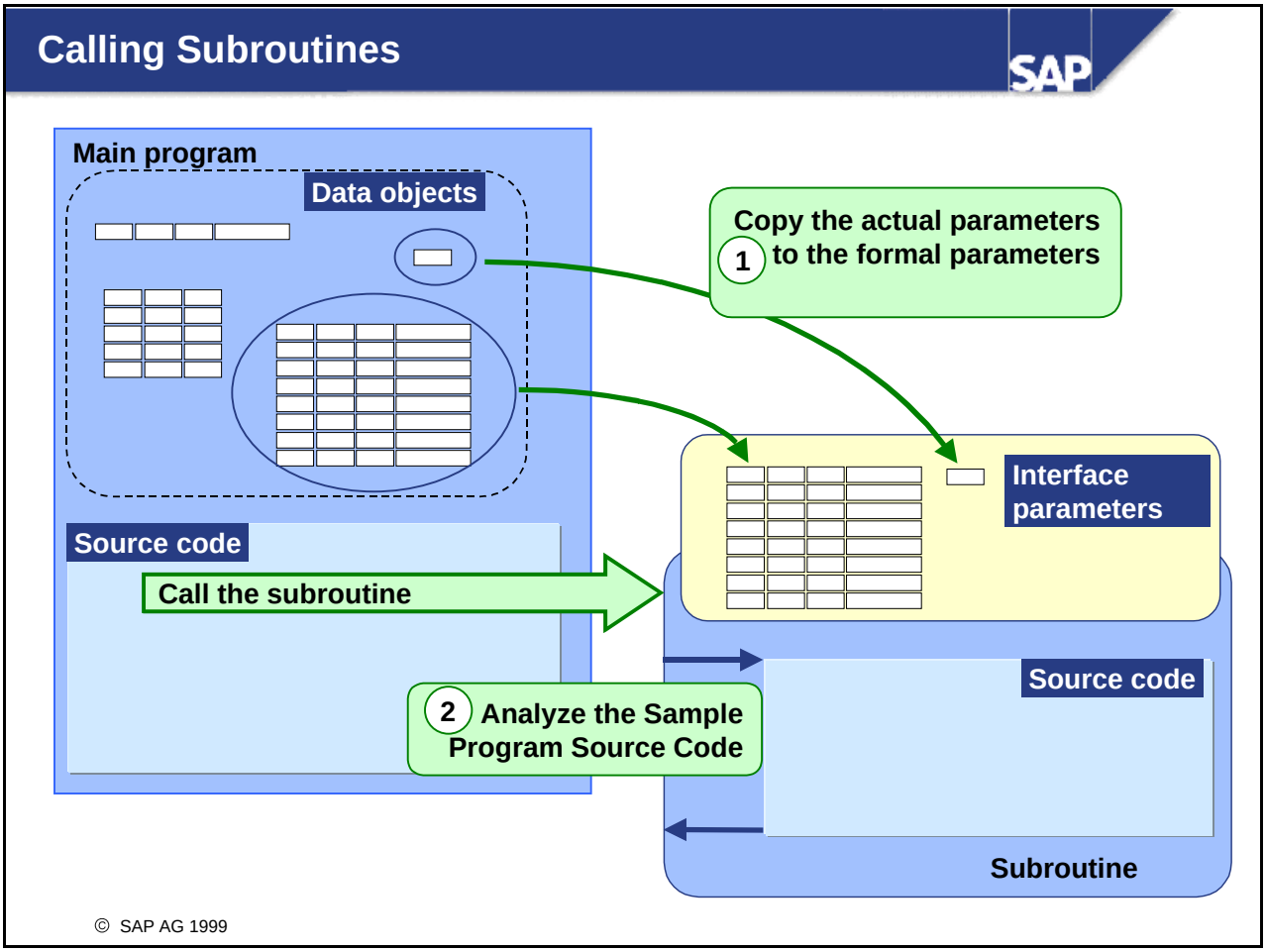





- You will be creating a ranked list of data for two different airlines. To make sure the program is flexible, keep the data for each airline in an internal table. You should create a three-part list.
- The source code for generating the list from the content of the internal tables is almost identical in each case. This means that, if the program is extended - for example, to add another column and display the contents in a list - you must change the source text in three different places.

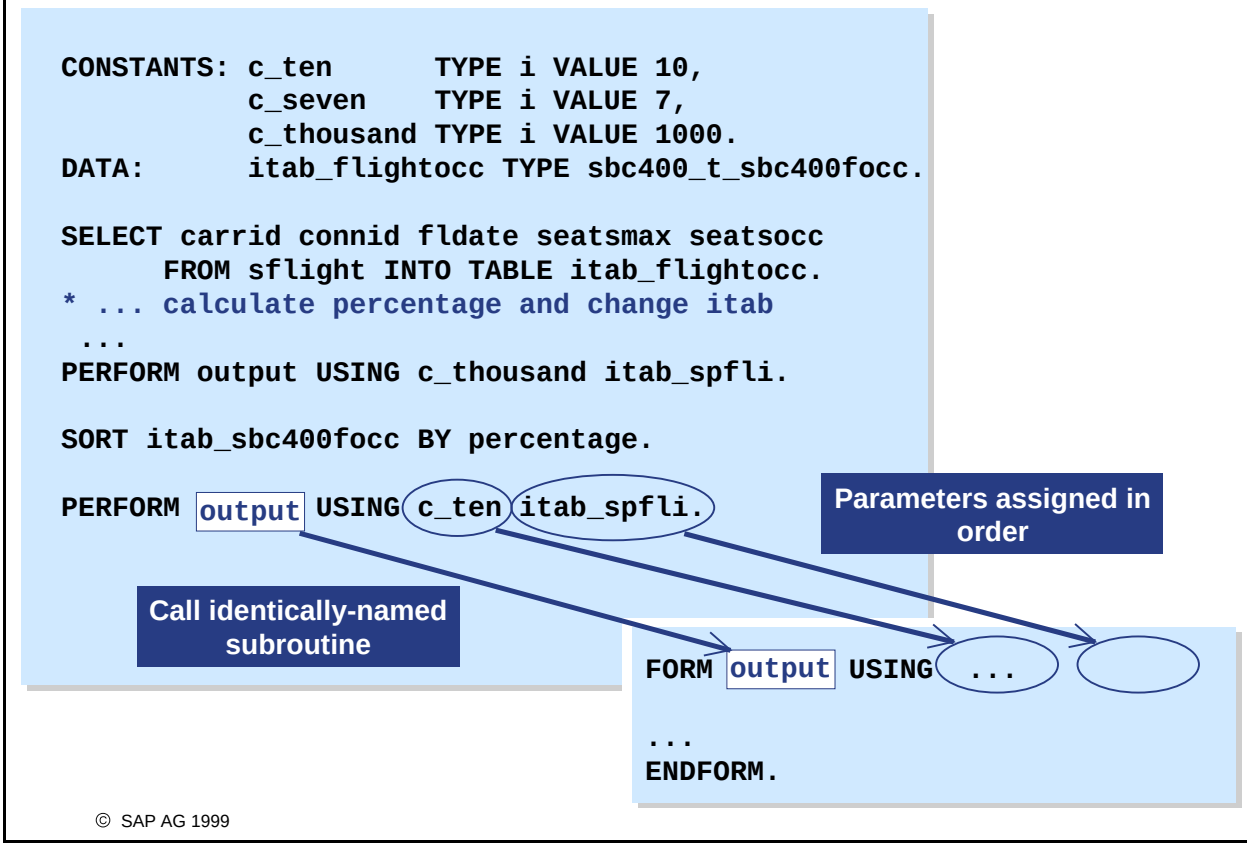



- The bigger the program, the more important it is to modularize it. The main advantages of this approach are that you can:
  - **Reuse the program:** If identical (or very similar) source code sections occur several times in a program, it would be nice to be able to implement them just once, and then branch to this source text from different parts of the program. Generically speaking, this is often possible using subroutines. That is, you implement a solution so that, when this subroutine is called, specific variables are filled with values using interface parameters. These values can come from different program variables, depending on the call made.
  - **Extend the program:** For example, if you have modularized the display of an internal table (see graphic), you can extend the program with very little effort to display another internal table with an identical line type, but for a different airline. You could also extend the program to allow the user to interact with it and thus change the display.
  - **Read and maintain the program more easily:** It is easier to find your way round a program, if it has been encapsulated in logical sections. Each subroutine can then be seen as a black box, until the programmer has to implement it in detail. Ideally, the purpose of the subroutine should be clear from its name, interface, and from comments.



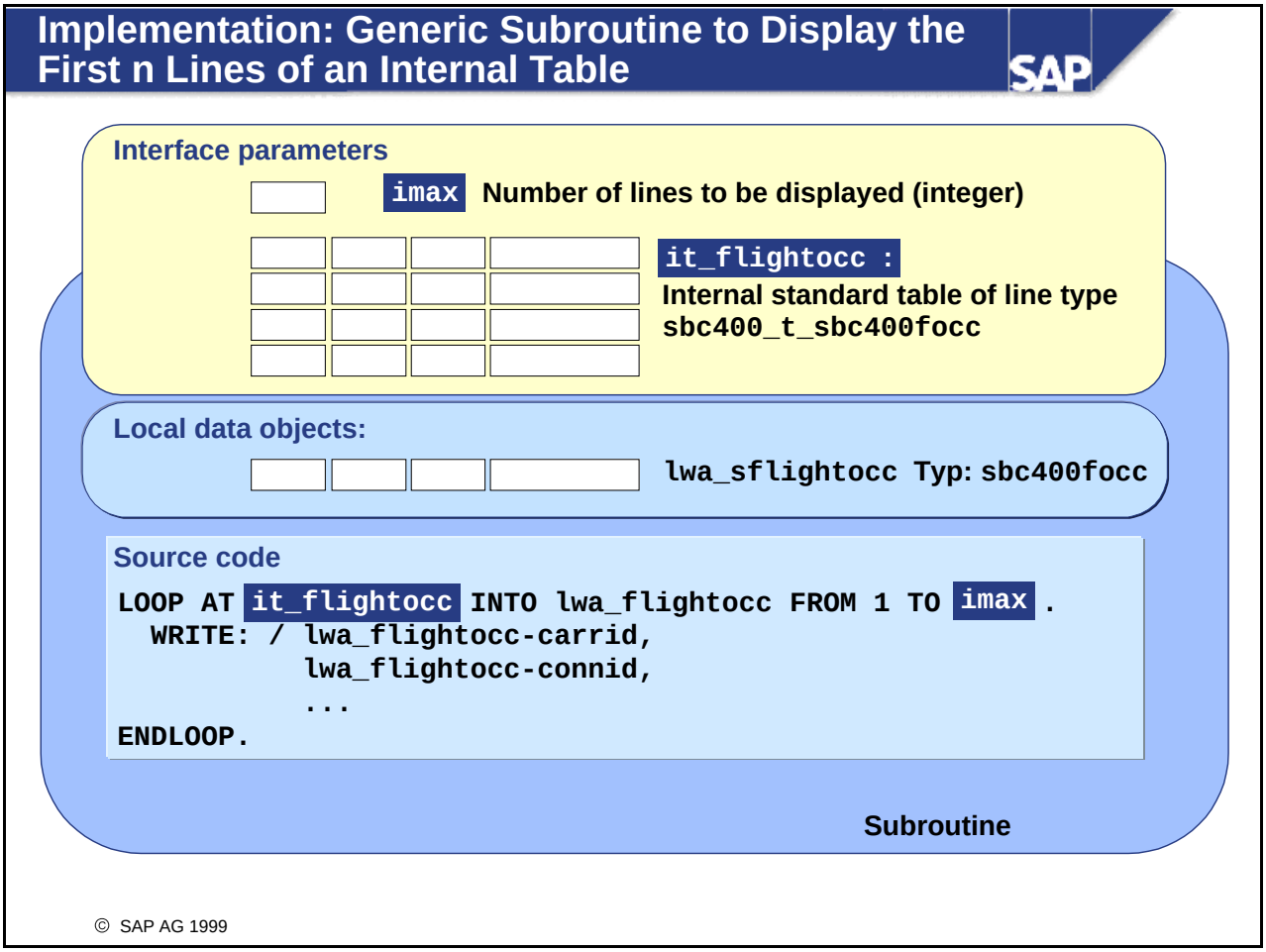

- First we will focus on a subroutine that only receives data when called.
- When the subroutine is called, a temporary runtime object is generated for the subroutine. For the runtime of the subroutine, memory is made available for the interface parameters and local data objects. All import parameters of the subroutine must be assigned to type-compatible data objects of the calling program.
  - When the subroutine is called, the values are copied to the interface parameters.
  - The subroutine source code is then processed sequentially.
  - After the subroutine has been processed, the system continues to execute the source code of the calling program, immediately after the call.

## Syntax Example: Calling the Subroutine



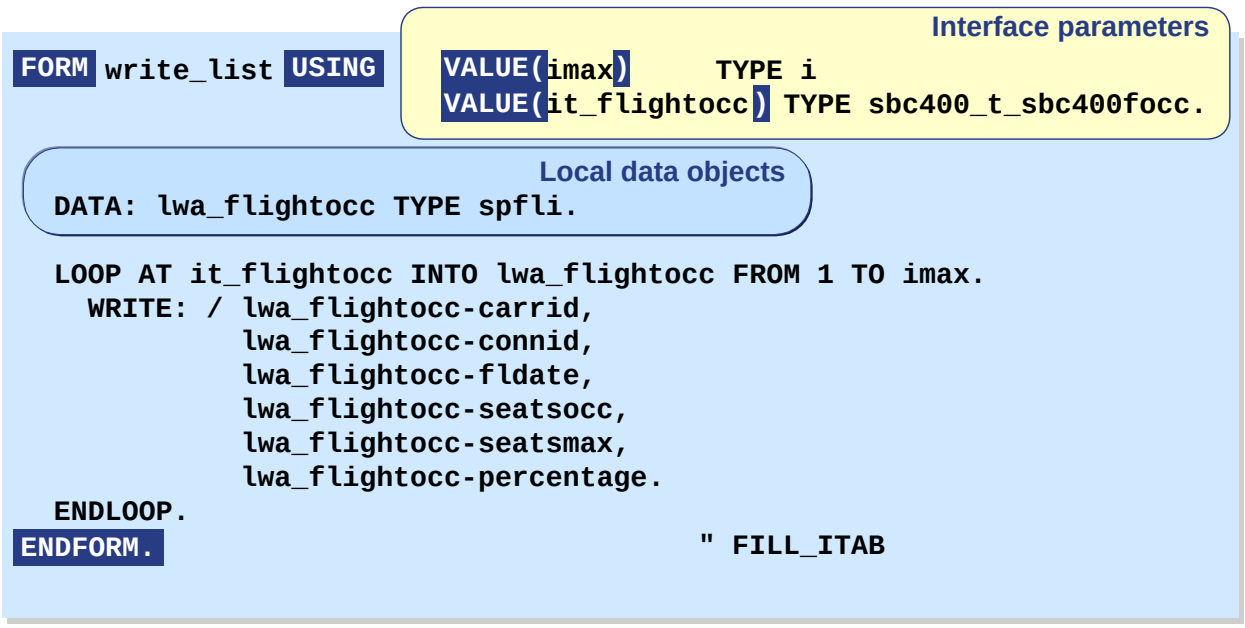



- Subroutine processing is triggered by the **PERFORM** statement.
  **PERFORM <name of subroutine>**
  **USING <data object 1> <data object 2><...><data object n>.**
  The name of the subroutine must be identical to the name in the PERFORM statement. The subroutine must be defined in the program. The data objects are assigned to the interface parameters in strict order.
- The next slides discuss defining a subroutine.



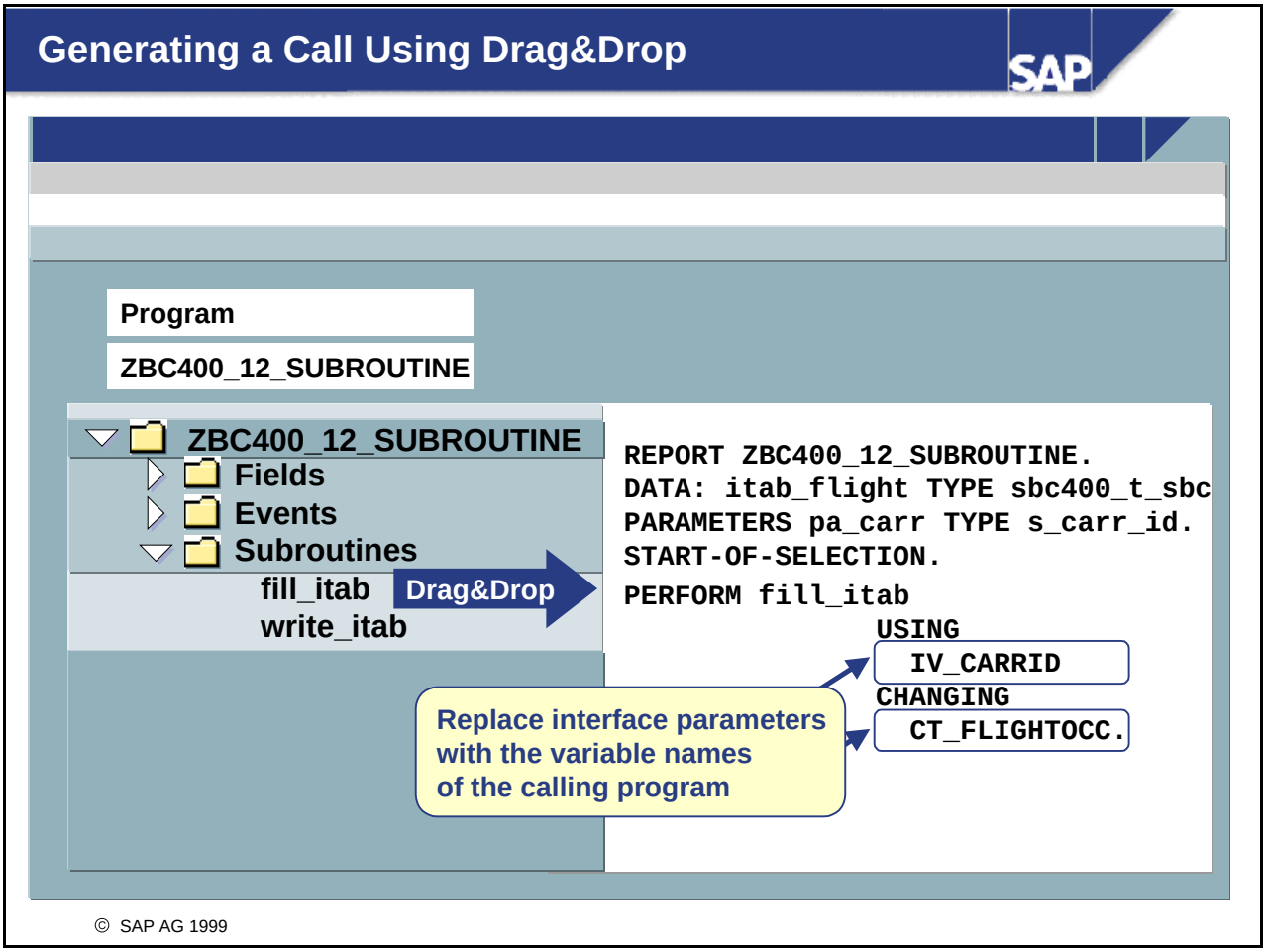

- When you define a subroutine, you specify which interface parameters are to be offered. You enter a name for each interface parameter and specify its type attributes.
- You define the data objects using the ABAP statement **DATA.** The memory for these data objects is made available as soon as the subroutine is called. At the end of the subroutine's runtime, the memory is released again.
- In the subroutine's source code, you can access the local data objects and interface parameters by name, a technique with which you should already be familiar.

## Syntax: Generic Subroutine to Display the First n Lines of an Internal Table

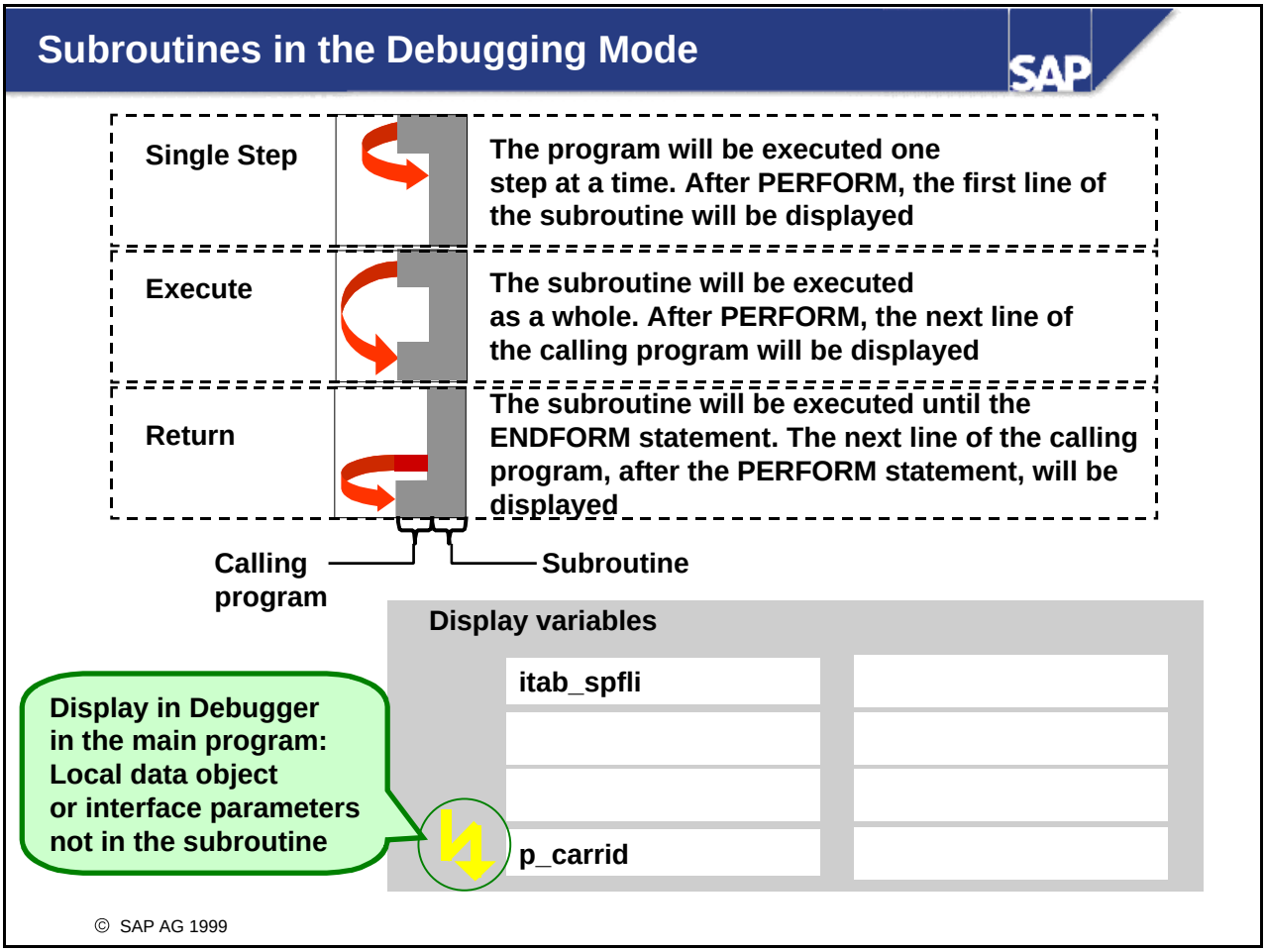




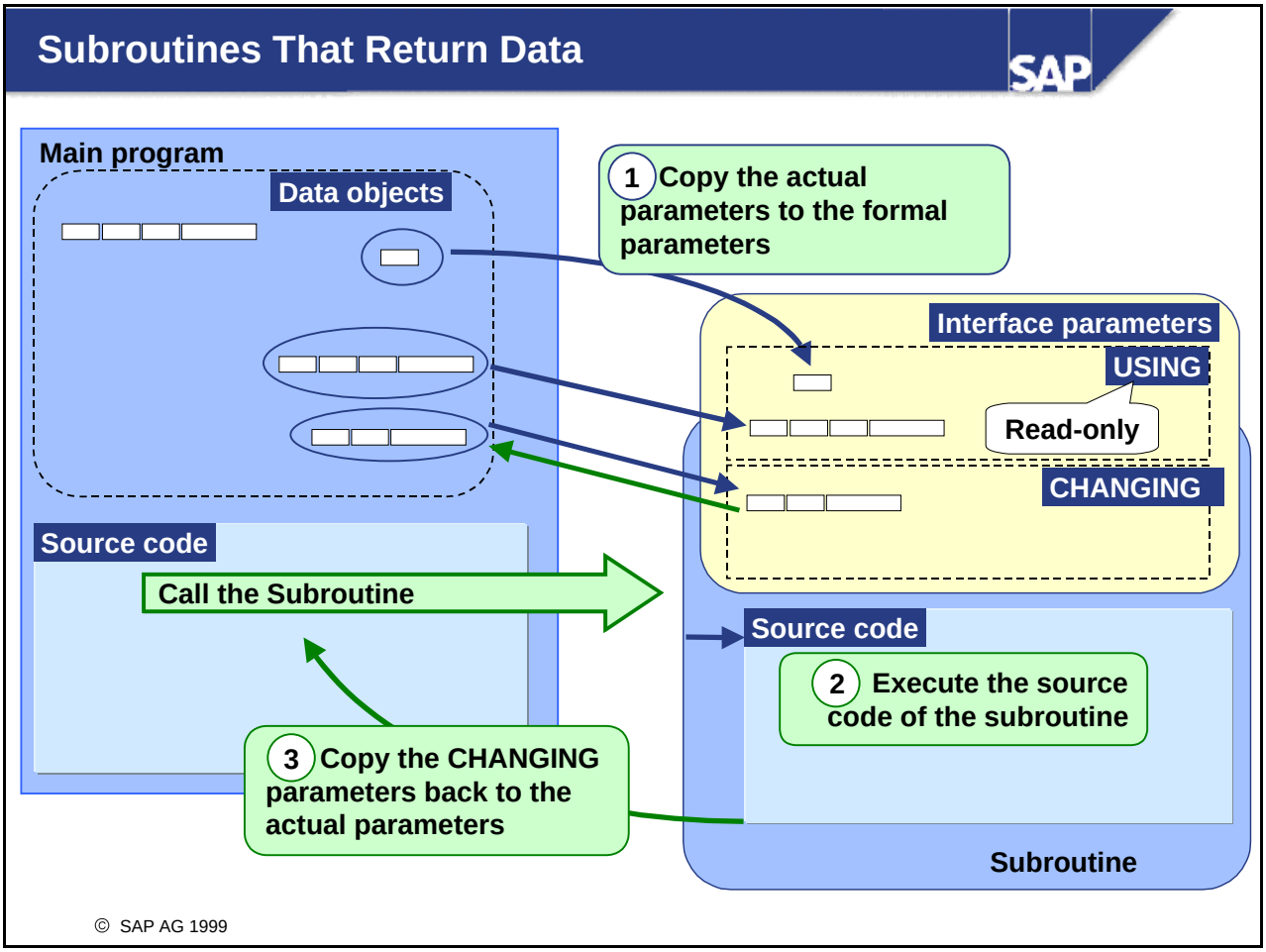

- You define the subroutine by reserving a source code section for it at the end of the program, using the **FORM ... ENDFORM** statements. **FORM <name of subroutine> USING <parameter1><parameter2>.**
  - **<name of subroutine>** specifies the name of the subroutine. The name must be a single word and begin with a letter, not a digit.
  - After **USING**, you list all of the parameters that the subroutine only needs to read.**<parameter>** is made up of the following:
    **VALUE(<parameter name>) TYPE <type>**.
    - You use **VALUE(...)**to specify that the value of the data object assigned to the interface parameter at runtime should be passed to the subroutine. (This is known as passing by value).
    - You can choose any name for the interface parameter **<parameter name>**. In the subroutine you can then use this name to access the value passed from the calling program.
    - After the name of the parameter you specify the type attributes of the interface parameter. In the **PERFORM** statement, if the assigned data object is not compatible with the type of the interface parameter, the syntax check returns an error.




- You can use the context menu for a subroutine to generate a where-used list for it. The system displays a list showing all the lines in the source code where the subroutine is called. You can double-click from any line in this list to the relevant line in the source code.
- You can call any available subroutine using:
  - **Drag&Drop:** In the tool area display the source code to which you want to add the subroutine call. (Make sure you are in change mode). In the navigation area, display the program object list and expand the *Subroutines* node. Now drag&drop the subroutine to the point where you want to call it. Replace the name of each interface parameter with the name of the data object that is to be assigned to each interface parameter.
  - **Pattern statement in the Editor:** In the tool area display the source code to which you want to add the subroutine call. (Make sure you are in change mode). Place your cursor in the source code where you want to insert the call. Choose the *Pattern* pushbutton and enter the name of the subroutine in the appropriate field. If you want you can use the possible entries help, which contains all the subroutines belonging to the program. Choose *Continue* to confirm your entries and replace the name of the interface parameters with the appropriate data objects.




- You can investigate subroutines in debugging mode as follows:
  - If you analyze the program step-by-step, each line of the subroutine is displayed in the order in which it is executed.
  - If you choose the *Execute* function, immediately before a **PERFORM** statement, the system executes the whole subroutine. The Debugger stops at the line immediately under the **PERFORM** statement.
  - If you choose the *Return* function inside a subroutine, this subroutine is executed until the **ENDFORM** statement. The Debugger stops at the line immediately after **PERFORM**.
- You can trace both the global and local variables of the subroutine in the field view. Since local data objects only exist for as long as the subroutine is being processed, and are only visible within the subroutine, local variables can only be displayed while the subroutine is running. If a variable is not visible, the system indicates this by displaying a yellow lightning bolt beside the input field.




- Generally, you also need to be able to change variables in subroutines. **CHANGING** interface parameters are available for this purpose. The runtime behavior of a subroutine call will then be as follows:
- When the subroutine is called, a temporary runtime object is generated for the subroutine. For the runtime of the subroutine, memory is made available for the interface parameters and local data objects. All **USING** (read-only access) and all **CHANGING** parameters of the subroutine must be assigned to type-compatible data objects of the calling program.
  - When the subroutine is called, the actual values are copied to the interface parameters. (This applies to both **USING** and **CHANGING** parameters).
  - The subroutine source code is then processed sequentially.
  - After the subroutine has been processed, the values of all the **CHANGING** parameters are copied back to the actual parameters and the system continues to execute the source code of the calling program, immediately after the call.

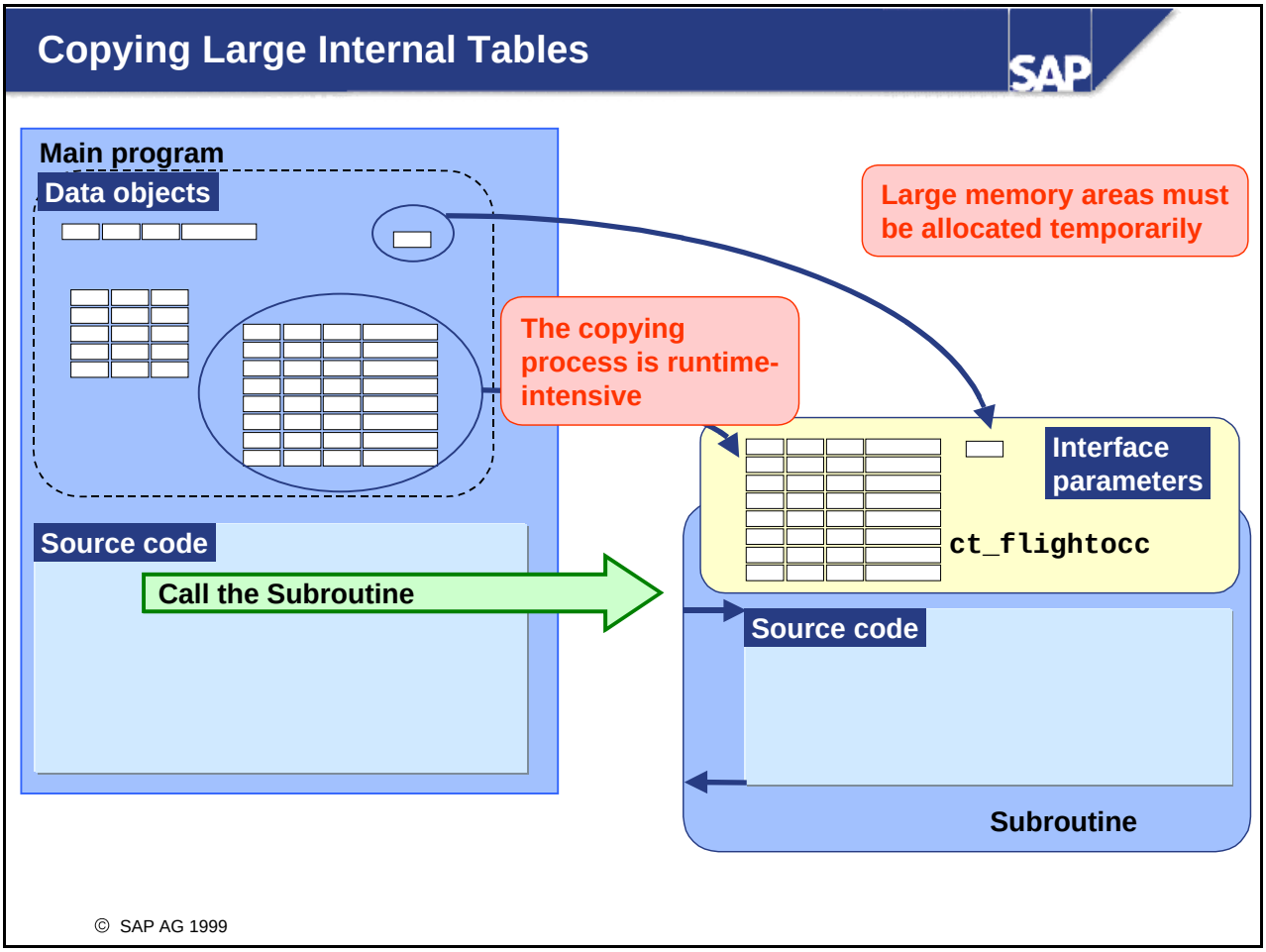
## Syntax Example: Subroutines with USING and CHANGING Parameters

```
*&---------------------------------------------------------------------*
*&      Form  FILL_ITAB
*&---------------------------------------------------------------------*
*       Filling internal table with records of  sflight  with
*       carrid = p_carrid, calculate percentage
*----------------------------------------------------------------------*
*    -->iv_carrid     carrier id
*    <--ct_flightocc  internal standard table with line type sbc400focc
*----------------------------------------------------------------------*
FORM fill_itab USING    value(iv_carrid)    TYPE s_carr_id
               CHANGING value(ct_flightocc) TYPE sbc400_t_sbc400focc.


 ...

ENDFORM.                               " FILL_ITAB
```

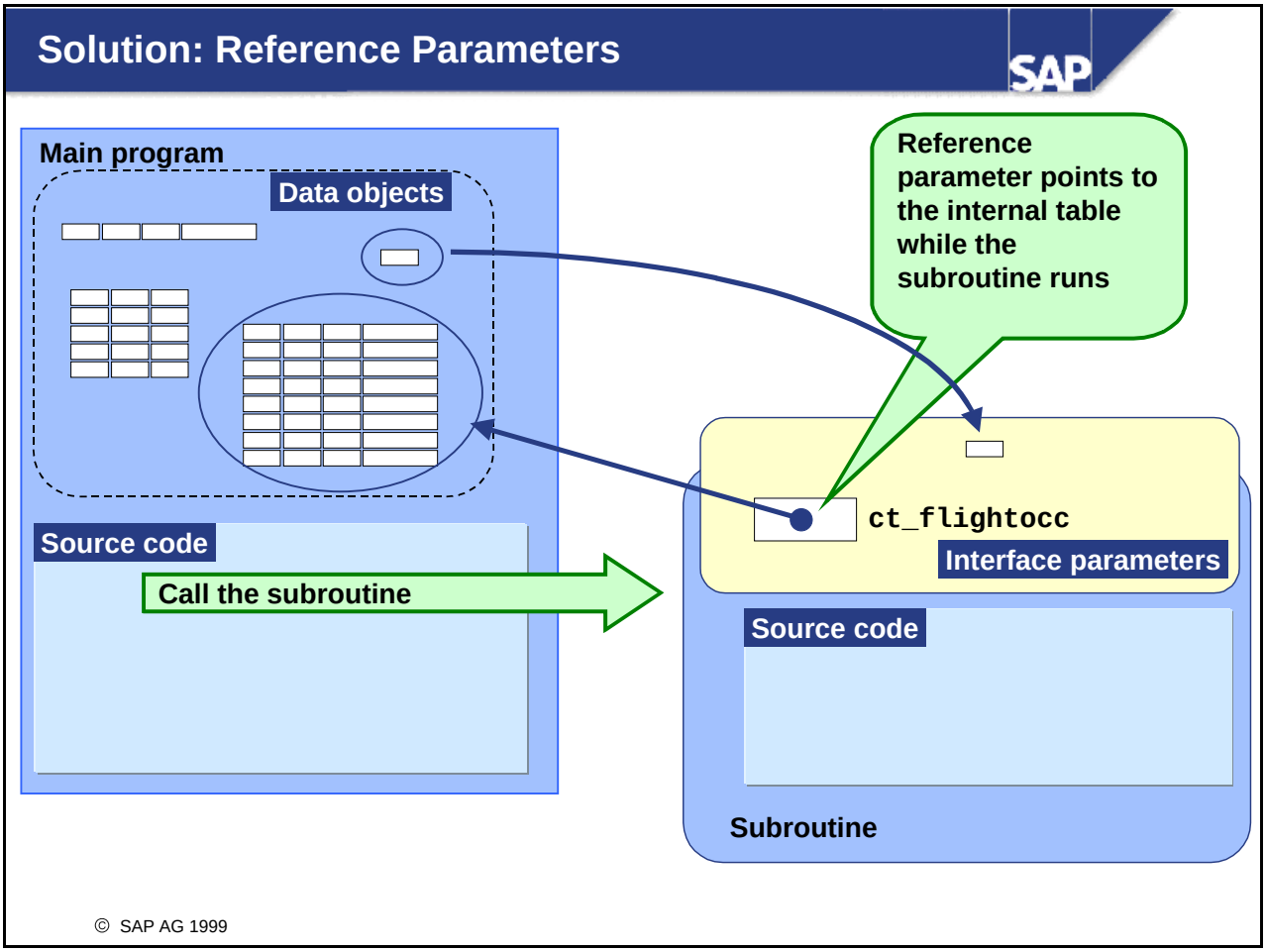


- When you define a subroutine, you list all the USING parameters after the USING addition. You then add the CHANGING addition, followed by all the CHANGING parameters. Enter the following for each parameter: The VALUE addition (if necessary), parameter name, and type attributes.
- **FORM <name of subroutine>**
  **USING <u-parameter 1> <...><u-parameter n>**
  **CHANGING <c-parameter 1> <...> <c-parameter m>.**
  **...**
  **ENDFORM.**
- Note: When specifying whether a parameter is only copied to the interface parameter when the program is called, or whether it is also copied back to the actual parameter after the subroutine has been executed, the main factor is the assignment to USING or CHANGING in the subroutine interface definition. You also specify the type attributes at this point.
- When you call a subroutine with **USING** and **CHANGING** parameters, the actual parameters should be listed appropriately after the **USING** and **CHANGING** additions.
  **PERFORM <name of subroutine>**
  **USING <actual parameter 1><...><actual parameter n>**
  **CHANGING <actual parameter n+1><...><actual parameter n+m>.**
  The assignment is always in strict order. For historical reasons, you can also omit the **CHANGING** addition when calling the subroutine.

- Literals and constants can only be assigned to **USING** parameters when the subroutine is called.

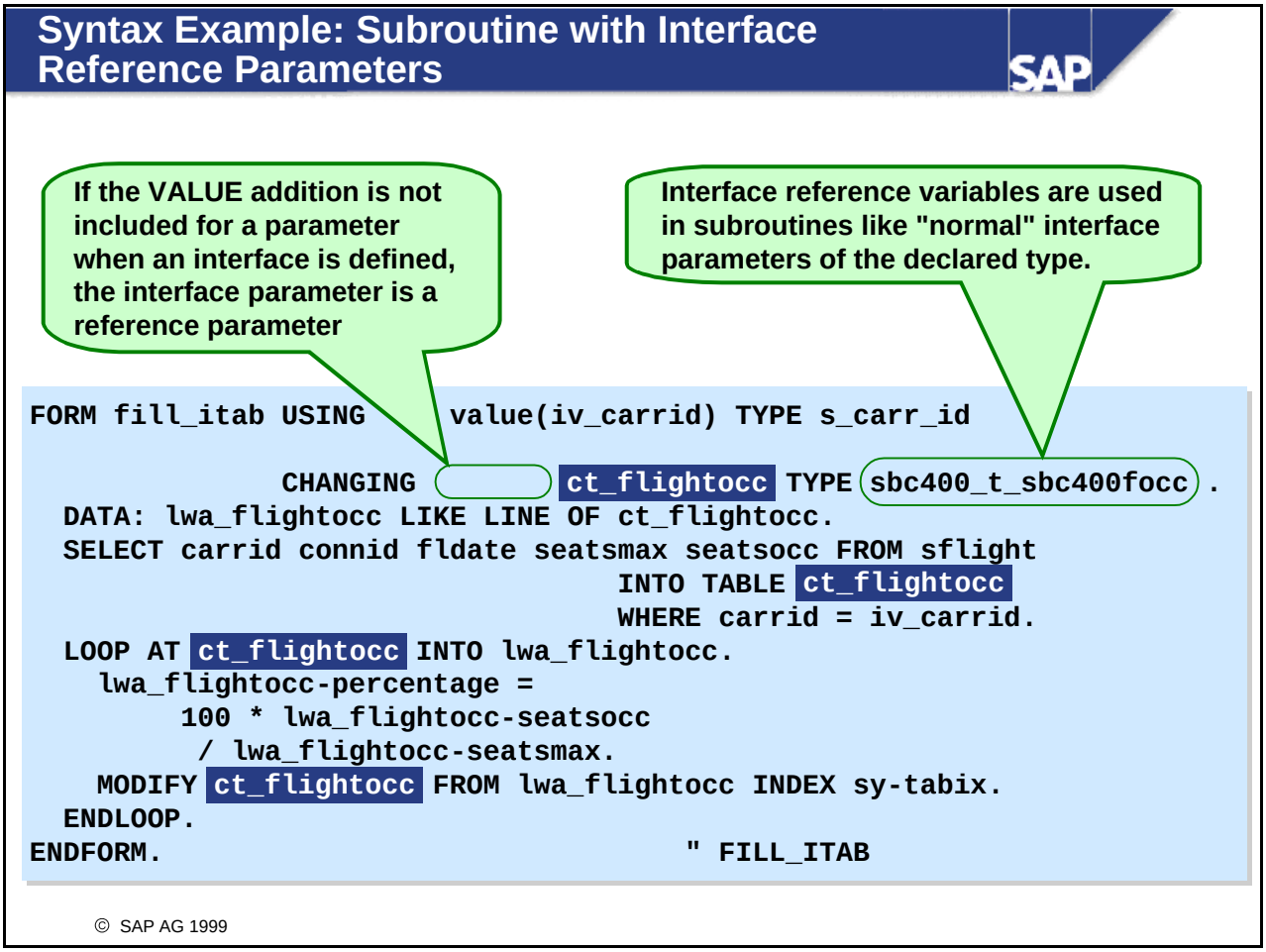



- Passing values during the copying process is very runtime-intensive for large data objects, particularly for large internal tables.



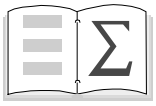

- Passing values during the copying process is very runtime-intensive for large data objects, particularly for large internal tables. For this reason, you can also pass references to global data objects to the subroutine, just as you can in other programming languages. Changes to the variable in the subroutine then cause the same change in the global data object.
- Within the subroutine, the variable is addressed using the interface parameter name while values are being passed.

## Syntax Example: Subroutine with Interface Reference Parameters




- You define a reference parameter in the interface, by omitting the **VALUE** addition.
- Reference parameters can be used as both **USING** and **CHANGING** parameters. Note that changes to reference parameters in the subroutine will also change the content of the data object in the main program. For this reason, **USING** reference parameters in subroutines must not be changed. Note that, if you do, the system will not return a syntax error, for compatibility reasons.

# Internal Program Modularization: Unit Summary



**You are now able to:**

- **Explain how a program containing event blocks functions at runtime using INITIALIZATION and START-OF-SELECTION as examples**
- **Encapsulate functions in a simple subroutine with interface**

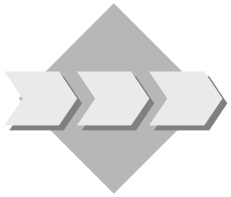
Exercises

**Unit: Internal Program Modularization**

**Topic: Subroutines**

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

- Create subroutines
- Use the subroutine interface to pass data



Change your program **ZBC400_##_SELECT_SFLIGHT_ITAB** (or the corresponding model solution) so that both the authorization check and the data output are encapsulated in subroutines.

**Program:** **ZBC400_##_FORMS**

**Model solution:** **SAPBC400PBS_FORMS**

1-1 Copy your program **ZBC400_##_SELECT_SFLIGHT_ITAB** or the corresponding model solution **SAPBC400DDS_AUTHORITY_CHECK_2** to the new program **ZBC400_##_FORMS**. Assign your program to **development class ZBC400_##** and the change request for your project “BC400…”. (where ## is your group number.)

1-2 Encapsulate the authorization check in a subroutine. Pass the airline code and the value required for the authorization field **ACTVT** in the interface. Pass **SY-SUBRC**, which is set by the authorization check, back to the main program via the interface. Specify types for the interface parameters of the subroutine. Possible ABAP Dictionary types are:

- Airline code: Data element **S_CARR_ID**
- Return value: System field **SY-SUBRC**
- Value of the authorization field **ACTVT**: Data element **ACTIV_AUTH**

1-3 Change the parts of the program that depend on the result of the authorization check: You can no longer query the value of **SY-SUBRC**. Instead, find out the value of the corresponding interface parameter from the subroutine.

1-4 Encapsulate the data output in a subroutine. Call the subroutine after the **SELECT** loop. Pass the internal table containing the read data using the interface. Specify the types of the

interface parameters. Display the data from the subroutine using a **LOOP… ENDLOOP** structure. To do this, create the required table work area as a local data object in the subroutine. To specify the type of the local structure, use the ABAP statement **DATA: <WA > LIKE LINE OF <ITAB>**.

**Unit: Internal Program Modularization**

**Topic: Subroutines**

**Model solution SAPBC400PBS_FORMS**

```
*&---------------------------------------------------------------------*
*& Report  SAPBC400PBS_FORMS                                           *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  sapbc400pbs_forms.
CONSTANTS actvt_display TYPE activ_auth VALUE '03'.
DATA: wa_flight TYPE sbc400focc,
      it_flight TYPE sbc400_t_sbc400focc.
PARAMETERS: pa_car TYPE sflight-carrid.
DATA:       returncode LIKE sy-subrc.


START-OF-SELECTION.
* Authority-Check:
  PERFORM authority_scarrid USING pa_car actvt_display
                            CHANGING returncode.
  CASE returncode.
* User is authorized
    WHEN 0.
      SELECT carrid connid fldate seatsmax seatsocc FROM sflight
           INTO CORRESPONDING FIELDS OF wa_flight
           WHERE carrid = pa_car.
        wa_flight-percentage =
        100 * wa_flight-seatsocc / wa_flight-seatsmax.
        APPEND wa_flight TO it_flight.
      ENDSELECT.
      PERFORM write_list USING it_flight.
```

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

```
*&---------------------------------------------------------------------*
*&      Form  AUTHORITY_SCARRID
*&---------------------------------------------------------------------*
*       text
*----------------------------------------------------------------------*
*      -->P_PA_CAR  text
*      -->P_LD_ACTVT  text
*      <--P_RETURNCODE  text
*----------------------------------------------------------------------*
FORM authority_scarrid USING    value(p_carrid)   TYPE s_carr_id
                        value(p_ld_actvt) TYPE activ_auth
                  CHANGING p_return         LIKE sy-subrc.
  AUTHORITY-CHECK OBJECT 'S_CARRID'
     ID 'CARRID' FIELD p_carrid
     ID 'ACTVT'  FIELD p_ld_actvt.
  p_return = sy-subrc.
ENDFORM.                    " AUTHORITY_SCARRID


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

# User Dialogs: Lists

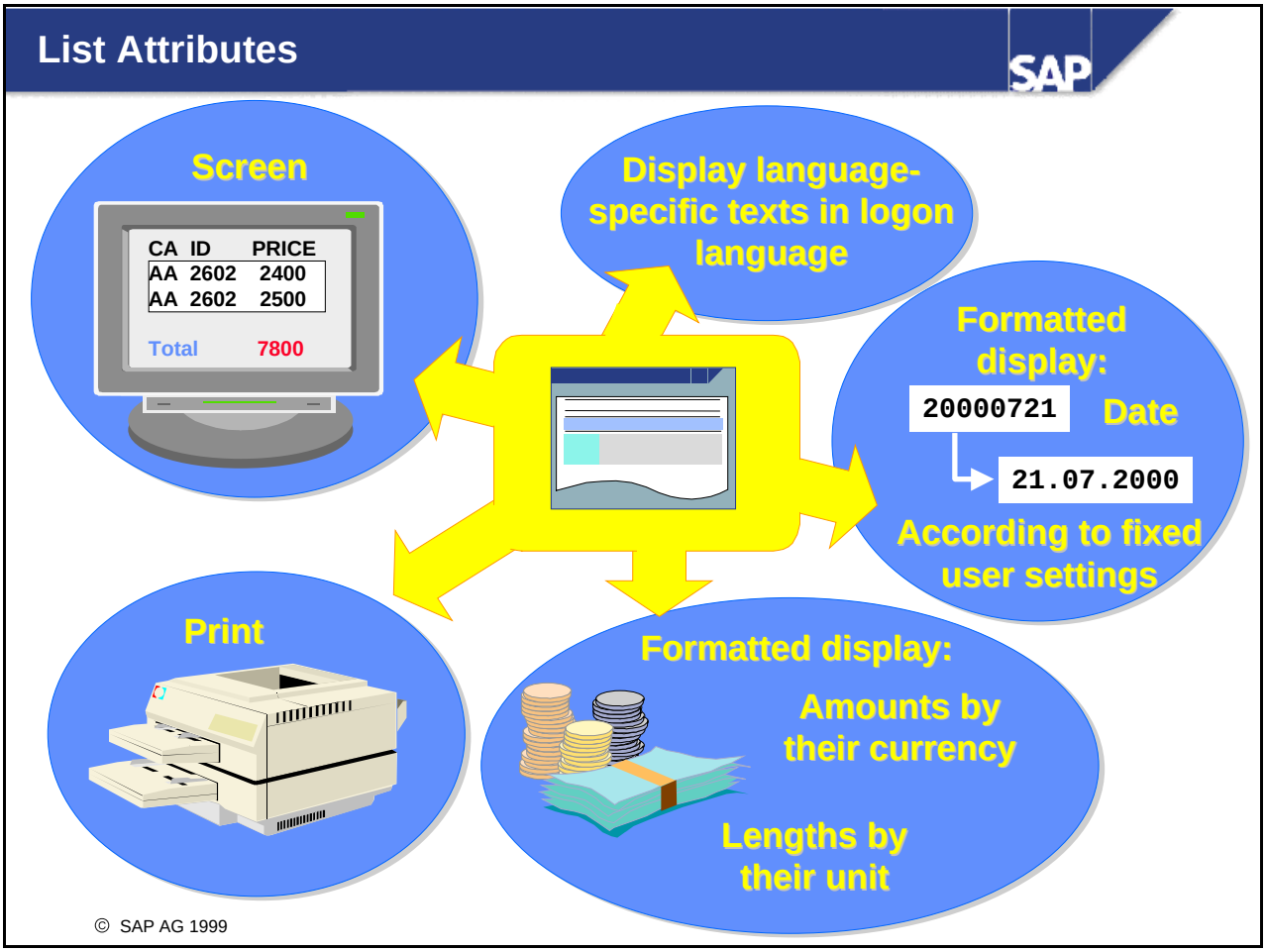
## Contents:

- List attributes and strengths
- Basic lists
- List events
- Interactive lists
- Example with syntax: detail lists

# User Dialogs: Lists: Unit Objectives

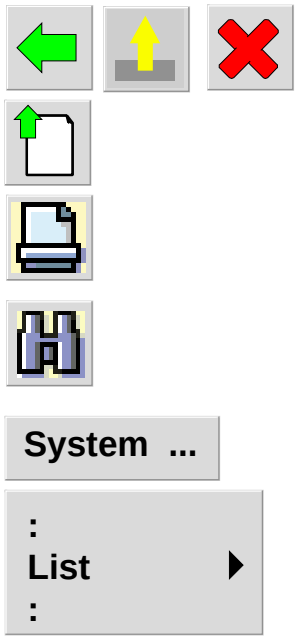
**At the conclusion of this unit, you will be able to:**

- **Describe list attributes and strengths,**
- **Write a program that displays the details of a specific line from your basic list to an interactive list whenever the user double clicks on that particular line, and**
- **Explain the runtime behavior of your program during the AT LINE-SELECTION event**




- The main purpose of a list is to output data in a manner that can be easily understood by the user; this output often takes the form of a table. Lists in R/3 take into account special business data requirements:
  - They are language-independent. Texts and headers appear in the logon language whenever the appropriate translation is available.
  - They can output monetary values in numerous currencies.
  - You can output list data in the following ways:
    - To the screen; here you can add colors and icons
    - To the printer
    - To the Internet/intranet: automatic conversion to HTML
  - you can also save lists in the R/3 System or output them for processing by external commercial software applications like spreadsheet programs

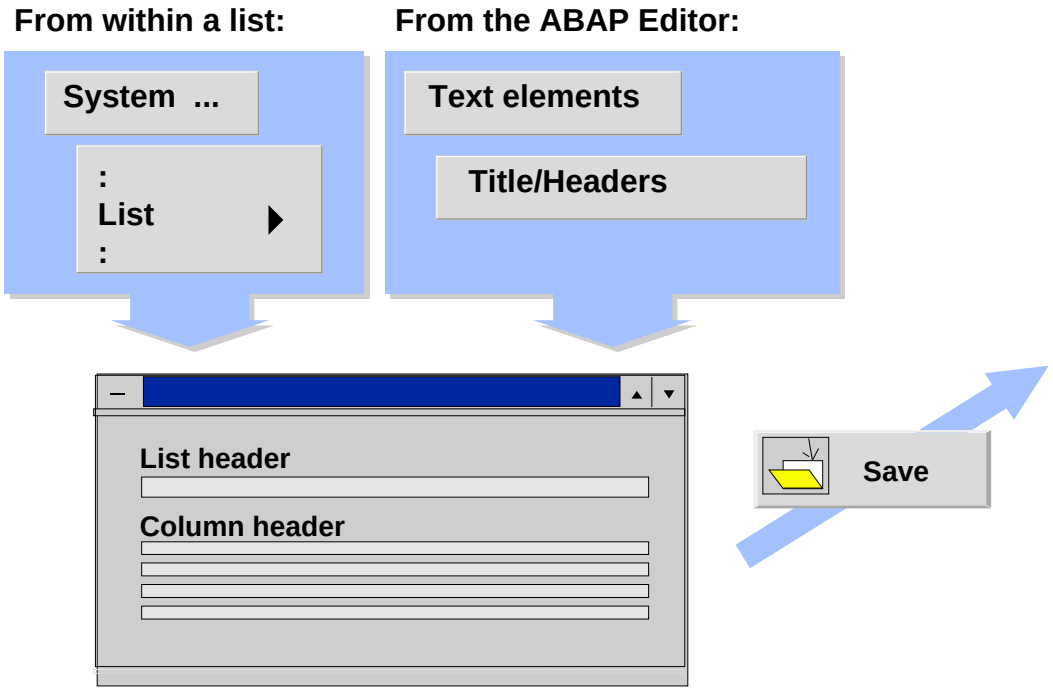
## Standard List Functions






- The standard list interface offers the user several navigation features:
  - Back
  - Exit
  - Cancel
  - Print
  - Find (in list)
  - Save: saves the list either as a file, in a report tree, or to the buffer
  - Send: sends the list in e-mail form
- For further information on how you can adjust the standard list interface to fit your individual needs see the relevant section in the *Dialogs: Interfaces* unit.

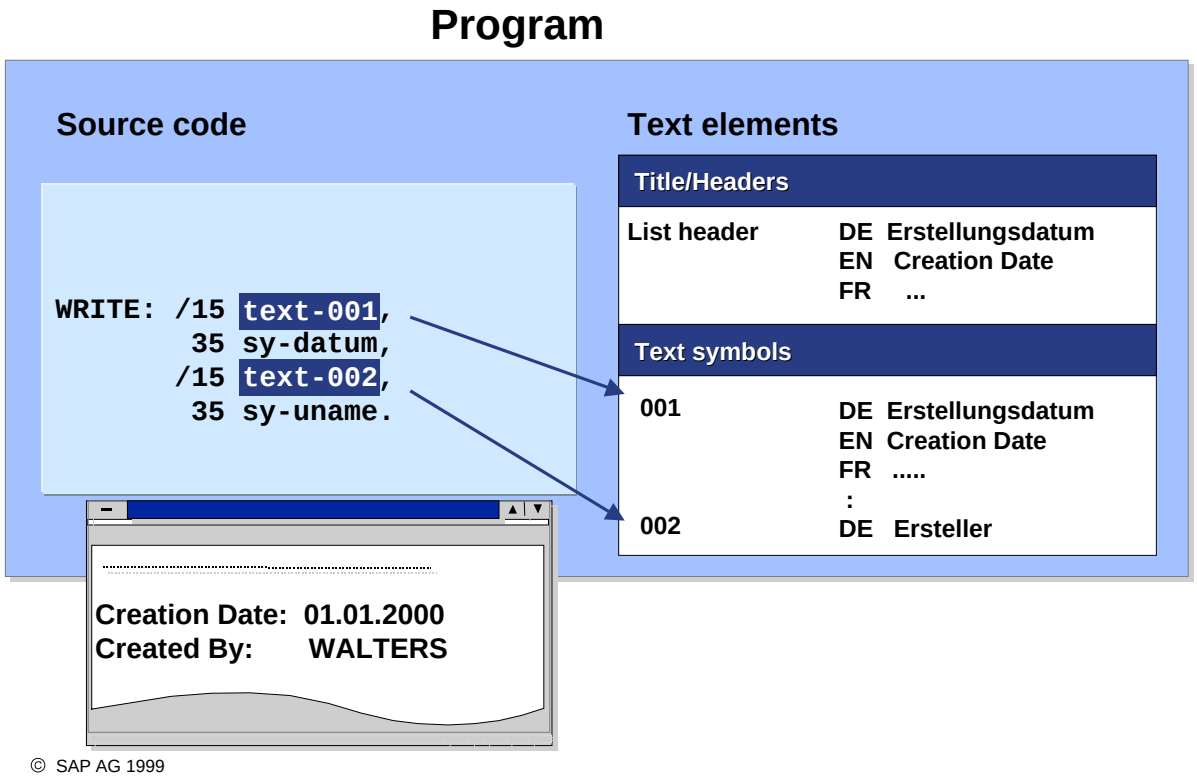
## Column Header in the Default Page Header






- Each list can have a **list header** and up to four lines of **column headers**. You can create these elements in two different ways:
  - From within the Editor using the text element maintenance functions
  - From within the list itself. If you save your **program**, **activate** it and then run it to create the list, you can enter both list and column headers by choosing the menu path *System -> List -> List Header -> Maintain Lists and Column Headers*. The main advantage of using this method is that the list is still displayed on the screen. This makes it easier to position column headers.
- The next time you start the program, the new headers will appear in the list automatically.
- When no header text is entered, the program title is inserted in the header.

# Multilingual Capability



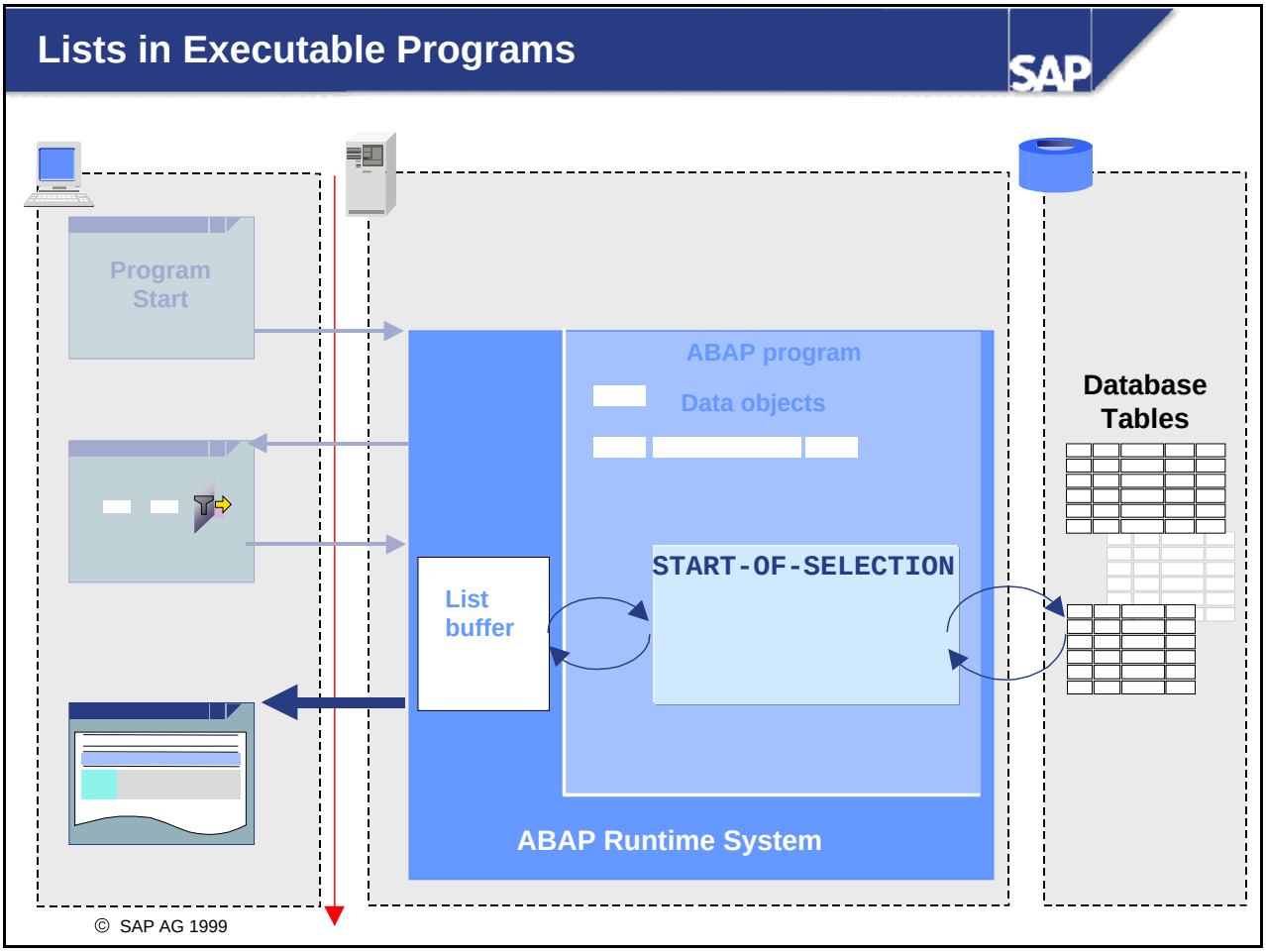



- **Titles and headers** are part of a program's text elements. You can translate all text elements into other languages. The logon language setting on the logon screen determines the language in which text elements will be displayed.
- **Text symbols** are another kind of text element. Text symbols are text literals that can be translated and which are assigned to the program. Text symbols allow you to create lists independent of language.
- You can write text symbols into your program in either of the following ways:

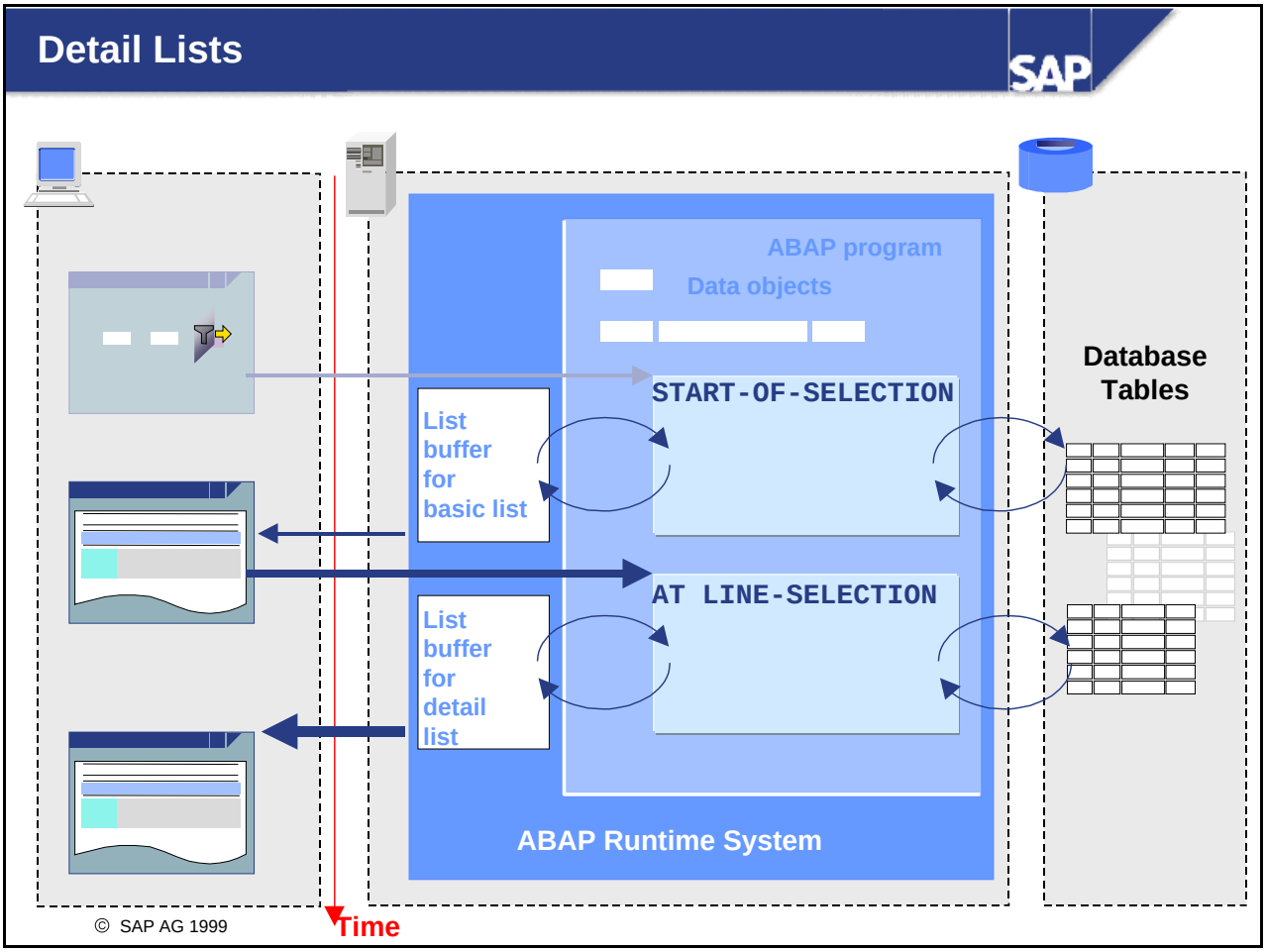
  - TEXT-<xxx> (where xxx is a three-character sequence)

  - '<Text>'(<xxx>) (where xxx is a three-character sequence).

- You can display texts in the Editor by choosing *Goto -> Text elements-> Text symbols* or by double-clicking the number of a text symbol.

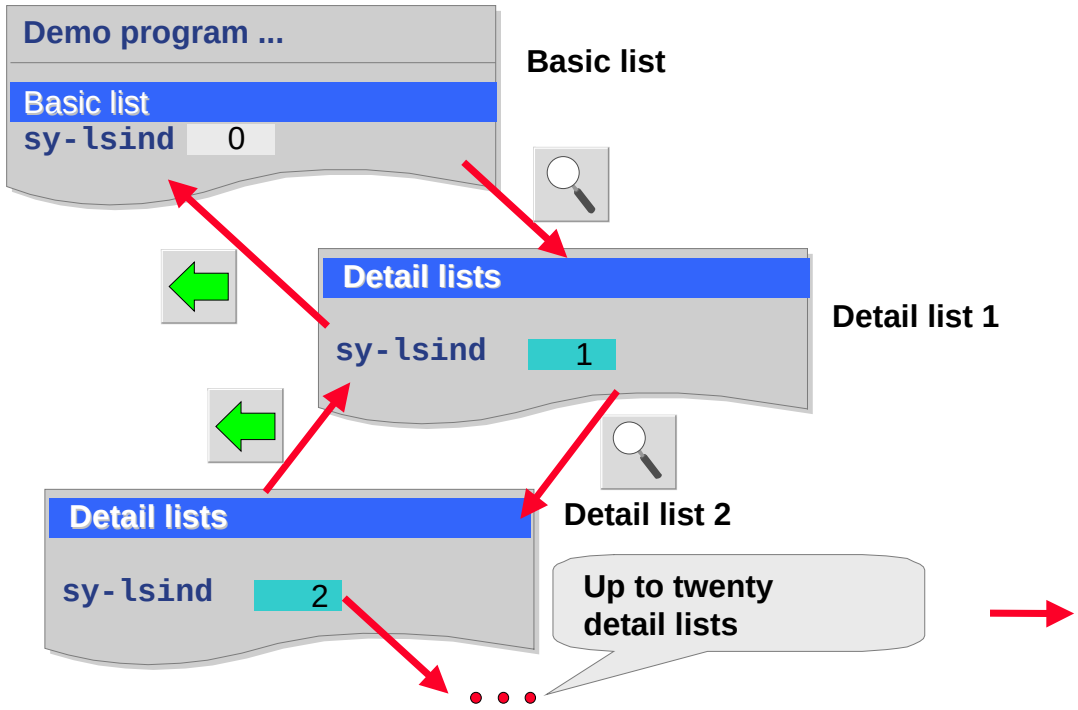



- In executable programs (type 1), lists are automatically displayed after their corresponding event blocks have been processed. These processing blocks must, however, contain a statement that writes to the list buffer. These statements include **WRITE, SKIP**, and **ULINE**.
- Event blocks are called in a sequence designed for list processing:
  - Prior to sending the selection screen: **INITIALIZATION**
  - After leaving the selection screen: **START-OF-SELECTION**
- All output from **START-OF-SELECTION** event blocks, subroutines, and function modules that is processed before a list is displayed is temporarily stored in the list buffer.
- Once all list creation processing blocks (for example **START-OF-SELECTION**) have been processed, all data from the list buffer is output in the form of a list.




- In executable programs, you can use the event block **AT LINE-SELECTION** to create details lists.
- The ABAP runtime system:
  - Displays basic lists after the appropriate event blocks have been processed (for example, after **START-OF-SELECTION**). In this case, system field **sy-lsind** contains the value 0.
  - Processes the event block **AT LINE-SELECTION** using the function code 'PICK' each time you double-click on an entry or choose an action for the system to perform. If you are using a standard status, this happens automatically every time you choose an icon, a menu function, or the function key F2.
  - Displays details lists after the **AT LINE-SELECTION** event block has been processed and increases the value contained in **sy-lsind** by one.
  - Displays the details list from the previous level in the list hierarchy (n-1) every time you choose the green arrow icon from the current details list (n).

## Example: A Simple Detail List






- The lists in the example program should behave as follows:
  - The basic list should display the text 'Basic List' and system field sy-lsind.
  - The user can navigate to a detail list by using any of the following:
    - Double-click
    - The *Detail list* icon (a magnifying glass) in the pushbutton bar
    - A menu function
    - The F2 function key
      Then the *Detail list* appears and the system field **sy-lsind** has the value 1.
  - Repeating this action should call the second details list, where system field **sy-lsind** contains the value 2 instead (representing the current details list level).
  - sy-lsind is incremented by 1 each time the user repeats this action (until sy-lsind reaches 20).
  - Choosing the green arrow takes the user back a single detail list level at a time until he or she reaches the basic list .

## Syntax: A Simple Detail List

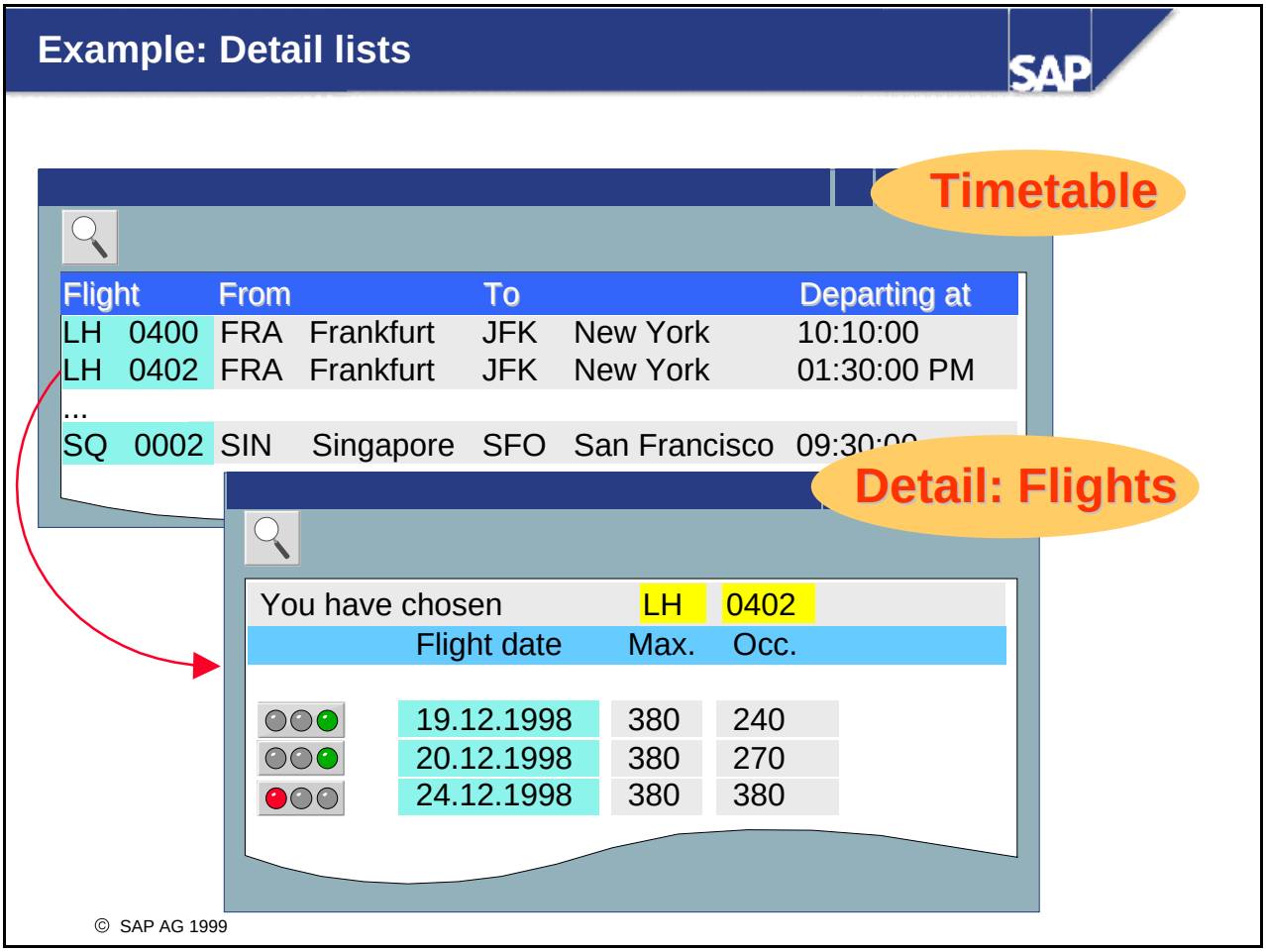
```
REPORT sapbc400udd_secondary_list_a.

START-OF-SELECTION.
   WRITE: / text-001 COLOR col_heading,
          / 'sy-lsind',
             sy-lsind color 2.


AT LINE-SELECTION.
   WRITE: / text-002 COLOR col_heading.
   ULINE.
   WRITE: / 'sy-lsind',
             sy-lsind color 4.
```

**Text symbols:**

| | |
|---|---|
| 001 | Basic list |
| 002 | Detail Lists |

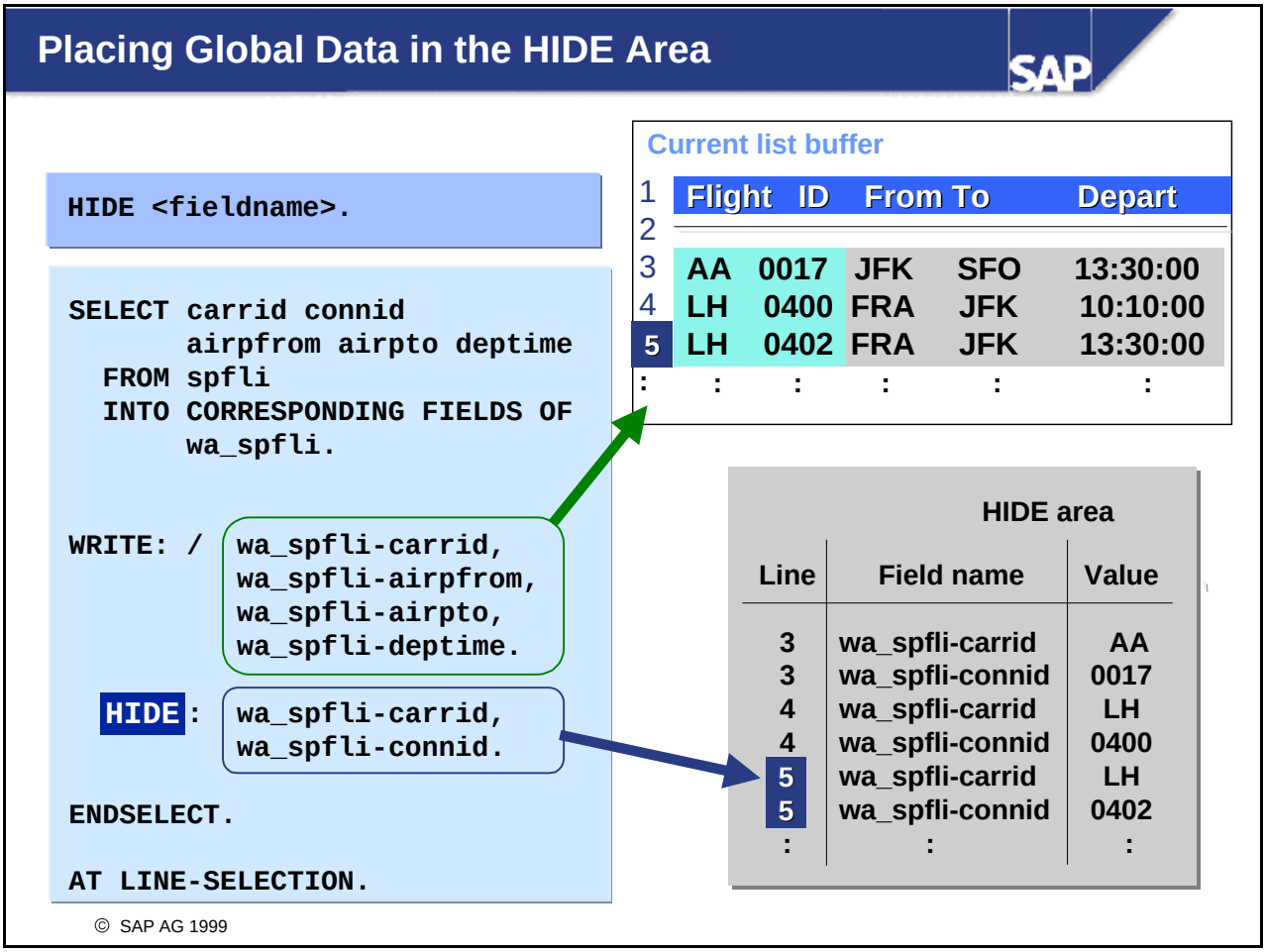


- A details list can be programmed as follows:
  - You create a basic list by filling the basic list buffer at an appropriate event block (here **START-OF-SELECTION**) using either **WRITE, SKIP**, or **ULINE**.
  - Use the event block **AT LINE-SELECTION** when programming details lists. Whenever you use **WRITE, SKIP**, or **ULINE** with this event block, you fill the details list buffer for the next level (the details list buffer with a level value one greater than the level on which the user performed his or her action).
  - You can manage the different levels of the detail list by querying the **sy-lsind** field in the **AT LINE-SELECTION** event block.



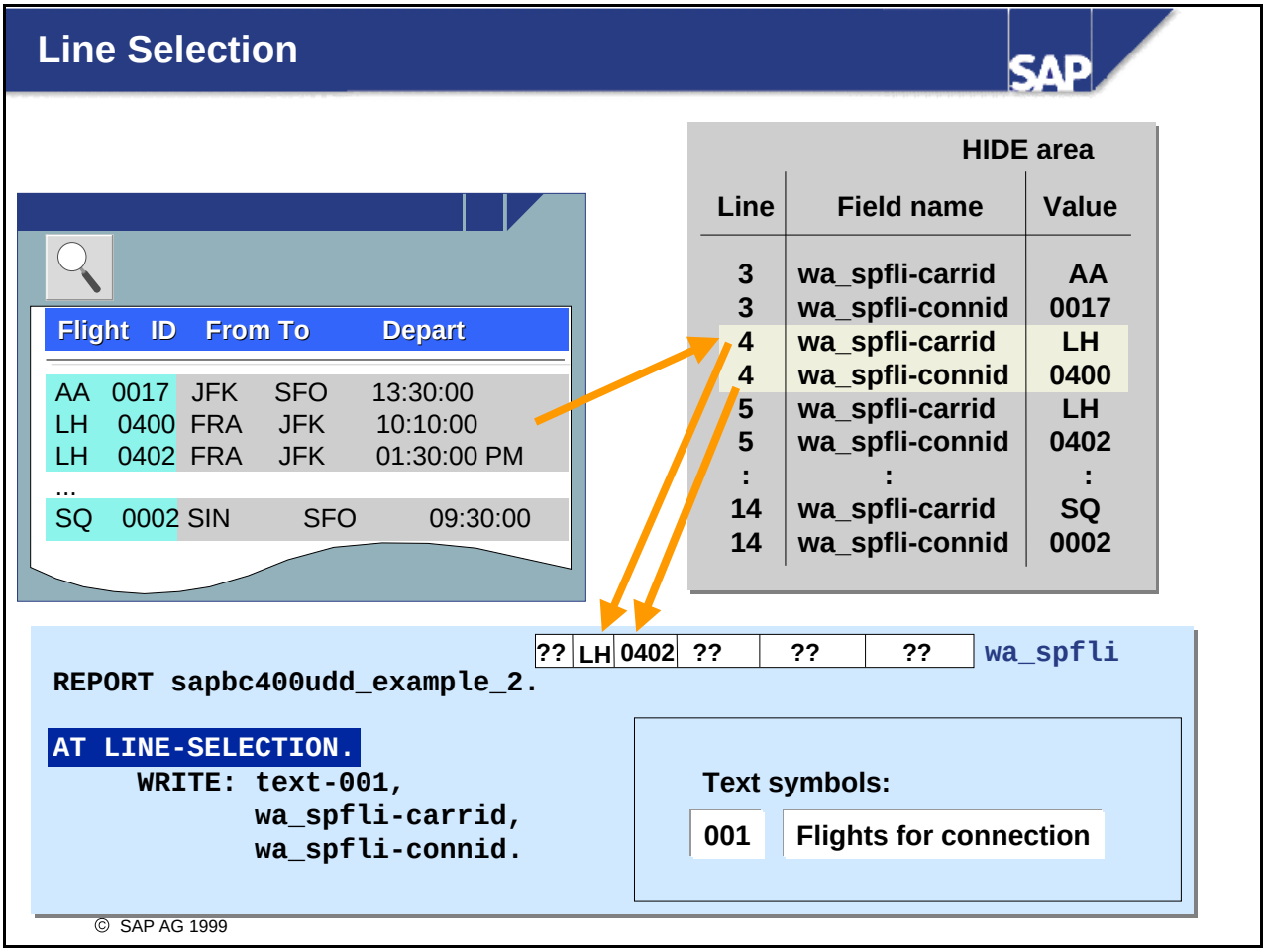

- We will now write a program using both basic lists and details lists:
- The basic list in your program should contain flight data such as carrier ID and flight numbers, departure city and airport, destination city and airport, as well as departure and arrival times. This data is stored in the database table **SPFLI**.
- The user should be able to access information about any particular flight by double-clicking its carrier ID and flight number. Flight date and occupancy should be displayed. This data is stored in the database table **SFLIGHT**. You must use the **SPFLI** key fields in this details list in order to read the appropriate data in **SFLIGHT**. The following slides demonstrate how this is done.
- The sample program is named **SAPBC400UDD_DETAIL_LIST** and is part of development class **BC400**.

## Placing Global Data in the HIDE Area






- When the event **AT LINE-SELECTION** is processed, a program's data objects contain the same values as before basic list display. A detail list, however, often needs data selected within the basic list itself. You can use the HIDE area to store certain data from the line that you have selected and then automatically insert where you need it in the corresponding data object for a details list.
  You can specify in advance which information should be classified by its line position, when you are creating the basic list.
- To do this, you use the ABAP keyword **HIDE** followed by a list of the data objects that you need. The runtime system automatically records the name and contents of the data object in relation to its line position in the list currently being created.
- The **HIDE global_field** statement stores the content of the global data field **global_field** with reference to the current display line.
- When this line is selected, the data field is filled automatically with the stored values.
- You do not have display the field beforehand, with a **WRITE** statement.
- The data field can also be a structure. However, the HIDE area does not support deep structures - that is, structures whose components include tables.




- As soon as interactive event (AT LINE-SELECTION in this example) is called by placing the cursor on a specific line and then either double-clicking or choosing a pushbutton, the values for this line stored in the HIDE area are inserted into their corresponding data objects.

## Line Selection: Syntax

```
REPORT sapbc400udd_example_2.
...

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

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

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

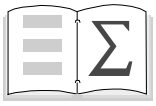
**Text symbols:**

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



- You create a details list by filling the details list buffer at the **AT LINE-SELECTION** event block using either **WRITE**, **SKIP**, or **ULINE**. In this sample program, the key fields for the airline are displayed and the flights available for this airline in the database table **SFLIGHT** are read using a **SELECT** loop. Note that the line-specific information on the airline is only available by double-clicking in the data objects if the relevant data objects have been placed in the **HIDE** area when the basic list was created.

# User Dialogs: Lists: Unit Summary



**You are now able to:**

- **Describe list attributes and strengths,**
- **Write a program that displays the details of a specific line from your basic list to an interactive list whenever the user double clicks on that particular line, and**
- **Explain the runtime behavior of your program during the AT LINE-SELECTION event**

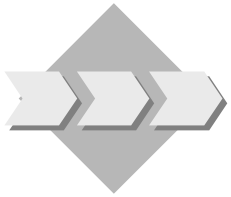
Exercises

**Unit: User Dialogs: Lists**

**Topic: Detail Lists**

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

- Create a detail list in a program



Extend your program **ZBC400_##_SELECT_SFLIGHT** or the corresponding model solution as follows:
Once the user has selected a flight on the basic list (double-click or F2 on the relevant list line), display a detail list containing all of the bookings for the selected flight.

**Program:** **ZBC400_##_DETAIL_LIST**

**Model solution:** **SAPBC400UDS_DETAIL_LIST**

1-1 Copy your program **ZBC400_##_SELECT_SFLIGHT** or the corresponding model solution **SAPBC400DDS_AUTHORITY_CHECK** to the new program **ZBC400_##_DETAIL_LIST**. Assign your program to **development class ZBC400_##** and the change request for your project “BC400…” (## is your group number).

1-2 Make sure that the key fields of the database table **SFLIGHT** are available to you for building up the detail list when the user selects a flight from the basic list (double-click or F2 on the corresponding list line).

1-3 Add the **AT LINE-SELECTION** event to your program to allow you to construct a detail list.

1-4 In the first line of the detail list, display key information from the selected line of the basic list. Under this line, display a horizontal line and a blank line.

1-5 Read all of the bookings from database table **SBOOK** for the selected flight. Make sure that you only read fields from the database table that you want to display in the list. to display the following fields of the database table **SBOOK** on the detail list:

**BOOKID,**
**CUSTOMID,**
**CUSTTYPE**

**CLASS,**
**ORDER_DATE,**
**SMOKER,**
**CANCELLED.**

1-6 **Optional:** Display the fields **LOCCURAM** and **LOCCURKEY** on the detail list: Ensure that the currency amount **LOCCURAM** is displayed with the appropriate formatting for the currency **LOCCURKEY**. Use the addition **CURRENCY <currency_key>** in the **WRITE** statement.

**Example**:
```
WRITE:  wa_sflight-price CURRENCY wa_sflight-currency,
        wa_sflight-currency.
```

1-7 **Optional:** Display the **BOOKID** field in the color **COL_KEY**.

**Unit: User Dialogs – Lists**

**Topic: Detail Lists**

**Model solution without optional exercises: SAPBC400UDS_DETAIL_LIST**

```
*&---------------------------------------------------------------------*
*& Report  SAPBC400UDS_DETAIL_LIST                                     *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  sapbc400uds_detail_list.

CONSTANTS: actvt_display TYPE activ_auth VALUE '03'.

DATA: wa_flight TYPE sbc400focc,
      wa_sbook  TYPE sbook.

PARAMETERS: pa_car TYPE s_carr_id.


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

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

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

* Key fields transported back from hide area to ABAP dataobjects
   WRITE: / wa_flight-carrid,
            wa_flight-connid,
            wa_flight-fldate.
   ULINE.
   SKIP.
* Selection of bookings, which depend on selected flight
   SELECT bookid customid custtype class order_date
          smoker cancelled loccuram loccurkey
          FROM sbook INTO CORRESPONDING FIELDS OF wa_sbook
          WHERE carrid = wa_flight-carrid
           AND  connid = wa_flight-connid
           AND  fldate = wa_flight-fldate.
* Creation of detail list
    WRITE: / wa_sbook-bookid,
             wa_sbook-customid,
             wa_sbook-custtype,
             wa_sbook-class,
             wa_sbook-order_date,
             wa_sbook-smoker,
             wa_sbook-cancelled.
   ENDSELECT.
```

```
ENDIF.
CLEAR wa_flight.
```

**Model solution with optional exercises: SAPBC400UDS_DETAIL_LIST1**

```
*&---------------------------------------------------------------------*
*& Report  SAPBC400UDS_DETAIL_LIST1                                    *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  sapbc400uds_detail_list1.

CONSTANTS: actvt_display TYPE activ_auth VALUE '03'.

DATA: wa_flight TYPE sbc400focc,
      wa_sbook  TYPE sbook.

PARAMETERS: pa_car TYPE s_carr_id.



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

        WRITE: / wa_flight-carrid,
                 wa_flight-connid,
                 wa_flight-fldate,
                 wa_flight-seatsocc,
                 wa_flight-seatsmax,
                 wa_flight-percentage,'%'.
* Hide key field values corresponding to the actual line
        HIDE: wa_flight-carrid, wa_flight-connid, wa_flight-fldate.
```

```
    ENDSELECT.
  WHEN OTHERS.
    WRITE: / 'Authority-Check Error'(001).
ENDCASE.
CLEAR wa_flight.

* Program continues here, if a line is selected on basic list
AT LINE-SELECTION.
 IF sy-lsind = 1.
* Key fields transported back from hide area to ABAP dataobjects
   WRITE: / wa_flight-carrid, wa_flight-connid, wa_flight-fldate.
   ULINE.
   SKIP.
* Selection of bookings, which depend on selected flight
   SELECT bookid customid custtype class order_date
          smoker cancelled loccuram loccurkey
          FROM sbook INTO CORRESPONDING FIELDS OF wa_sbook
          WHERE carrid = wa_flight-carrid
          AND   connid = wa_flight-connid
          AND   fldate = wa_flight-fldate.
* Creation of detail list
    WRITE: / wa_sbook-bookid COLOR COL_KEY,
             wa_sbook-customid,
             wa_sbook-custtype,
             wa_sbook-class,
             wa_sbook-order_date,
             wa_sbook-smoker,
             wa_sbook-cancelled,
             wa_sbook-loccuram CURRENCY wa_sbook-loccurkey,
             wa_sbook-loccurkey.
  ENDSELECT.
 ENDIF.
 CLEAR wa_flight.
```

# User Dialogs: Selection Screens

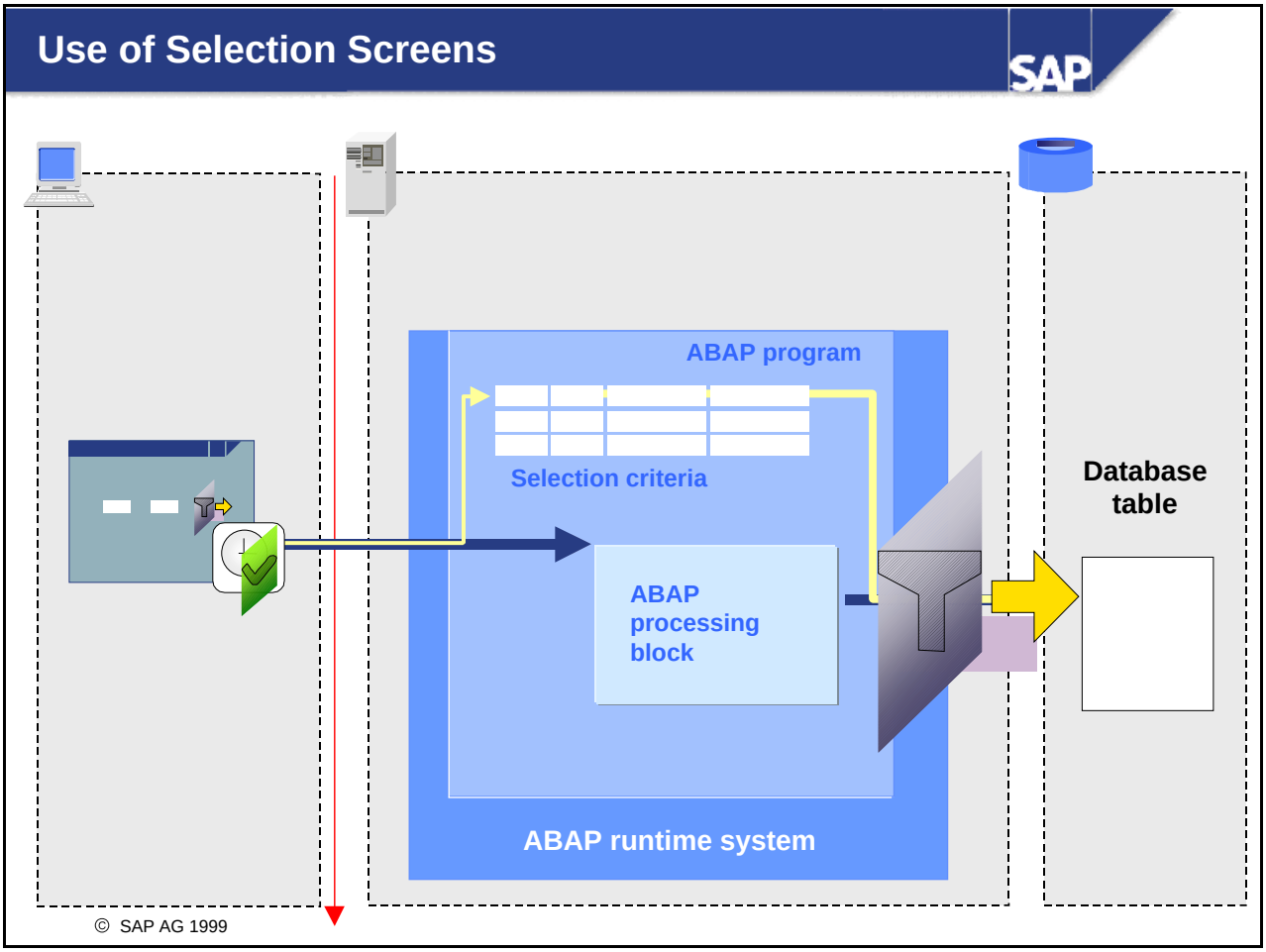
## Contents:

- Selection screen attributes and strengths
- Defining selection screens
- Evaluating user input to restrict database selection
- Selection screen events
- Example with syntax: additional input checks with error dialog

# Selection Screens: Unit Objectives

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

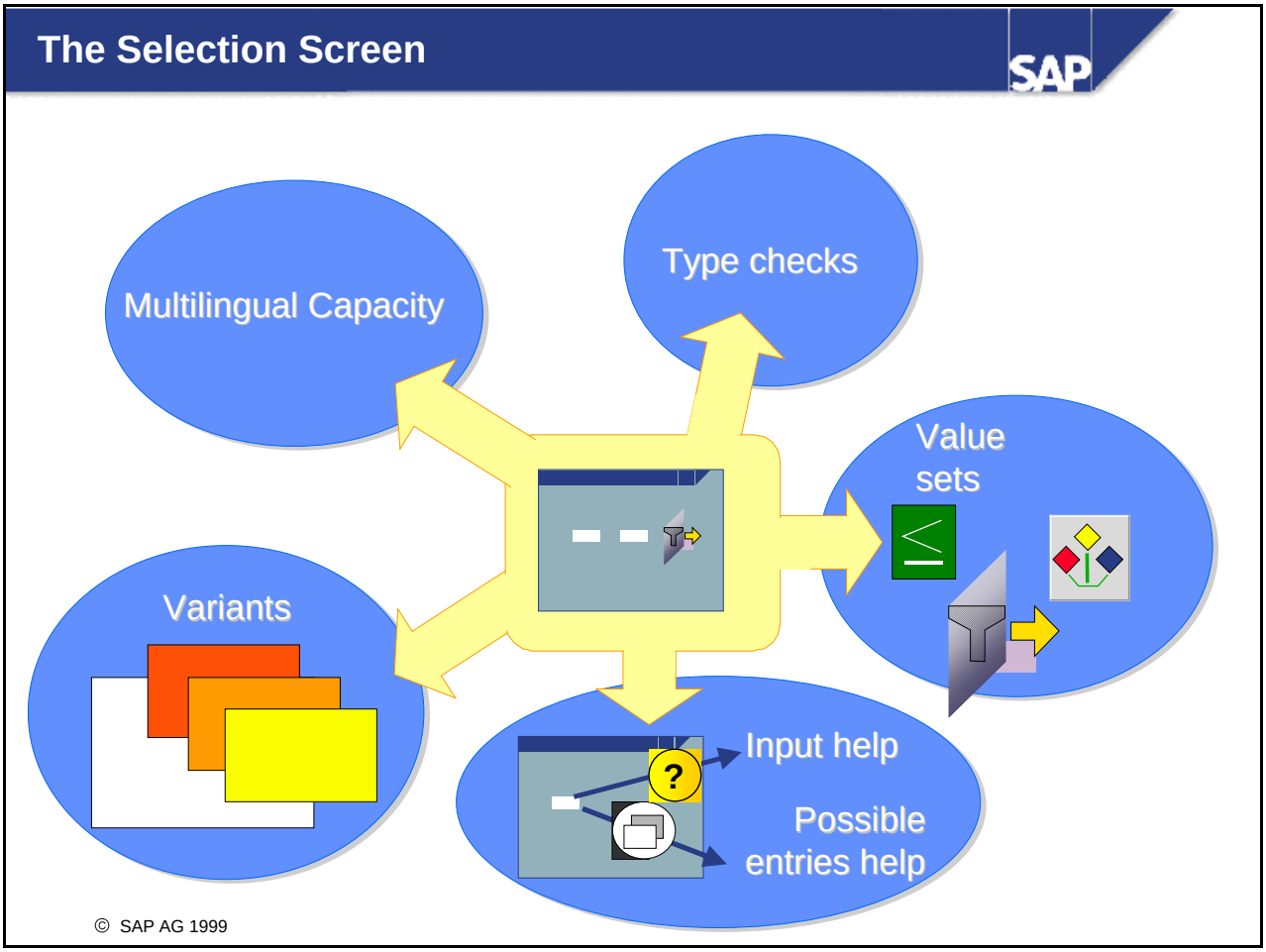
- **Describe selection screen attributes and strengths**
- **Write a program that allows the user to input intervals on a selection screen that can be used to restrict the number of data records retrieved from the database**
- **Write a program containing additional input checks on its selection screen which re-send the selection screen when an error occurs**



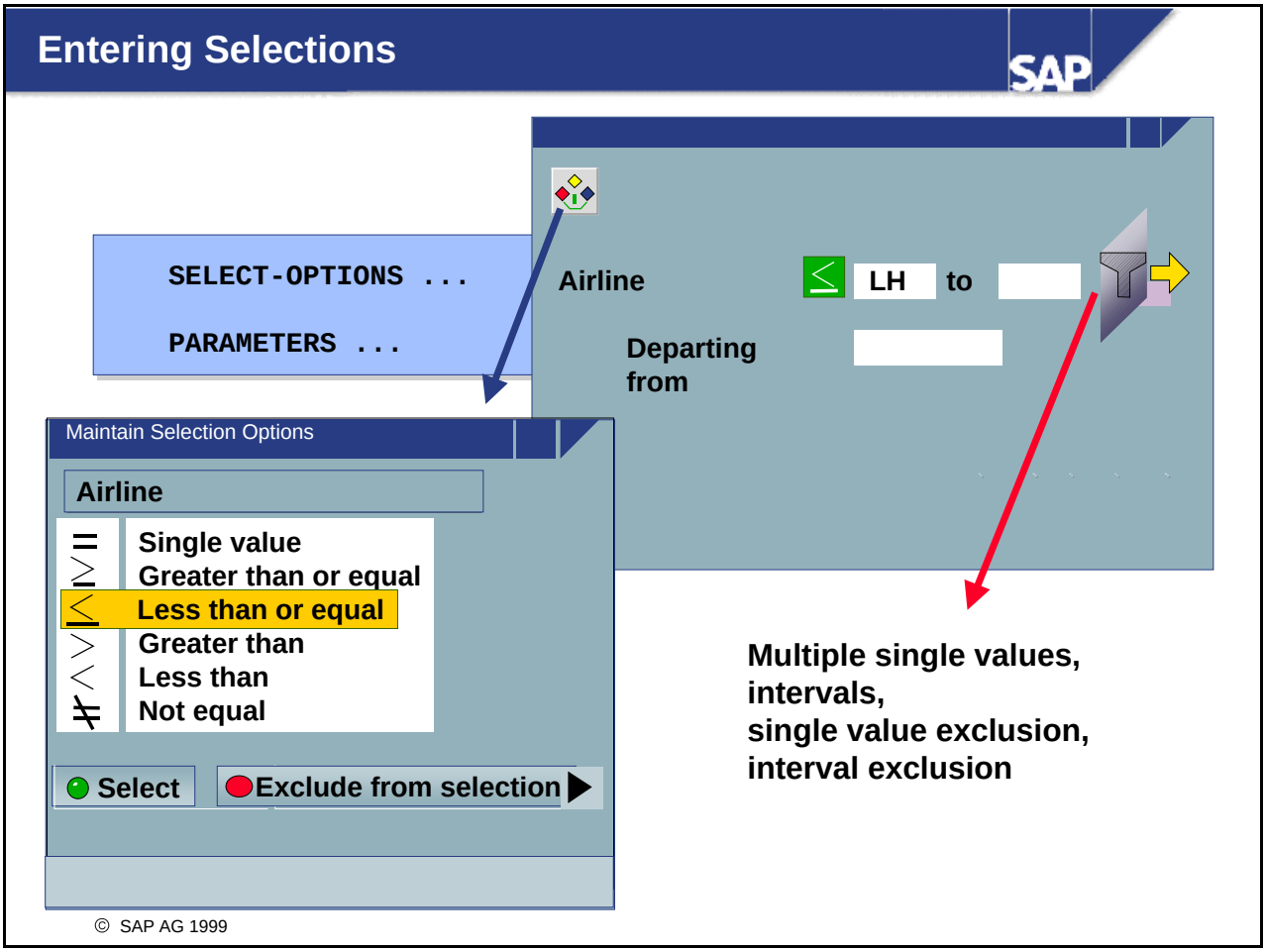

- Selection screens allow users to enter selection criteria required by the program.
- For example, if you create a list containing data from a very large database table, you can use a selection screen to restrict the amount of that data that is selected. At runtime, the user can enter intervals for one of the key fields, and only data in this interval is read from the database and displayed in the list. This considerably reduces the load on the network.

# Screen Attributes

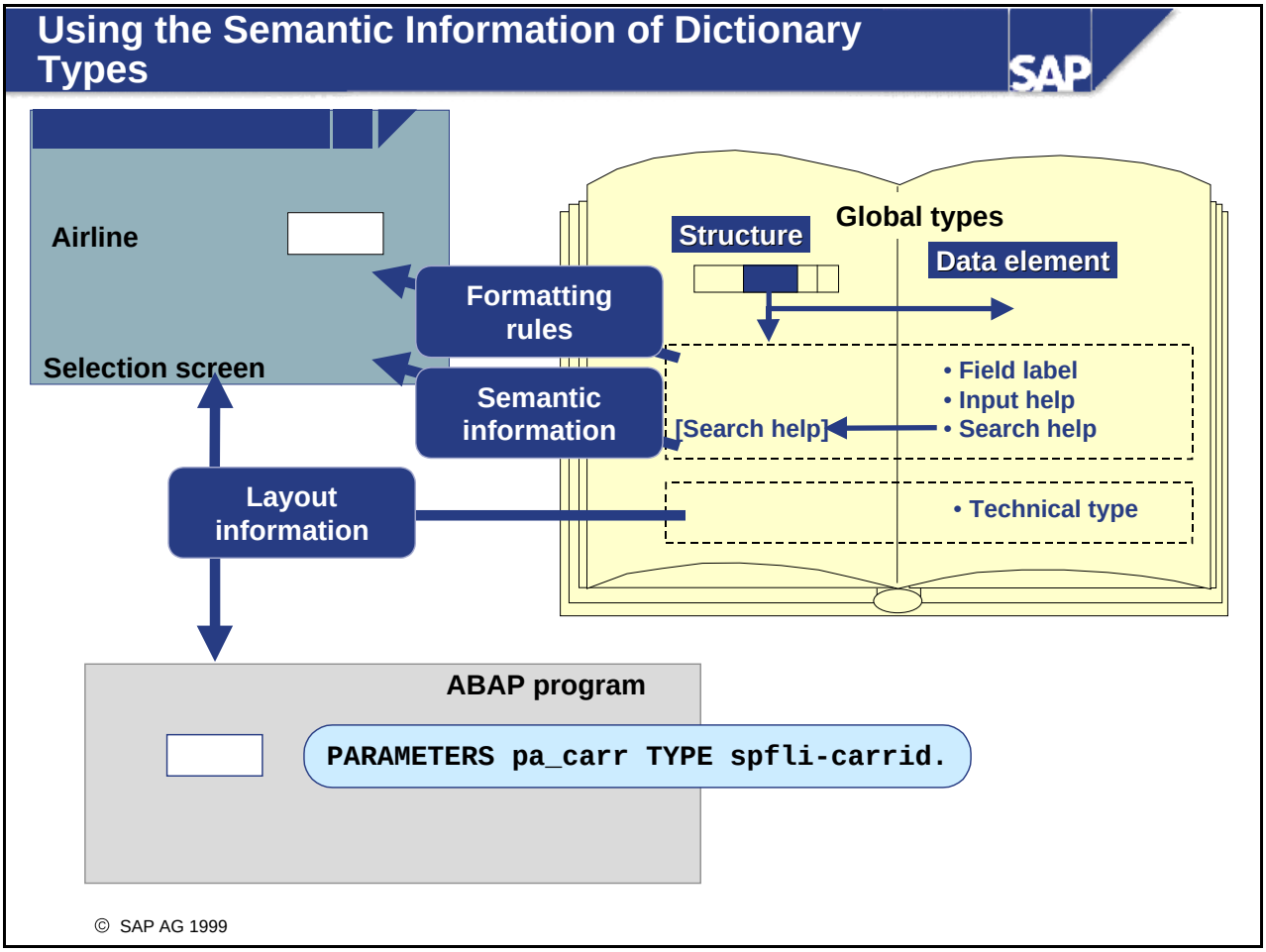
- Selection screen attributes
- Single fields (PARAMETERS)
- Value sets (SELECT-OPTIONS)
- Selection screen events



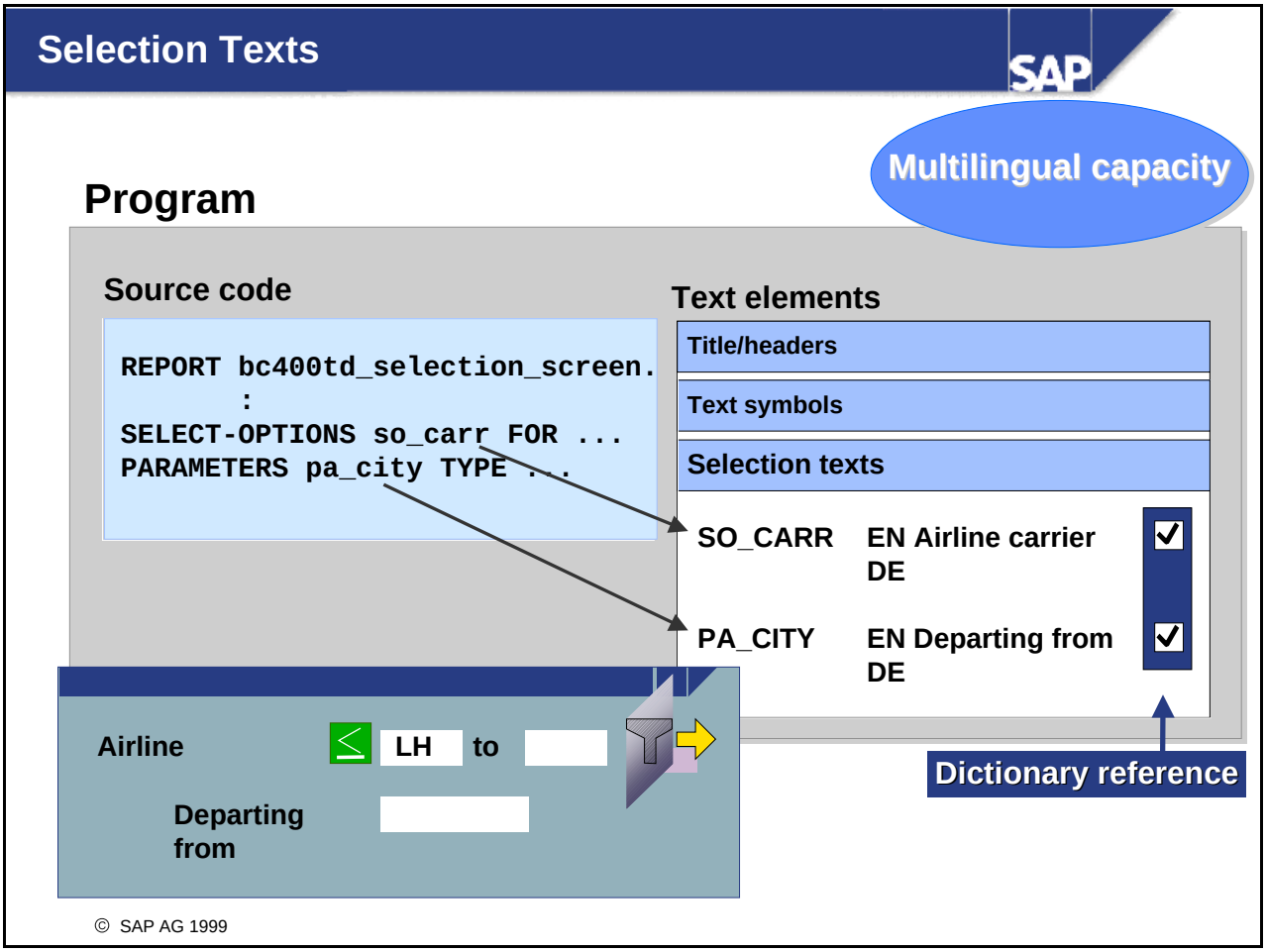

- Selection screens are designed to present users with an input template. This allows them to enter selections, which reduce the amount of data that has to be read from the database. The user has the following options:
  - Enter values in single fields
  - Make **complex entries:** intervals, operations, patterns
  - Save selections fields filled with values as **variants**
  - Access input help and search helps by choosing the F4 function key or the possible entries pushbutton
- You can translate **selection texts** into other languages so that they are then displayed in the language in which the user is logged on.
- The system checks types automatically. If you enter a value with an incorrect type, the system displays an error message and  makes the field ready to accept your corrected entry.




- Selection screens allow you to enter complex selections as well as single-value selections. The syntax for programming selection options, dealt with in this unit, includes the following topics:
  - Setting selection options
  - Entering multiple values or intervals
  - Defining a set of  exclusionary criteria for data selection
- Every selection screen contains an information icon. Choose this icon to display additional information.

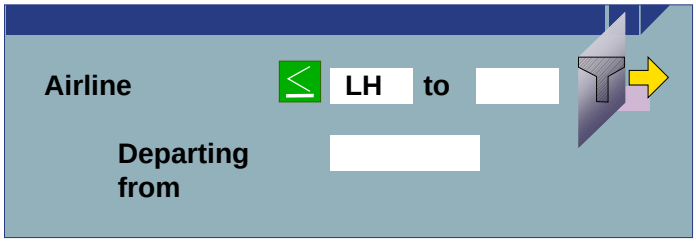


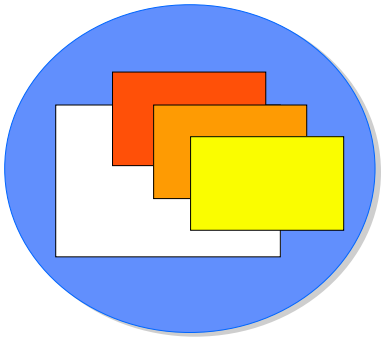

- If an input field is typed with a **data element**, the following semantic information is available:
  - You can use the **field name** as a selection text
  - **Input help** (F1 help) from the data element is available automatically
  - **Possible entries help** (F4 help) is available automatically, provided the data element has been coupled with a search help. A search help is a standalone object defined in the Dictionary, which controls the dialog with and data retrieval for the possible entries help.
- If an input field is typed with a **structure field**, the following semantic information is available:
  - **Field names** and **input help** (F1 help) are copied from the data element that has been used to type the Dictionary structure field.
  - If a structure field is coupled with a search help, then this is the search help that is used for the possible entries help (F4 help) - that is, it obscures the data element search help. If there is search help coupled with the structure field, the system uses the data element search help.
- Bear in mind the Dictionary type you choose to provide a type for an input field affects the semantic information available to the user.
- For more information, refer to the online documentation for the ABAP Dictionary.




- On the selection screen, the names of the variables appear next to the input fields. However, you can replace these with selection texts, which you can then translate into any further languages you require. Selection texts are displayed in the user's logon language.
- If the input field is typed directly or indirectly with a data element, you can copy the field name from one of the texts stored in the Dictionary.

## Variants



**Create a variant:**

- **Fill out the selection screen fields**
- **Save as a variant**
  - **Enter a variant name: <Name>**
  - **Meaning: Enter a <descriptive short text>**
  - **You can also choose other attributes**

**Use the variant:**

- **Start the program**
- **Choose the**  **icon**
- **Choose a variant by name**
- **The system copies the values to the input fields**

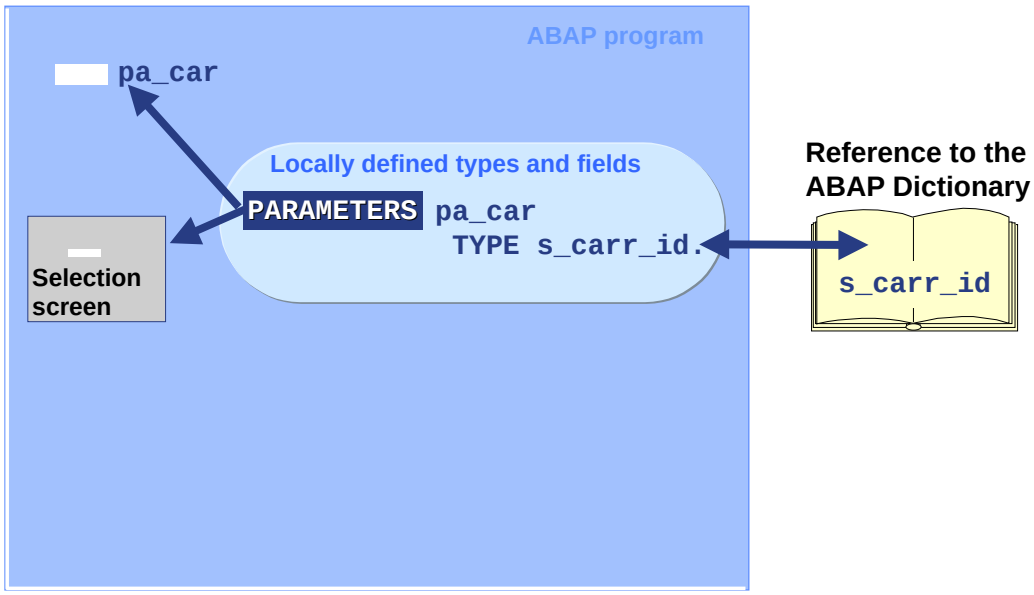


- If you want to save the values (or some of the values) on a selection screen that you have filled out, you can do so by creating a variant. When you start the program again later, you can get these values from the variant and display them in the selection screen.
- You can define and save variants for any selection screen. You do this by starting the program and choosing *Variants -> Save as variant*.
- Variants allow you to make selection screens easier to use by:
  - Pre-assigning values to input fields
  - Hiding input fields
  - Saving these settings for reuse
- A single variant can refer to more than one selection screen.
- Variants are client-specific.
- If you choose the information icon (on any selection screen), the system will display more information about variants. You can also find out more in course BC405 *Techniques of List Processing and InfoSet Query*.

# Single Fields (PARAMETERS)

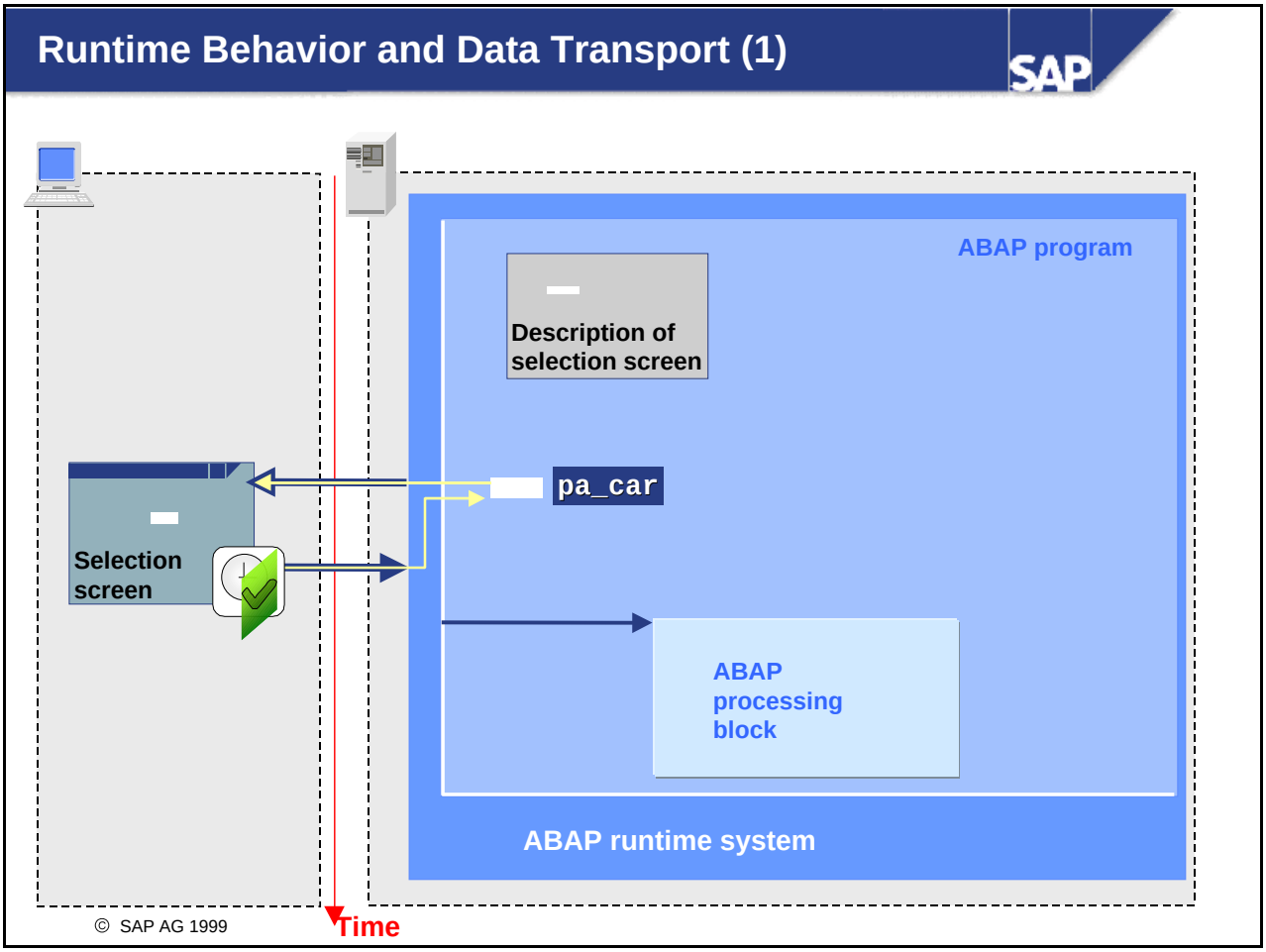
- Selection screen attributes
- Single fields (PARAMETERS)
- Value sets (SELECT-OPTIONS)
- Selection screen events

# Effect of the PARAMETERS Statement






- In an executable program, a single **PARAMETERS** statement is sufficient to generate a standard selection screen.
- The **PARAMETERS** <name> **TYPE** <type> and **PARAMETERS** <name> **LIKE** <data object> statements generate a simple input field on the selection screen, and a data object <name> with the type you have specified.
- If the user enters a value and chooses *Execute*, that value is placed in the internal data object <name> in the program. The system will only permit entries with the appropriate type.




- Once the **INITIALIZATION** event block has been processed, the selection screen is sent to the presentation server. The runtime system transports the data object values that are defined using **PARAMETERS** to the selection screen input fields of the same name.
- The user can then change the values in the input fields. If the user then clicks on the *Execute* function, the input field values are transported to the program data objects with the same name and can be evaluated in the ABAP processing blocks.

## Using Parameters When You Access the Database

```
REPORT ...
DATA wa_spfli TYPE spfli.
PARAMETERS pa_car  TYPE s_carr_id.
...
SELECT carrid connid cityfrom cityto ...
          FROM spfli
          INTO CORRESPONDING FIELDS OF wa_spfli
          WHERE carrid = pa_car .
      WRITE: / wa_spfli-carrid, wa_spfli-connid,
               wa_spfli-fldate, ...  .
ENDSELECT .
```

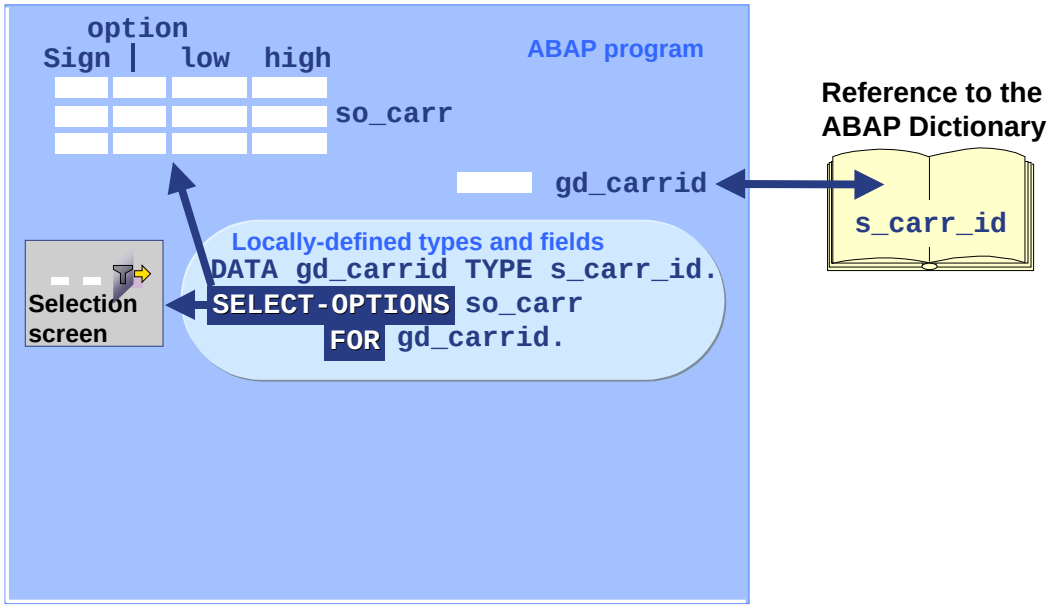


- If you have used the **PARAMETERS** statement to program an input field as a key field for a database table, you can use a **WHERE** clause in the **SELECT** statement to limit data selection to this value.
- In the example above, the system only reads a record from the database table **SPFLI** if that record's key field **CARRID** has the same value as is contained in data object **pa_car** at runtime.

# Value Sets (SELECT-OPTIONS)

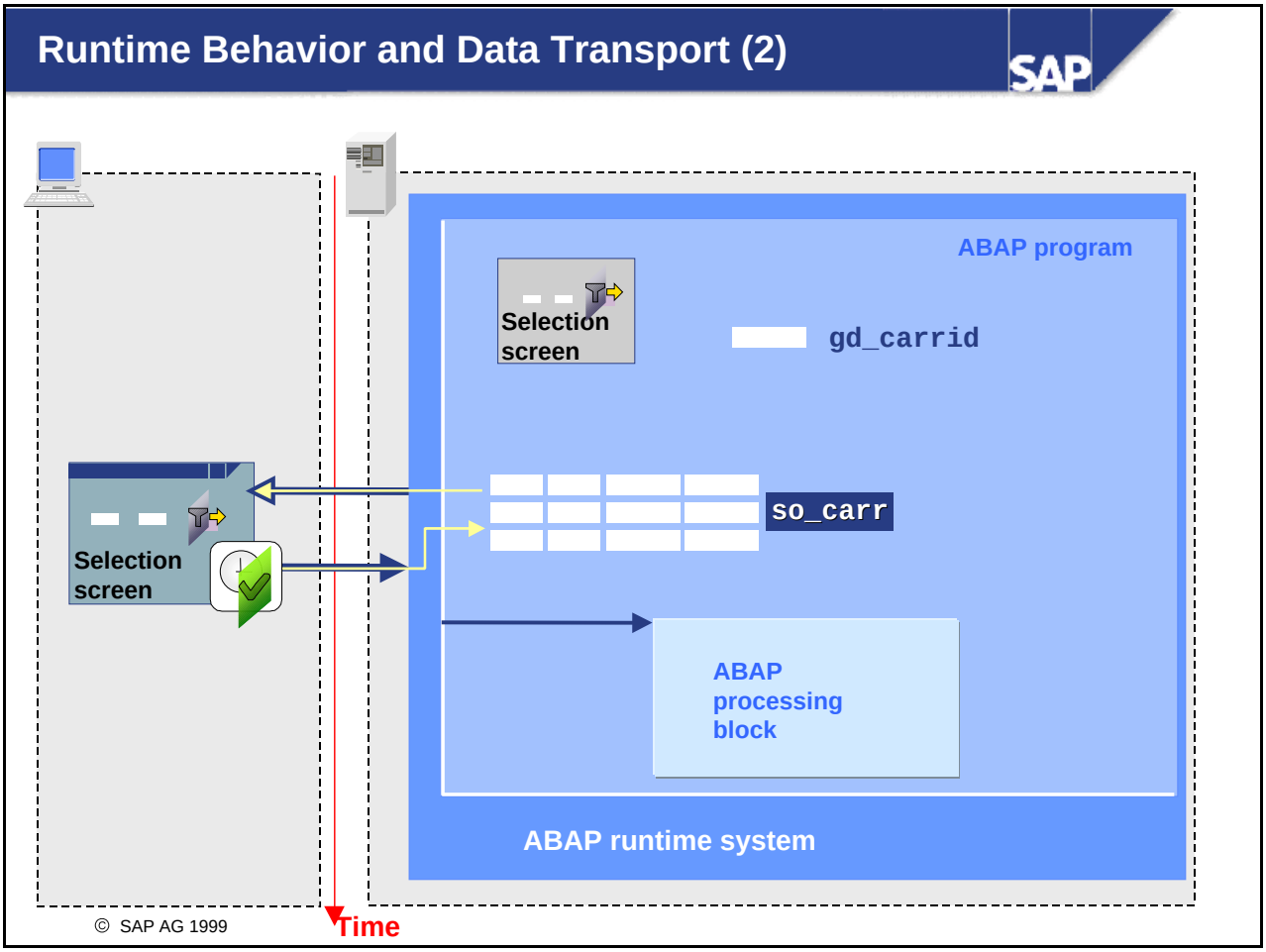
- Selection screen attributes
- Single fields (PARAMETERS)
- Value sets (SELECT-OPTIONS)
- Selection screen events

## Effect of SELECT-OPTIONS



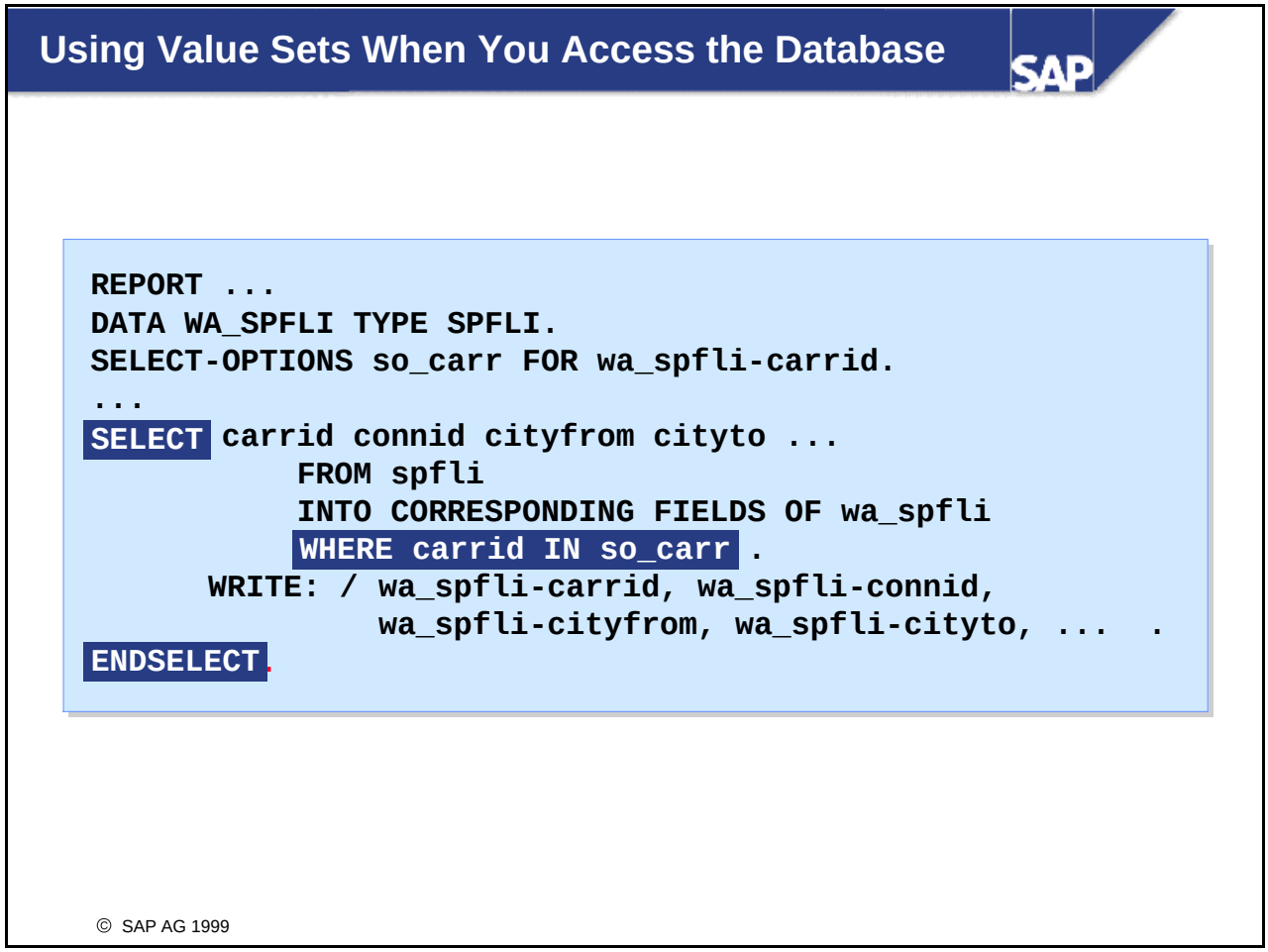



- The statement **SELECT-OPTIONS** <name> **FOR** <data_object> defines a selection option: This places two input fields on the selection screen, with the same type that you have defined in the reference. This enables users to enter a value range or complex selections. The statement also declares an internal table <name> within the program, with the following four columns:
  - **sign**: This field designates whether the value or interval should be included in or excluded from the selection.
  - **option**: This contains the operator: For a list of possible operators, see the keyword documentation for the **SELECT-OPTIONS** statement.
  - **low**: This field contains the lower limit of a range, or a single value.
  - **high**: This field contains the upper limit of a range.
- Selection table <name> always refers to a data object that has already been defined. The data object is used as a target field during database selection, while the selection table is a set of possible values. For this reason, a special version of the **WHERE** clause exists for database selection. It determines whether or not the database contains the corresponding field within the **value set**.




- If the user enters several values or intervals for a selection option and chooses *Execute*, the system places them in the internal table.

## Using Value Sets When You Access the Database



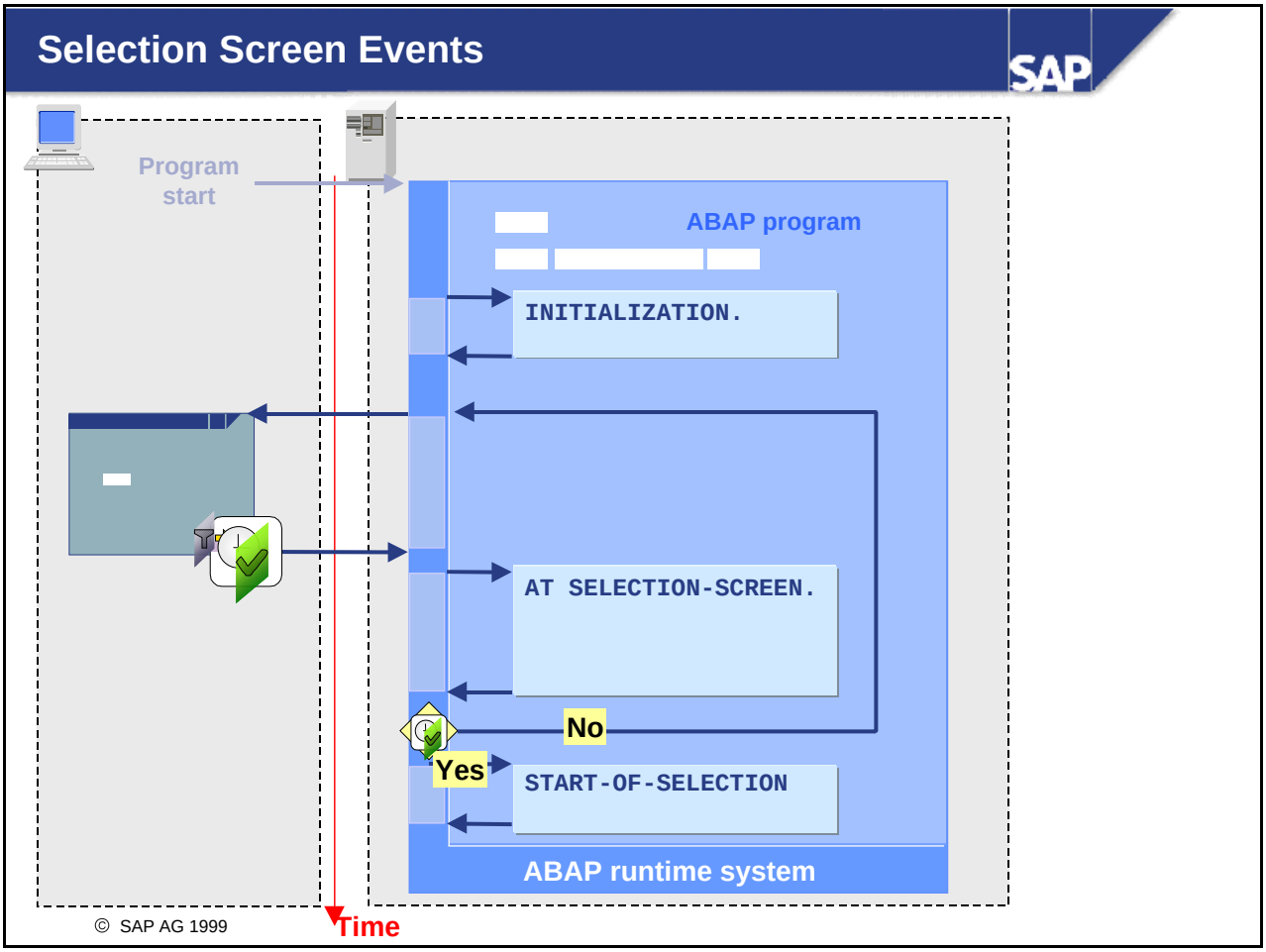
```
REPORT ...
DATA WA_SPFLI TYPE SPFLI.
SELECT-OPTIONS so_carr FOR wa_spfli-carrid.
...
SELECT carrid connid cityfrom cityto ...
          FROM spfli
          INTO CORRESPONDING FIELDS OF wa_spfli
          WHERE carrid IN so_carr .
      WRITE: / wa_spfli-carrid, wa_spfli-connid,
               wa_spfli-cityfrom, wa_spfli-cityto, ...  .
ENDSELECT.
```



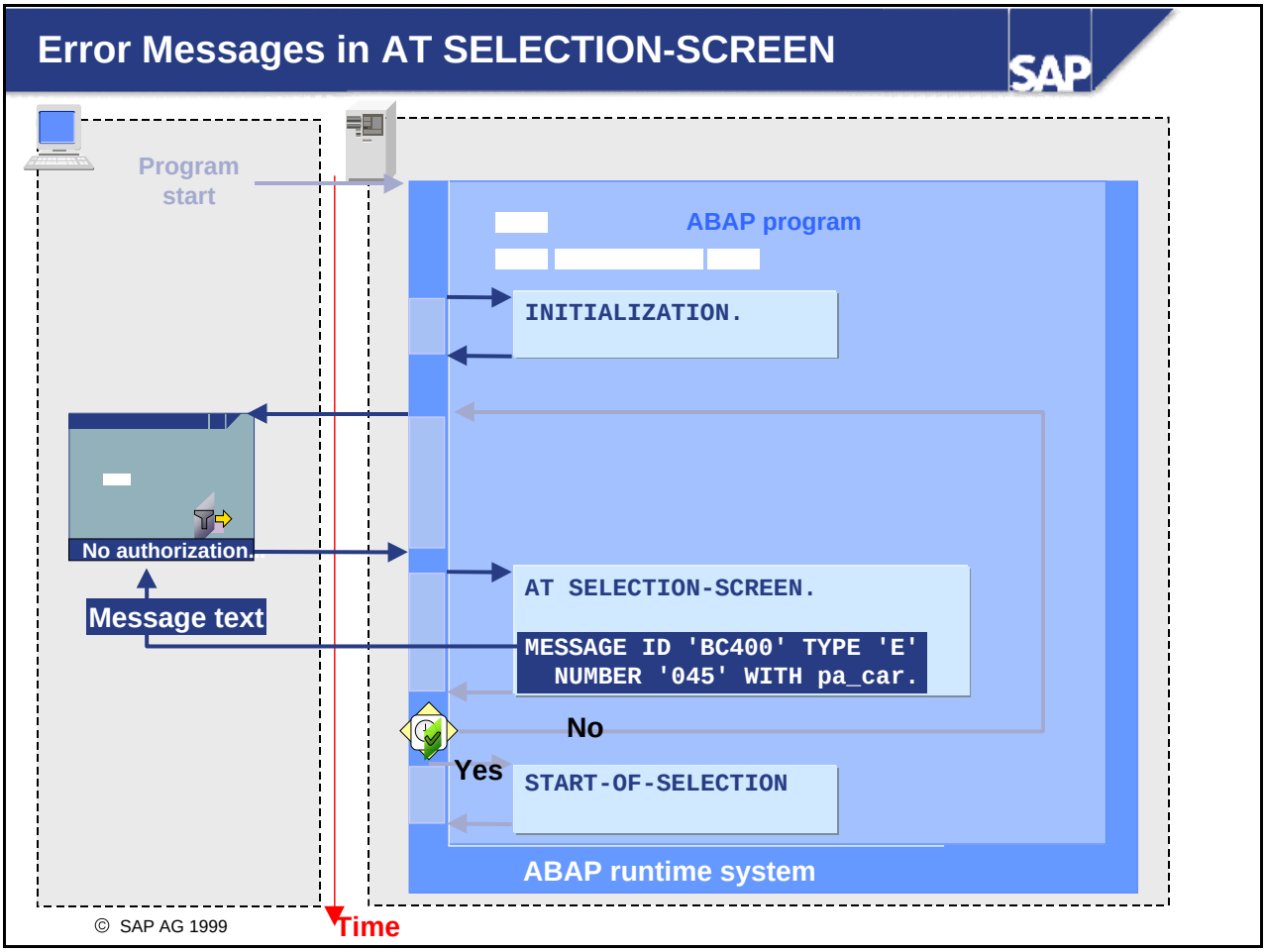
- The above example shows how you can restrict database selection to a certain range using a selection table.
- Conditions in an internal table declared using **SELECT-OPTIONS** are interpreted as follows:
  - If the internal table is empty, the condition <field> **IN** <selname> is always true.
  - If the internal table only contains simple inclusive conditions **i1, ..., in**, the result is the composite condition ( **i1 OR ... OR in** ).
  - If the internal table only contains simple exclusive conditions e1, ..., em, the result is the composite condition ( **NOT e1 ) AND ... AND ( NOT em** ).
  - If the internal table contains both the simple inclusive conditions **i1, ..., in** and the simple exclusive conditions **e1, ..., em**, the result is the composite condition **( i1 OR ... OR in ) AND ( NOT e1 ) AND ... AND ( NOT em ).**

# Selection screen events

- Selection screen attributes
- Single Fields (PARAMETERS)
- Value sets (SELECT-OPTIONS)
- Selection screen events




- In an executable program, the ABAP runtime system generates a standard selection screen as long as you have written at least one **PARAMETERS** or **SELECT-OPTIONS** statement. The selection screen belongs to the event block **AT SELECTION-SCREEN.**
- The selection screen is displayed after the **INITIALIZATION** event block **.**
- Each time the user chooses *Enter*, a pushbutton, a function key, or a menu function, the system carries out a type check. If the entries do not have the correct type, the system displays an error message, and makes the fields ready for input again. When the data types have been corrected, the system triggers the **AT SELECTION-SCREEN** event.
- Subsequent program flow depends on the user's actions:
  - If the user chose F8 or *Execute*, the next event block is called: In this case, **START-OF-SELECTION.**
  - If the user chose any other function, the selection screen is redisplayed.




- Use the event block **AT SELECTION-SCREEN** whenever you want to program additional input checks for a standard selection screen.
- The event block **AT SELECTION-SCREEN** is triggered by each user action. If an error dialog is triggered, the system jumps back to the selection screen and automatically resets all input fields to ready for input and displays a message in the status line.
- Input errors can be caught using the **MESSAGE** statement. See the following slides for an example of this.
- For more detailed information on the **MESSAGE** statement, refer to the keyword documentation.
- Additional information can be found in the keyword documentation for **AT SELECTION-SCREEN.**

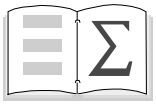
## Syntax Example for AT SELECTION-SCREEN

```
PARAMETERS: pa_car TYPE s_carr_id.


* First event processed after leaving the selection screen
AT SELECTION-SCREEN.
  AUTHORITY-CHECK OBJECT 'S_CARRID'
    ID 'CARRID' FIELD pa_car
    ID 'ACTVT'  FIELD actvt_display.
  IF sy-subrc <> 0.
* Show selection screen again and show message in status bar
    MESSAGE ID 'BC400' TYPE 'E' NUMBER '045' WITH pa_car.
  ENDIF.
```



- As an example of an additional input check with error dialog, you need to add an input field for the airline ID to the program:
- The system carries out an authorization check on the selection screen.
  - If the user has display authorization for the airline entered, the program continues.
  - If the user does not have display authorization, then the selection screen is displayed again and an error message appears in the status bar.

# Selection Screens: Unit Summary



**You are now able to:**

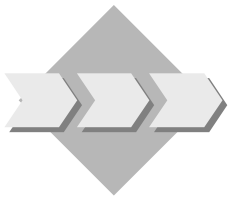
- **Describe selection screen attributes and strengths**
- **Write a program that allows the user to input intervals on a selection screen that can be used to restrict the number of data records retrieved from the database**
- **Write a program containing additional input checks on its selection screen that re-send the selection screen when an error occurs**

## Selection Screen: Exercises

**Unit: Selection Screen**

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

- Use the ABAP statement **SELECT-OPTIONS** to enter complex values on a standard selection screen.
- Take account of complex values in a database selection.
- Program an error message for a standard selection screen



Extend your program **ZBC400_##_DETAIL_LIST** or the corresponding model solution as follows:
Provide the user with a means of entering a complex value set for the flight number. Take the values into account in your database selection.
Additionally, change your program so that the user can only progress from the selection screen if the authorization check for the desired airline is successful.

**Program:** **ZBC400_##_SEL_SCREEN**

**Model solution:** **SAPBC400UDS_SEL_SCREEN**

1-1 Copy your program **ZBC400_##_DETAIL_LIST** or the corresponding model solution **SAPBC400_UDS_DETAIL_LIST** to the program **ZBC400_##_SEL_SCREEN**. Assign your program to **development class ZBC400_##** and the change request for your project "BC400…" (## is your group number).

1-2 Extend your selection screen to allow the user to enter a complex value range for the **flight number CONNID**.

1-3 Use the complex value selection to restrict the amount of data selected from the database table **SFLIGHT**.

1-4 Change your program so that the user cannot progress from the selection screen if the authorization check against the authorization object **S_CARRID** fails. If the authorization check fails, display a suitable error message from message class BC400, and allow the user to enter a different value on the selection screen.

Solutions

**Unit: Selection screen**

**Model solution: Program SAPBC400UDS_SEL_SCREEN**

```
*&---------------------------------------------------------------------*
*& Report        SAPBC400UDS_SEL_SCREEN                        *
*&                                                             *
*&---------------------------------------------------------------------*

REPORT  sapbc400uds_sel_screen.

CONSTANTS: actvt_display TYPE activ_auth VALUE '03'.

DATA: wa_flight TYPE sbc400focc,
      wa_sbook  TYPE sbook.

PARAMETERS: pa_car TYPE s_carr_id.
* Data field for complex restrictions applied to connection id
SELECT-OPTIONS: so_con FOR wa_flight-connid.


* First event processed after leaving the selection screen
AT SELECTION-SCREEN.
  AUTHORITY-CHECK OBJECT 'S_CARRID'
    ID 'CARRID' FIELD pa_car
    ID 'ACTVT'  FIELD actvt_display.
  IF sy-subrc <> 0.
* Return to selection screen again and display message in status * bar
    MESSAGE ID 'BC400' TYPE 'E' NUMBER '045' WITH pa_car.
  ENDIF.
```

```
START-OF-SELECTION.
  SELECT carrid connid fldate seatsmax seatsocc FROM sflight
         INTO CORRESPONDING FIELDS OF wa_flight
         WHERE carrid = pa_car
         AND   connid IN so_con.
    wa_flight-percentage =
    100 * wa_flight-seatsocc / wa_flight-seatsmax.

    WRITE: / wa_flight-carrid,
             wa_flight-connid,
             wa_flight-fldate,
             wa_flight-seatsocc,
             wa_flight-seatsmax,
             wa_flight-percentage,'%'.
    HIDE: wa_flight-carrid, wa_flight-connid, wa_flight-fldate.
  ENDSELECT.

AT LINE-SELECTION.
 IF sy-lsind = 1.
   WRITE: / wa_flight-carrid, wa_flight-connid, wa_flight-fldate.
   ULINE.
   SKIP.
   SELECT bookid customid custtype class order_date
          smoker cancelled loccuram loccurkey
          FROM sbook INTO CORRESPONDING FIELDS OF wa_sbook
          WHERE carrid = wa_flight-carrid
          AND   connid = wa_flight-connid
          AND   fldate = wa_flight-fldate.
     WRITE: / wa_sbook-bookid,
              wa_sbook-customid,
              wa_sbook-custtype,
              wa_sbook-class,
              wa_sbook-order_date,
              wa_sbook-smoker,
              wa_sbook-cancelled,
              wa_sbook-loccuram CURRENCY wa_sbook-loccurkey,
              wa_sbook-loccurkey.
   ENDSELECT.
 ENDIF.
```

# User Dialogs: Screens

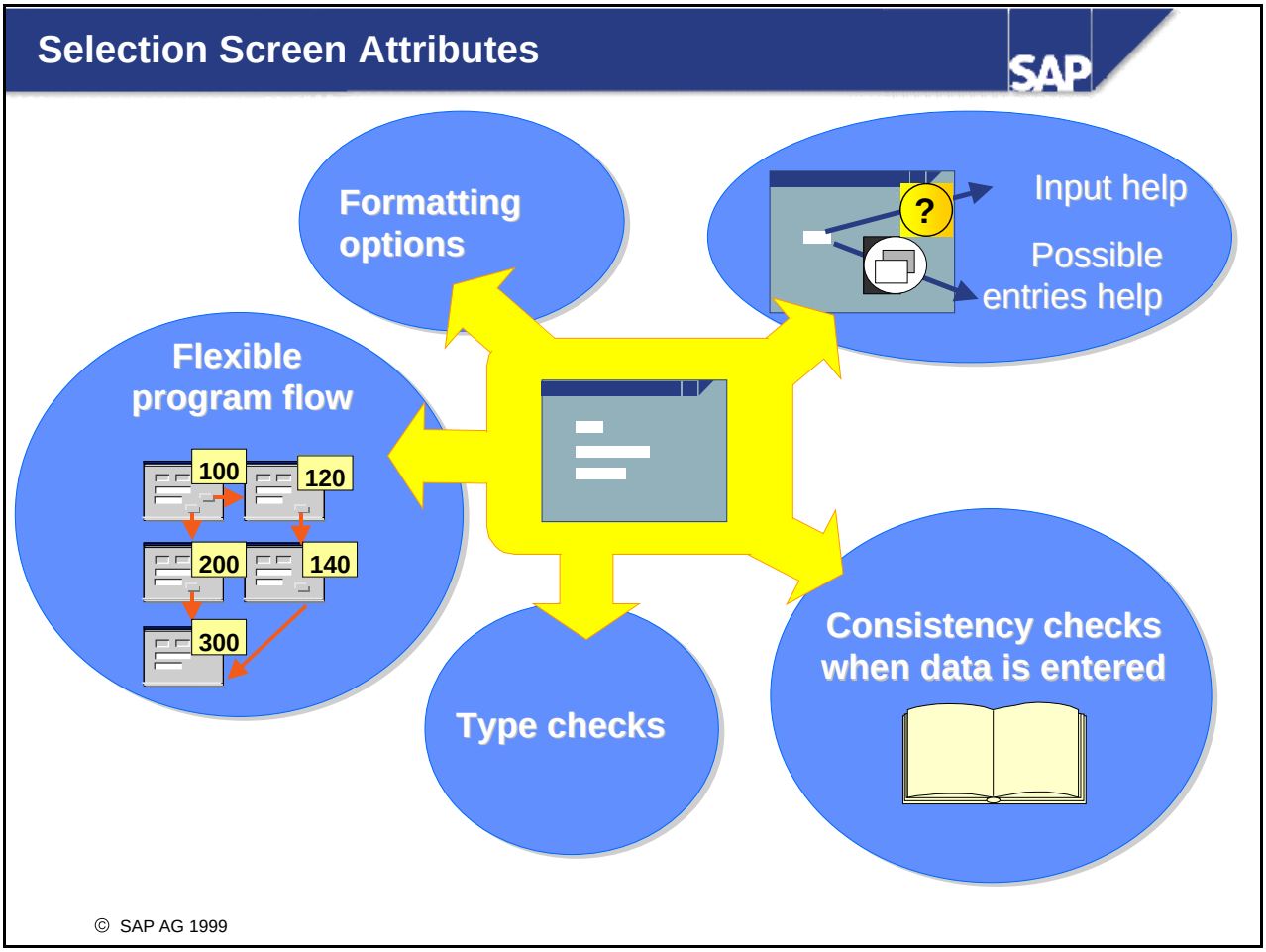
## Contents:

- Screen attributes and strengths
- Creating screens
  - Layout
  - Field attributes
  - Flow logic
- Data transport
- Using pushbuttons and evaluating user actions

# Screens: Unit Objectives

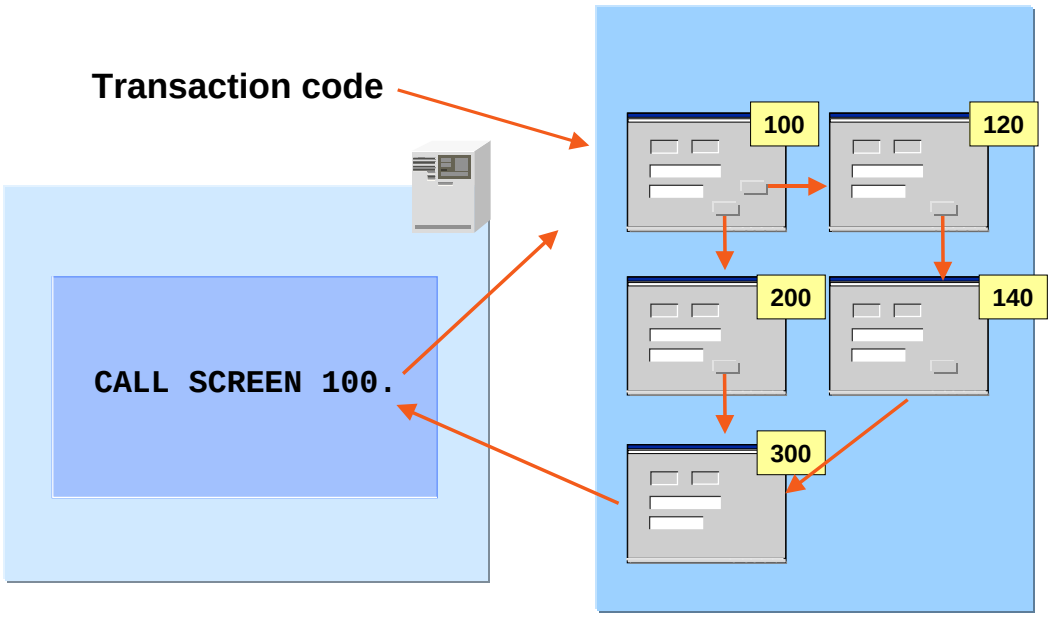
**At the conclusion of this unit, you will be able to:**

- **Describe screen attributes and strengths**
- **Write a program that:**
  - **Displays data on a screen**
  - **Allows the user to change some of that data**
  - **Allows the user to influence further program processing using pushbuttons**



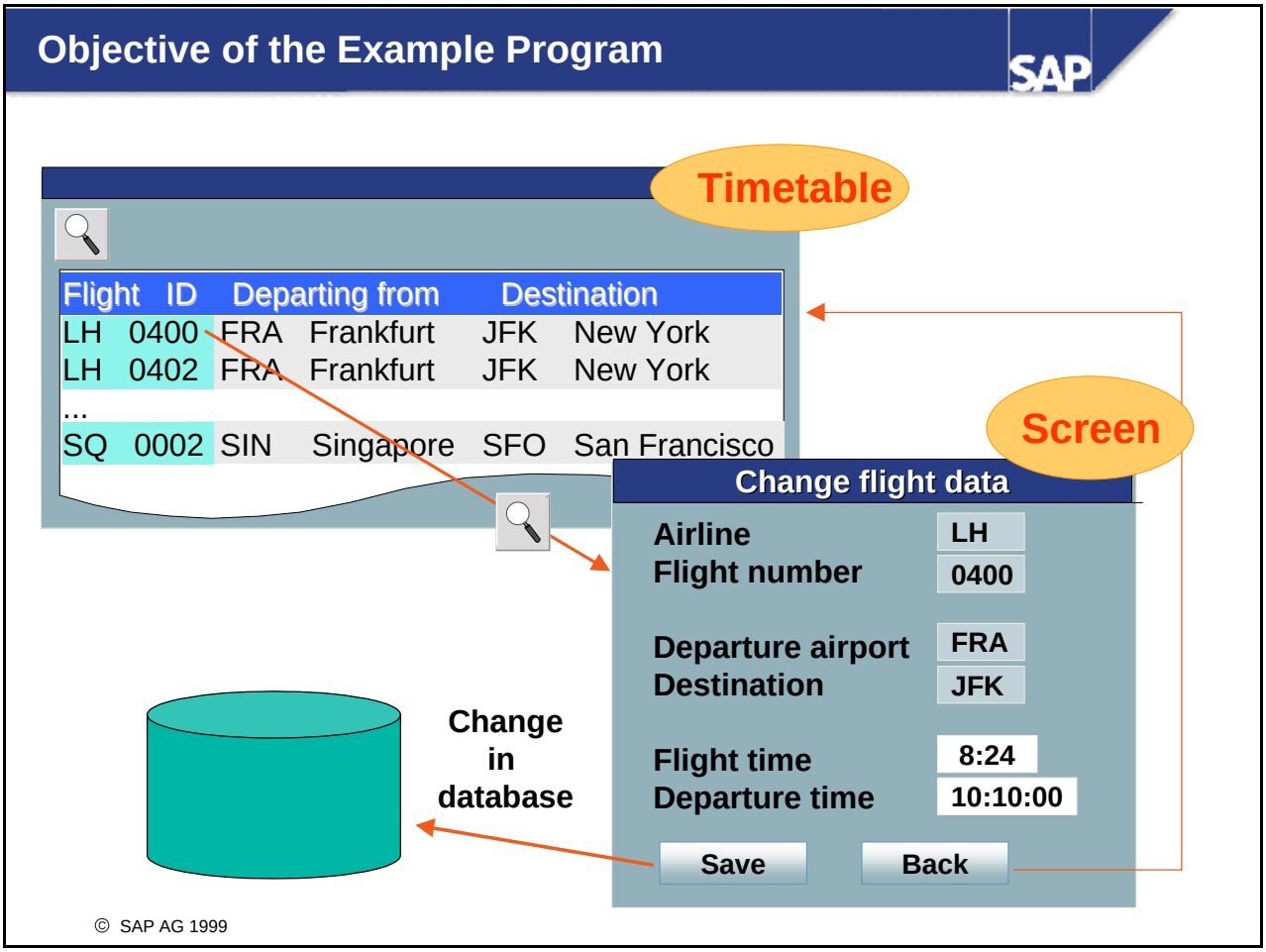

- Screens are made up of more than just a monitor display with input and output fields.
- Screens' integration with the ABAP-Dictionary allows the system to perform consistency checks for their input fields automatically. These checks include required input check, type checks, foreign key checks, and fixed value checks. All of these checks rely upon ABAP Dictionary information.
- 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 when appropriate.
- Screen layout is also very flexible. Input fields, output fields, radio buttons, check boxes, and even pushbuttons can be placed on screens. They allow users to determine the direction in which the program will proceed.
- On the whole, such user influence on program progression allows for more program flexibility in those programs that do contain screens.
- If an input field is typed with a global Dictionary type, you can use any search help or input help (assigned using a data element) associated to it.
- Screens offer 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 currency field; and lengths, weights, and so on are formatted according to the content of a unit field.

## Options for Calling Screens



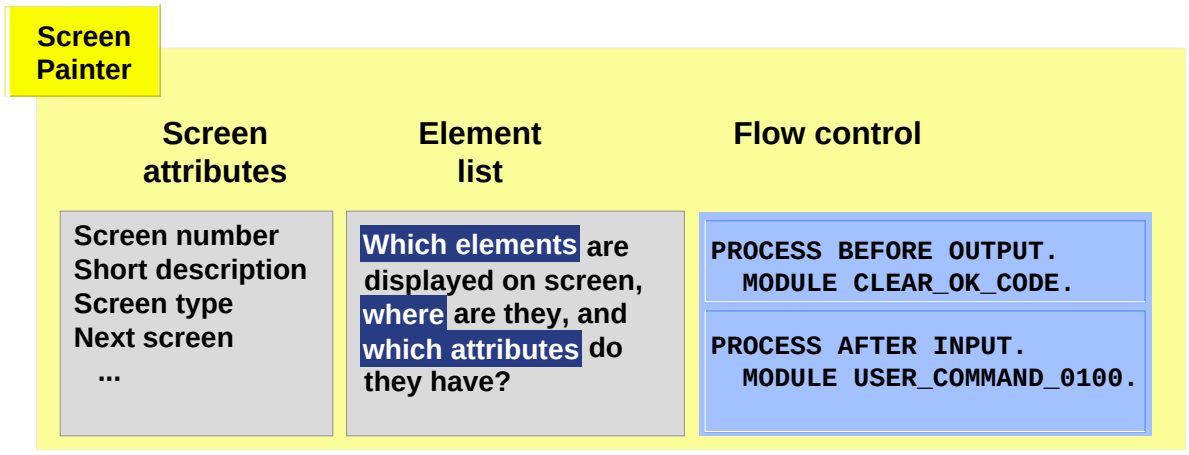



- You can call screens from any ABAP processing block that you want.
- You can create sequences of screens and call them as a unit by calling an initial screen. You must then implement the subsequent program flow by means of the screens' flow logic. There are two ways to call an initial screen:
  - Create a transaction code of the *Dialog transaction* type and enter the initial screen number with the program name. You then add this transaction code to the SAP Easy Access Menu or enter it directly in the input field in the toolbar, which then triggers the sequence of screens.
  - You can call the initial screen for a program from any ABAP processing block.



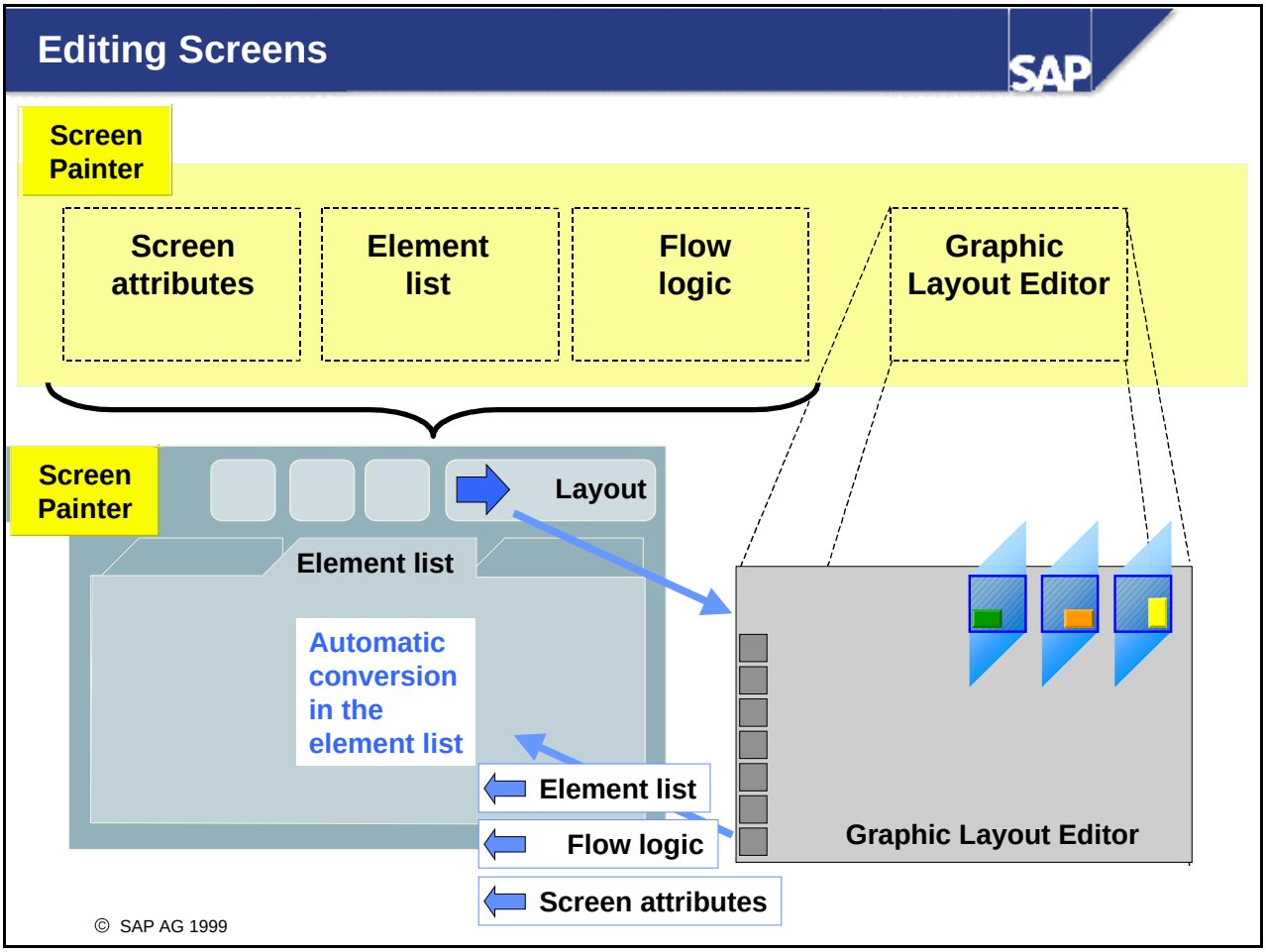

- In the following units you will develop a program step-by-step, which changes standard flight data.
  - Double-click on an entry in the basic list timetable to reach a screen. This screen displays data from the line you selected, as well as additional information about the airline. You can change flight and departure times.
  - Choosing *Back* takes the user back to the basic list without changing any data
  - Choosing *Save* changes the data in the database.
- Changes to the database can be made using function modules. See the unit on the *Database Dialogs* for more about this process.

## Parts of a Screen



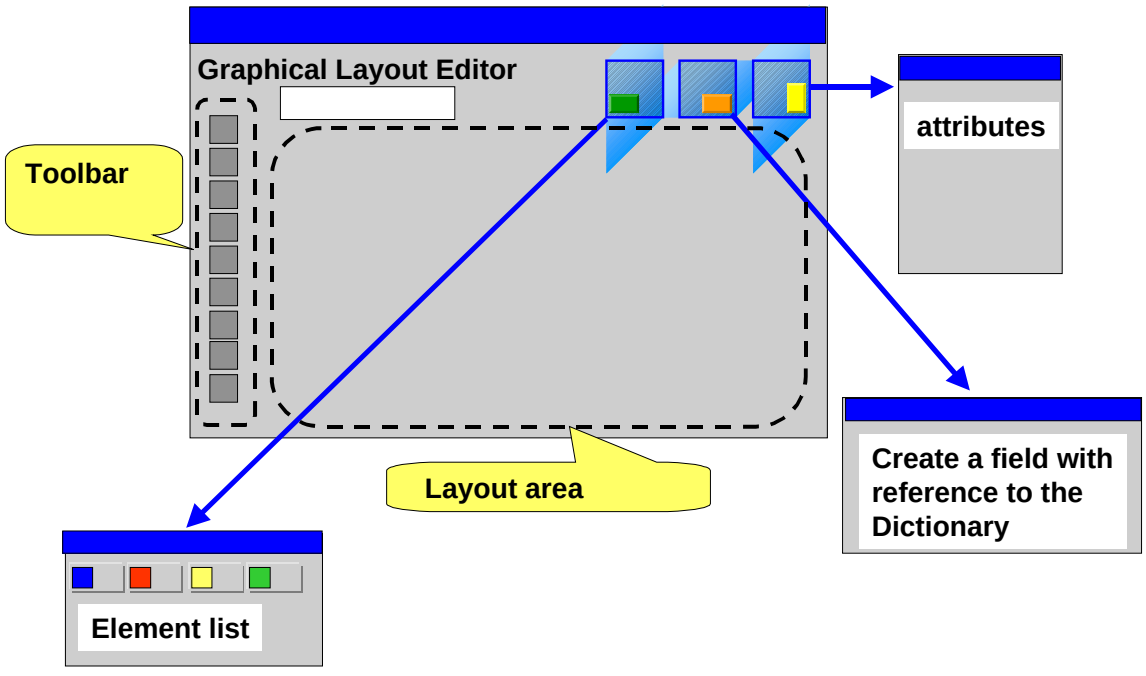



- Each project requires the following information:
  - **Screen attributes** contain, for example, a four-digit number (the screen "name"), a short text, and the screen type (*normal* for full screen, or *modal dialog box* otherwise).
  - The **element list** contains information on which elements appear on the screen, where, and what attributes they have. Elements that are displayed on the screen are called screen objects.
  - The **Flow logic** contains information on the logic that the system must execute before sending the screen to the presentation server, along with the logic that it must execute after control returns to the application server (that is, after the user has performed an action).




- Technically, all the data that belongs to a screen is stored in the screen attributes, the element list, and the flow logic.
- You can use the Graphic Layout Editor to create the screen layout. In the Screen Painter, choose *Layout.* The Graphic Layout Editor appears. As soon as you leave it - for example, by choosing the *Previous* icon (the blue arrow pointing left) - the system copies the graphical information from the Editor to the technical values and you can continue working in the Screen Painter.
- Note: To avoid inconsistencies, the Screen Painter is locked for input while you are using the Graphic Layout Editor (that is, the system displays an egg-timer).

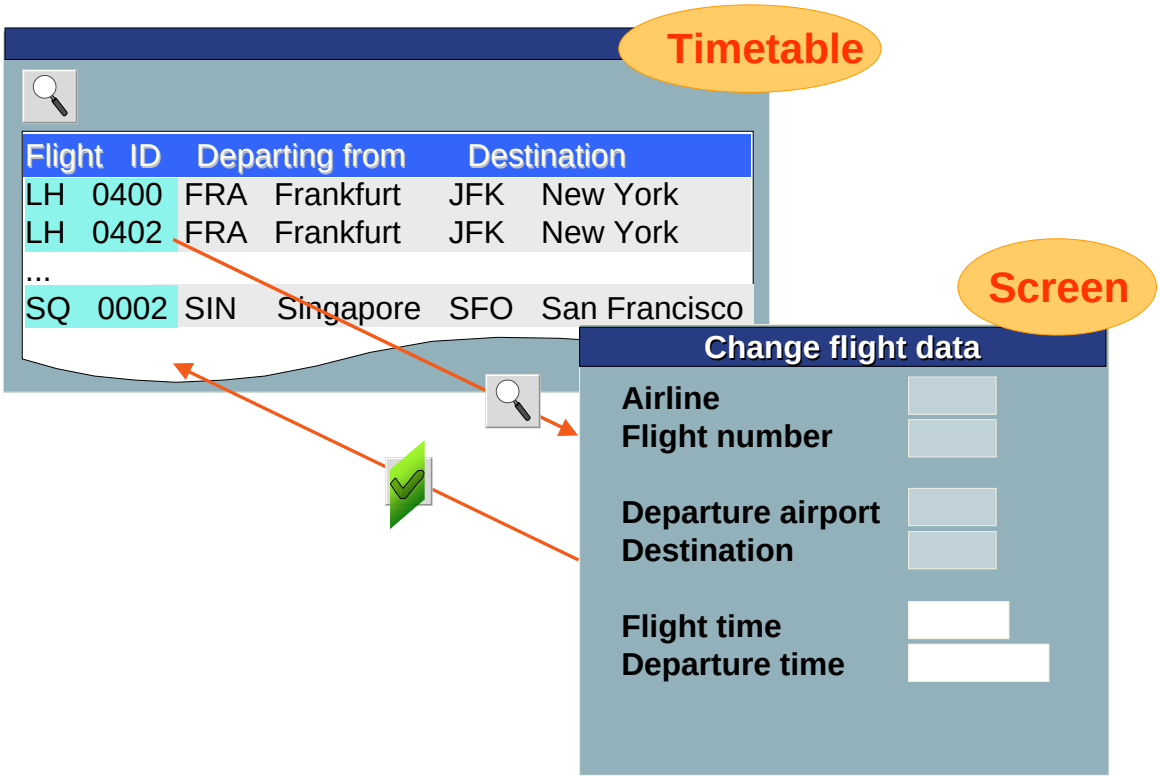
## The Editing Window in the Graphical Layout Editor






- There are three additional dialog boxes available in the Graphic Layout Editor:
  - **Element attributes:** displays all the attributes for a screen object, some of which you can change in this dialog box. For example, you can specify whether or not an input/output field is to be input-ready.
  - **Dictionary/program fields:** Allows you to generate fields that have either a global type or the same type as a data object in the program.
  - **Element list**: shows all the elements displayed on the screen (**screen objects**) with their attributes. You can also change attributes here.

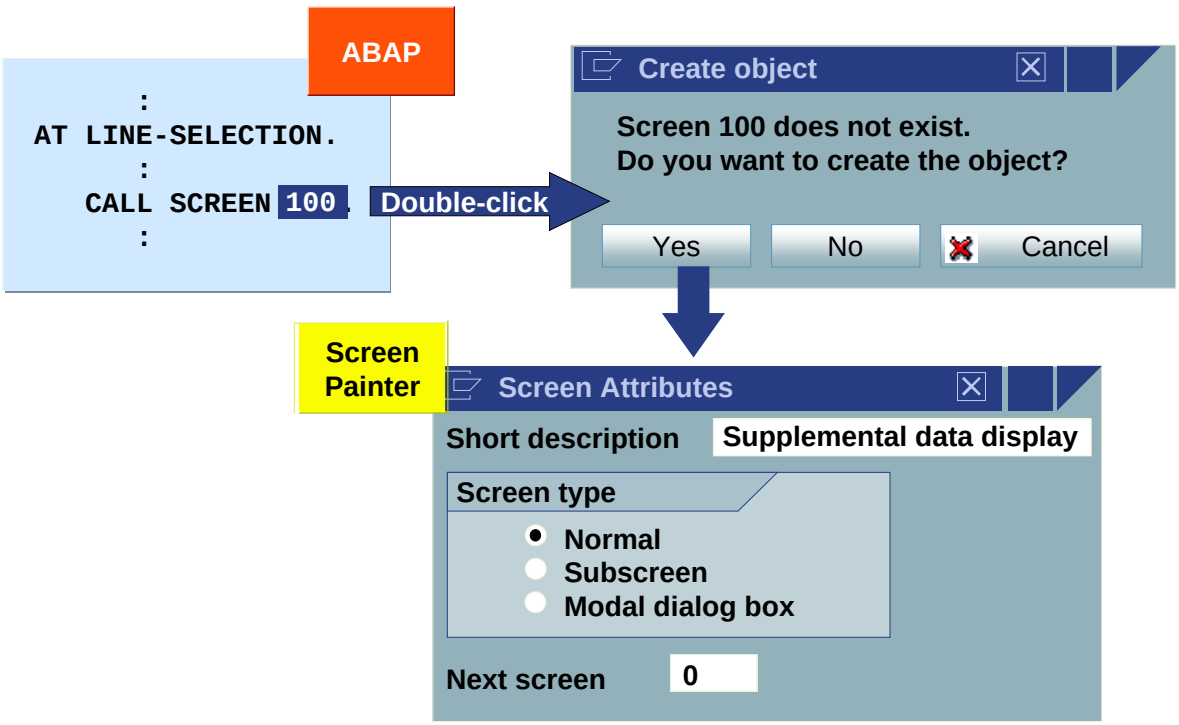
## Example, Step 1: Creating a Screen





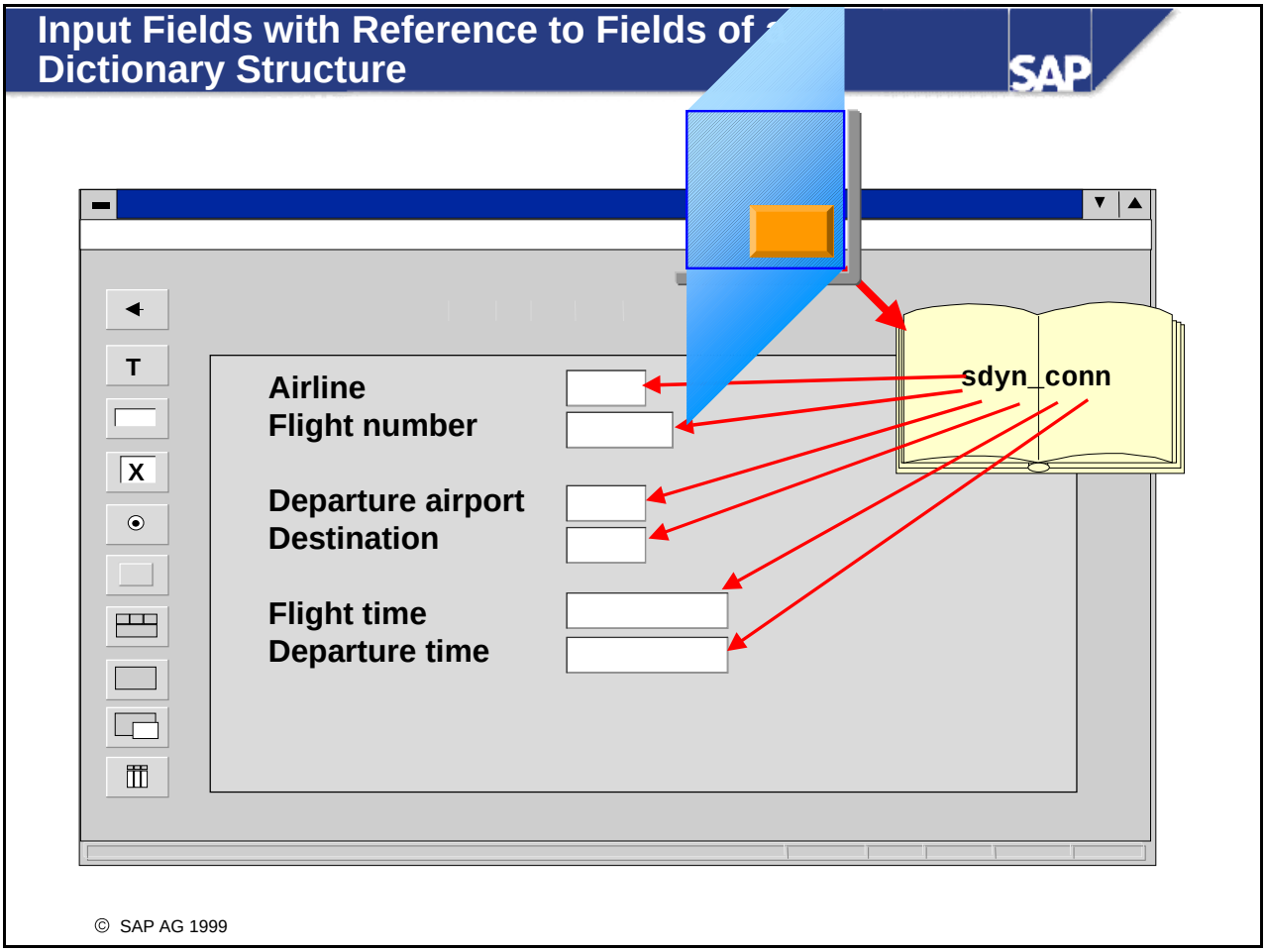

- Your first step is to create a screen, specify its layout, and define its field attributes. The fields: *Airline, Flight Number, Departure Airport*, and *Arrival Airport* should appear as output fields, *Flight Time* and *Departure Time* as input fields.
- You should be able to call your screen by double-clicking a line within the basic list and you should be able to return to the basic list by choosing the appropriate function key on the screen.

# Creating a Screen: Screen Attributes

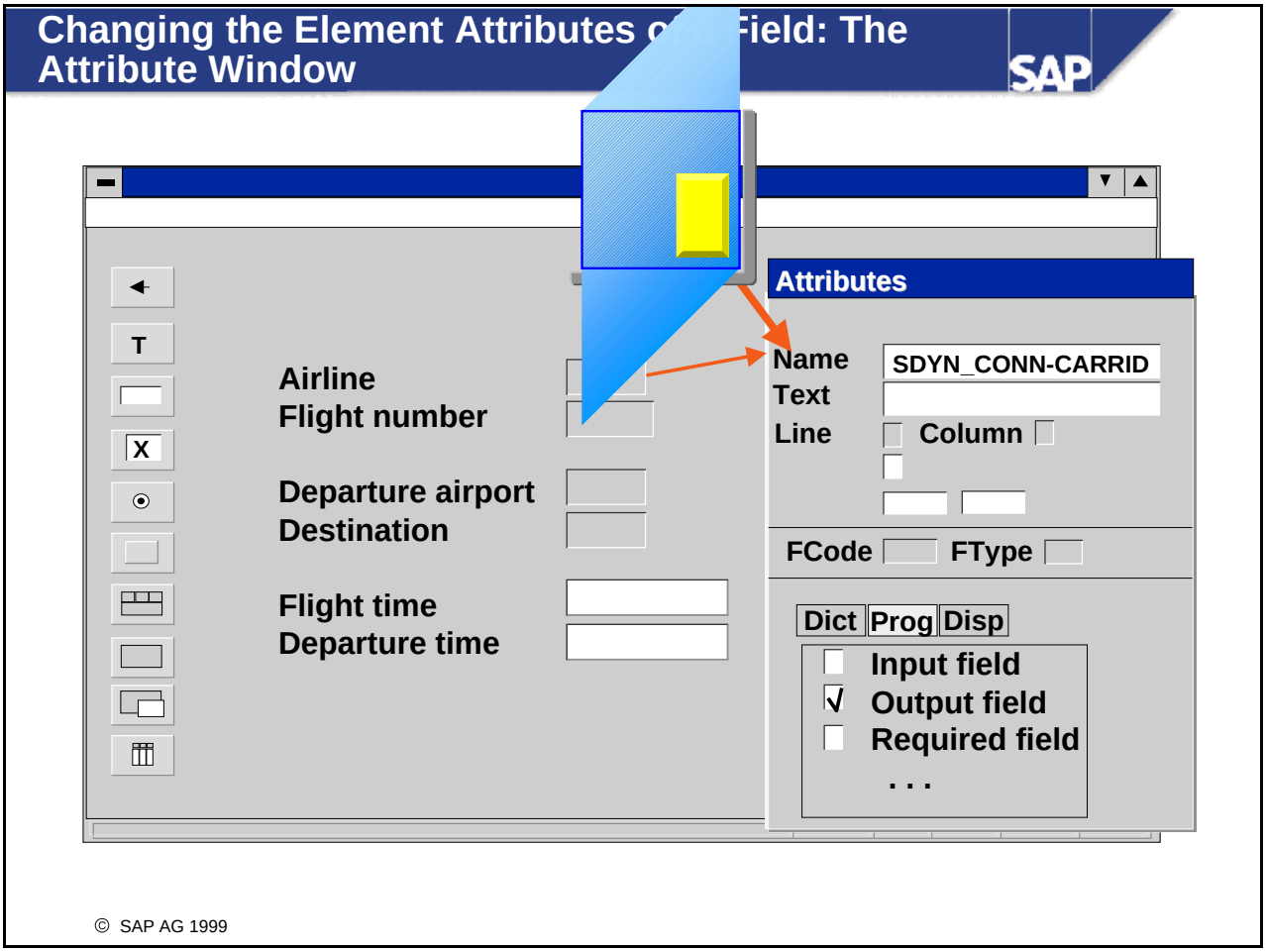





- There are several ways to create screens:
  - **Forward Navigation:** You can create screens from within the ABAP Editor by double-clicking on the screen number. This transfers you into Screen Painter automatically
  - **Object Navigator**: You can also create a screen from the object list in the Object Navigator
- When creating a screen for the first time the system will ask you to enter **screen attributes**. Enter a **short description** of the screen, select screen type *Normal* and enter the number of the subsequent screen in the *Next Screen* input field.
- If you enter 0 or leave the *Next Screen* field blank, the system first processes your screen completely and then returns to processing the program at the point immediately following the screen call. Be aware that in the input field of the next screen 0 is suppressed as an initial value during display.
- In this example the screen you create is supposed to be called from within a basic list. Therefore **CALL SCREEN 100** must belong to the event block **AT LINE-SELECTION**.

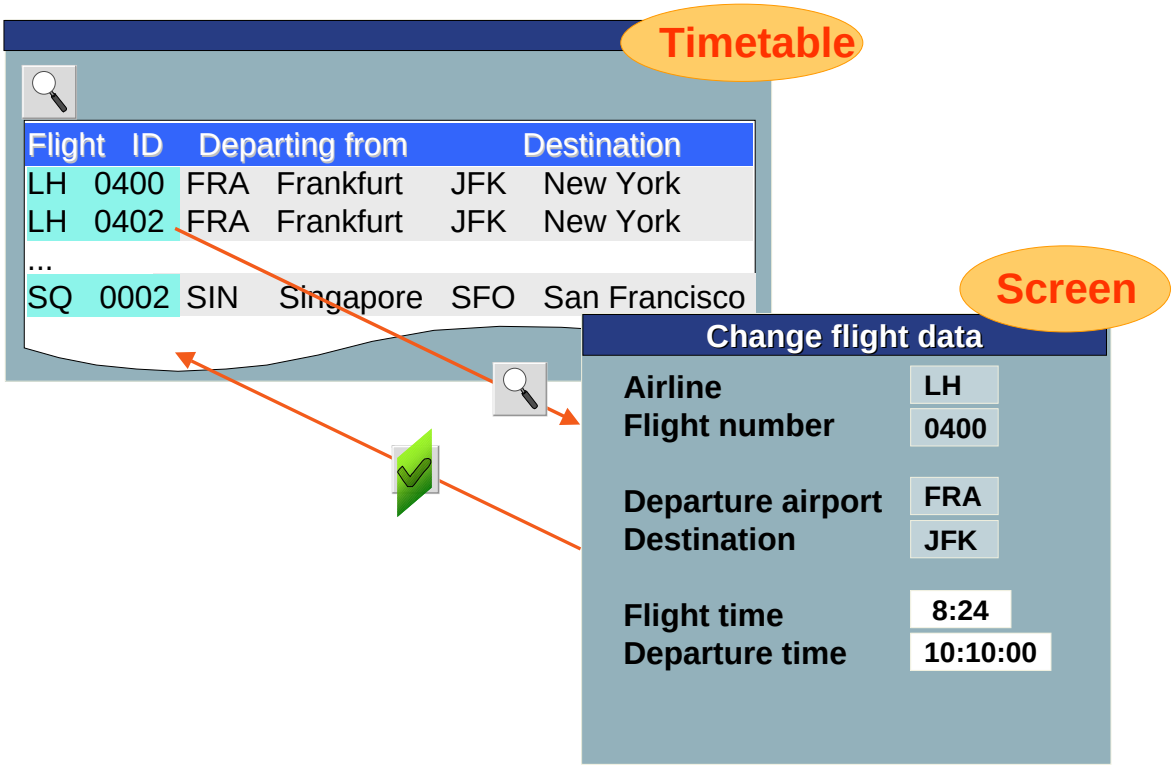



- There are two ways of assigning field attributes to screen fields:
  - **Adopt them from the Dictionary**: You can adopt types and field attributes from existing ABAP Dictionary structures. This makes all information about the object available to you, including semantic information about its data elements and foreign key dependencies. The name of the Dictionary field is automatically adopted as a field name.
  - **Adopt them from a program**: You can adopt field attributes from data objects already defined within a program. To do this, however, an activated copy of the program must already exist. The name of the data object is automatically adopted as a field name.
- The Graphical Screen Painter's interface allows you to define screen elements (for example, input and output fields, keyword texts, borders, and so on) with relative ease. Choose the desired screen element from the column on the left and then place it on the screen using your mouse.
- You can delete screen elements simply by selecting them with your mouse and then choosing *Delete*.
- You can move screen elements by holding down your left mouse button and dragging them to a new position.




- You can maintain screen field attributes by selecting a field and choosing **Attributes**.
- You can classify certain fields as 'mandatory' (*Required field*). A question mark is displayed at runtime if the field is initial.
- If not all required fields have been filled at runtime and a user action is performed, an error dialog is triggered and all input fields are once again displayed ready for input.

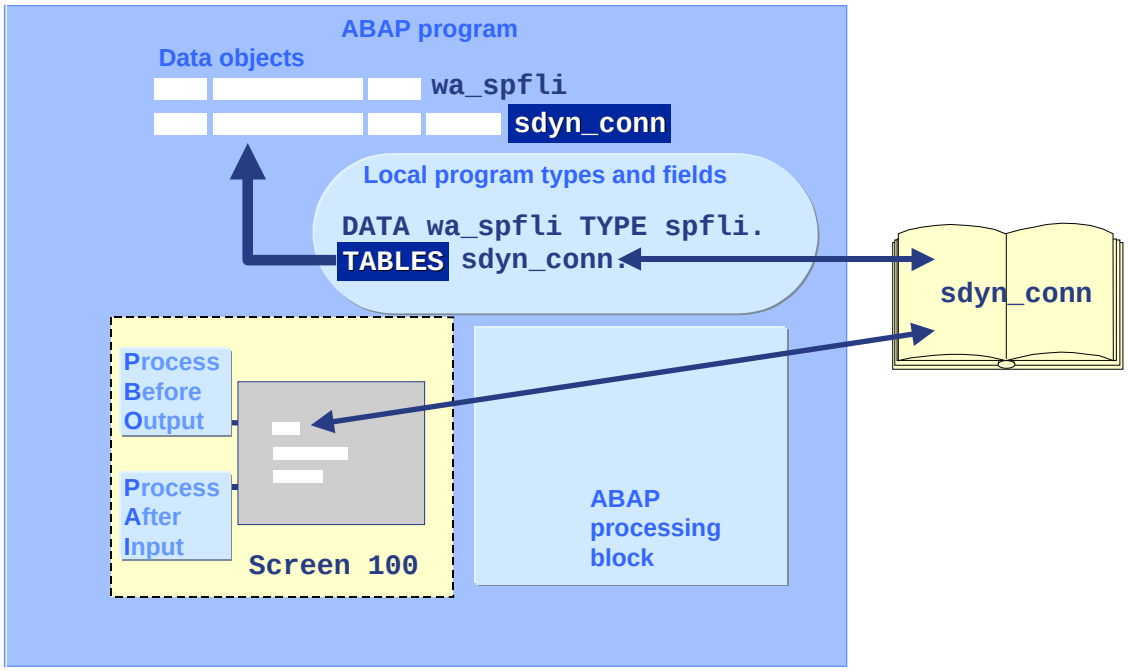
## Example, Step 2: Displaying Data





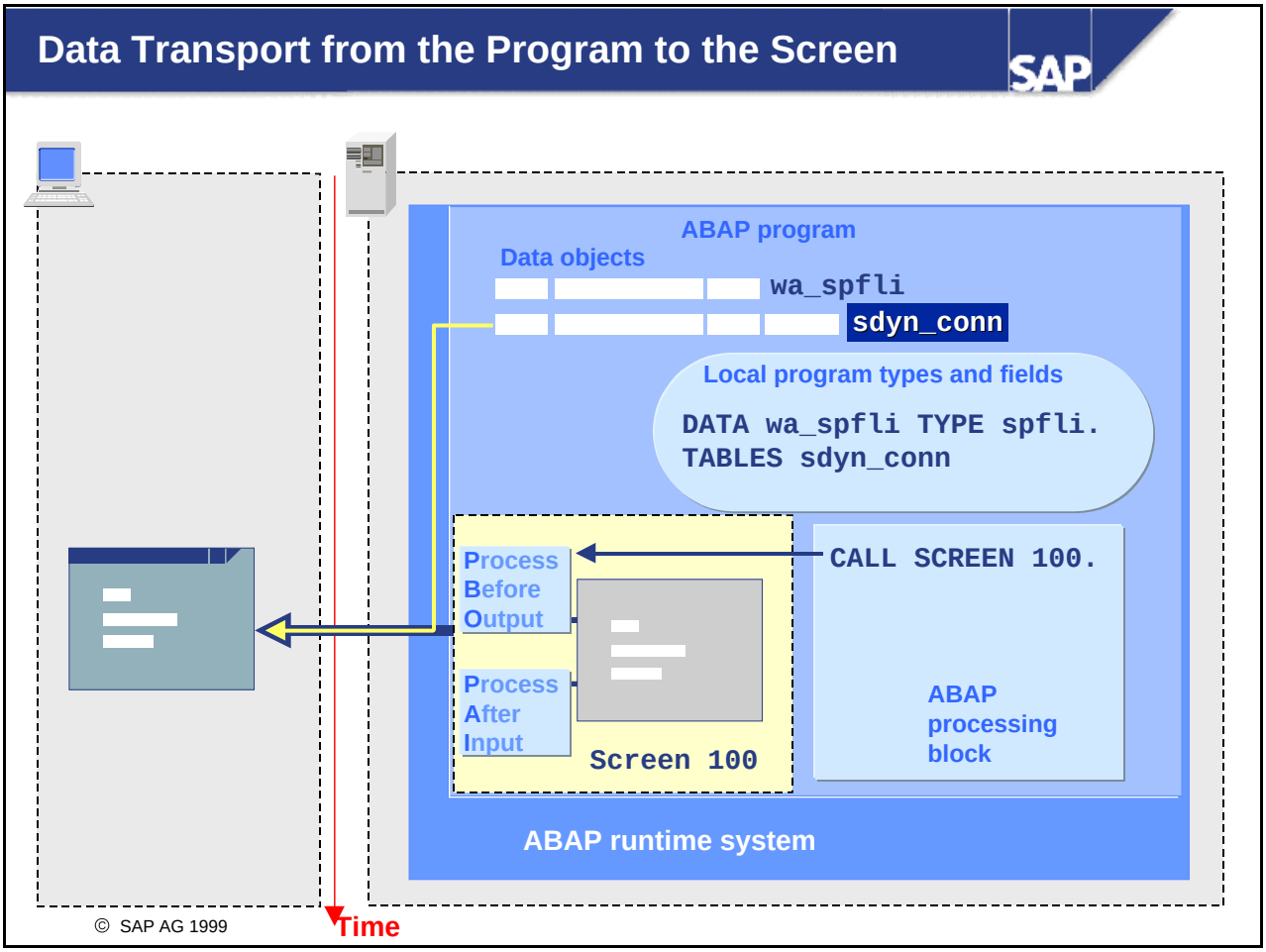

- In step two you will learn how to program data transport from a basic list onto your screen.
- For the user, the program works in the following manner:
  - By double-clicking on a line in the basic list the user branches to a screen. On this screen the most important bits of information for the connection he or she has chosen are displayed. The flight time and departure time are displayed in a field that is ready for input and hence can be changed.
  - The user can return to the basic list in one of several ways.
- With this in mind, this part of the unit will deal with:
  - Prerequisites for automatic data transport between programs and screen fields
  - Defining the screen interface and programming data transport to the interface's data objects

## Screen Interfaces

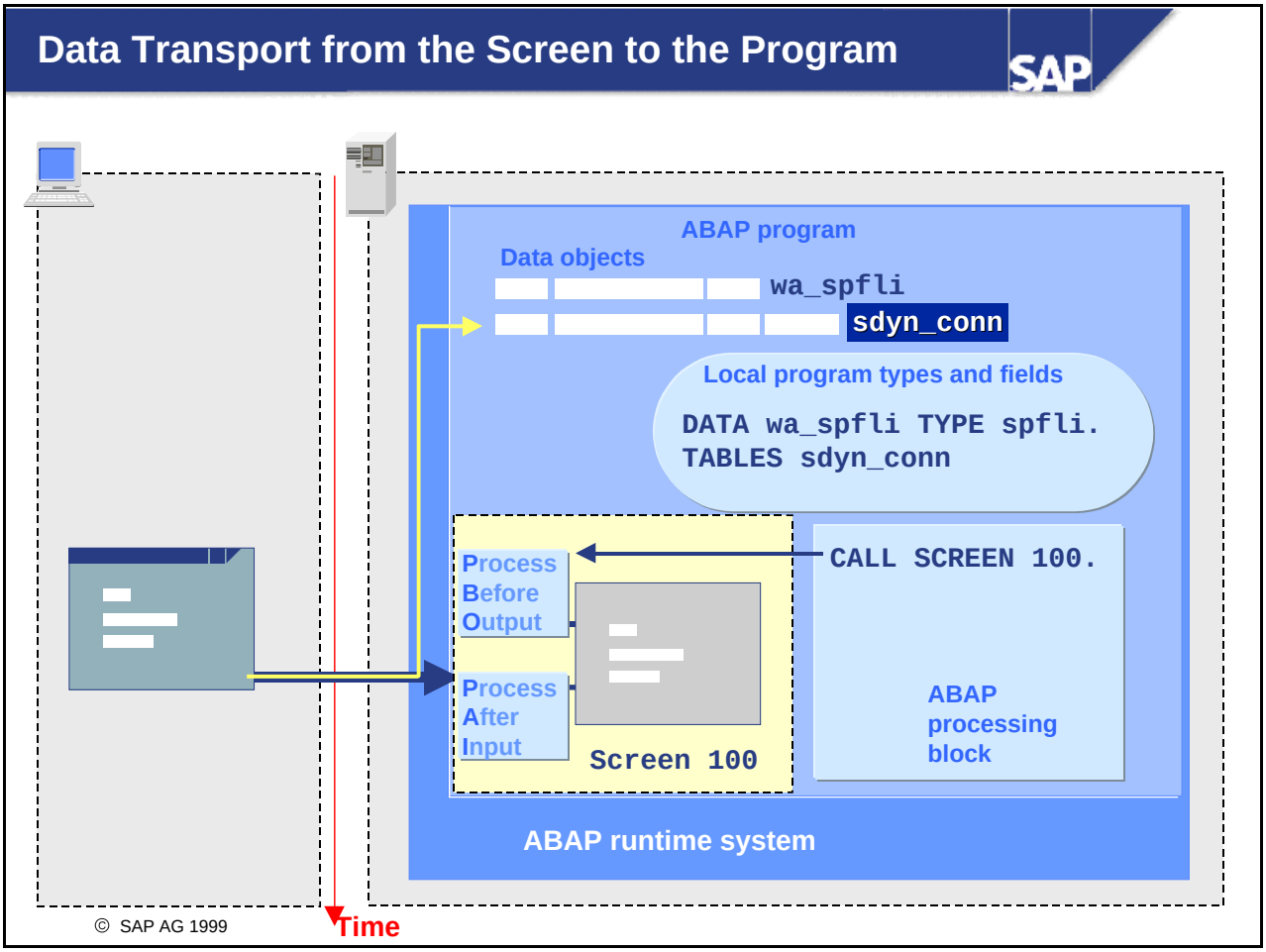




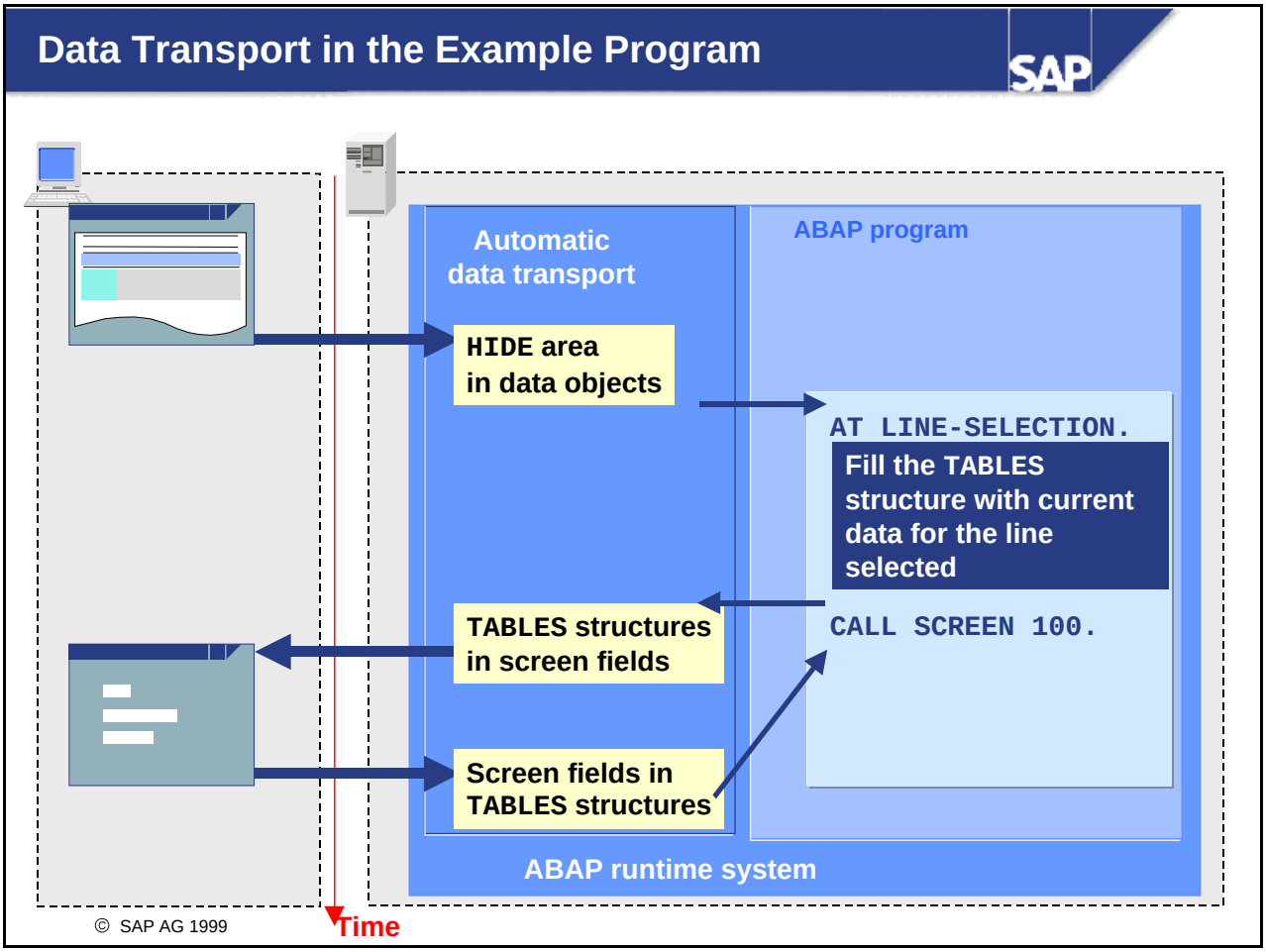

- The statement **TABLES** declares an internal data object that is used as an interface for the screen. **TABLES** always refers to a structure defined in the ABAP Dictionary.
- If a **TABLES** statement and a screen field both refer to the same Dictionary structure, this data object's data is transported to the screen fields every time the screen is called. Any new entries or changes that the user makes on the screen are then transferred back into this data object.
- Normally the ABAP Dictionary contains structures with fields that correspond to several different tables. These tables in turn correspond to the business view of particular applications. The flight data programs being created in this course use one structure for master data maintenance (**sdyn_conn**) and another for bookings data (**sdyn_book**). Using your own structures as interfaces usually helps make a program easier to understand and help you avoid errors.




- Data transport takes place automatically between screens and program data objects of the same name:
  - Immediately before a screen is sent to the presentation server (**after** all **PBO** event modules have been processed) the system copies field contents out of the ABAP work area into their corresponding fields in the screen work area.
- ABAP statements facilitate data transport between program data objects and the work area designated as the screen interface.




- Data transport takes place automatically between screens and program data objects of the same name:
  - Immediately after a user action (**before** the first **PAI** module has been processed) the system copies field contents out of the screen work area and into their corresponding fields in the ABAP work area.
- ABAP statements facilitate data transport between program data objects and the work area designated as the screen interface.




- The example program should display data appropriate to the line selected in the basic list.
- If data objects and their values were stored in the **HIDE** area when the basic list was created, the data belonging to the selected line will be placed in the corresponding data objects.
- You must copy the data to be displayed to a **TABLES** structure. which you can do in any ABAP processing block processed before the screen is sent to the presentation server. There are two ways of doing this:
  - You start by reading all the data to be displayed before retrieving the basic list from the database and displaying it. You then place all the necessary data in the **HIDE** area. Then, at the **AT LINE-SELECTION** event, you only after to copy the data from other data objects into the **TABLES** structure.
    **Advantage:** You only have to read data from the database into the program once.
    **Disadvantage:** You have to read data from the database that the user may not even look at. If detailed data has changed between creating the basic list and displaying the screen, the system will display the wrong data.
  - You store the key fields in the basic list in the **HIDE** area when you create the basic list, and read the data for the selected key from the database using **SELECT SINGLE.**
    **Advantage:** You reduce the volume of data that you need to read from the database when you create the basic list. The detailed data on the screen is up-to-date.
    **Disadvantage:** The system sends a query to the database every time the user double-clicks the list.

- The data is copied from the TABLES structure to the identically-named screen fields immediately before the screen is sent to the presentation server.

## Data Availability

| Structure: | wa_spfli | | sdyn_conn | |
|---|---|---|---|---|
| Fields: | Basic list | HIDE area | Screen: Output Field | Input Field |
| MANDT | | | | |
| CARRID | ✓ | ✓ | ✓ | |
| CONNID | ✓ | ✓ | ✓ | |
| COUNTRYFR | | | | |
| CITYFROM | | | | |
| AIRPFROM | ✓ | | ✓ | |
| COUNTRYTO | | | | |
| CITYTO | | | | |
| AIRPTO | ✓ | | ✓ | |
| FLTIME | – | | ✓ | ✓ |
| DEPTIME | – | | ✓ | ✓ |
| ARRTIME | | | | |
| DISTANCE | | | | |
| DISTID | | | | |
| FLTYPE | | | | |

**Before calling the screen:**
```
SELECT SINGLE * FROM spfli ...
```



- As a last step, we will extend the program so that users can change data in the database - specifically, the fields **FLTIME** and **DEPTIME**. To allow the user to change data for several airlines, the system should display a basic list of all airlines that he or she is authorized to change. The user reaches the maintenance screen by double-clicking. Once the changes have been made successfully, he or she returns to the basic list. However, the system does not create a new basic list. Therefore the data that can be changed should not appear on the basic list.
- In order to ensure that the database data that is displayed on the screen is up-to-date, the record is read again from the database at the beginning of **AT LINE-SELECTION**.
- The main advantages of this method are:
  - For the basic list, only those columns of the database table that are displayed on the list need to be read. This can improve performance with large lists.
  - The data that is displayed on the screen is always up-to-date, even if the data record selected has only just been changed using this program. This would not happen if all screen data was placed in the **HIDE** area when the basic list is created.
  - Changes made to the database using the screen do not lead to incorrect values in the basic list, as the modifiable fields are not contained in the list.
  - Looking ahead to the lock concept: The lock times can be shortened. You can find more detailed information on this topic in the *Database Dialogs* unit.

- The program can be extended: You can display additional information from the data record on the screen without having to make many changes.

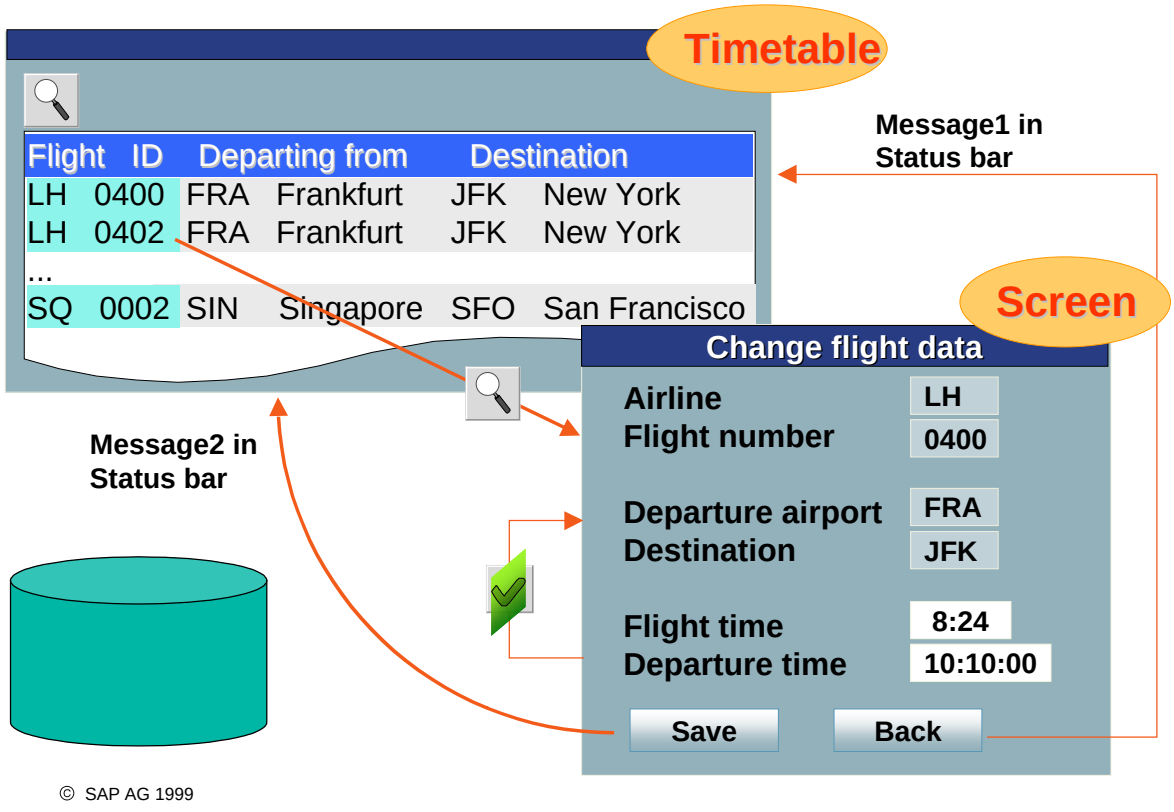
## Syntax: Example Program with Data Transport

```
START-OF-SELECTION.
*
  SELECT carrid connid airpfrom cityfrom airpto cityto
                INTO CORRESPONDING FIELDS OF wa_spfli
                FROM spfli.
    WRITE: / wa_spfli-carrid COLOR COL_KEY,
             wa_spfli-connid COLOR COL_KEY,
             ...                          .

 * Buffering key fields
     HIDE: wa_spfli-carrid, wa_spfli-connid.

   ENDSELECT.

 AT LINE-SELECTION.
   SELECT SINGLE * FROM spfli
                 INTO  wa_spfli
                 WHERE carrid = wa_spfli-carrid
                   AND connid = wa_spfli-connid.
   MOVE-CORRESPONDING wa_spfli to sdyn_conn.

   CALL SCREEN 100.
```



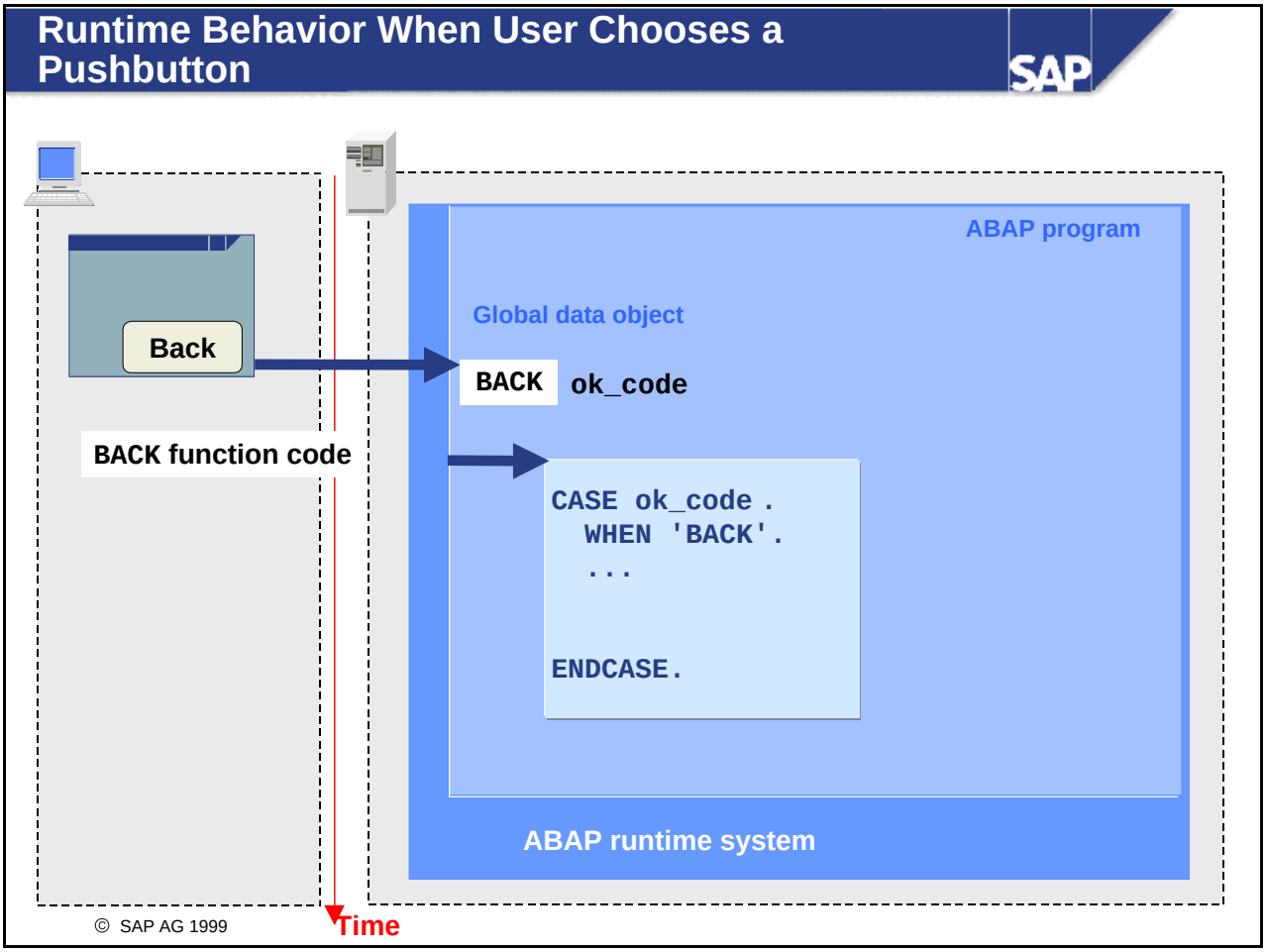
- To display data on the screen, the **TABLES** structure must be filled with current data before the screen is sent to the presentation server. The example above shows one way of doing this.
- The **HIDE** statement is used to place key fields of database tables with reference to the list line in the **HIDE** area. Then the current data for the line selected is available in fields **wa_spfli-carrid** and **wa_spfli-connid** at event **AT LINE-SELECTION**.
- The data record is read from the database using **SELECT SINGLE**. This ensures that the structure contains current data, even if the user has just changed the data. The structure is assigned the same type as the database table line type, so that suitable fields are available for all data in the data record.
- The corresponding fields are copied to the **TABLES** structure **sdyn_conn** using **MOVE-CORRESPONDING**. The system transports the structure data to the screen fields automatically.
- Alternatively, you can place the data in the **TABLES** structure directly when the database is accessed, using the **INTO CORRESPONDING FIELDS** addition.

## Example, Step 3: Defining Pushbuttons

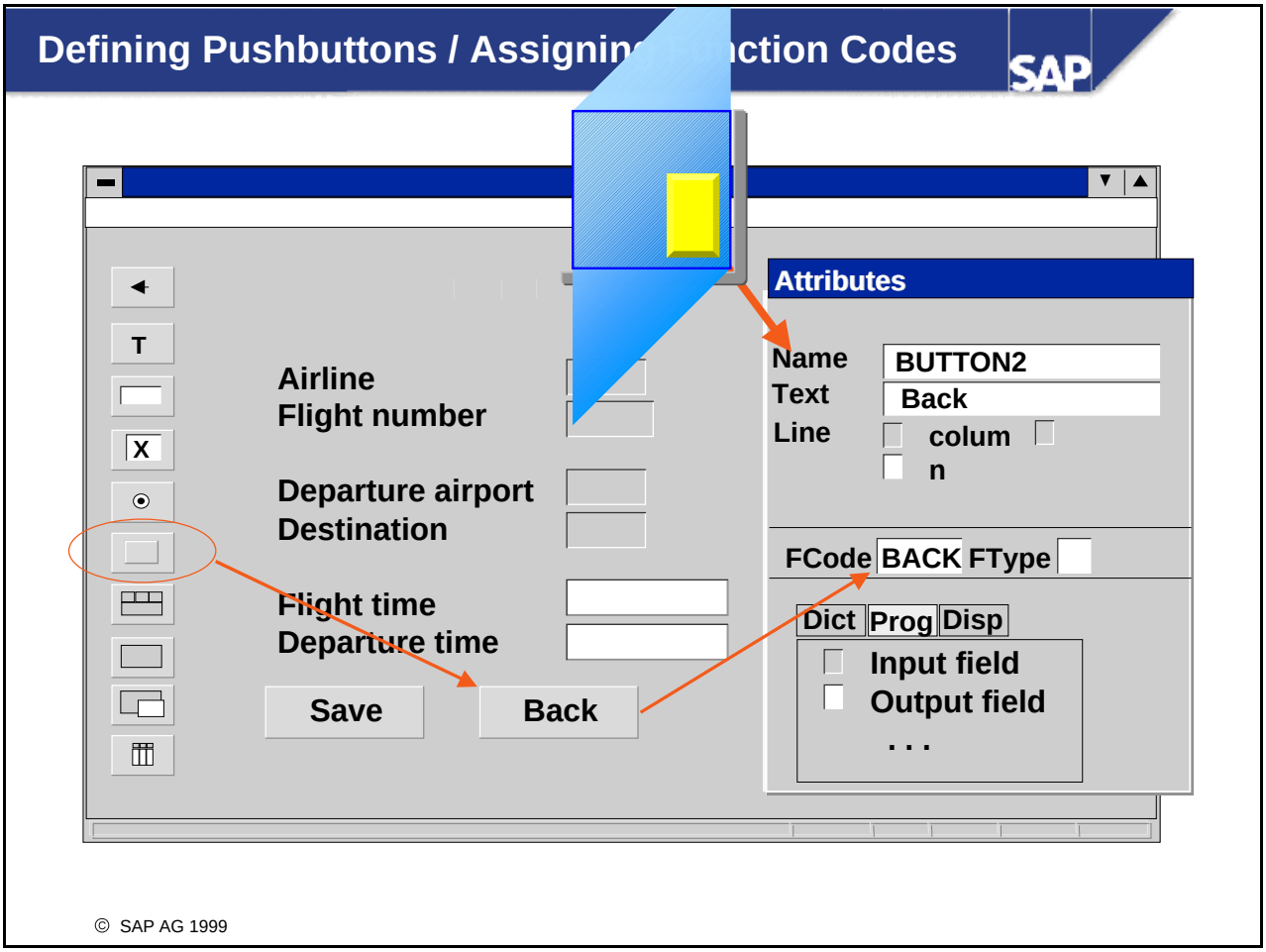



- In step three you will learn how to designate pushbutton functions. These functions allow different kinds of program logic to be processed according to user choice.
- For the user, the program works in the following manner:
  - When the user double-clicks a line in the basic list, the system displays a new screen. he most important bits of information for the connection he or she has chosen are displayed on this screen. The flight time and departure time can be changed.
  - When the user chooses the *Back* pushbutton, he or she returns to the basic list without writing any changes to the database. The message 'Screen was left without any changes being made' is displayed in the status bar of the basic list.
  - When the user chooses *Save,* the system writes all his or her changes to the database. We will take a closer look at this step in the unit *Database Dialogs*. In the following section, the pushbutton has already been prepared. After choosing the pushbutton, the user should return to the basic list and the sysetm should display a message in the status bar.
  - After pressing *Enter*, the screen is displayed again.
- Changes to the database are discussed in the unit *Database Dialogs.*
- With this in mind, this part of the unit deals with:
  - Flow logic in PBO and PAI event blocks

- Using PBO and PAI modules as ABAP processing blocks for screen programming
- How to control how the program continues according to the pushbutton chosen by the user.

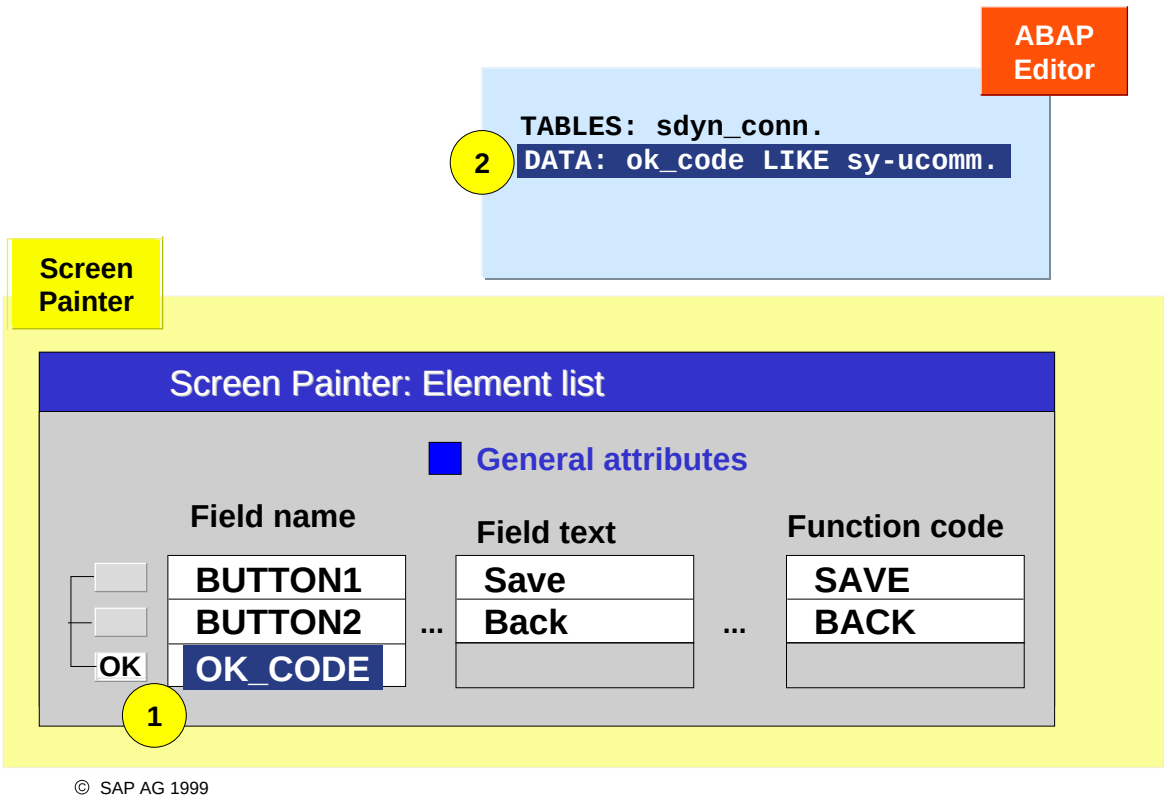



- If the user chooses a pushbutton, the runtime system copies the associated function code to a special program data object  This data object is usually called the **ok_code**. The content of this **ok_code** field is then evaluated in an ABAP processing block, which allows you to create a program flow that depends on the user's actions.  The following slides deal with: how you declare the **ok_code** field; how you create pushbuttons and assign function codes to them; and how you can, for example, change the sequence of screens depending on the user's actions.



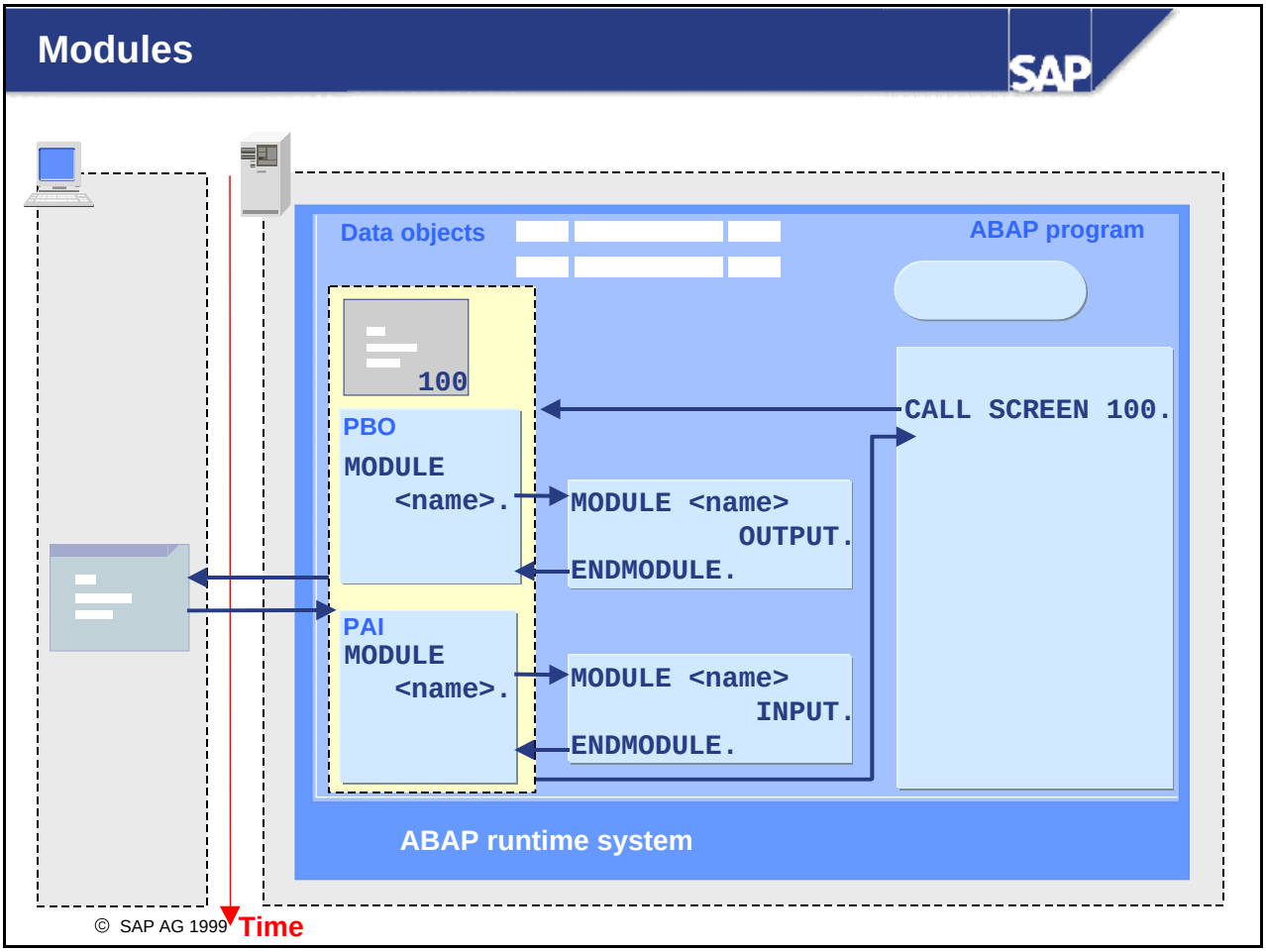

- To define functions for specific pushbuttons, you must assign them to function codes first. You can do this either on the attributes screen or in the field list in the graphical Layout Editor.

# Making the Command Field Usable



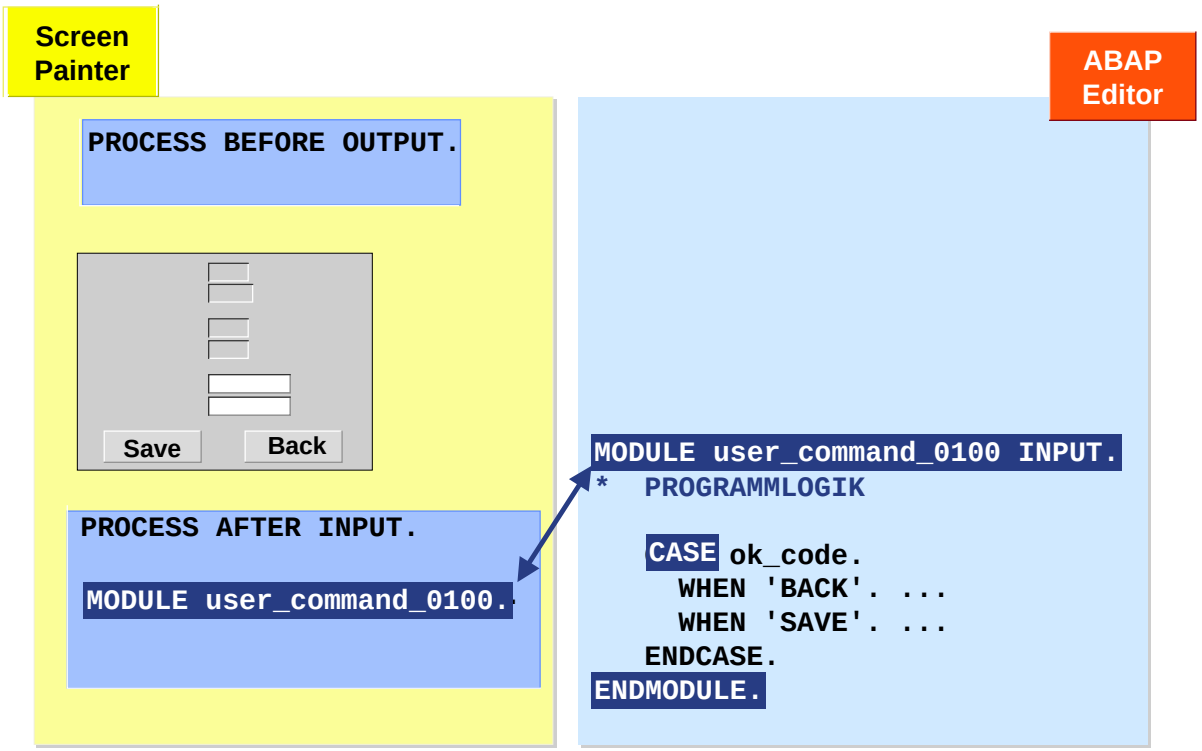



- The **OK_CODE** field is a data object which the system fills with the corresponding function codes after every user action.
- The name OK_CODE must be inserted as the last line in **every screen's** field list. Generally, you use the name **OK_CODE**.
- If you define a corresponding data object of the same name in a program's declaration area, the system places the function code of the pushbutton chosen by the user in the data object at runtime. You can use field **sy-ucomm** as a reference field.



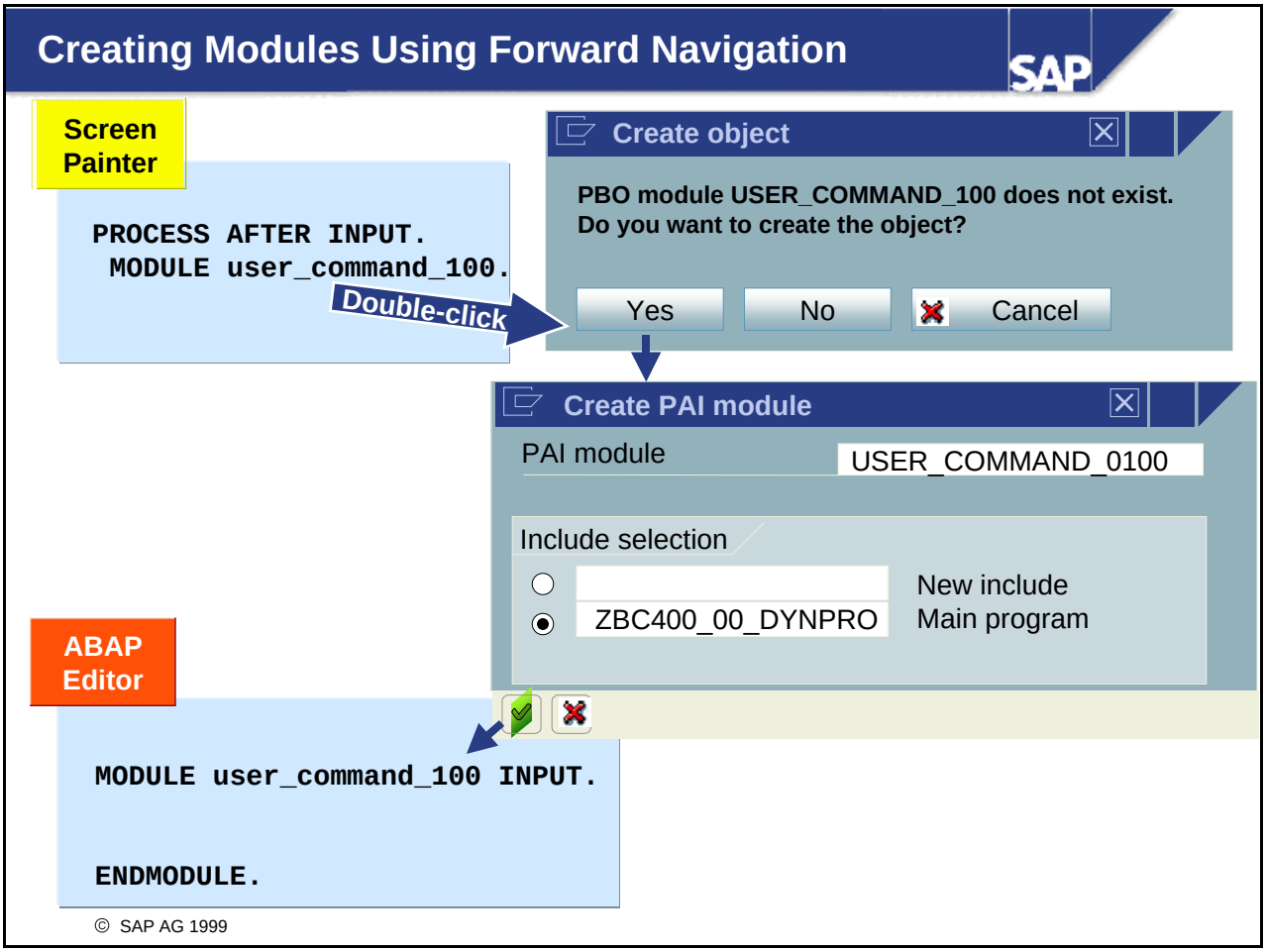

- The ABAP statement **CALL SCREEN <nnnn>** interrupts processing block processing and calls a screen.
- Each screen has two corresponding event blocks:
  - **PROCESS BEFORE OUTPUT (PBO)** is processed immediately before a screen is displayed. At this time modules are called that take care of tasks such as inserting recommended values into input fields.
  - **PROCESS AFTER INPUT (PAI)** is processed immediately after a user action. All program logic that is influenced by user action must be processed at **PAI.**
- Note: The code for the events **PBO** and **PAI** is written using the **Screen Painter** and **not** the ABAP Editor. These two event blocks make up a screen's **flow logic**.
  When programming flow logic, use the set of commands called Screen ABAP. **MODULE** <ABAP module name> is the most important Screen ABAP command. It calls a special ABAP processing block called a **module**.
- **Modules** are ABAP processing blocks with no interface that can only be called from within a program's flow logic. Modules begin with the ABAP statement **MODULE** and end at **ENDMODULE**.
- Program logic that logically belongs to a specific screen should normally be processed at the screen's **PBO** and **PAI** events.

## The user_command_<nnnn> PAI Module



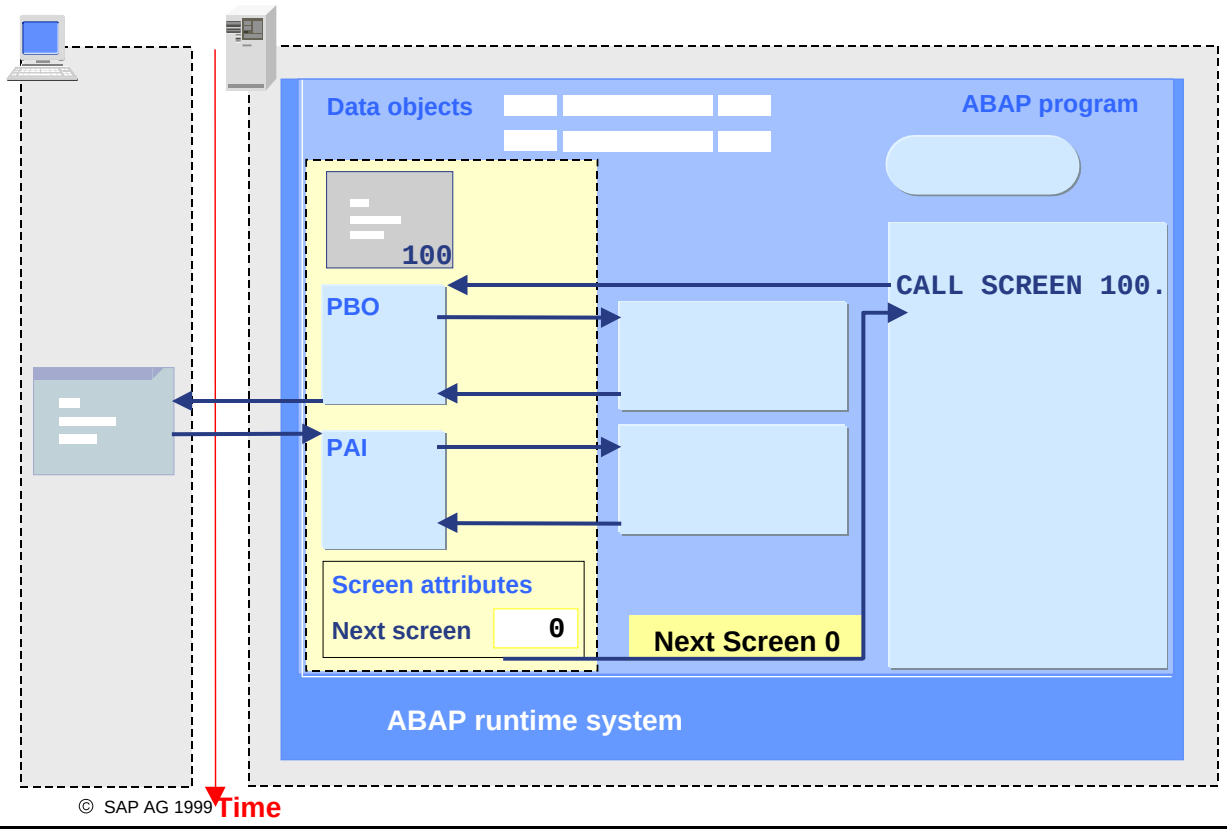



- You evaluate user actions in a PAI module, usually called the **user_command_<nnnn>** (where **<nnnn>** is the screen number). To do this, you evaluate the function code in the command field.
- Note: You cannot include any ABAP statements in the flow logic. Instead, you must call an ABAP module using the **MODULE** statement. For historical reasons, modules have no interface and no local variables. You can access all the ABAP program's global data in modules.

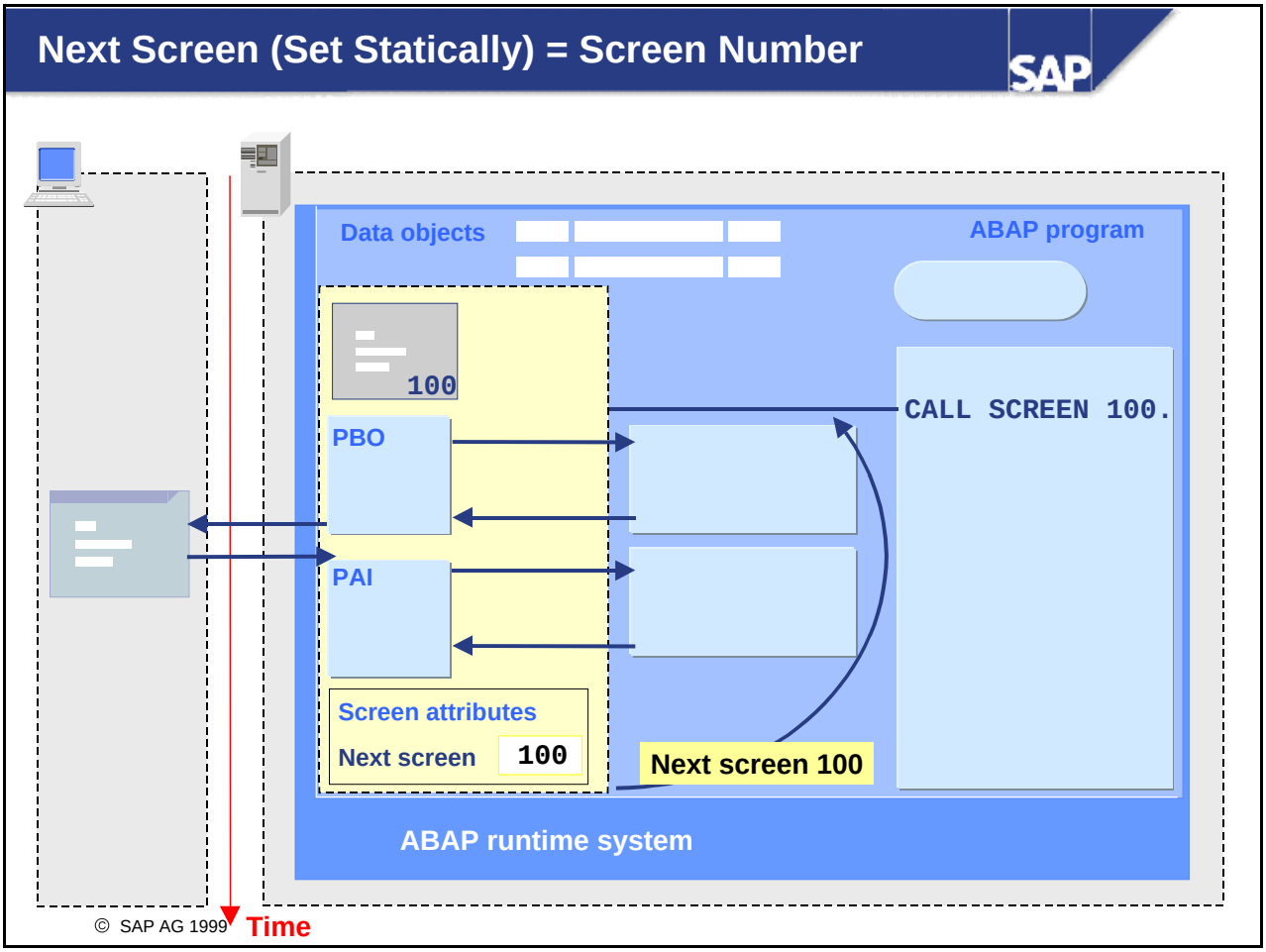



- You can implement calls such as MODULE within a screen's flow controls (PBO and PAI events). The modules themselves are, however, created using ABAP.
- There are two ways to create a module:
  - Using **forward navigation**: Double-click the module name from within the Screen Painter Editor to create the module.
  - Using the **Object Navigator**: If you want to create a module using the object list in the Object Navigator, first display your program, then choose 'PBO module' or 'PAI module' in the *ProgramObjects* display and create a new development object by selecting the *create* icon.
- A module can be called from more than one screen. (Reusability)
- Be aware that modules called at PBO events must be defined using the **MODULE** ... **OUTPUT** statements whereas modules defined using **MODULE ... INPUT** are defined at PAI.

## Next Screen (Set Statically) = 0

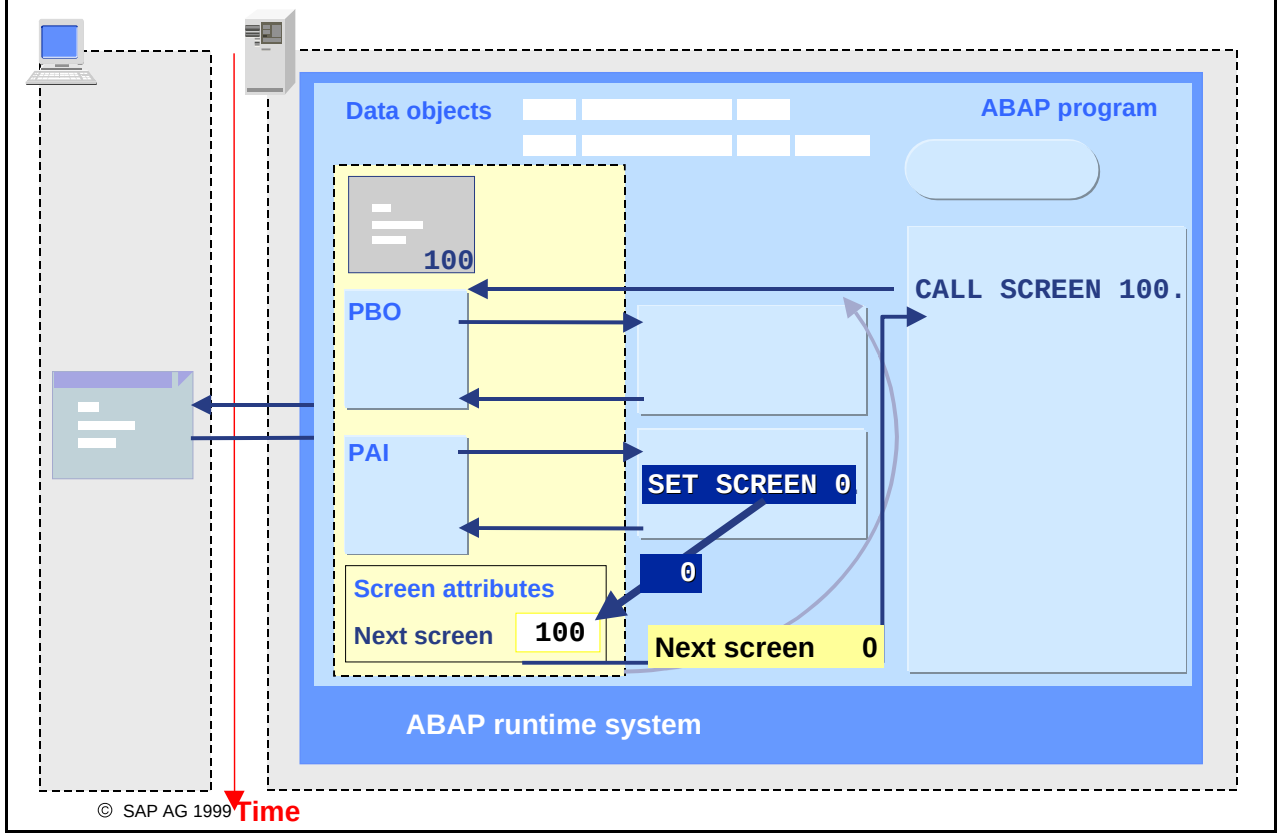



- If you enter 0 or leave the *Next Screen* field blank, the system first processes your screen completely and then carries on processing the program from where the screen was called.




- If you set the *Next screen* of screen 100 to 100, the system processes the screen again, after it has finished processing the **PAI** module.

## Setting the Next Screen Dynamically



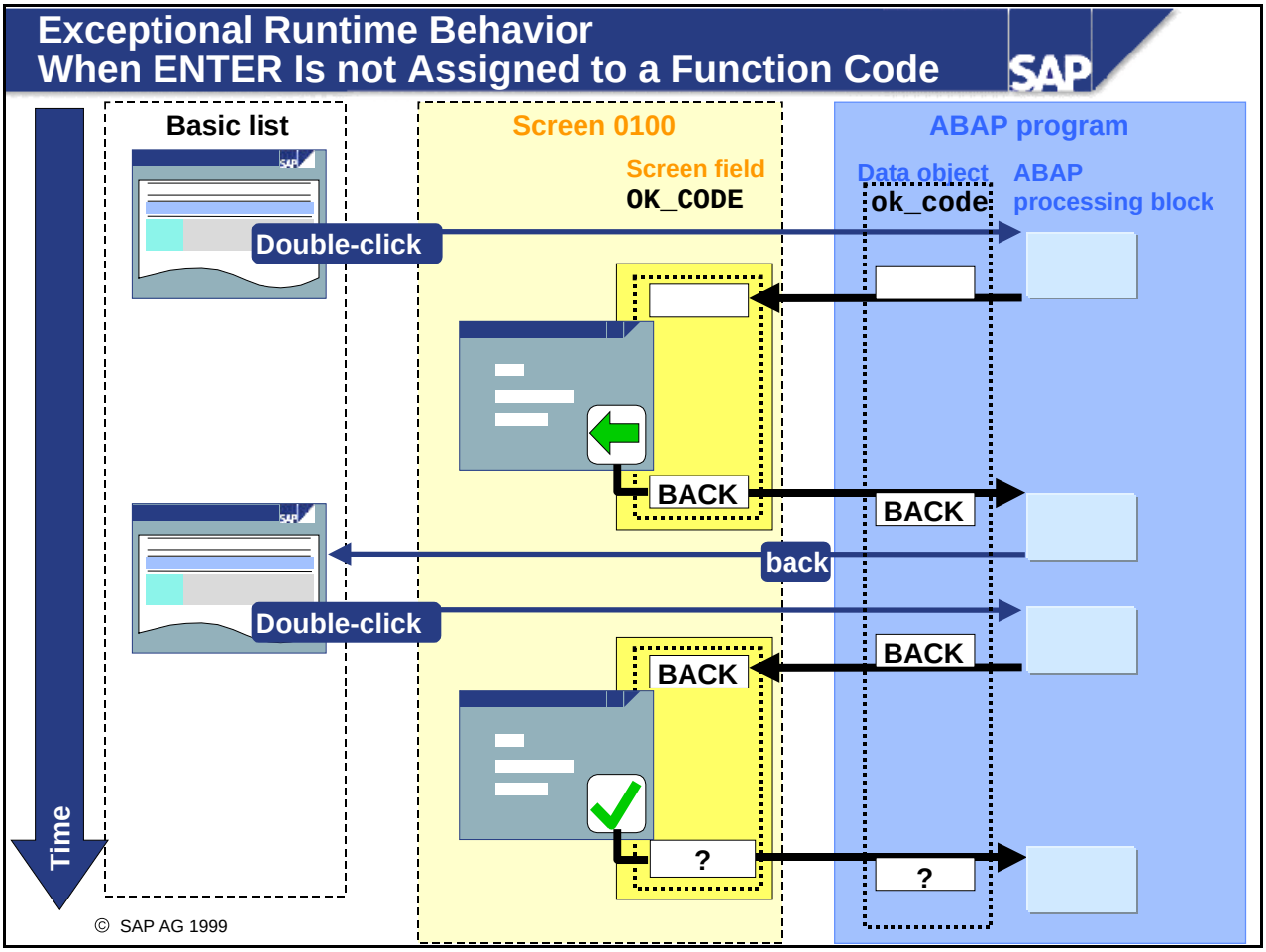

- You can use the ABAP statement SET SCREEN <nnnn> within a PAI module to set the *Next screen* value automatically.
- Often the same screen number is entered in both the *Screen number* and *Next screen* fields. In this case, when you choose *Enter,* a field check is performed and the system returns you to the same screen. In order to leave the screen, an appropriate pushbutton must be defined that then triggers a *Next screen* change within the PAI module.
- Note that, if the system processes the same screen again, it also runs through all the PBO modules again. If you decide to fill the TABLES structure by means of a PBO module, you must make sure that you do not overwrite changes that the user has made on screen to the data, if the module gets called twice.

## Syntax Example: The user_command_100 Module

```
DATA: ok_code LIKE sy-ucomm.


 .
 .

MODULE USER_COMMAND_100 INPUT.
CASE ok_code.
    WHEN 'BACK'.
       SET SCREEN 0.
       MESSAGE ID 'BC400' TYPE 'S' NUMBER '057'.
    WHEN 'SAVE'.
* Calling a function module to save changes is left out for
* didactical reasons until chapter 'Database Dialogs 2'
       SET SCREEN 0.
       MESSAGE ID 'BC400' TYPE 'S' NUMBER '058'.
 ENDCASE.
ENDMODULE.
```

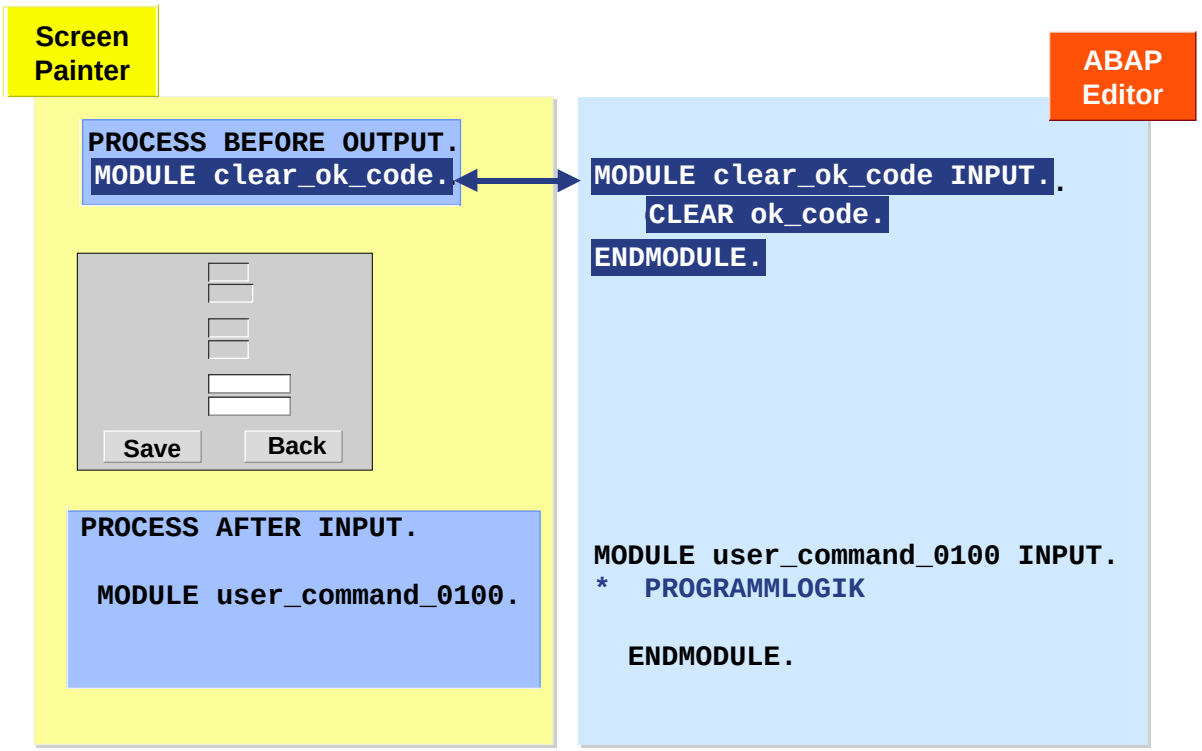


- In this example program two pushbuttons should trigger changes in the *Next screen* value:
  - Choosing '**BACK**' should automatically set this value to 0. This sends the user back to the last screen called before the present one. In your sample program, you return to a basic list if the detail list buffer has not been filled, or, if it has been filled, a detail list is displayed. Message 057 appears in the status bar of the screen subsequently displayed.
  - Choosing **'SAVE'** causes an S message to be displayed and the system then displays a basic list or a detail list, the same as when the user chooses **'BACK'**. We will write these changes to the database in the unit on *Database Dialogs II.*




- At runtime, the behavior is as follows:
- The user starts the program and double-clicks to display detailed information on the screen. It is clear that all the data is correct so he or she returns to the basic list by choosing the green arrow.
- The system stores the function code **'BACK'**, assigned to the pushbutton, in the command field. This function code is then evaluated in a PAI module. The next screen is set to 0. Then **AT LINE-SELECTION** is processed further. If there is no **WRITE** statement, the system displays the basic list again.
- The user then displays details for another record by double-clicking it. Automatic field transport copies the data object, **ok_code**, to the identically-named screen field and the appropriate screen is displayed.
- If the user now chooses ENTER, he or she should once again branch to that screen. However, ENTER has not been assigned to a function code so the command field has not been over-written. The function code **BACK** remains in the command field and is copied to the program command field at the beginning of he PAI event. This function code is then evaluated in a PAI module. Consequently (as described above) the system goes back to the basic list, instead of re-displaying the screen as the user expected.

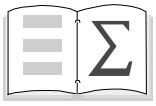
## Possible Solution: Deleting the Command Field in a PBO Module






- If the command field (the OK_CODE) field is not initialized, errors can occur, since the system does not force you to assign a function code to every pushbutton. There are two ways of initializing the command field:
  - In a PBO module. Then it is set to the initial value at PAI, unless the user has carried out a user action to which a function code is assigned. In this case, the **OK_CODE** field contains the function code.
  - Use an auxiliary field and copy the contents of the **OK_CODE** field to the auxiliary field in a PAI module, and then initialize the **OK_CODE** field. In this case, the auxiliary field must be queried in the PAI module for the function code evaluation.

# Screens: Unit Summary



**You are now able to:**

- **Describe screen attributes and strengths**
- **Write a program that:**
  - **Displays data on a screen**
  - **Allows the user to change some of that data**
  - **Allows the user to influence further program processing using pushbuttons**

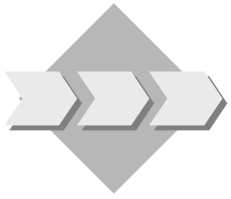
Exercises

**Unit: Screens**

**Topic: Creating Screens**

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

- Create screens
- Call existing from the program



Program **SAPBC400UDT_DYNPRO_A** displays all bookings made by one agency as a list.
Extend the program as follows:
Double-clicking on a line in the basic list should call a screen. This screen should contain input fields for specific booking data that is not displayed on the list. This screen should also contain output fields for booking information that is already displayed on the list. Any user action should result in the basic list being displayed again

**Program:** **ZBC400_##_DYNPRO**

**Model solution:** **SAPBC400UDS_DYNPRO_A**

**Template:** **SAPBC400UDT_DYNPRO_A**

1-1 Copy the template SAPBC400UD**T**_DYNPRO_1 to your program **ZBC400_##_DYNPRO**. Assign the program to **development class ZBC400_##** and the change request for the project "BC400…" (replacing ## with your group number).

1-2 Become familiar with the program. Test the program using the agency number 1## (**##** is your group number).

1-3 Selecting a line on the basic list (by double-clicking or using F2) should call a screen. Create this screen (screen number 100) using forward navigation.

1-4 For the attributes, assign screen number 0 as the number of the next screen, so that after any user action on screen 100, the user returns to the basic list.

1-5 Create input/output fields on the screen. When you are assigning field types, refer to ABAP Dictionary structure **SDYN_BOOK**.

- The booking table key fields **CARRID**, **CONNID**, **FLDATE**, and **BOOKID** should be copied with their field labels.

- The customer name **NAME** should be copied without a field label and be displayed next to the customer number.

- The fields **CUSTOMID**, **CUSTTYPE**, **SMOKER**, **CLASS**, **LOCCURAM**, and **LOCCURKEY** should be copied with field labels.

1-6 Maintain the screen field attributes:

- Fields **CARRID**, **CONNID**, **FLDATE**, **BOOKID** and **CUSTOMID** should be displayed as output fields (*Output field* attribute).

- The customer name **NAME** should be displayed next to the customer number without text (*Output only* attribute).

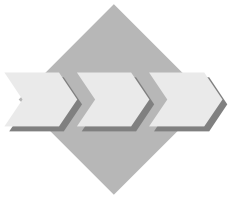
- The fields **CUSTOMID CUSTTYPE**, **SMOKER**, **CLASS**, **LOCCURAM** and **LOCCURKEY** are input/output fields (*Input field/Output field* attribute).

**Unit: Screens**

**Topic: Data transport**

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

- Fill the screen fields with data from the program



Extend your program, **ZBC400_##_ DYNPRO**:
Double-clicking on a line of the basic list displays details of the selected booking on the screen. If the user changes data on the screen, then these changes should be available in the program once the user has left the screen.

**Program:** **ZBC400_##_DYNPRO**

**Model solution:** **SAPBC400UDS_DYNPRO_B**

2-1 Extend your program, **ZBC400_##_DYNPRO**, or copy the relevant model solution **SAPBC400UDS_DYNPRO_A** and give it the name **ZBC400_##_DYNPRO_B**. Assign your program to the development class **ZBC400_##** and to the transport request for this project, BC400… (replacing ## with your group number).

2-2 Use a work area as an interface between the program and the screen. Since you used a reference to a Dictionary structure type when assigning screen field types, you must use the **TABLES** declarative statement.

2-3 Ensure that the **SBOOK** database table key fields and the customer name are still available (**HIDE:** ...) in the **AT LINE-SELECTION** event block after a line has been selected on the basic list (double click or F2).

2-4 You should then extend your program so that data can be changed on the database. Ensure that the screen can only be processed if the user has change authorization for the airline selected.
In order to ensure that double-clicking a line in the basic list displays up-to-date data, the data record must be read from the database table **SBOOK** before the screen is processed. Therefore, before calling the screen, copy the most recent data for the booking selected from the database table **SBOOK** to a structure that has the same line structure as the database

table. If the data record cannot be read, the system must display information message 176 from message class **BC400**. If the record is successfully read, call the screen.

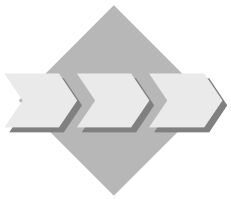
2-5 Immediately before calling the screen, copy the relevant data to the **TABLES** work area that serves as an interface to the screen.

**Unit: Screens**

**Topic: Field Transports and Next Screen Processing**

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

- Create pushbuttons on screens
- Process the system code triggered when the user clicks on a pushbutton and control the program flow
- Set the next screen dynamically



Extend your program, **ZBC400_##_ DYNPRO**:
The user should be given a choice of two pushbuttons on the screen that control the program flow.

**Program:** **ZBC400_##_DYNPRO**

**Model solution:** **SAPBC400UDS_DYNPRO_C**

3-1 Extend your program, **ZBC400_##_DYNPRO**, or copy the relevant model solution **SAPBC400UDS_DYNPRO_B** and give it the name **ZBC400_##_DYNPRO_C**. Assign your program to the development class **ZBC400_##** and the task that has already been created for you (replacing ## with your group number).

3-2 Define two pushbuttons on the screen that allow the user to either return to the basic list (**PUSH_BACK**) or to save changes to data (**PUSH_SAVE**):

| Name of pushbutton | **Text** | Function code |
|---|---|---|
| PUSH_BACK | Back | **BACK** |
| PUSH_SAVE | *Save*<br>or icon<br>ICON_SYSTEM_SAVE | **SAVE** |

3-3 Name the **OK_CODE** field on the screen and define a data object of the same name (and corresponding type) in the program.

3-4 Navigate in the flow logic. Create a module for function code processing (using forward navigation) at **PROCESS AFTER INPUT**:

| Function code | **Action** | Next screen |
|---|---|---|
| BACK | None | List |
| SAVE | **First:**<br>Information message No.<br>060(BC400) | List |
| Other | None | Screen 100 |

3-5 Ensure that pressing 'Enter' always displays screen 100, regardless of the navigation history. Initialize the **OK_CODE** field in a PBO module.

Solutions

**Unit: Screens**

**Topic: Creating Screens**

**Model solution: Program SAPBC400UDS_DYNPRO_A**

```
*&---------------------------------------------------------------------*
*& Report        SAPBC400UDS_DYNPRO_A                                  *
*&                                                                     *
*&              Define and Call a Screen                               *
*&---------------------------------------------------------------------*

REPORT  sapbc400uds_dynpro_a.
CONSTANTS actvt_display TYPE activ_auth VALUE '03'.

*  Definition of selection screen
PARAMETERS pa_anum TYPE sbook-agencynum.


* workarea for select
DATA: wa_booking TYPE sbc400_booking.


START-OF-SELECTION.

* selecting data using a dictionary view to get the data from sbook and
* the customer name from scustom
  SELECT carrid connid fldate bookid customid name
        FROM sbc400_booking
        INTO CORRESPONDING FIELDS OF wa_booking
          WHERE agencynum = pa_anum.

   AUTHORITY-CHECK OBJECT 'S_CARRID'
      ID 'CARRID' FIELD wa_booking-carrid
```

```
      ID 'ACTVT'  FIELD actvt_display.
   IF sy-subrc = 0.
*Output
     WRITE: / wa_booking-carrid,
              wa_booking-connid,
              wa_booking-fldate,
              wa_booking-bookid,
              wa_booking-name.

   ENDIF.
 ENDSELECT.

AT LINE-SELECTION.
 CALL SCREEN 100.
```

1-3 Create a screen using forward navigation
(double-click “100” in the **CALL SCREEN 100** statement).

1-4 Maintain screen attributes
Enter a descriptive short text
Set the next screen to 0

1-5 Layout
- Navigate to the graphical Layout Editor
Choose ***Dict/Program fields***
- Enter **SDYN_BOOK**
Choose the ***Get from Dictionary*** icon
- Check the fields you want,
choose *Enter* to confirm, and drag these fields to the screen

Block 1
**Key fields: CARRID, CONNID, FLDATE** and **BOOKID**
Copy with field names
Block 2
**Customer name NAME**
Copy without a field name (choose the *Without text* radio button)
Block 3
Copy the fields: **CUSTOMID, CUSTTYPE, SMOKER, CLASS,**

**LOCCURAM**, and **LOCCURKEY**

with field names

1-6 Change the field attributes -

for example, by double-clicking the input field

- Adapt the short texts:

The fields **CARRID, CONNID, FLDATE**, **BOOKID**, and **CUSTOMID**

should be displayed as output fields (the *Output field*) attribute.

The customer name **NAME** should be displayed next to the customer number without text (*Output only* attribute).

The fields **CUSTTYPE**, **SMOKER**, **CLASS**, **LOCCURAM**, and

**LOCCURKEY** should both input-ready and output-ready (the *Input/Output field* attribute).

**Unit: Screens**

**Topic: Data transport**

**Model solution: Program SAPBC400UDS_DYNPRO_B**

```
*&---------------------------------------------------------------------*
*& Report        SAPBC400UDS_DYNPRO_B                                  *
*&                                                                     *
*&  Define and Call a Screen and display data on the screen           *
*&---------------------------------------------------------------------*

REPORT  sapbc400uds_dynpro_b.
CONSTANTS: actvt_display TYPE activ_auth VALUE '03',
           actvt_change TYPE activ_auth VALUE '02'.
*  Definition of selection screen
PARAMETERS pa_anum TYPE sbook-agencynum.


* workarea for list
DATA wa_booking TYPE sbc400_booking.
* workarea for single booking to be changed
DATA wa_sbook TYPE sbook.
* workarea for dynpro
TABLES sdyn_book.

START-OF-SELECTION.

* selecting data using a dictionary view to get the data from sbook and
* the customer name from scustom
  SELECT carrid connid fldate bookid customid name
         FROM sbc400_booking
         INTO CORRESPONDING FIELDS OF wa_booking
           WHERE agencynum = pa_anum.
```

```
   AUTHORITY-CHECK OBJECT 'S_CARRID'
      ID 'CARRID' FIELD wa_booking-carrid
      ID 'ACTVT'  FIELD actvt_display.
   IF sy-subrc = 0.
*Output
     WRITE: / wa_booking-carrid,
              wa_booking-connid,
              wa_booking-fldate,
              wa_booking-bookid,
              wa_booking-name.

     HIDE:    wa_booking-carrid,
              wa_booking-connid,
              wa_booking-fldate,
              wa_booking-bookid,
              wa_booking-name.


   ENDIF.
 ENDSELECT.

AT LINE-SELECTION.
  IF sy-lsind = 1.
    AUTHORITY-CHECK OBJECT 'S_CARRID'
       ID 'CARRID' FIELD wa_booking-carrid
        ID 'ACTVT'  FIELD actvt_change.
    IF sy-subrc = 0.

     SELECT SINGLE *
          FROM sbook
          INTO wa_sbook
          WHERE carrid = wa_booking-carrid
            AND connid = wa_booking-connid
            AND fldate = wa_booking-fldate
            AND bookid  = wa_booking-bookid.
     IF sy-subrc = 0.
       MOVE-CORRESPONDING wa_sbook TO sdyn_book.
       MOVE wa_booking-name TO sdyn_book-name.
```

```
      CALL SCREEN 100.
    ENDIF.
  ELSE .
    MESSAGE ID 'BC400' TYPE 'S' NUMBER '047' WITH wa_booking-carrid.
  ENDIF.
ENDIF.
```

```
* INCLUDE bc400uds_dynpro_bo01.

* INCLUDE bc400uds_dynpro_bi01.
```

**Unit: Screens**

**Topic: Field Transports and Next Screen Processing**

**Model solution: Program SAPBC400UDS_DYNPRO_C**

```
*&---------------------------------------------------------------------*
*& Report        SAPBC400UDS_DYNPRO_C                                  *
*&                                                                     *
*&  Define and Call a Screen and display data on the screen           *
*&---------------------------------------------------------------------*

REPORT  sapbc400uds_dynpro_c.
CONSTANTS: actvt_display TYPE activ_auth VALUE '03',
           actvt_change TYPE activ_auth VALUE '02'.
*  Definition of selection screen
PARAMETERS pa_anum TYPE sbook-agencynum.


* workarea for list
DATA wa_booking TYPE sbc400_booking.
* workarea for single booking to be changed
DATA wa_sbook TYPE sbook.
* workarea for dynpro
TABLES sdyn_book.
* variable for function code of user action
DATA ok_code LIKE sy-ucomm.

START-OF-SELECTION.

* selecting data using a dictionary view to get the data from sbook and
* the customer name from scustom
  SELECT carrid connid fldate bookid customid name
         FROM sbc400_booking
```

```
        INTO CORRESPONDING FIELDS OF wa_booking
         WHERE agencynum = pa_anum.

   AUTHORITY-CHECK OBJECT 'S_CARRID'
      ID 'CARRID' FIELD wa_booking-carrid
      ID 'ACTVT'  FIELD actvt_display.
   IF sy-subrc = 0.
*Output
     WRITE: / wa_booking-carrid,
              wa_booking-connid,
              wa_booking-fldate,
              wa_booking-bookid,
              wa_booking-name.

     HIDE:    wa_booking-carrid,
              wa_booking-connid,
              wa_booking-fldate,
              wa_booking-bookid,
              wa_booking-name.



   ENDIF.
  ENDSELECT.
  CLEAR wa_booking.

AT LINE-SELECTION.
  IF sy-lsind = 1.
    AUTHORITY-CHECK OBJECT 'S_CARRID'
       ID 'CARRID' FIELD wa_booking-carrid
       ID 'ACTVT'  FIELD actvt_change.
    IF sy-subrc = 0.

     SELECT SINGLE *
           FROM sbook
           INTO wa_sbook
           WHERE carrid = wa_booking-carrid
             AND connid = wa_booking-connid
             AND fldate = wa_booking-fldate
             AND bookid  = wa_booking-bookid.
```

```
    IF sy-subrc = 0.
      MOVE-CORRESPONDING wa_sbook TO sdyn_book.
      MOVE wa_booking-name TO sdyn_book-name.
      CALL SCREEN 100.
    ENDIF.
  ELSE .
    MESSAGE ID 'BC400' TYPE 'S' NUMBER '047' WITH wa_booking-carrid.
  ENDIF.
ENDIF.
CLEAR: wa_sbook, wa_booking.

INCLUDE bc400uds_dynpro_co01.
INCLUDE bc400uds_dynpro_ci01.
```

```
*&---------------------------------------------------------------------*
*& INCLUDE BC400UDS_DYNPRO_CO01 .                                      *
*&---------------------------------------------------------------------*
*&      Module  CLEAR_OK_CODE  OUTPUT                                  *
*&---------------------------------------------------------------------*
*&      text                                                           *
*&---------------------------------------------------------------------*
module clear_ok_code output.
clear ok_code.
endmodule.                 " CLEAR_OK_CODE  OUTPUT
```

```
*&---------------------------------------------------------------------*
*& INCLUDE BC400UDS_DYNPRO_CI01 .                                      *
*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT                               *
*&---------------------------------------------------------------------*
*&      text                                                           *
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.

  CASE ok_code.
    WHEN 'BACK'.
      LEAVE TO SCREEN 0.
    WHEN 'SAVE'.
      MOVE-CORRESPONDING sdyn_book TO wa_sbook.
      MESSAGE ID 'BC400' TYPE 'I' NUMBER '060'.
      LEAVE TO SCREEN 0.

  ENDCASE.
ENDMODULE.                          " USER_COMMAND_0100  INPUT
```

# Interfaces

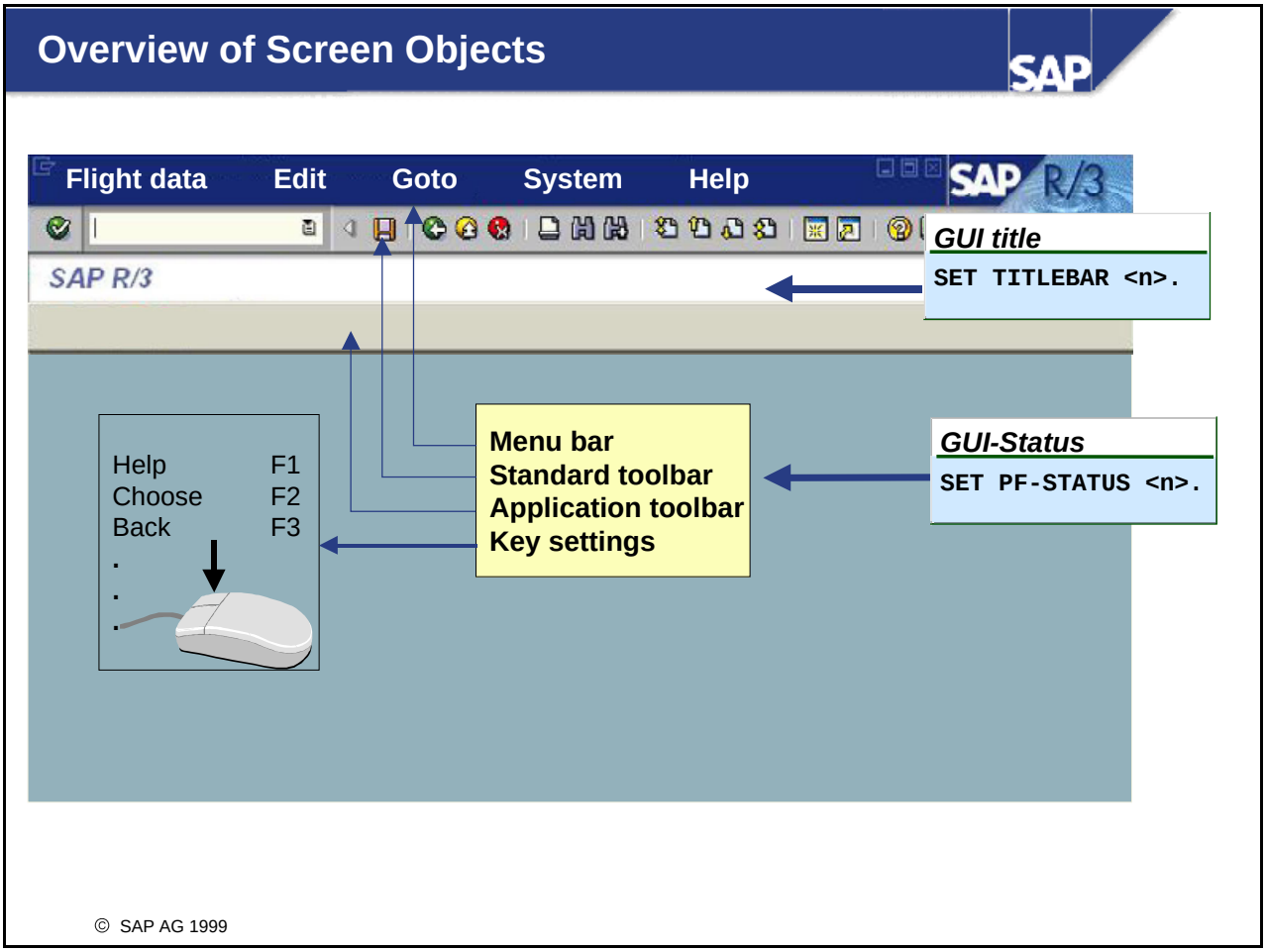
## Contents:

- Creating interfaces using the Menu Painter
  - Title
  - Menu bar
  - Standard toolbars
  - Application toolbars

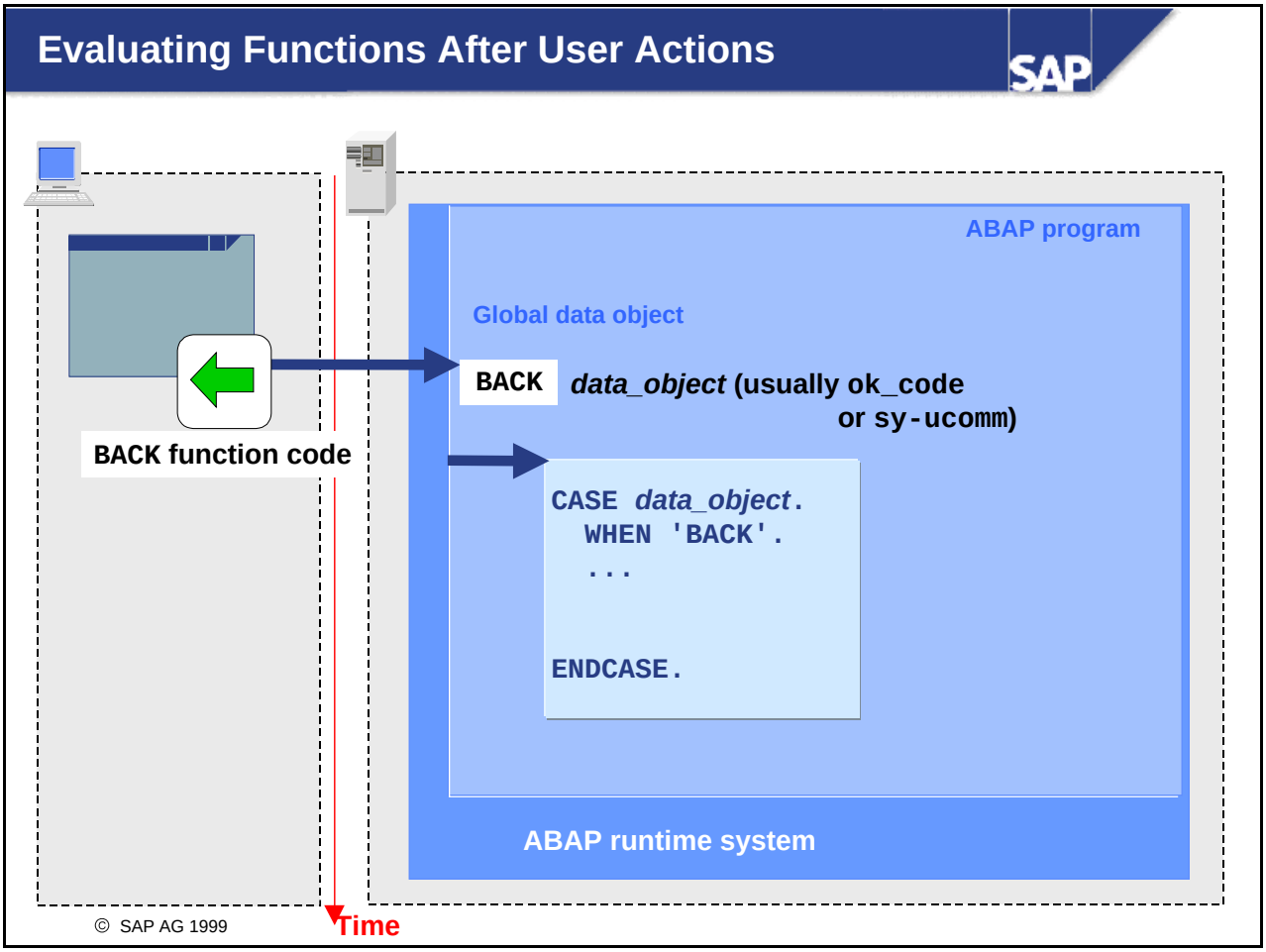
# Interfaces: Unit Objectives

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

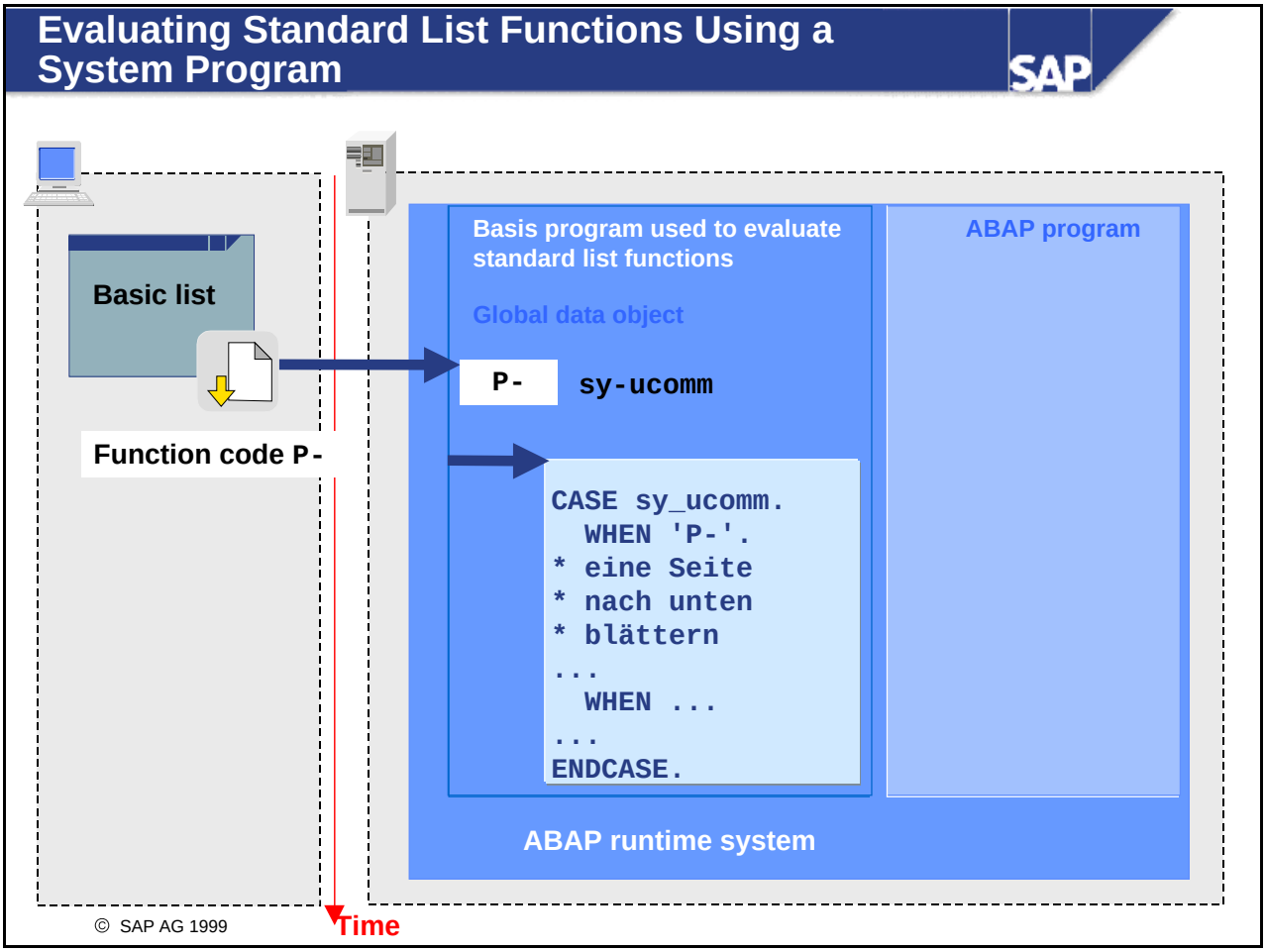
- **Create a GUI title**
- **Create GUI statuses for lists and screens that contain the following:**
  - **Menu bars**
  - **Standard toolbars**
  - **Application toolbars**
  - **Function key settings**




- All user interfaces include the following elements:
  - A **title bar** containing the title of the screen, selection screen, or list currently being displayed
  - A **menu bar** with expandable menus
  - **Menus** containing the executable functions for the current program. Menus can also contain submenus. The menus 'System' and 'Help' can be found on every screen in R/3 and always contain the same functions. Neither of these menus may be changed or hidden.
  - A **standard toolbar** containing icons for those functions most often used. The R/3 standard toolbar always contains the same icons with standard functions assigned to them. Those standard functions that cannot be accessed from a particular interface are grayed out.
  - **Function key settings**, which can be displayed by clicking on your right mouse button. Ideally, you should be able to execute all menu functions by way of function keys as well.
  - An **application toolbar** containing icons and pushbuttons for those functions most often used on the current screen.
- Each program is created with an interface containing all of the tools/objects listed above. Different views of this interface (GUI statuses) are then created for a program's individual screens, selection screens, and lists.

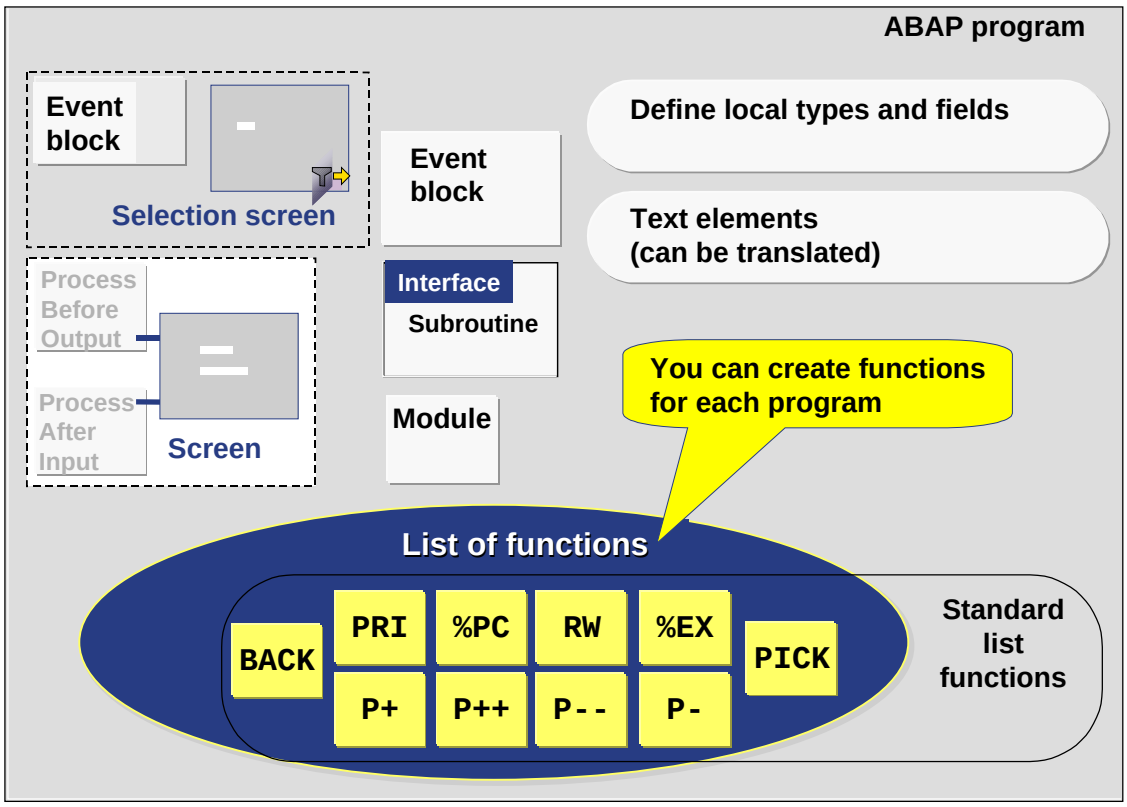



- The principles you learnt for processing screens also apply to lists and selection screens. und Dynpro.
  - A function code is assigned to each pushbutton and menu entry that the user can choose to trigger a user action.
  - When this user action is triggered, the function code is stored in the corresponding data object, known as the command field or ok_code field. In screen processing, you generally use a data object with the name ok_code. In list processing, you use the system field sy-ucomm (for **U**ser **COMM**and).
  - The runtime system then triggers sequential processing of an ABAP processing block that evaluates the command field.
- Apart from the traditional techniques for user dialogs, you can also use Controls technology. A great deal of the screen display logic is stored at the front end. It can be managed using methods of objects of global classes called from ABAP programs. When you use this technique, user actions can trigger events. If you have implemented a special method in your program to handle an event, then this event triggers the processing of this method.



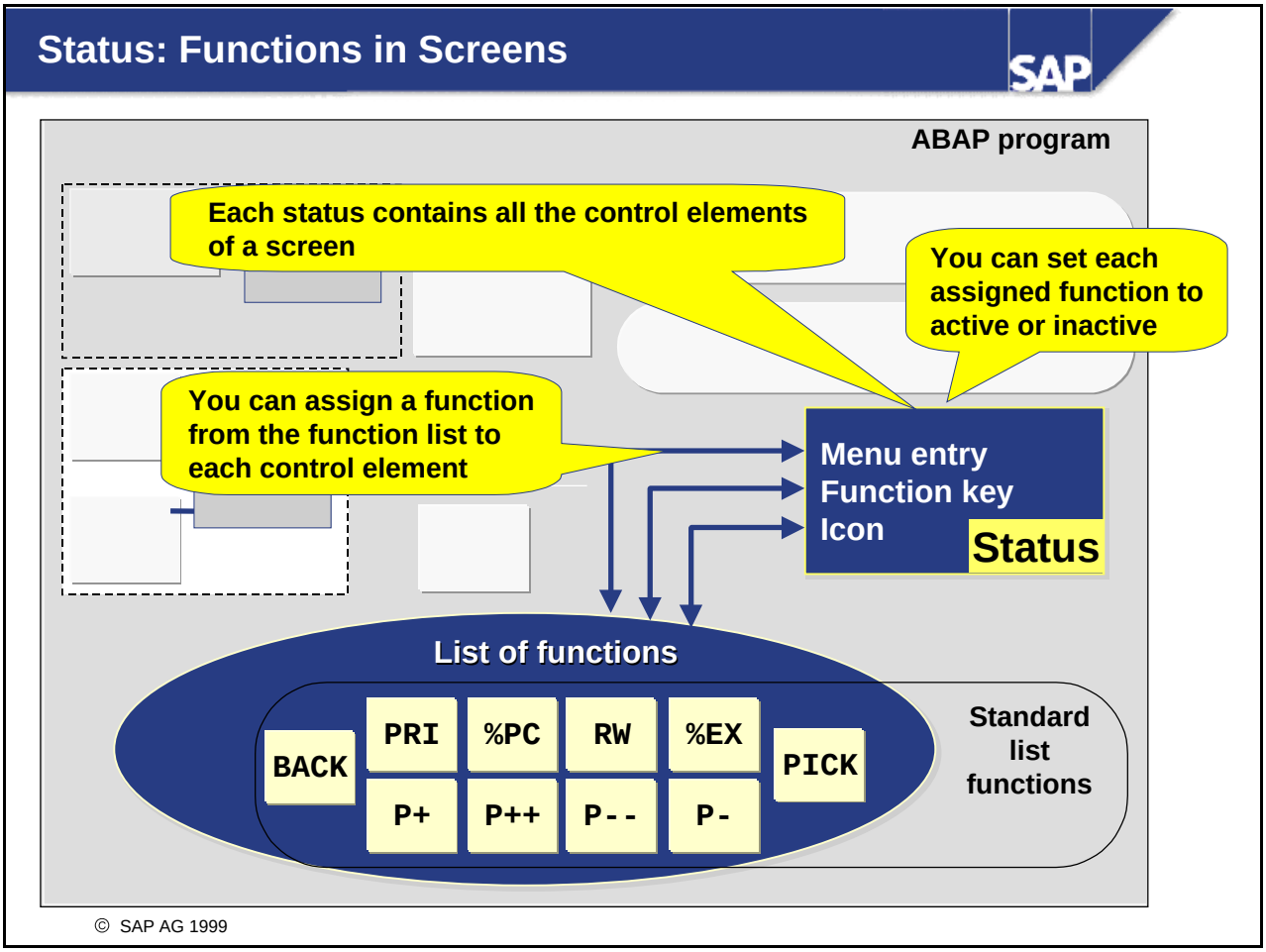

- The same principle applies as to processing standard list functions - system programs evaluate the command field sy-ucomm.
- If you want to extend the interface for a list by adding your own functions, you must ensure that the standard functions are assigned to the icons for the standard list functions. If this is not the case, the system will not perform the correct function when the user chooses the icon.

# Functions in ABAP Programs



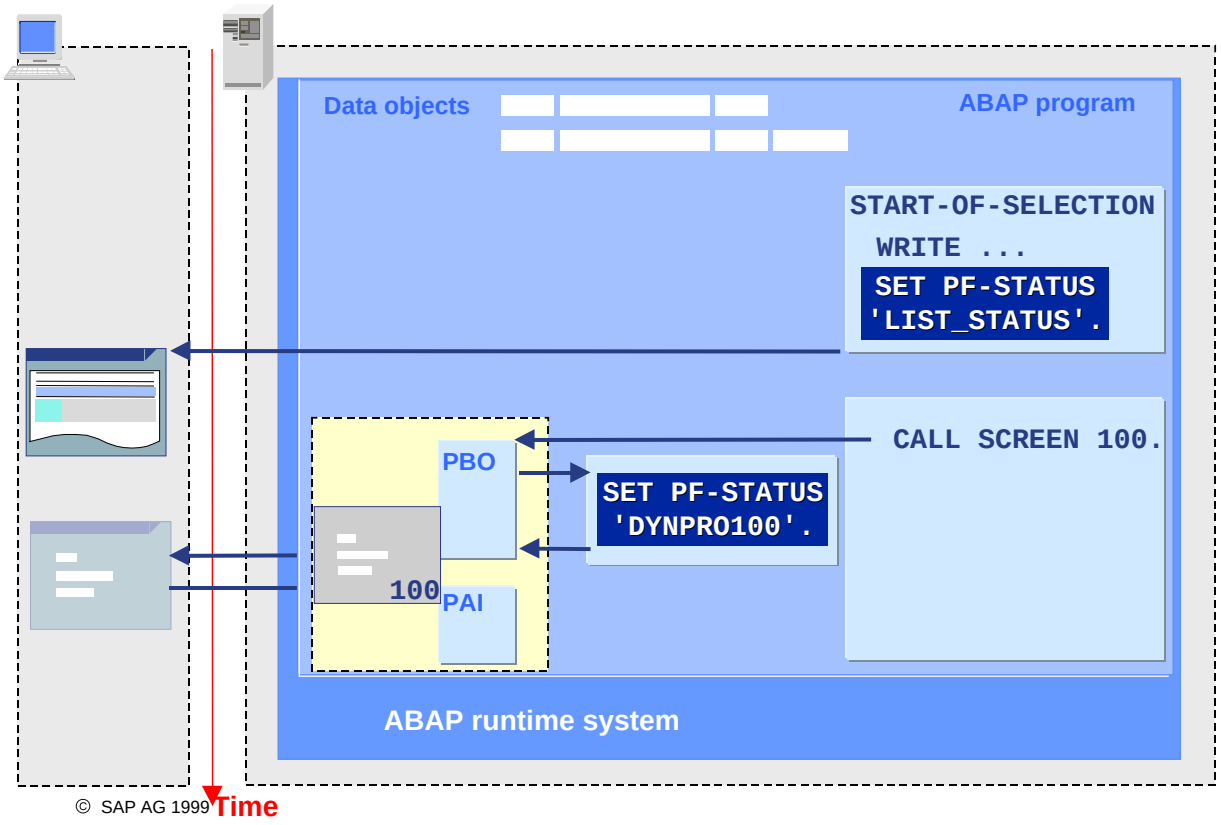



- In the program, you create each function that you want to make available - either as a menu function, or in a toolbar - as a "standalone" function. This also applies to the standard list functions, which you can generate automatically.

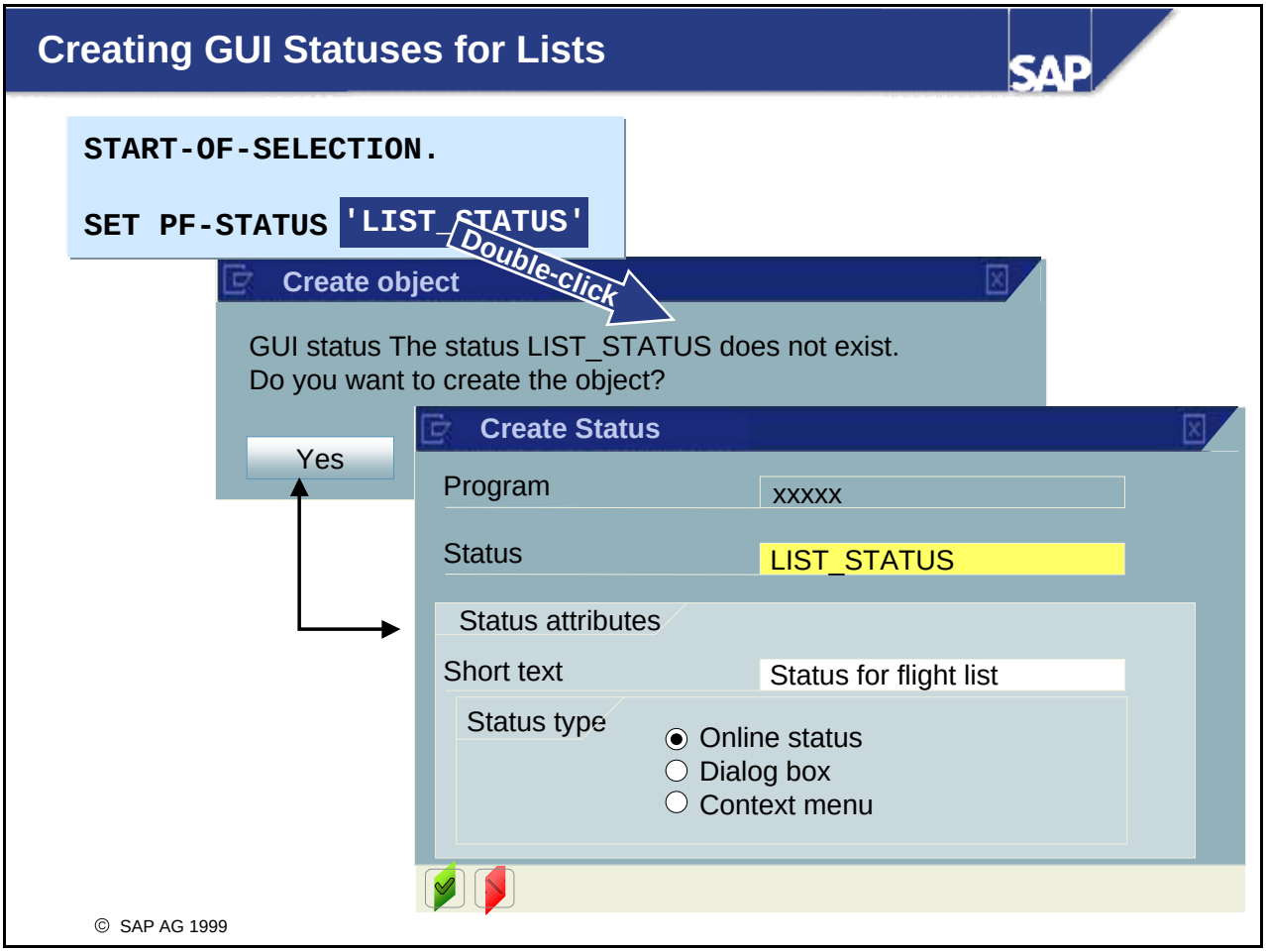



- A **status** is the actual form that a menu bar, a standard toolbar, and an application toolbar take on for a particular screen within your program. A status determines which functions are **active** or executable or **inactive** and not able to be executed for a particular screen. You can change a screens status at PBO: **Example:** If you use the ABAP Editor, which is itself written in ABAP, you can toggle between Display and Change mode. This both alters the background color of the ABAP source code and determines which menu functions are active. You implement this by changing the status dynamically.
- You can create several statuses
  for each program

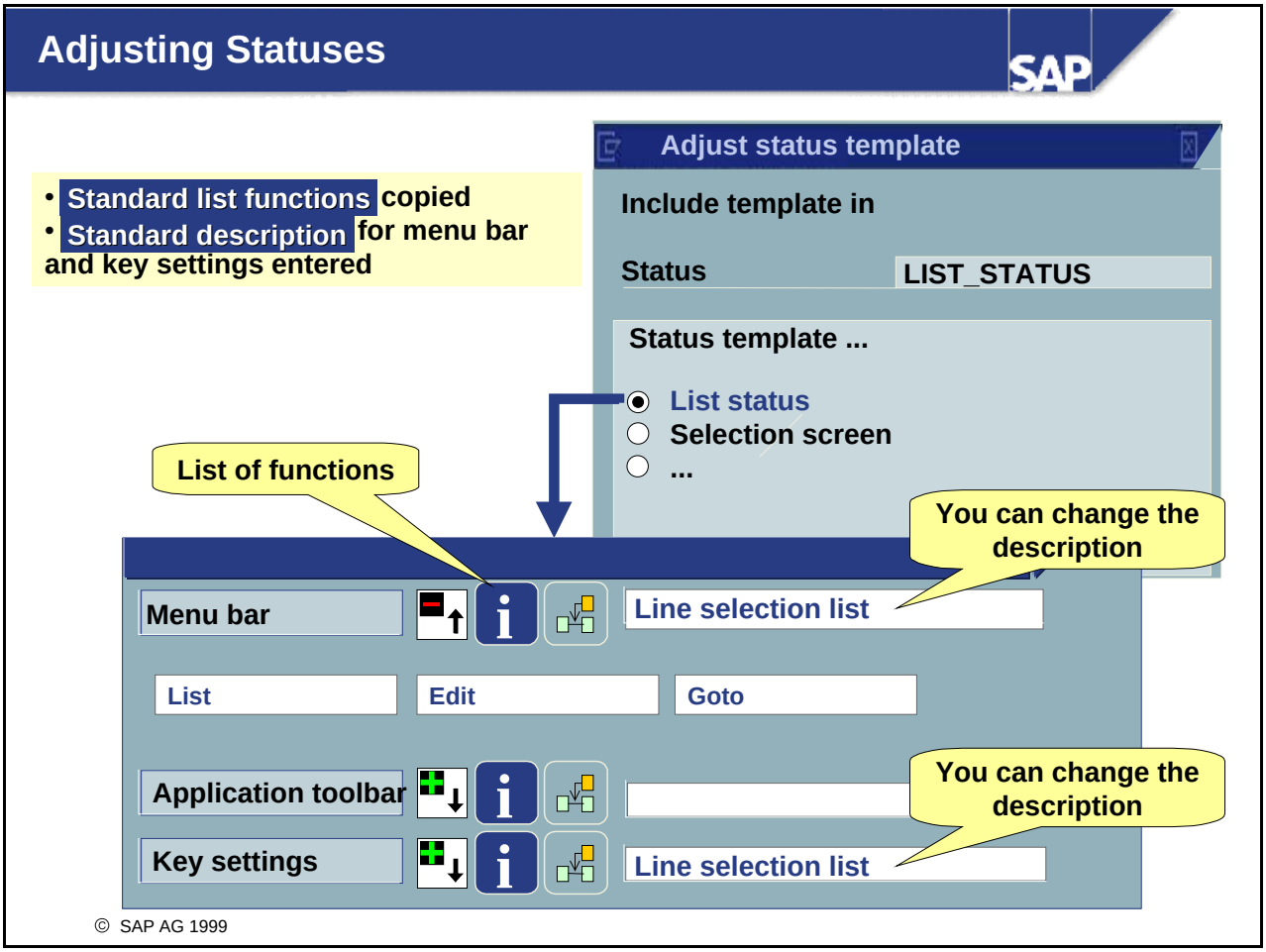
## Runtime Behavior: Setting a Status before Displaying a Screen



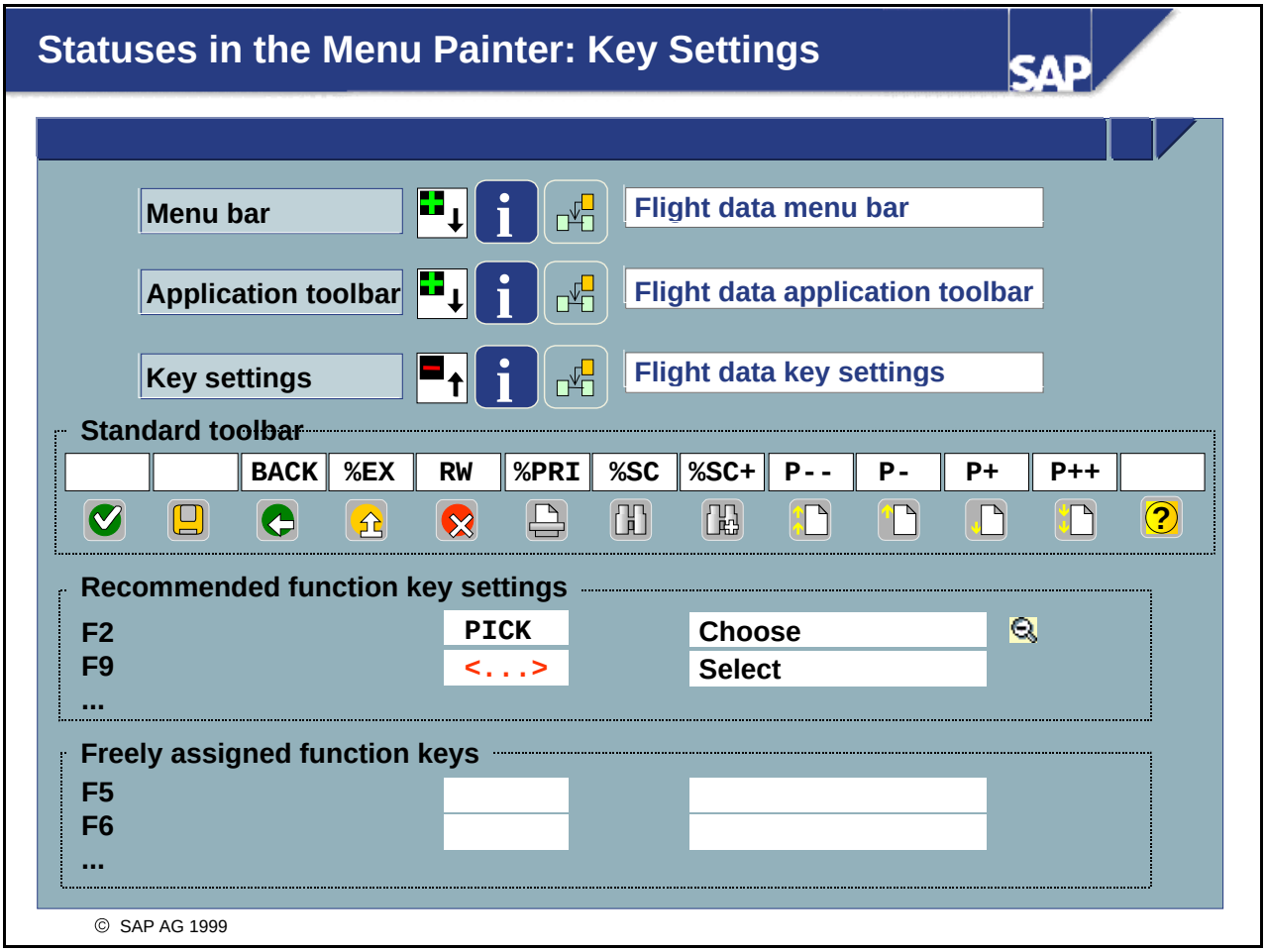

- You can set a status in ABAP processing blocks that are processed before the screen is sent to the presentation server. That is, you choose a status that already exists for the program, by name. This status then specifies the menu, the application toolbar, and the standard toolbar for the screen to be displayed.
- You set a status using the ABAP statement.
  **SET PF-STATUS <status name>.**
  You can create the name of the status as a text literal in upper-case. You can also create a constant that contains the name of the status in upper-case, or you can assign the name of a variable.



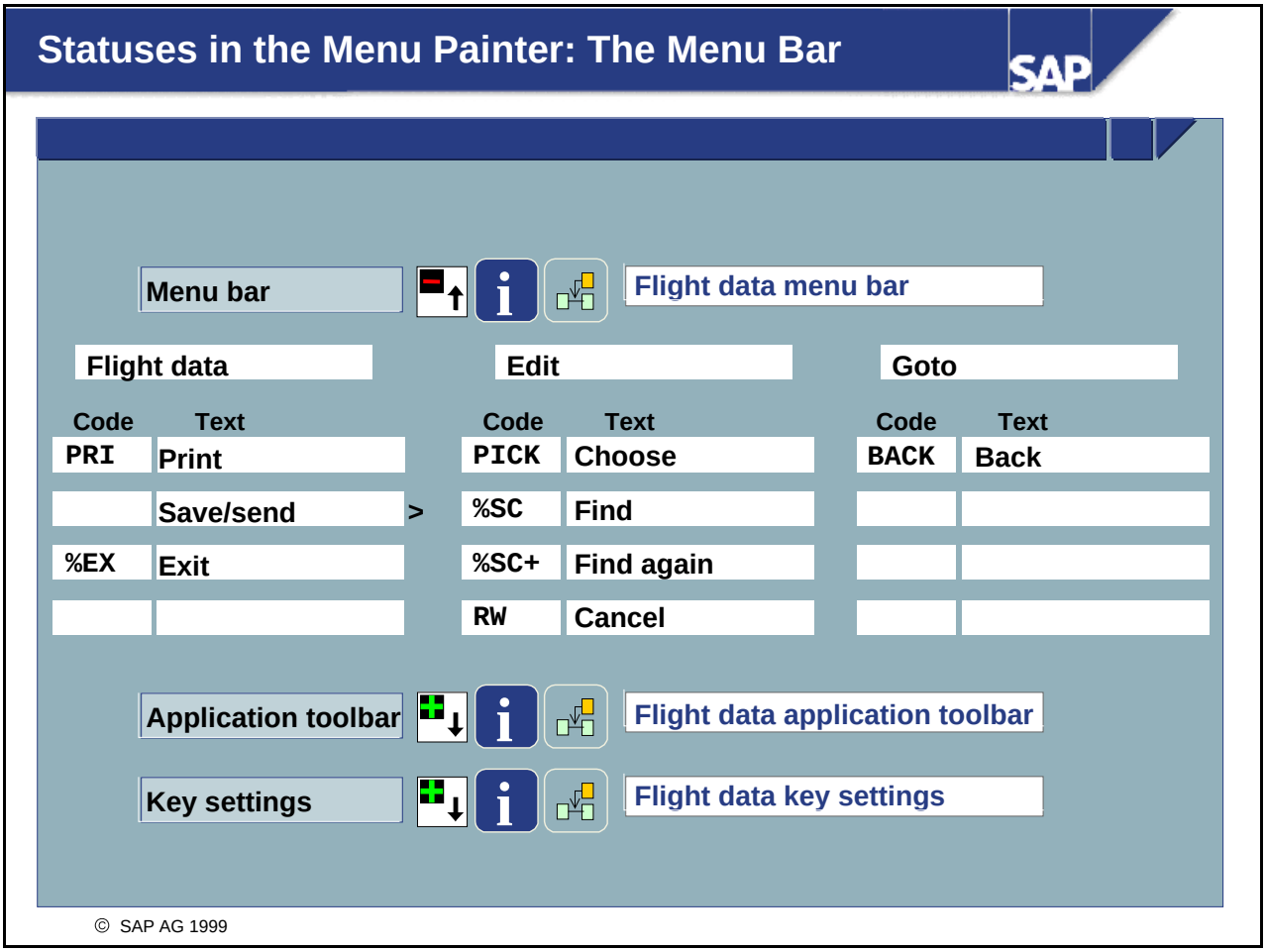

- You can create and maintain statuses in one of three different ways:
  - By using the **object list** of the Object Navigator
  - By using **forward navigation** in the ABAP Editor
  - By directly using the **Menu Painter**.
- When creating a status you can either create a new menu bar, application toolbar, and new key settings yourself (top down), or use existing objects for your interface (bottom up), or a use combination of both methods.
- Status names can have a maximum of 20 characters. (Letters must be upper-case).
- By choosing a status type, you determine whether a status refers to a normal screen (*Online status* in Release 4.6C), a dialog box, or a context menu. The functions you can use subsequently depend on the status type you have chosen.



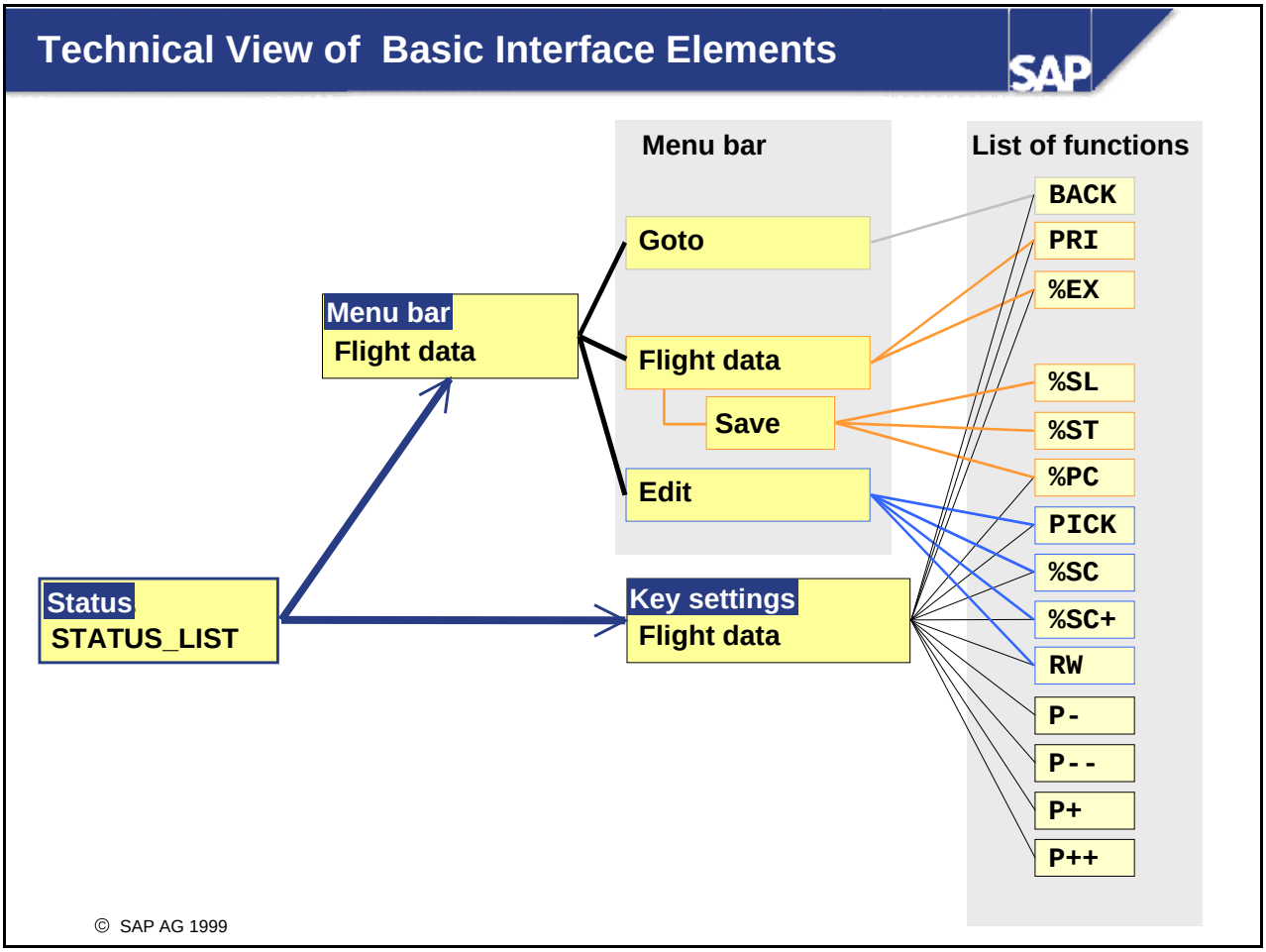

- The *Adjust template* function in the *Extras* menu allows you to include standardized function codes in your status. This function also allows you to include objects from a status of your choice with the existing status. This allows you to choose norms for list statuses and selection screens or use any other status you want from another ABAP program.
- If you choose *Adjust status->List status*, the system generates the standard entries for the menu, application toolbar, and function keys settings. Each menu bar, application toolbar, and function key setting is also given a default name, which you should replace with an explanatory text.
- You can display the functions that have been generated by choosing the *Information* icon.



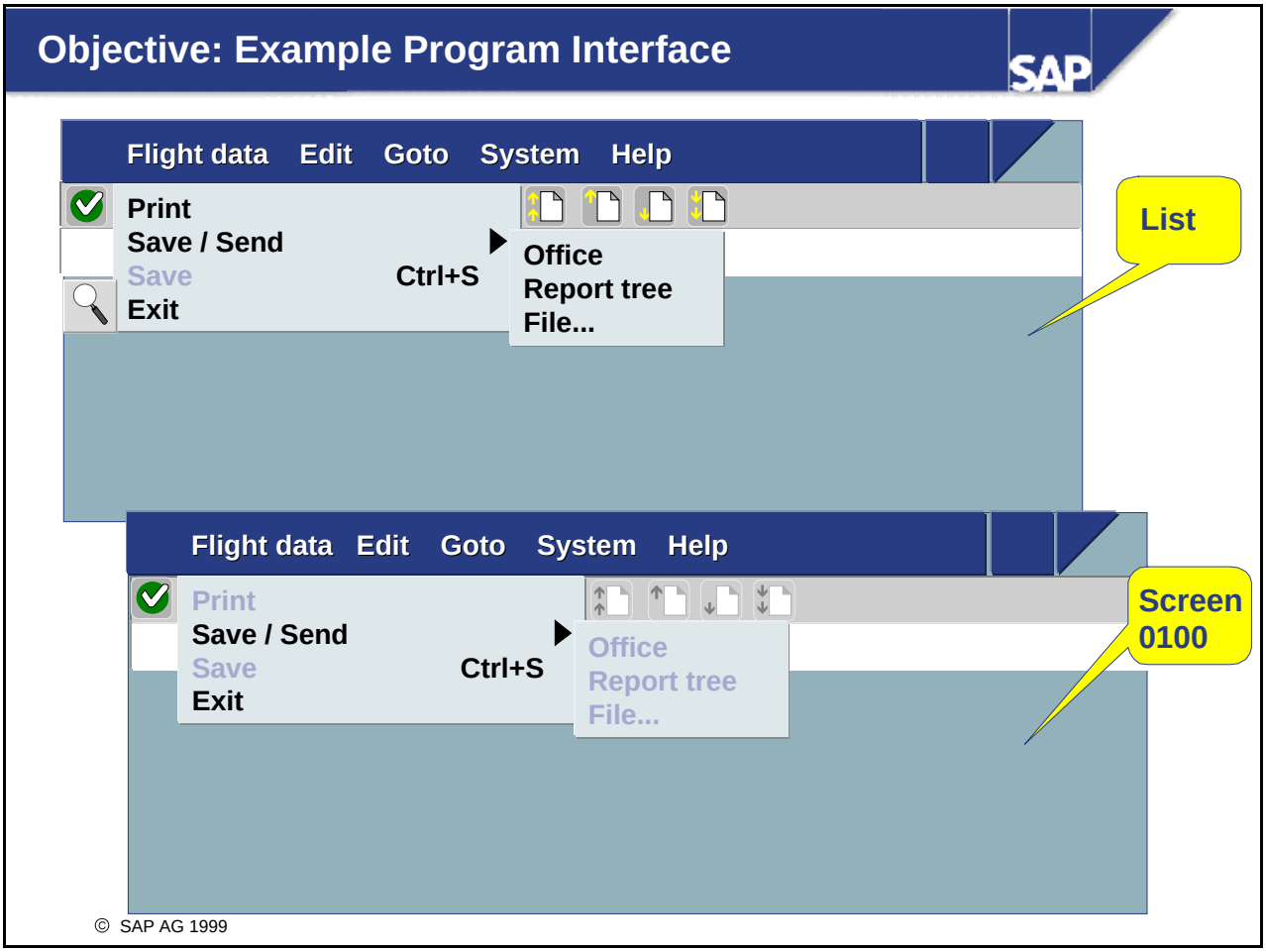

- Key settings can be divided into three areas:
  - **Standard toolbar:** Certain pre-defined function codes are mandatory for the functions Save, Back, Exit program, Cancel, Print, the Scroll icons, and the Enter pushbutton. Simply assign these codes to the standard toolbar icon and they will be automatically assigned to their corresponding pushbutton.
  - **Recommended function key settings:** The system proposes the functions that you should generally assign to specific function keys.
  - **Freely assigned function keys:** The system lists all the remaining function codes that are not assigned to a standard function key. You can then choose appropriate function keys from this list, for your program-specific functions.
- You can also define buttons on a button bar for those function keys that are used most often. These pushbuttons can either be icons or pushbuttons with text.



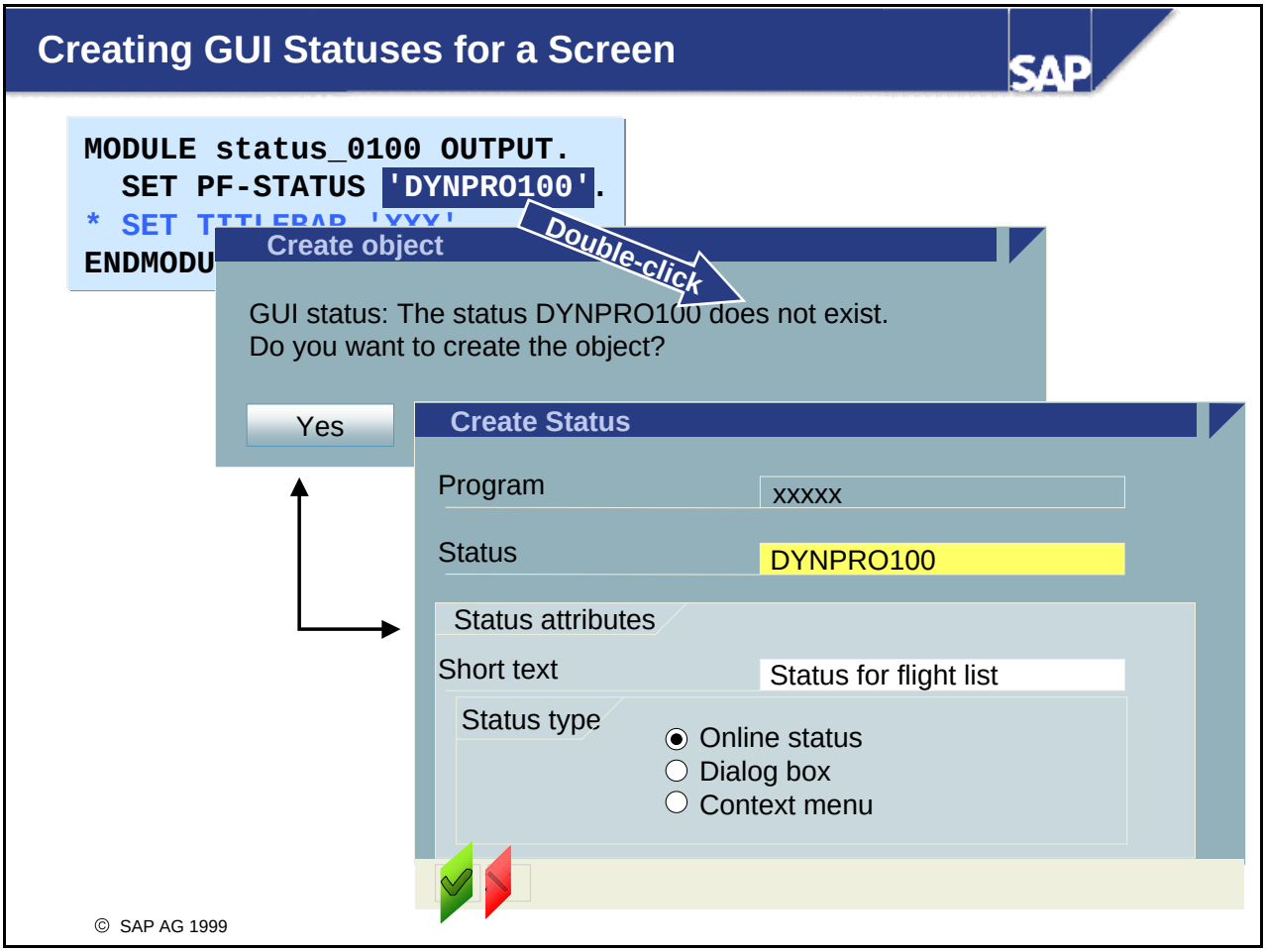

- If you want, you can make the system suggest texts for your menu bar. You can then modify them as appropriate.
- Menu bars can contain up to eight menus. You can define up to six of these yourself, but the *System* and *Help* menus are added automatically by the system.




- The above slide shows a technical view of the interface that you have created so far. You have created the sub-objects by adding a list standard status. You have renamed the menu bar and function key settings. Ideally, you should be able to execute all menu functions using function keys as well. You have not yet added pushbuttons to the application toolbar.
- From a technical point of view, a status is always a **reference** to a particular menu bar, standard toolbar, and application toolbar.

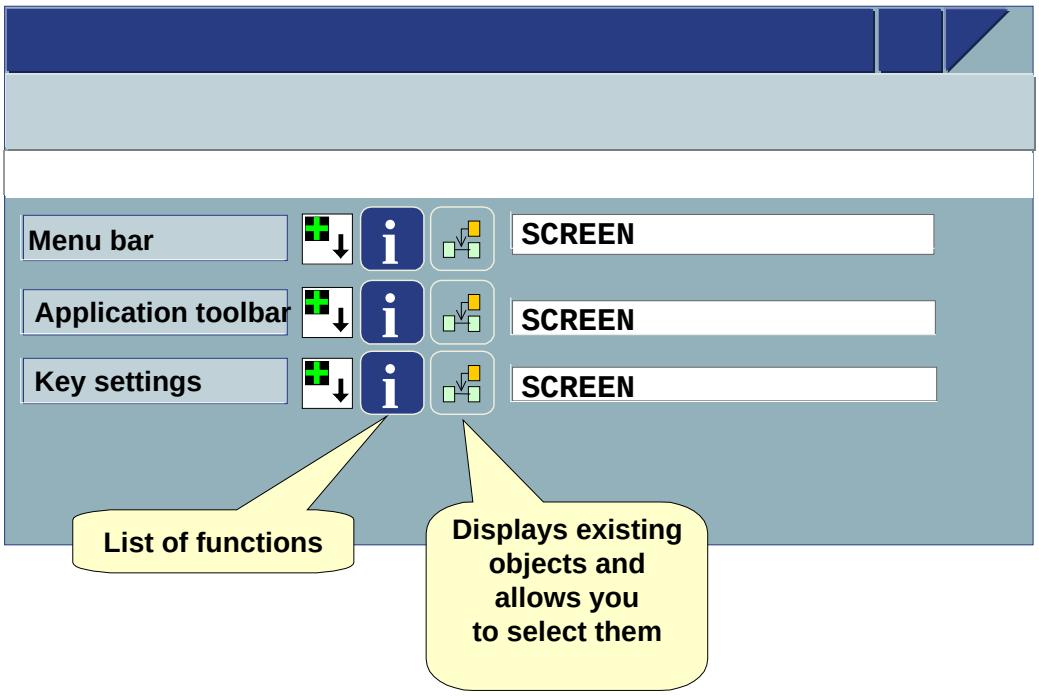



- As a final step, we will define an interface for the entire example program. The various statuses will have the following characteristics:
  - The same menu bar will be available both in the list and on the screen. Only those menu functions that can be executed will appear in black.
  - In the standard toolbar, only those functions that can be executed will appear in color. Screen functions will not be altered in any way.
  - The screen will have its own title.



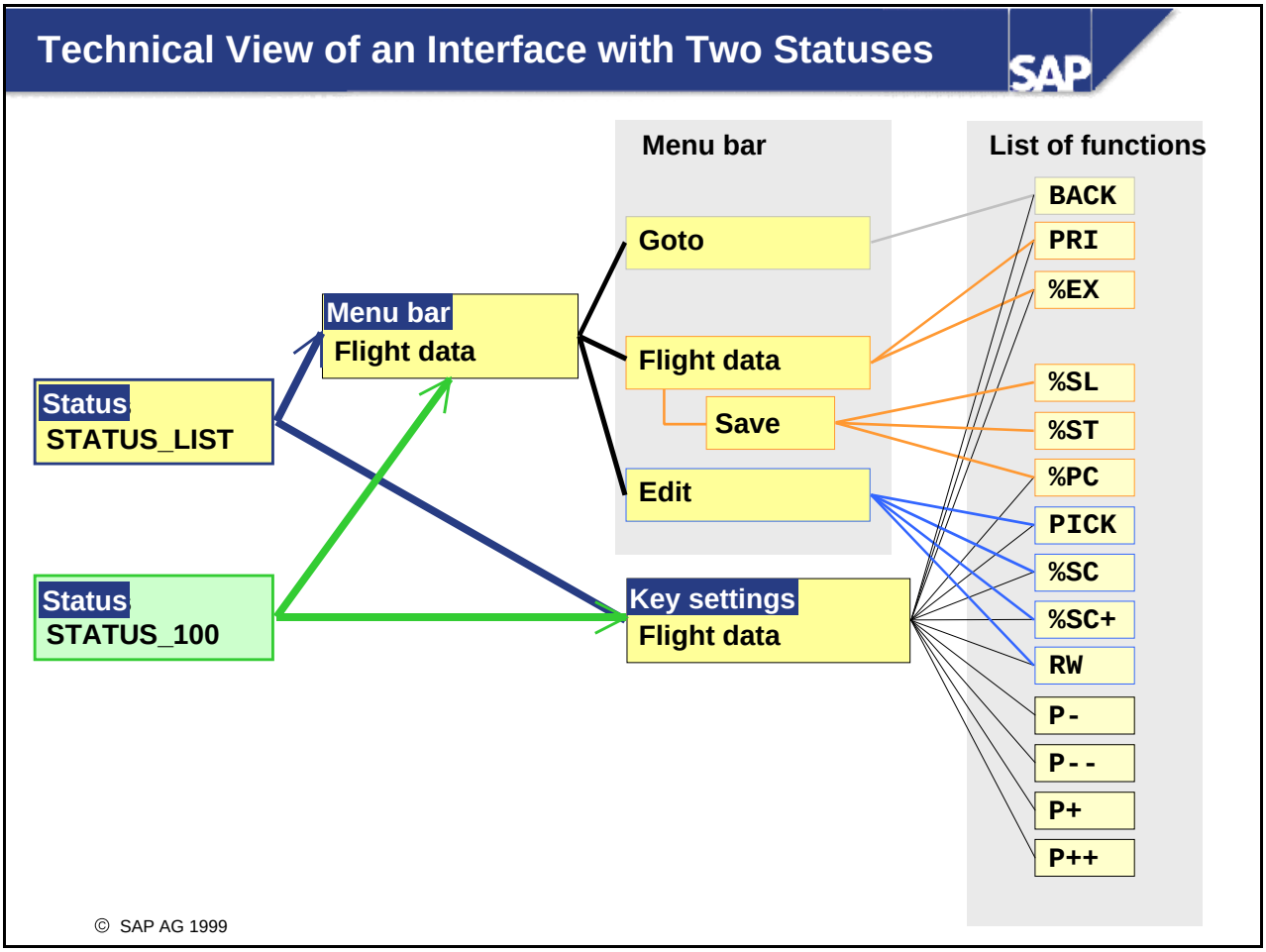

- To create a status for a screen:
  - Create a PBO module containing the statement **SET PF-STATUS '<NAME>'**. **<NAME>** can contain up to twenty digits or letters (which must be upper-case). This statement is pre-generated whenever you create the module **status_nnnn** using forward navigation.
  - Create the status using forward navigation
  - Choose *Online dialog* (that is, a normal screen) as your status type

## Including Existing Elements



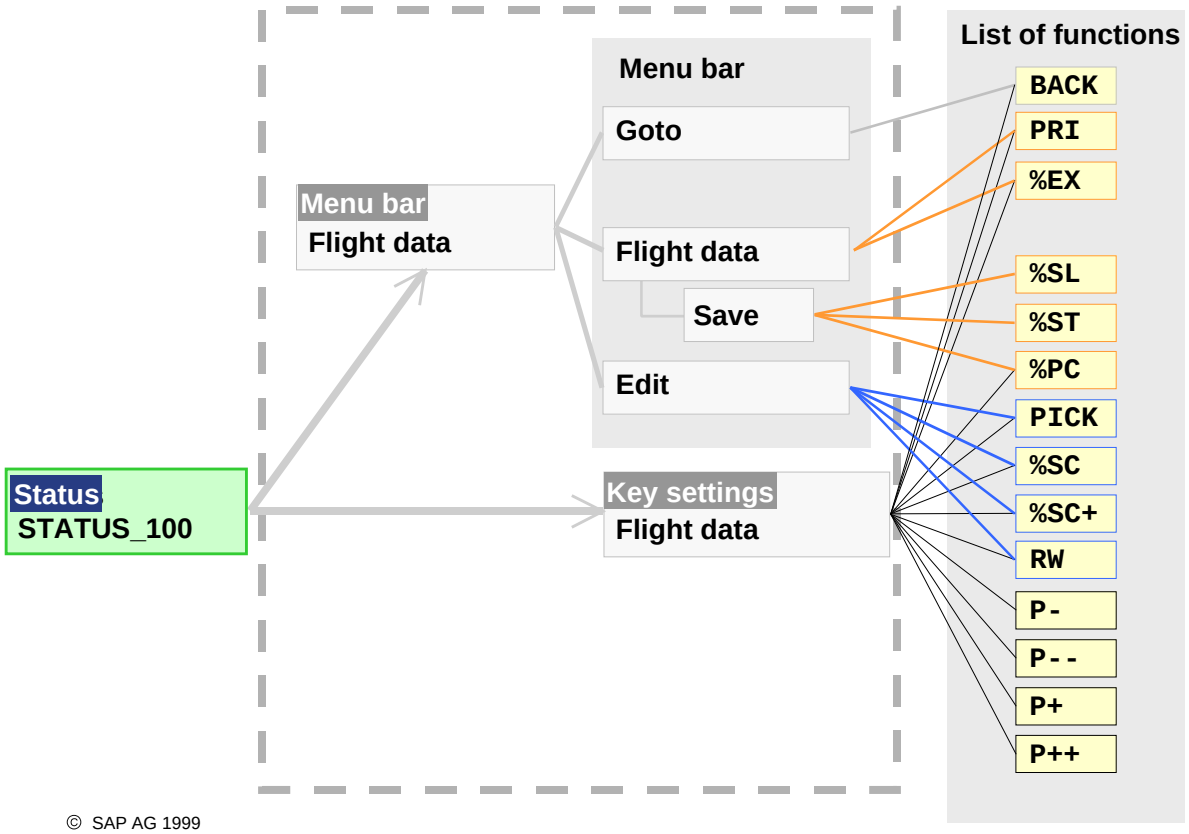


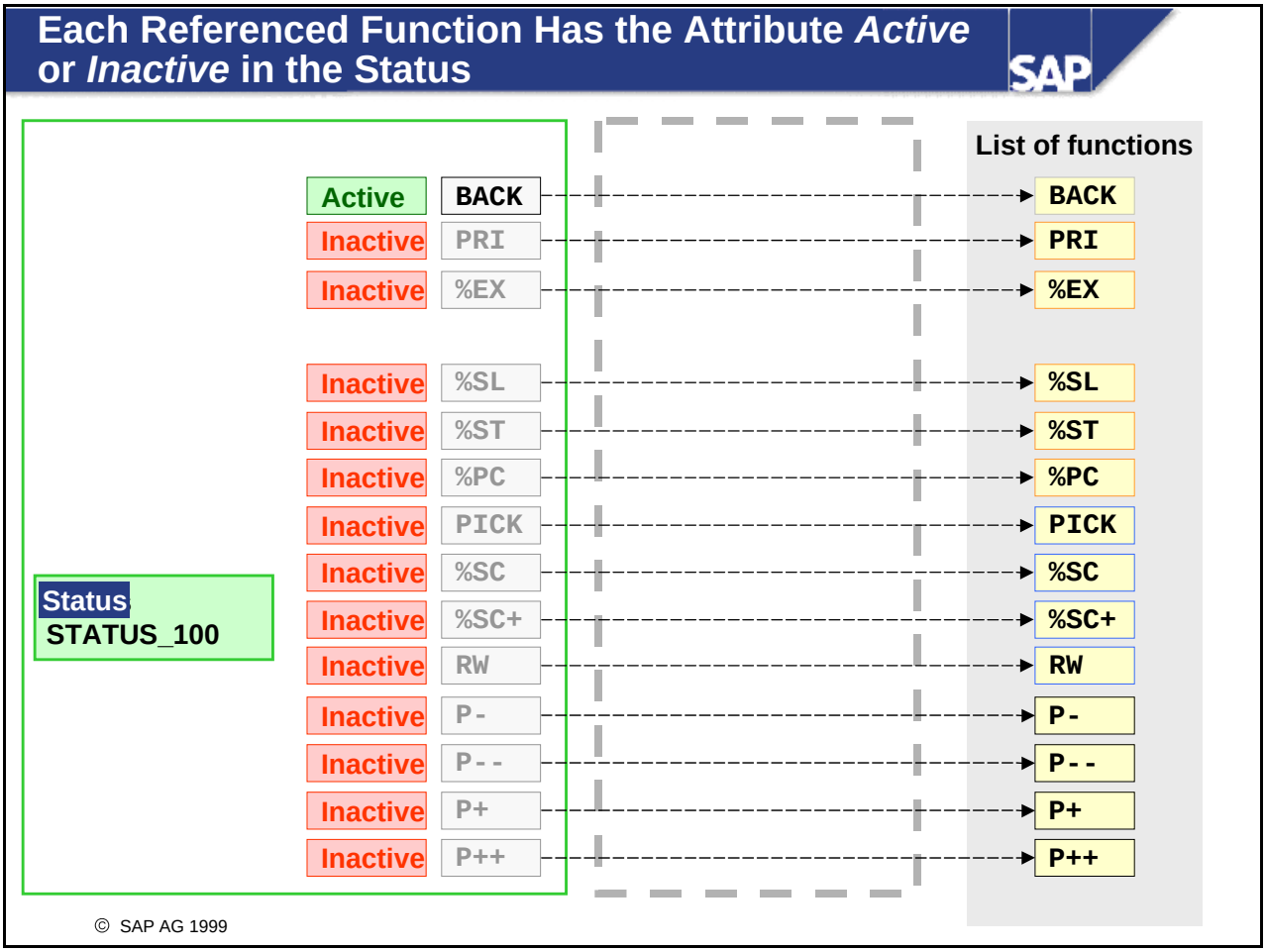

- The screen status must reference the same menu bar and function key assignment as the list status. You can display all the menu bars and function key assignments that are already defined using the icon shown above. The example program shows only one menu bar and one function key assignment, which you can choose by double-clicking.




- The above slide shows a technical view of the interface that you have created so far. In addition to the basic list status, you have created created another status and a title for the screen. If you have also created references to the existing menu bar and key settings using icons, then the status will reference the same functions as the list status. The menu entries are also identical, which makes it easier for users to find their way round the system.
- This techniques allows you to fulfil fundamental ergonomic requirements simply:
  - You should be able to execute **all menu functions by way of function keys** as well. Within each program, functions should be **assigned consistently** to function keys. This is why you should create a set of function key settings for each program, and assign each function to a function key there. You can create different statuses for different screens and have them reference these (function) key settings.
  - **All functions should be available in the menu** (except in the template, where you can only browse from the standard toolbar). Functions are generally distributed in discrete sets in different menu lists. Always follow ergonomic guidelines when assigning functions. You can then create one or more menu bars, which contain the menus whose functions can be executed. Each menu bar is referenced in one status. Within each program, functions should be **assigned consistently** to menu entries.

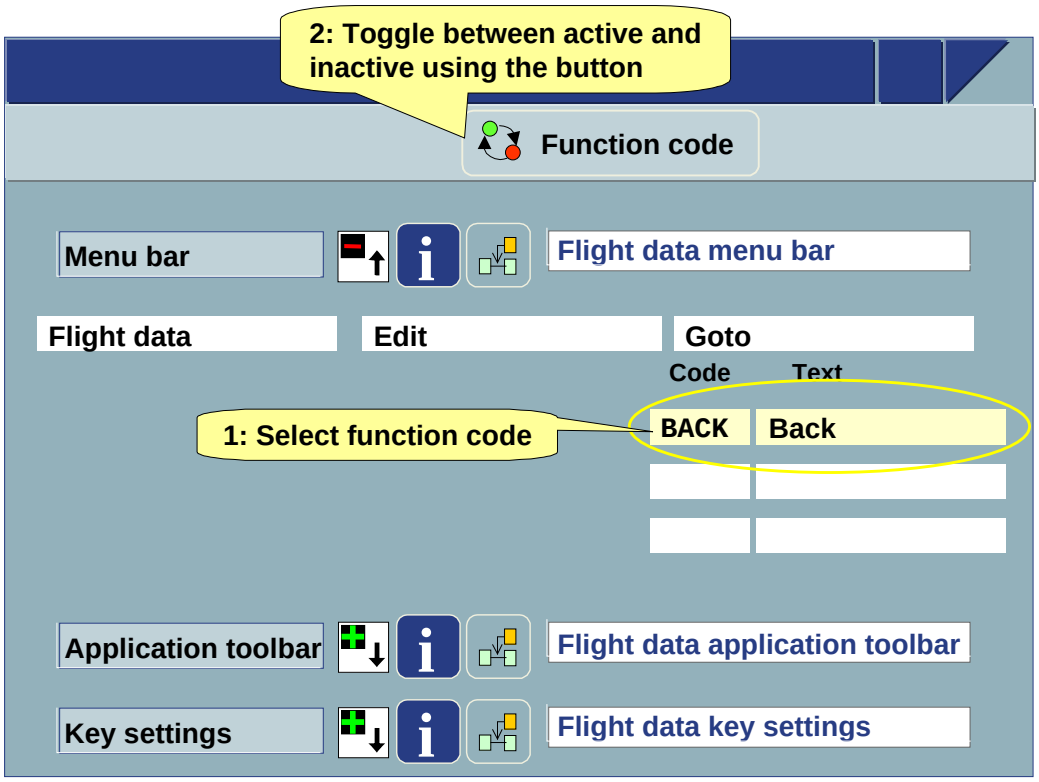
# Each Status References Functions (Indirectly)





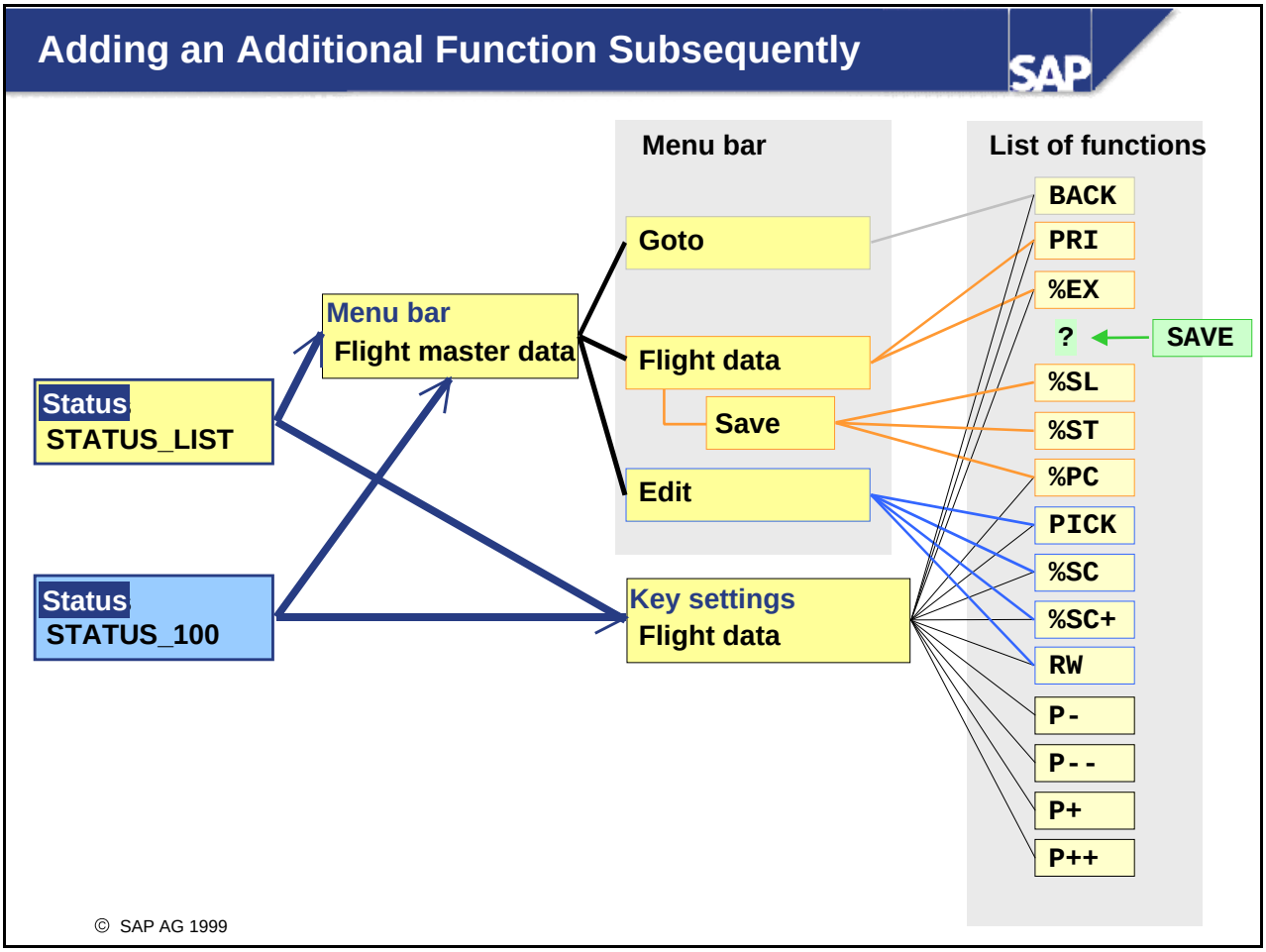

- You can specify whether each function is active or inactive in the status. The system then behaves as follows:
  - In the menu, the short text for each active function is displayed in black. The user can perform these functions. If they choose a menu entry in black type, its function code is sent to the command field. On screens, PAI is triggered and in lists, the system program is triggered. In the menu, the short text for each inactive function is displayed in gray. Nothing happens if the user chooses one of these menu entries.
  - If the user chooses a function key associated with an active function, the PAI and system list program are triggered and the function code is stored in the command field. If, however, he or she chooses an inactive function, nothing happens.
  - Icons associated with inactive functions in the application toolbar are always grayed out. Again, if the user chooses an inactive icon, nothing happens.
- Note: You assign the attribute *active* or *inactive* to functions, **not** to a menu entry or function key. The Menu Painter tool ensures that possible user actions are interpreted consistently.
- Note: If you create a status with reference to an existing menu bar and key settings, all the functions are set to inactive. You then set the functions you want to *active*. (See also the slide entitled *Setting Functions to Active or Inactive in the Status*).

## Setting Functions to Active or Inactive in the Status

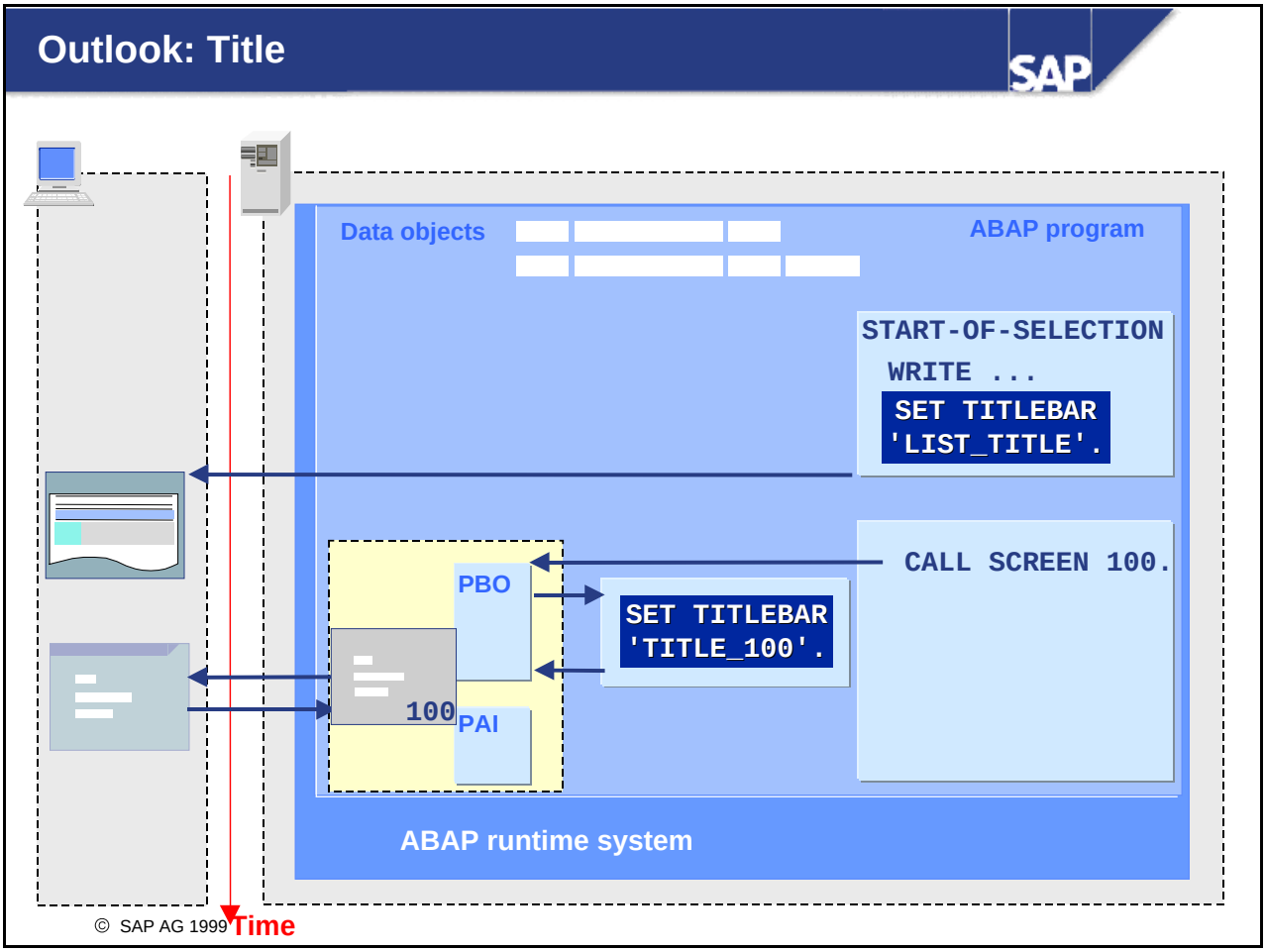





- The status for the screen contains the same objects as the basic list status, but initially all these objects are inactive. Activate the attribute for the **BACK** function, which will be evaluated in the **USER_COMMAND_0100** PAI module, as follows:
  - Choose *Change* mode by double-clicking the *Display/Change* button.
  - Then place the cursor on the function code you wish to activate and choose the *Active/Inactive* icon on the button bar.
- Later, we will need to activate the **SAVE** function, which will be evaluated in the PAI module, but which is not part of the standard list functions. For more details, refer to *Adding an Additional Function Subsequently*.

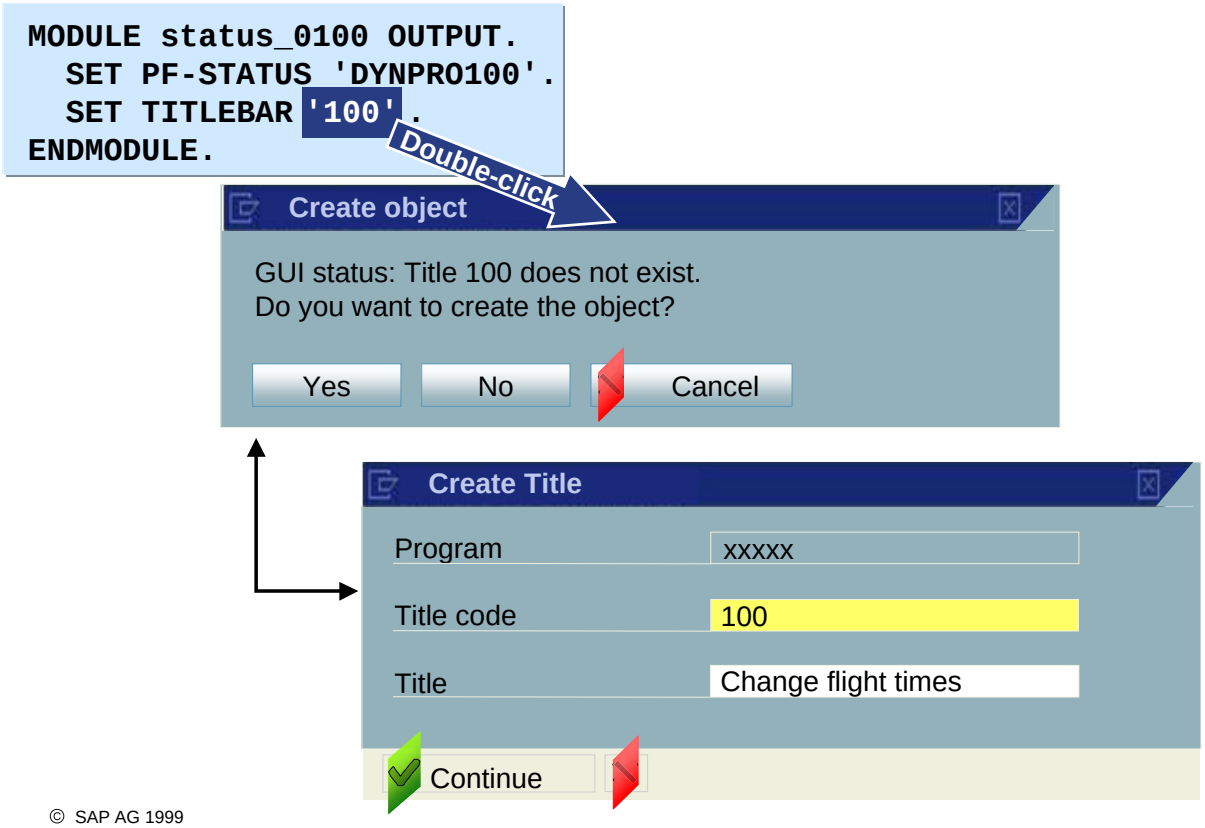



- There are several ways to add a function to a status subsequently.
- **Normally:** You want to insert the function in the same place in all your statuses. You carry out this change in one status. The function is active there. A modal dialog box then appears, asking you whether you want to change all referenced statuses. Choose *Continue* to confirm that you do. Technically, the following then occurs: You create a new function using forward navigation. (You can check the codes available using the information icon). You also change a referenced menu or key settings using forward navigation. The change takes effect in all statuses that reference the menu or key settings. In the other statuses, the new function is set to inactive but you can now change this in each status as necessary.
- **In exceptional cases:** You want to change one status only. In this case, you create copies of objects. For more detailed information, refer to the documentation or the training course BC410: *Developing User Dialogs*.




- You can create a title for a screen using the ABAP statement **SET TITLEBAR <title name>**, where <title name> is a technical name for the title (of up to 20 characters), to which you can then assign texts in various languages. Selection texts are displayed in the user's logon language.

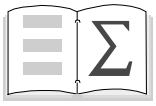
## Creating GUI Titles for a Screen






- You can create and maintain titles in one of three different ways:
  - By using the **object list** of the Repository Browser
  - By using **forward navigation** in the ABAP Editor
  - By using the **Menu Painter**.
- Status names can have a maximum of 20 characters.
- Note: If you create the suggested PBO module **status_nnnn** using forward navigation, the **SET TITLEBAR 'XXX'** statement is automatically generated in the module. It is commented out, however. You activate the statement by erasing the star and entering the number of the title. Now you can create the title using forward navigation.

# Interfaces: Unit Summary



**You are now able to:**

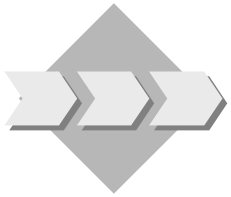
- **Create a GUI title**
- **Create GUI statuses for lists and screens that contain the following:**
  - **Menu bars**
  - **Standard toolbars**
  - **Application toolbars**
  - **Function key settings**

Exercises

**Unit: Interfaces**

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

- Create user interfaces for a program
- Include a GUI status and GUI title in a program



Extend your program, **ZBC400_##_DYNPRO**:
The system displays the texts that belong to the list and screen in the title bar.
You must specify the correct status for the list and screen.

**Program:** **ZBC400_##_DYNPRO**

**Model solution:** **SAPBC400UDS_DYNPRO_D**

1-1 Extend your program, **ZBC400_##_DYNPRO**, or copy the relevant model solution **SAPBC400UDS_DYNPRO_C** and give it the name **ZBC400_##_DYNPRO_D**. Assign your program to the development class **ZBC400_##** and to the transport request for this project, BC400… (replacing ## with your group number).

1-2 **Status for the list:**
First define a status for the list. Create the status (type: dialog status) using forward navigation. Include the appropriate ABAP statement in the **START-OF-SELECTION** processing block. The status you create must correspond to the standard list status. To create your function key assignment, button bar, and menu bar, choose *Extras* → *Adjust template* → *List status* and add your function codes. Change the name of the left menu list from List to Postings.

1-3 **Status for the screen**

**1-3-1**
Assign a status to the screen (type: *online status*). Call a module at **PROCESS BEFORE OUTPUT**, and include the appropriate ABAP statement in it.

Do **not** create a new function key assignment, button bar, or menu bar. Instead,

reference them to the objects you created in the last exercise (1-2).
Activate the function code **BACK**. Save the interface.

1-3-2 Add the function code **SAVE**. The user should be able to trigger this by choosing the **Save icon in the standard toolbar** or by choosing the correct entry in the *Bookings* menu.

1-4 Check the status for the list:
The **SAVE** function should appear, inactive, in the same place on each menu.
Activate the program with all its subobjects and test it.

1-5 **Optional:**
Assign a GUI title to the basic list and screen (using forward navigation) and generate the interface.

Solutions

**Unit: Interfaces**

1-2 In the program source text, add the **SET PF-STATUS 'LIST'** statement to the **START-OF-SELECTION** event.

(Note: Text between the two apostrophes must be in uppercase).

- In the program line **SET PF-STATUS 'LIST'**double-click the **LIST** status name.

  The system displays a dialog box containing the words "*The GUI interface status 'LIST' does not exist. Do you want to create it?*"

  **Choose**  to **confirm.**

  **Enter a** *short text*.

  **Choose** the *Online status* status type.

- Choose *Extras →Adjust template →List status*
- Adapt the name of the menu bar and function key assignment to your own needs.
- Change the name of the menu list to *Bookings*.
- Activate the status.

1-3

1-3-1 Display screen 100 in change mode (by selecting the screen number in the program object list and using the alternate mouse button). In the flow logic for the PROCESS BEFORE OUTPUT event, add the statement:

**MODULE set_status_0100.**

- Create the module using forward navigation. Double-click the name of the module and choose *Yes* to confirm the settings in the dialog box. In the next dialog box, choose *Main program* and add the following source text:

```
*&---------------------------------------------*
*&      Module  SET_STATUS_0100  OUTPUT
*&---------------------------------------------*
  MODULE set_status_0100 OUTPUT.
 SET PF-STATUS 'DYNPRO'.
ENDMODULE.
```

- Create the status using forward navigation. (Double-click 'DYNPRO' in the SET PF-STATUS 'DYNPRO' statement).
- Choose *Yes* to confirm the settings in the dialog box.
- In the next dialog box that appears, enter a short text and choose *Online status*.

- Place the cursor on the menu bar and choose the  icon. Create the reference to the existing menu bar by double-clicking.
- Place the cursor on the function key and choose the  icon Create the reference to the existing function key assignment by double-clicking.
- Activate the two functions **SAVE** and **BACK** by placing the cursor on the function code in the menu or on the function key assignment and choosing  .
- Save your entries and activate the status.
- Activate the screen.

1-3-2 Navigate to the DYNPRO status. Make sure you are in *Change* mode.

- Double-click the function-key assignment. Enter the function code **SAVE** in the standard toolbar above the  icon.
- To make the function available in the *Bookings* menu:

- Double-click the *Bookings* menu in the menu bar.
- Create a new line in the menu.
- Enter the code **SAVE**
- Confirm your entries.
- Choose  to leave the dialog box.

1-5 Extend the program source text in the START-OF- SELECTION. event:

```
SET PF-STATUS 'LIST'.
SET TITLEBAR 'LIST'.
```

Create the title using forward navigation and enter the text in the dialog box that appears.

Extend the source text of the **SET_STATUS_0100** module as follows:

```
*&---------------------------------------*
*&     Module  SET_STATUS_0100  OUTPUT
*&---------------------------------------*
  MODULE set_status_0100 OUTPUT.
  SET PF-STATUS 'DYNPRO'.
  SET TITLEBAR 'DYNPRO'.
  ENDMODULE.
```

Create the title using forward navigation and enter the text in the dialog box that appears.

Activate the status and the program.

**Source text for the program SAPBC400UDS_DYNPRO_D**

```
*&---------------------------------------------------------------------*
*& Report        SAPBC400UDS_DYNPRO_D                                  *
*&                                                                     *
*&---------------------------------------------------------------------*

REPORT  sapbc400uds_dynpro_d.
CONSTANTS: actvt_display TYPE activ_auth VALUE '03',
           actvt_change TYPE activ_auth VALUE '02'.
*  Definition of selection screen
PARAMETERS pa_anum TYPE sbook-agencynum.

* workarea for list
DATA wa_booking TYPE sbc400_booking.
* workarea for single booking to be changed
DATA wa_sbook TYPE sbook.
* workarea for dynpro
TABLES sdyn_book.
* variable for function code of user action
DATA: ok_code LIKE sy-ucomm.
```

```
START-OF-SELECTION.

  SET PF-STATUS 'LIST'.



* selecting data using a dictionary view to get the data from sbook and
* the customer name from scustom
  SELECT carrid connid fldate bookid customid name
        FROM sbc400_booking
        INTO CORRESPONDING FIELDS OF wa_booking
          WHERE agencynum = pa_anum.

    AUTHORITY-CHECK OBJECT 'S_CARRID'
       ID 'CARRID' FIELD wa_booking-carrid
       ID 'ACTVT'  FIELD actvt_display.
    IF sy-subrc = 0.
*Output
      WRITE: / wa_booking-carrid,
               wa_booking-connid,
               wa_booking-fldate,
               wa_booking-bookid,
               wa_booking-name.

      HIDE:    wa_booking-carrid,
               wa_booking-connid,
               wa_booking-fldate,
               wa_booking-bookid,
               wa_booking-name.



    ENDIF.
  ENDSELECT.
  CLEAR wa_booking.

AT LINE-SELECTION.
  IF sy-lsind = 1.
    AUTHORITY-CHECK OBJECT 'S_CARRID'
```

```
        ID 'CARRID' FIELD wa_booking-carrid
         ID 'ACTVT'  FIELD actvt_change.
    IF sy-subrc = 0.

      SELECT SINGLE *
           FROM sbook
           INTO wa_sbook
           WHERE carrid = wa_booking-carrid
             AND connid = wa_booking-connid
             AND fldate = wa_booking-fldate
             AND bookid  = wa_booking-bookid.
      IF sy-subrc = 0.
        MOVE-CORRESPONDING wa_sbook TO sdyn_book.
        MOVE wa_booking-name TO sdyn_book-name.
        CALL SCREEN 100.
      ENDIF.
    ELSE .
      MESSAGE ID 'BC400' TYPE 'S' NUMBER '047' WITH wa_booking-carrid.
    ENDIF.
  ENDIF.
  CLEAR: wa_sbook, wa_booking.

  INCLUDE bc400uds_dynpro_do01.

  INCLUDE bc400uds_dynpro_di01.



*---------------------------------------------------------------------*
*  INCLUDE BC400UDS_DYNPRO_DO01 .                                     *
*---------------------------------------------------------------------*
*&---------------------------------------------------------------------*
*&      Module  STATUS_0100  OUTPUT                                    *
*&---------------------------------------------------------------------*
*       text                                                          *
*---------------------------------------------------------------------*
MODULE STATUS_0100 OUTPUT.
  SET PF-STATUS 'DYNPRO'.
  SET TITLEBAR '100'.     "optional
```

```
ENDMODULE.                 " STATUS_0100  OUTPUT


*&---------------------------------------------------------------------*
*&      Module  CLEAR_OK_CODE  OUTPUT                                  *
*&---------------------------------------------------------------------*
*&       text                                                          *
*----------------------------------------------------------------------*
module clear_ok_code output.
clear ok_code.
endmodule.                 " CLEAR_OK_CODE  OUTPUT




*&---------------------------------------------------------------------*
*& INCLUDE BC400UDS_DYNPRO_DI01 .                                      *
*&---------------------------------------------------------------------*
*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT                               *
*&---------------------------------------------------------------------*
*&      text                                                          *
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CASE ok_code.
    WHEN 'BACK'.
      LEAVE TO SCREEN 0.
    WHEN 'SAVE'.
      MOVE-CORRESPONDING sdyn_book TO wa_sbook.
      MESSAGE ID 'BC400' TYPE 'I' NUMBER '060'.
      LEAVE TO SCREEN 0.

  ENDCASE.
```

```
ENDMODULE.                    " USER_COMMAND_0100  INPUT
```

# Reuse Components

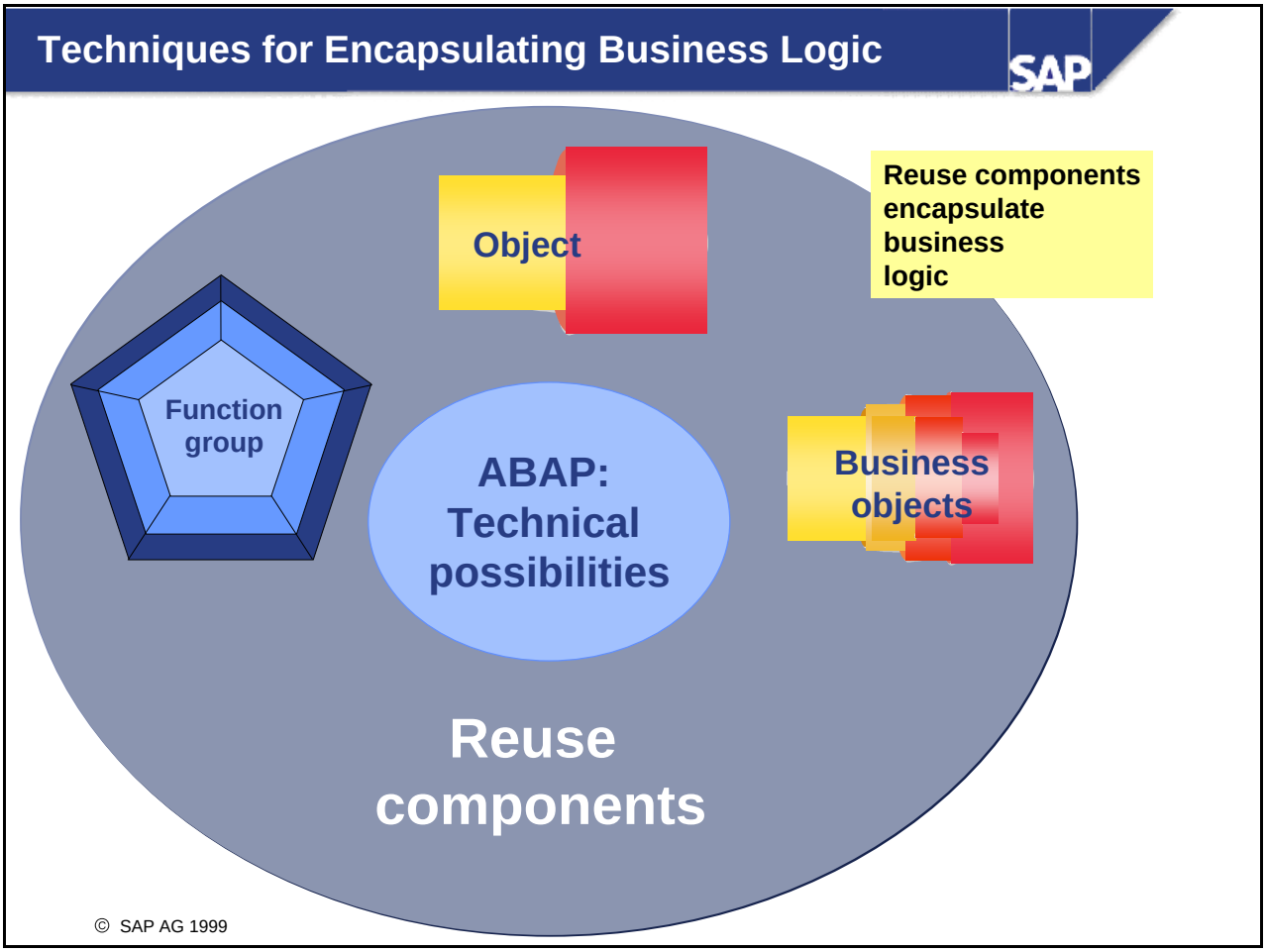
## Contents:

- Function Groups and Function Modules
- Objects and Methods
- Business Objects and BAPIs

# Reuse Components: Unit Objectives

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

- **Find and use function modules**
- **Display a simple list using the ALV grid control (objects from global classes in the Class Builder)**
- **Use a BAPI and find BAPIs using the BAPI Browser**




- The R/3 System offers several techniques that you can use to make business logic available for reuse.
- **Function modules** can be called from any ABAP program. Parameters are also passed to the interface. Function modules that belong together are combined to form function groups. Program logic and user dialogs can be encapsulated in function modules.
- **Objects:** You can use the compatible extension "ABAP objects" to create objects at runtime, with reference to central classes.
- **BAPI**s are business objects that are made available using the Business Object Repository (BOR).

# Overview: Function Groups and Function Modules

Function Groups and Function Modules

Business Objects and BAPIs

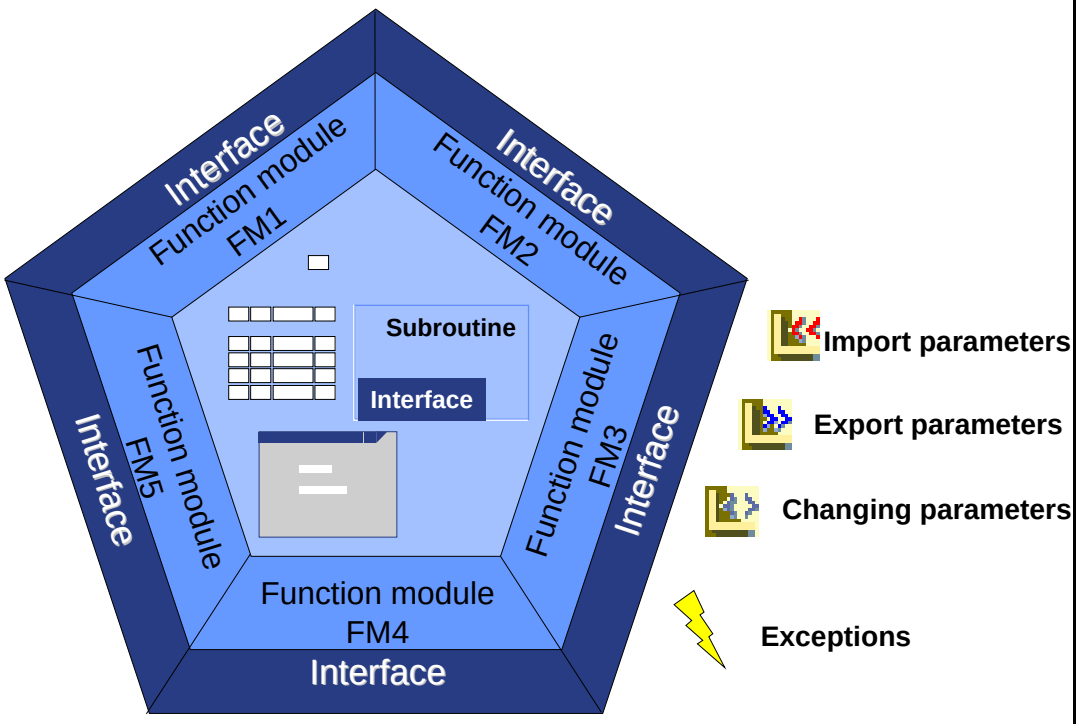
Objects and Methods

# Function Groups and Function Modules: Course Objectives

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

- **Describe the various ways of finding a function module**
- **Find out important information about a function module using the Function Builder**
- **Write a program containing a standard user dialog that is encapsulated in a function module**

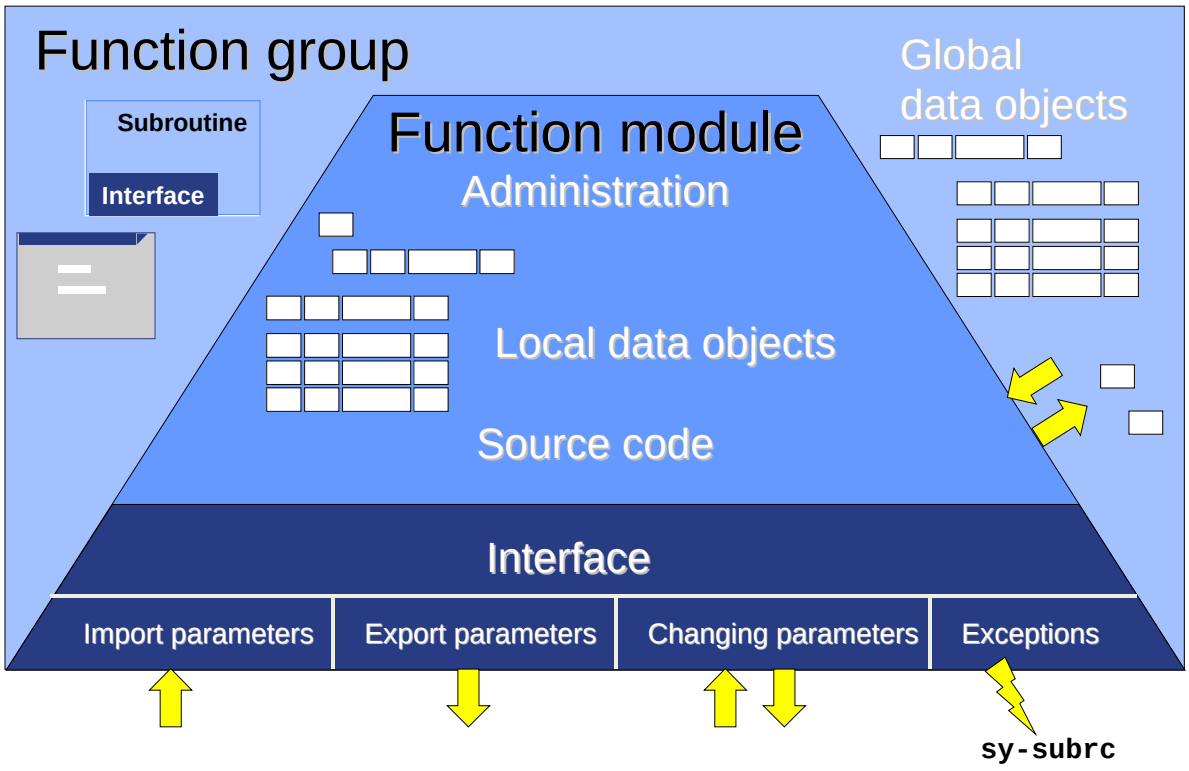
## Function Groups





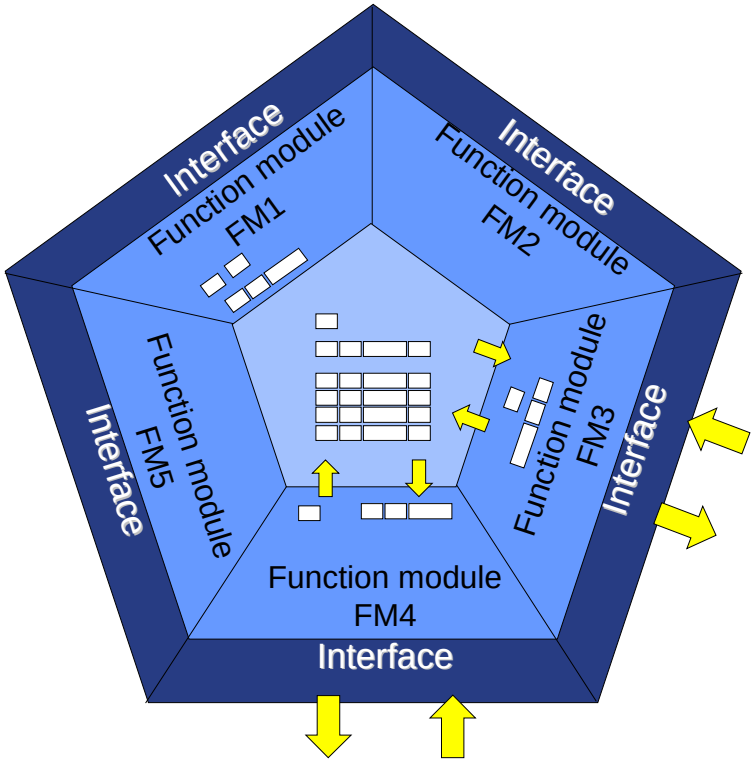

- A **function group** is an ABAP program with type F, which is a program created exclusively for containing function modules. Function modules are modular units with interfaces that can be called from any ABAP Program. Function modules that operate on the same objects are combined to form function groups.
- Each function group can contain:
  - **Data objects**, which can be seen and changed by all the function modules in the group. These data objects remain active as long as the function group remains active.
  - **Subroutines**, which can be called by any of the function modules in the group.
  - **Screens**, which can be called by any of the function modules in the group.

## Function Modules




- Function modules are modular units with interfaces. The interface can contain the following elements:
  - **Import parameters** are parameters passed to the function module. In general, these are assigned standard ABAP Dictionary types. Import parameters can also be characterized as optional.
  - **Export parameters** are passed from the function module to the calling program. Export parameters are always optional and for that reason do not need to be accepted by the calling program.
  - **Changing parameters** are passed to the function module and can be changed by it. The result is returned to the calling program after the function module has executed.
  - **Exceptions** are used to intercept errors. If an error triggers an exception in a function module, the function module stops. You can assign exceptions to numbers in the calling program, which sets the system field SY-SUBRC to that value. This return code can then be handled by the program.
- Each function module can contain local data objects and access global data objects belonging to its function group. All the subroutines and screens in the function group can be called by the function module.

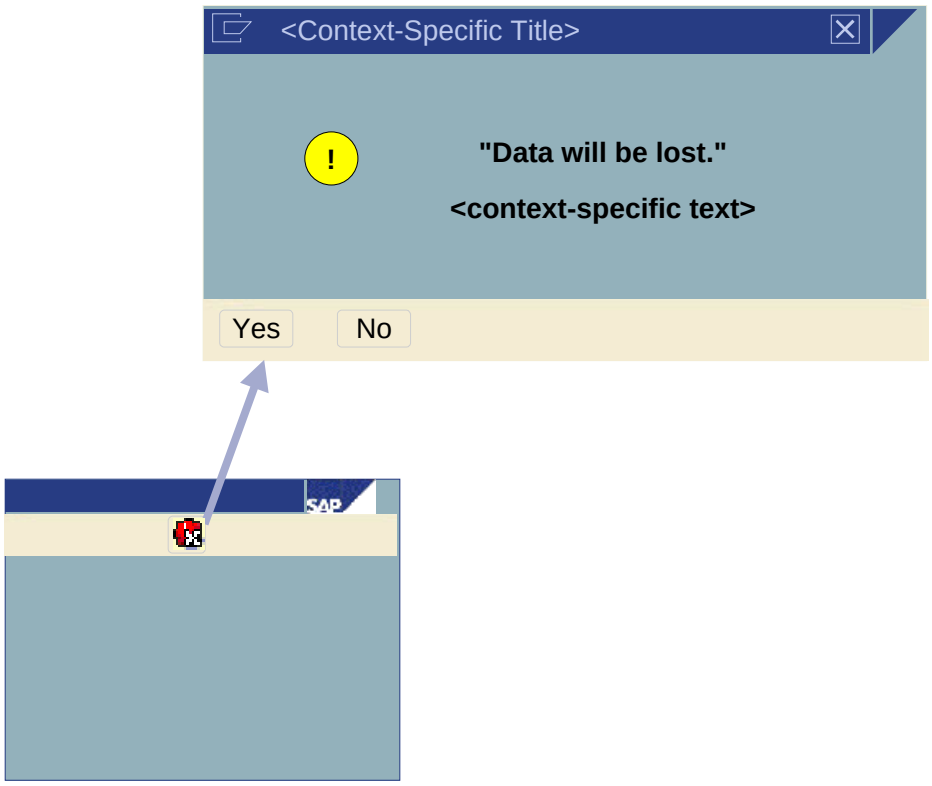
## Function Groups: Data Flow





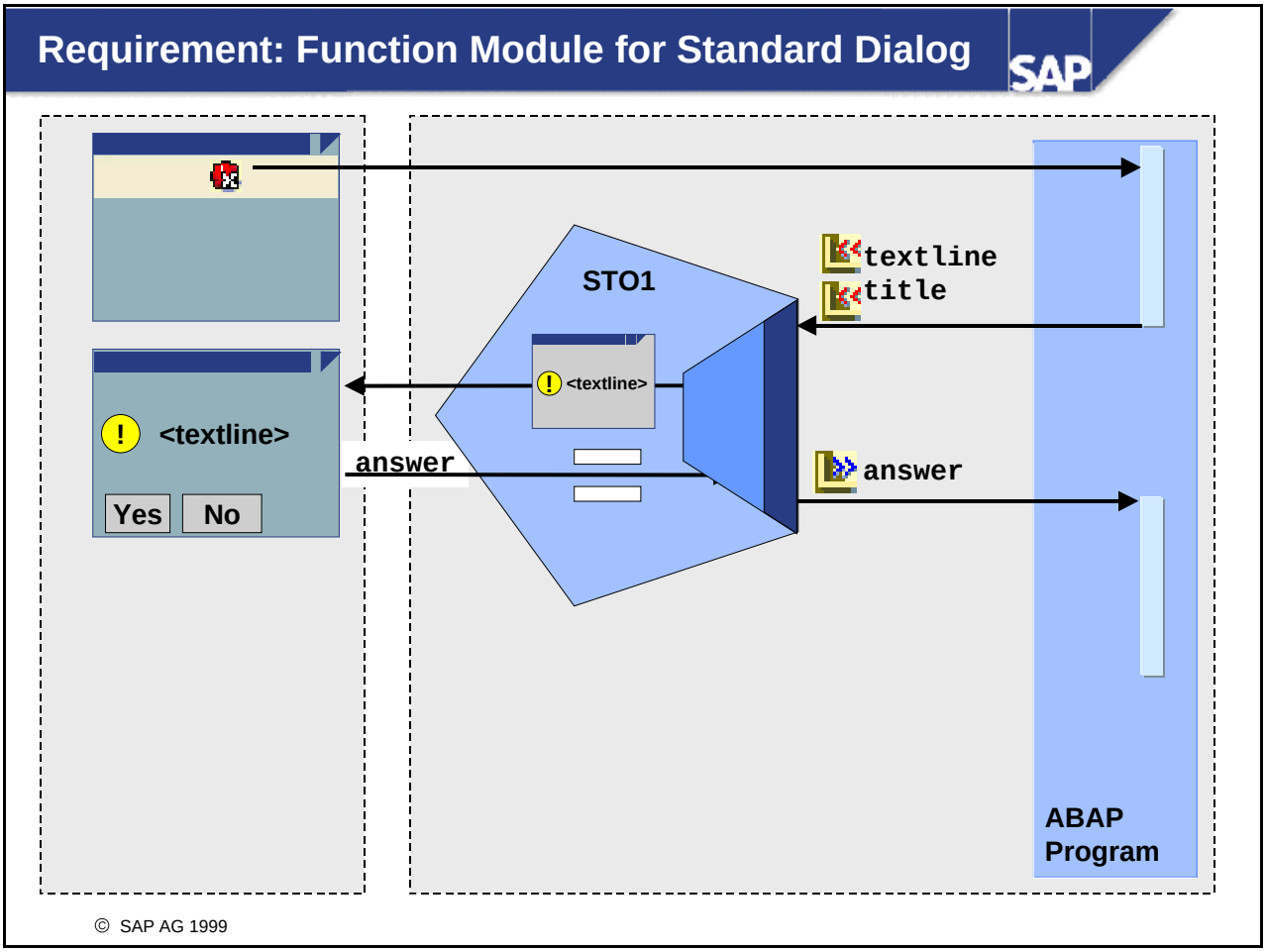

- The global data in the function group remains after the function module has been called. The function group remains active for as long as the calling program is active. Thus, if a function module is called that writes values to the global data, other function modules in the same function group can access this data when they are called by the program.

## Example: The *Cancel* Dialog Box

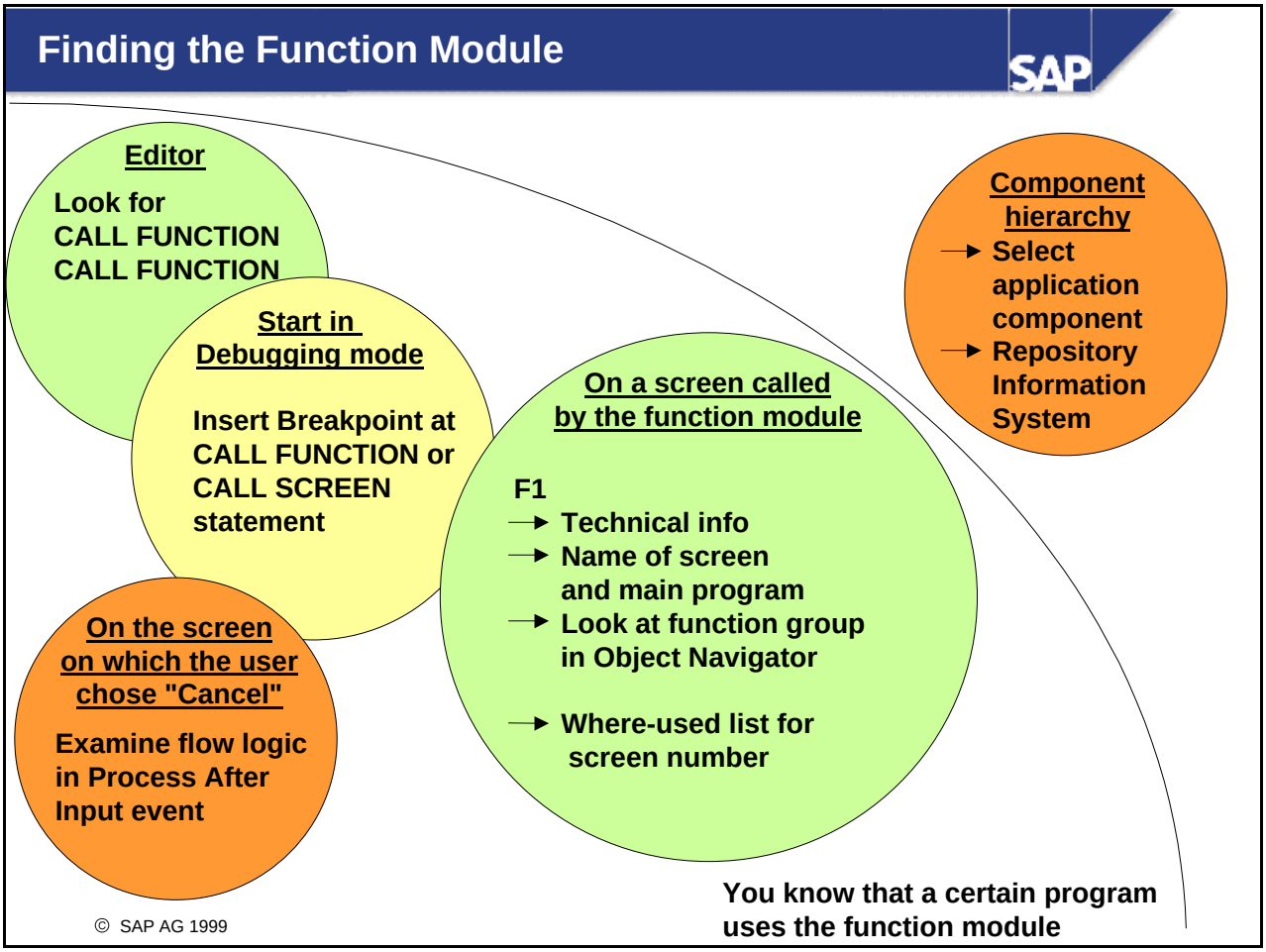





- In many programs a standard dialog box appears after the user has chosen *Cancel*. This dialog box always contains the sentence: "Data will be lost." The two lines following it are context-specific, as is the title. The user can choose from one of two options - "Yes" or "No."
- This dialog box is encapsulated in a function module.

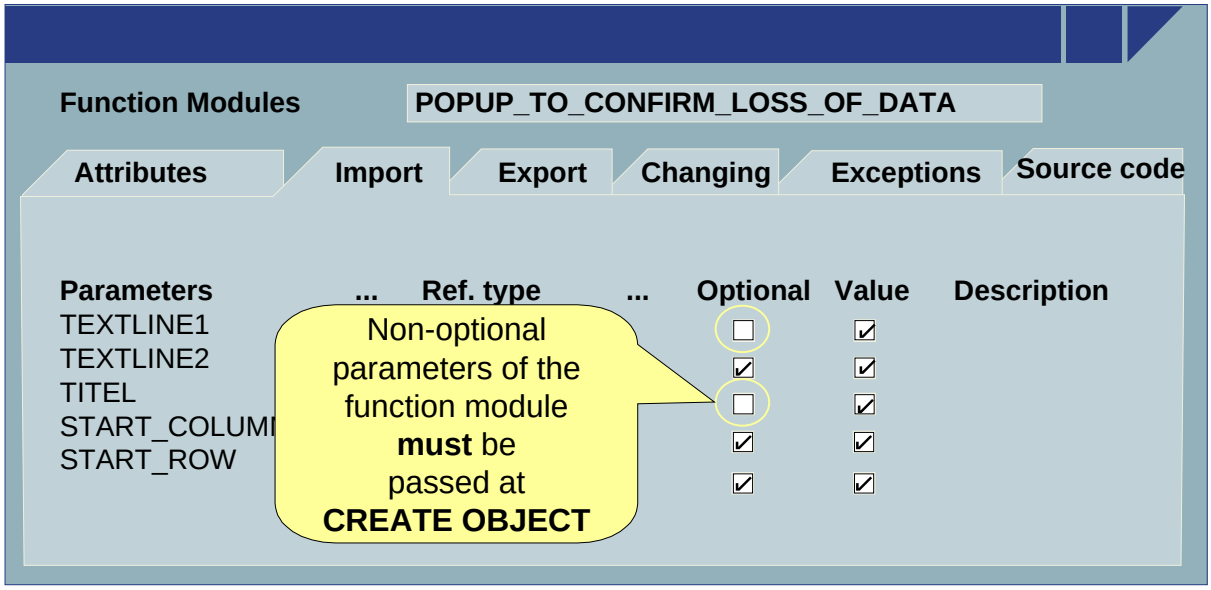



- You can avoid programming this dialog box, if you can find an existing function module with the following properties:
  - Import parameters for the title and the two variable text lines
  - An export parameter to record whether the user has chosen "Yes" or "No"
  - The ability to call a screen in the function group that displays the two variable text lines and the title, and contains the "Yes" and "No" buttons.



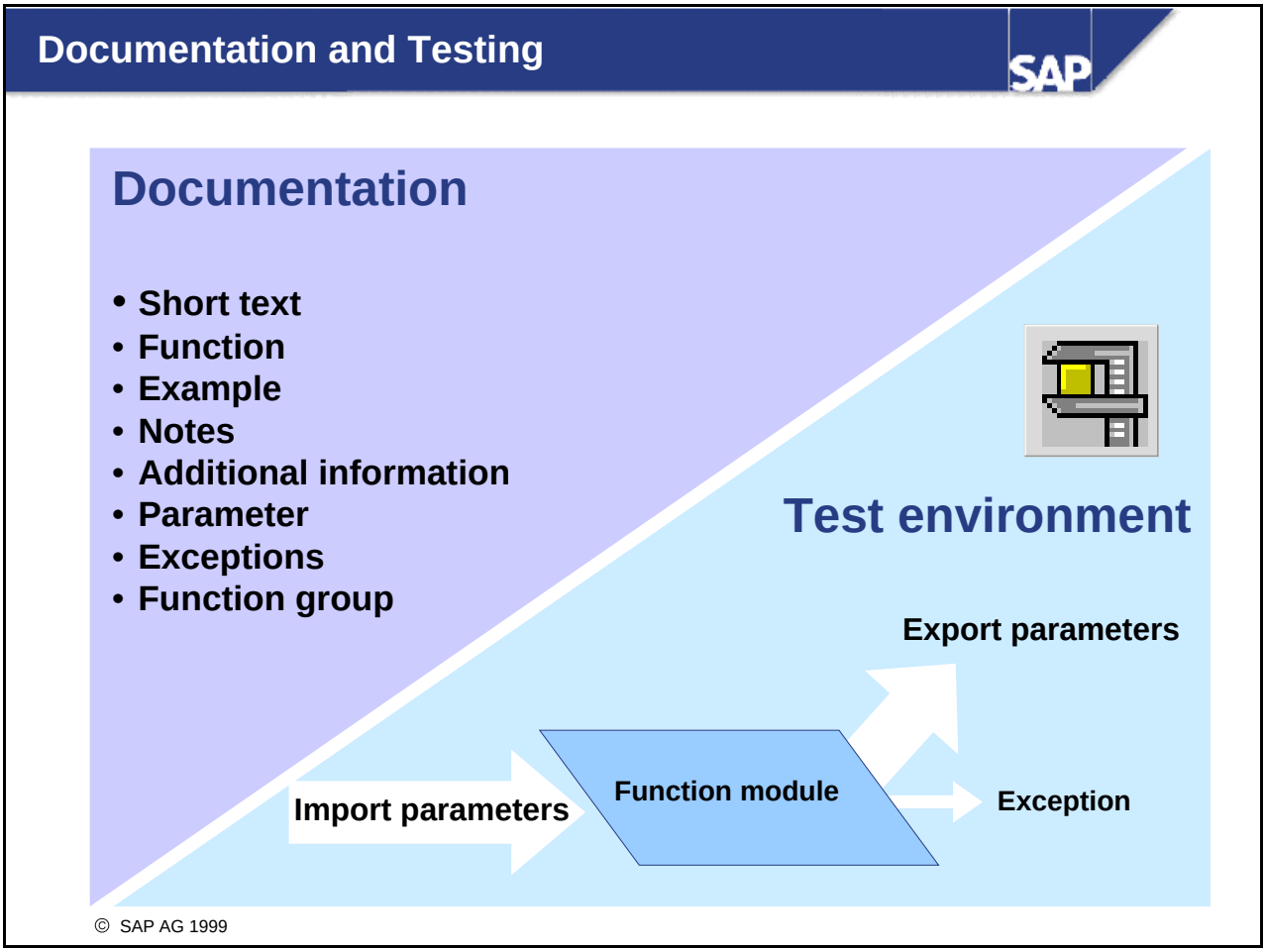

- Scenario: You are creating a program in the Object Navigator and leave the Attributes screen. You want to know if it is encapsulated in reusable form You can use the following methods:
  - In the Debugger, set a breakpoint at 'CALL SCREEN', if successful, under 'CALL' identify the program and subroutine as well as the function module. Then, display the call and the data that is passed to the interface in the Object Navigator.
  - In the Debugger, set a breakpoint at "CALL FUNCTION". If successful, the result is the same as in method 1.
  - Click a text field in the standard dialog box, then press F1 and choose "Technical info". Navigate to the screen and display a where-used list for programs, then look at the function modules that use it.
  - In the *Save* dialog box, display the F1 help and then *Technical info*. Navigate to the screen, then examine the flow logic and its modules.
  - In the application hierarchy, determine the component (Basis Services), select it, navigate to the Repository Information System, look under *Programming -> Function Library -> Function modules*, and select *Only released*.

# Function Module Interface



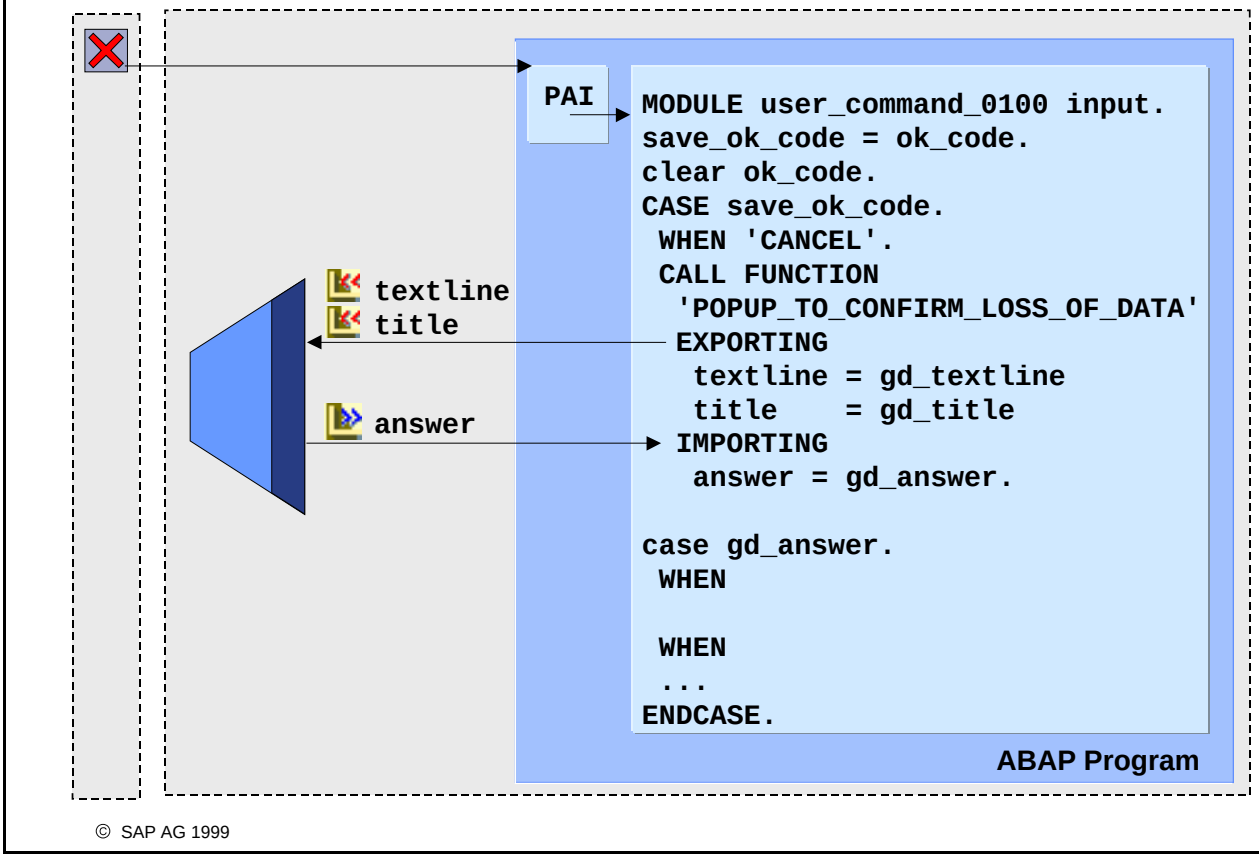



- Once you have found a function module, you must find out more about its interface.
- Non-optional parameters in the function module must be passed in the **CALL FUNCTION** statement. To find out handling the other parameters, refer to the function module documentation and the documentation on interface parameters.
- If the documentation is not specific enough, or is not available in your logon language, you can analyze the source code for the function module by choosing the *Source code* tab.



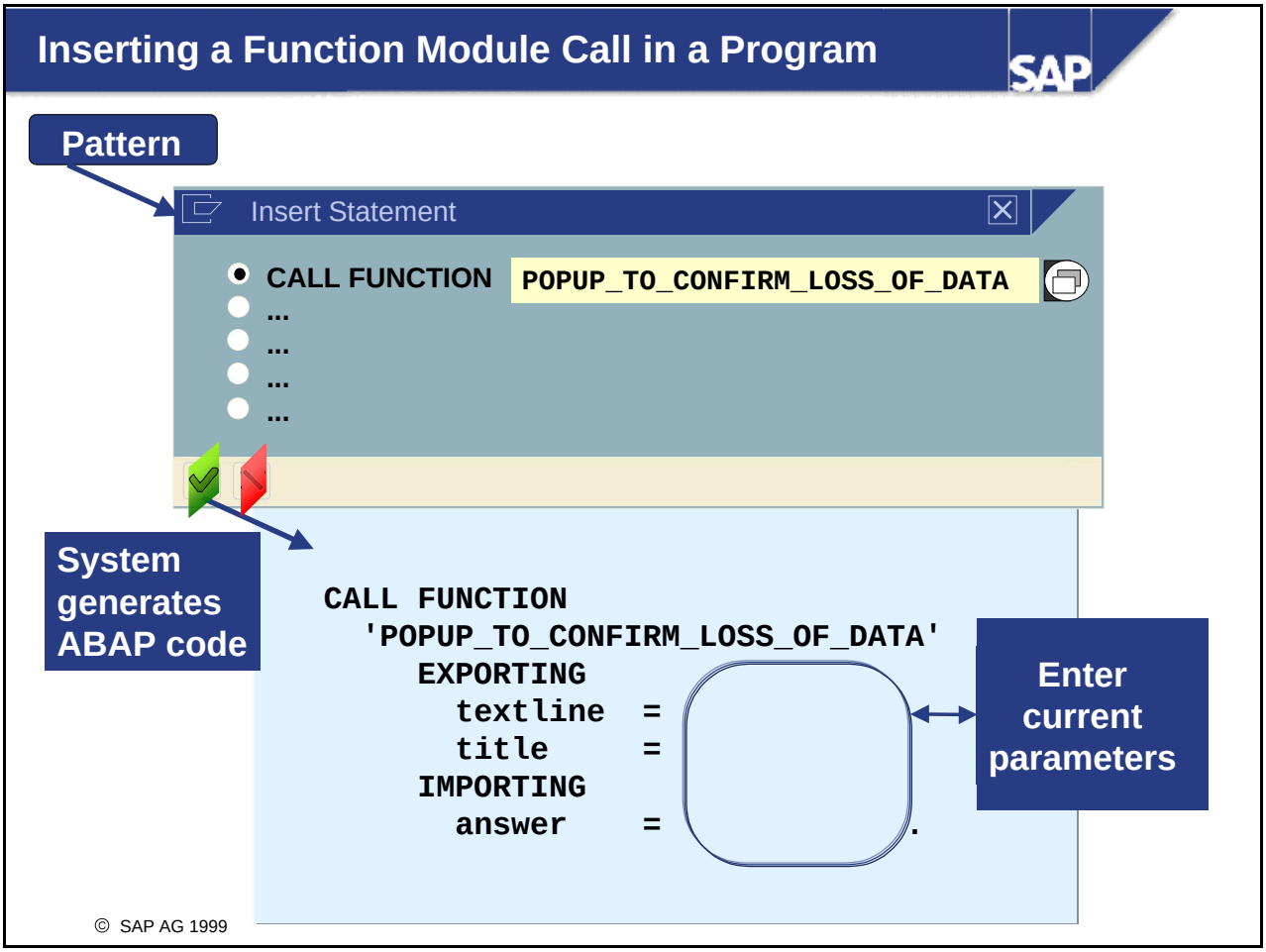

- You can test function modules using the test environment. An input template allows you to specify the **IMPORT** parameters. The result is transferred to the **EXPORT** parameters and displayed.
- If an error occurs, the system notes which exception was triggered.
- The runtime for the function module is displayed in microseconds. These values are subject to the same conditions as the runtime analysis transaction. You should therefore repeat the test several times using the same data.
- You can store test data in a test data directory.
- You can use the test function of the Function Builder to test function modules with table parameters.
- You can create test sequences.

## Syntax: Calling the Function Module






- You call function modules from ABAP programs using the **CALL FUNCTION** statement. The name of the function module is displayed in single quotation marks. After **EXPORTING**, the system assigns the parameters that are passed/ to be passed to the function module. After **IMPORTING**, the system assigns the parameters that are passed from the function module to the program. Most function modules support additional exceptions. If so, after **EXCEPTIONS**, the exceptions are assigned to values that will be set in the system field **sy-subrc**, if a system error occurs. On the left side of the *Parameter assignment* screen, the system displays the names of the interface parameters, while the right side of the screen displays the program's data objects.




- To do this, use a statement pattern in the ABAP Editor (the *Pattern* pushbutton), and enter the name of the function module.
- The system then generates an ABAP statement **CALL FUNCTION** '<**function module name**>', including the interface of the function module, and inserts it in the program at the current cursor position.
- Fill in the actual parameters, and write the statements that will handle any exceptions that occur. Interface parameters values are assigned explicitly by the name of the actual parameter. From the calling program the parameters that are to be passed to the function module are exported; those passed from the function module to the program are imported. You do not have to assign an actual parameter to an optional parameter. In this case, you can delete the line containing the optional parameter.
- Note that - during parameter assignment - the function module parameter is always on the left and the actual parameter on the right.

# Overview: Business Objects and BAPIs

Function Groups and Function Modules

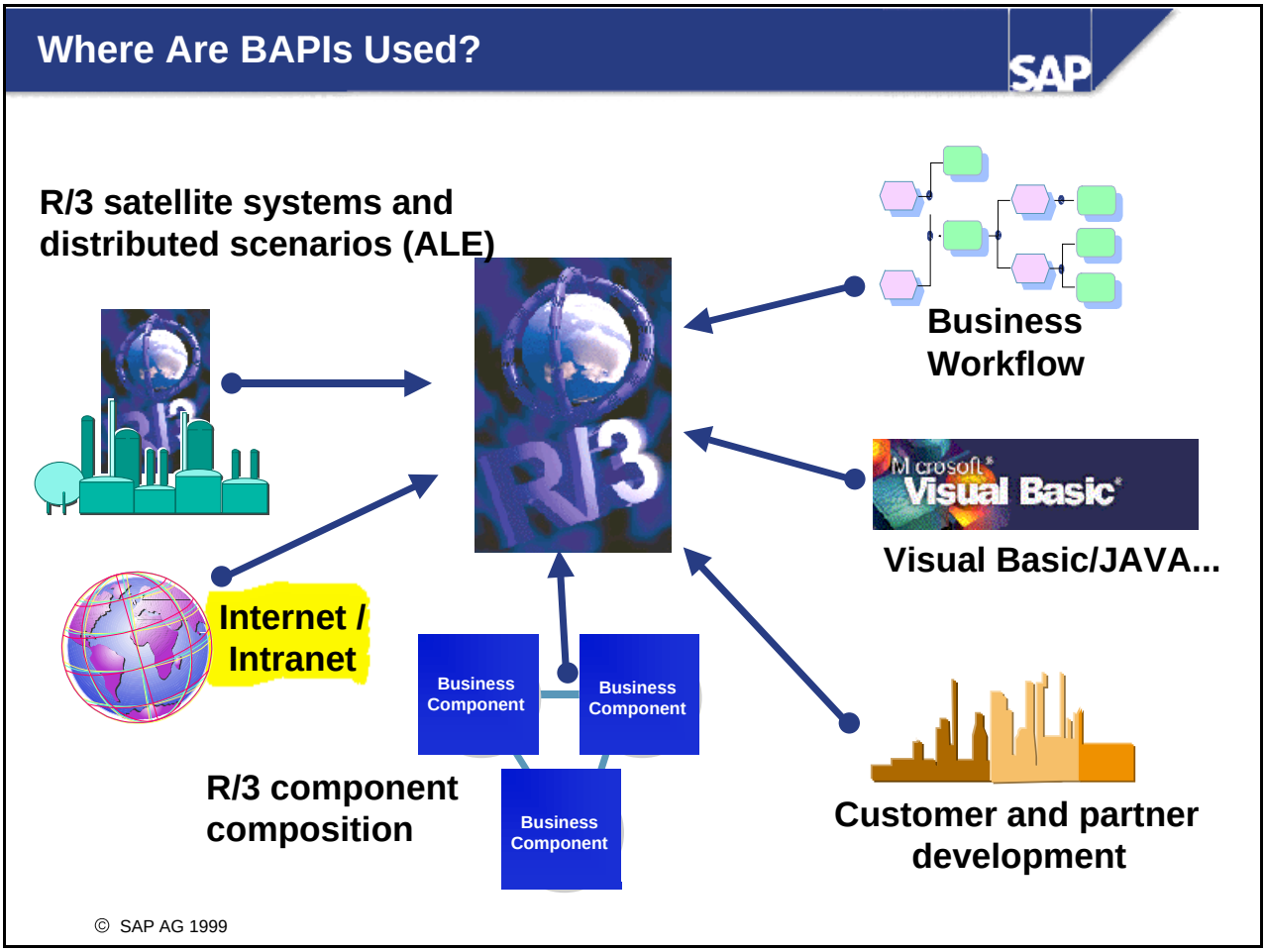
Business Objects and BAPIs

Objects and Methods

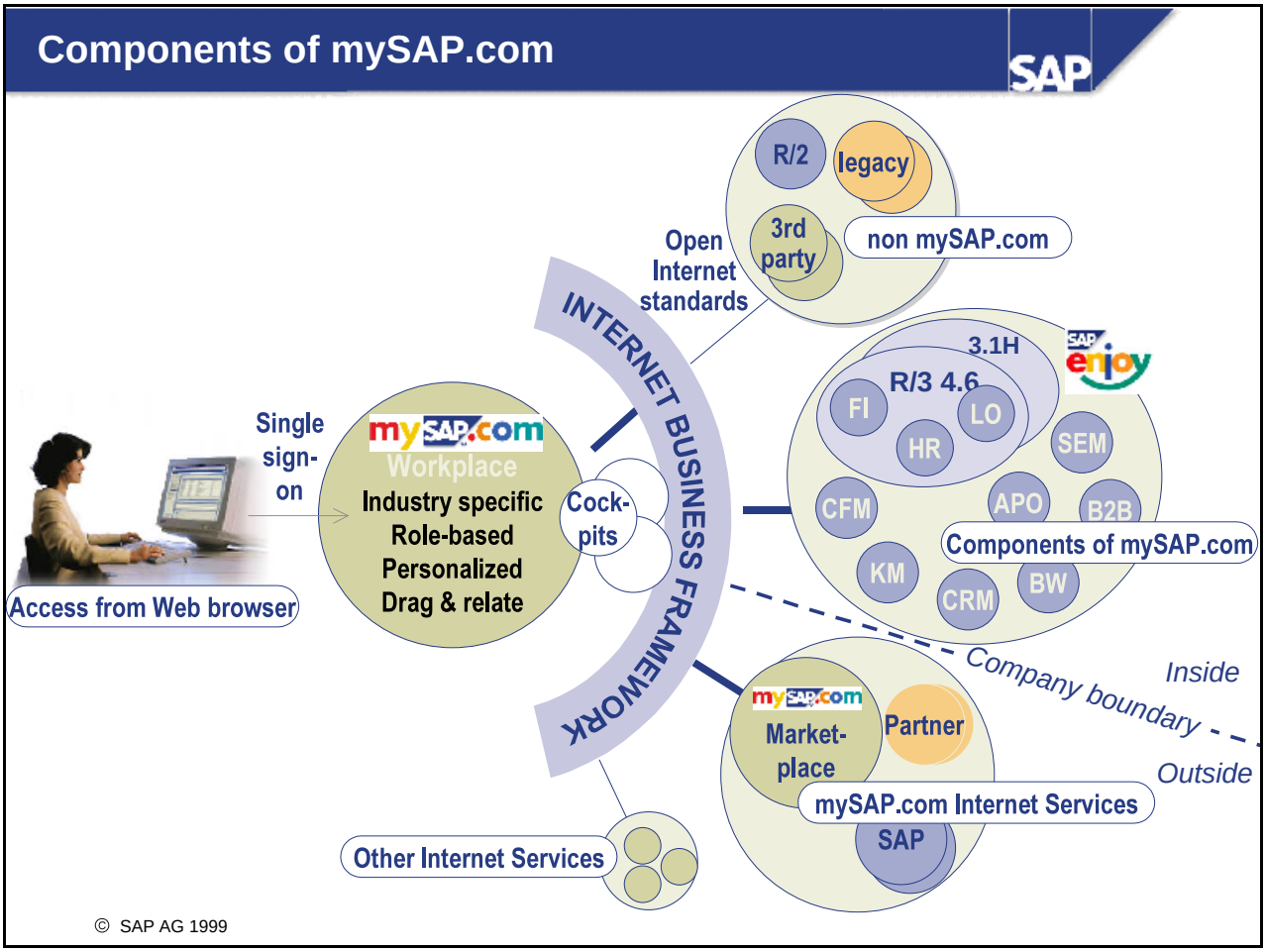
# Business Objects and BAPIs: Course Objectives

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

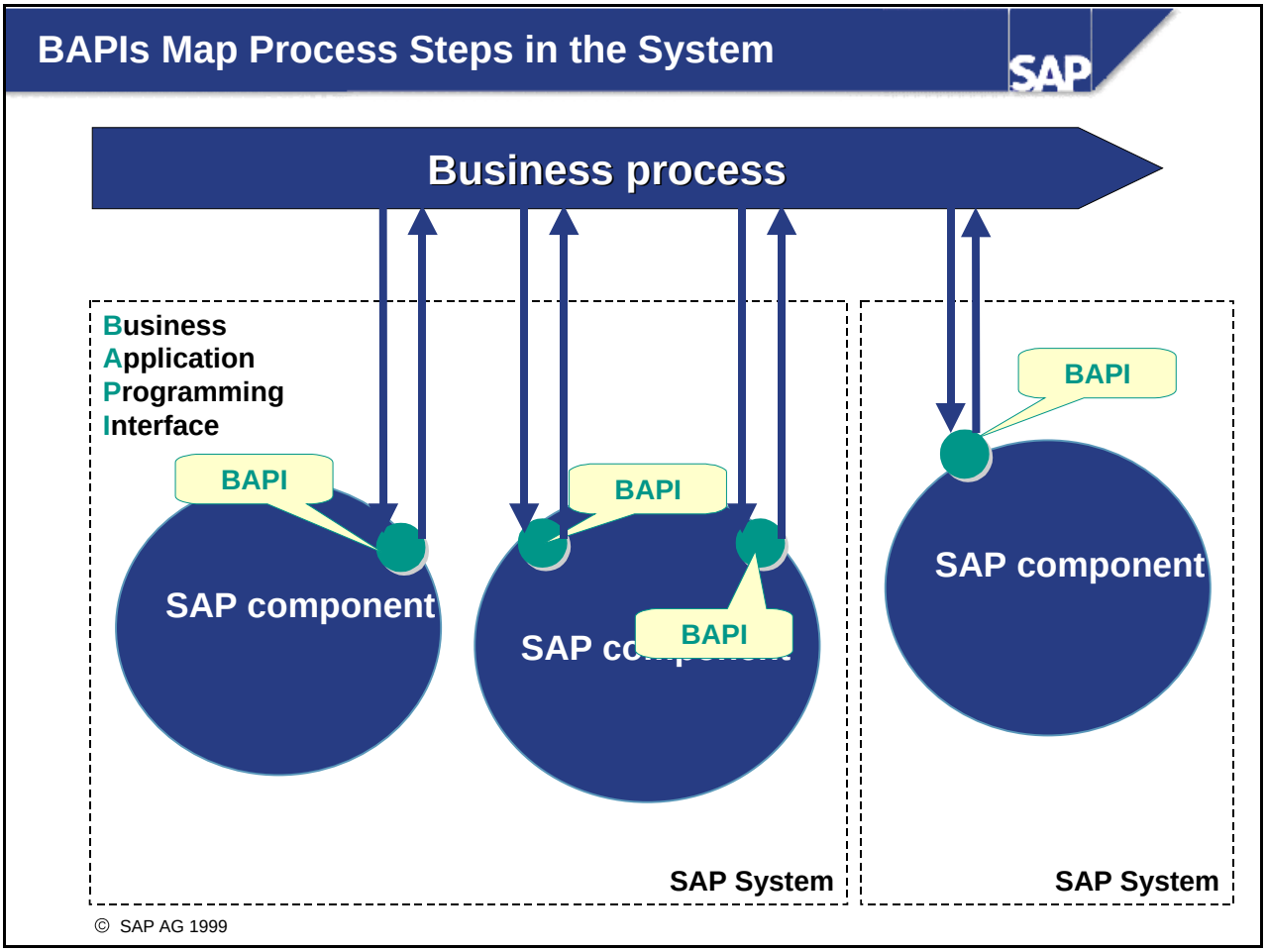
- **Find out important information about business objects and their methods (BAPIs) using the BAPI Explorer and Business Object Builder**



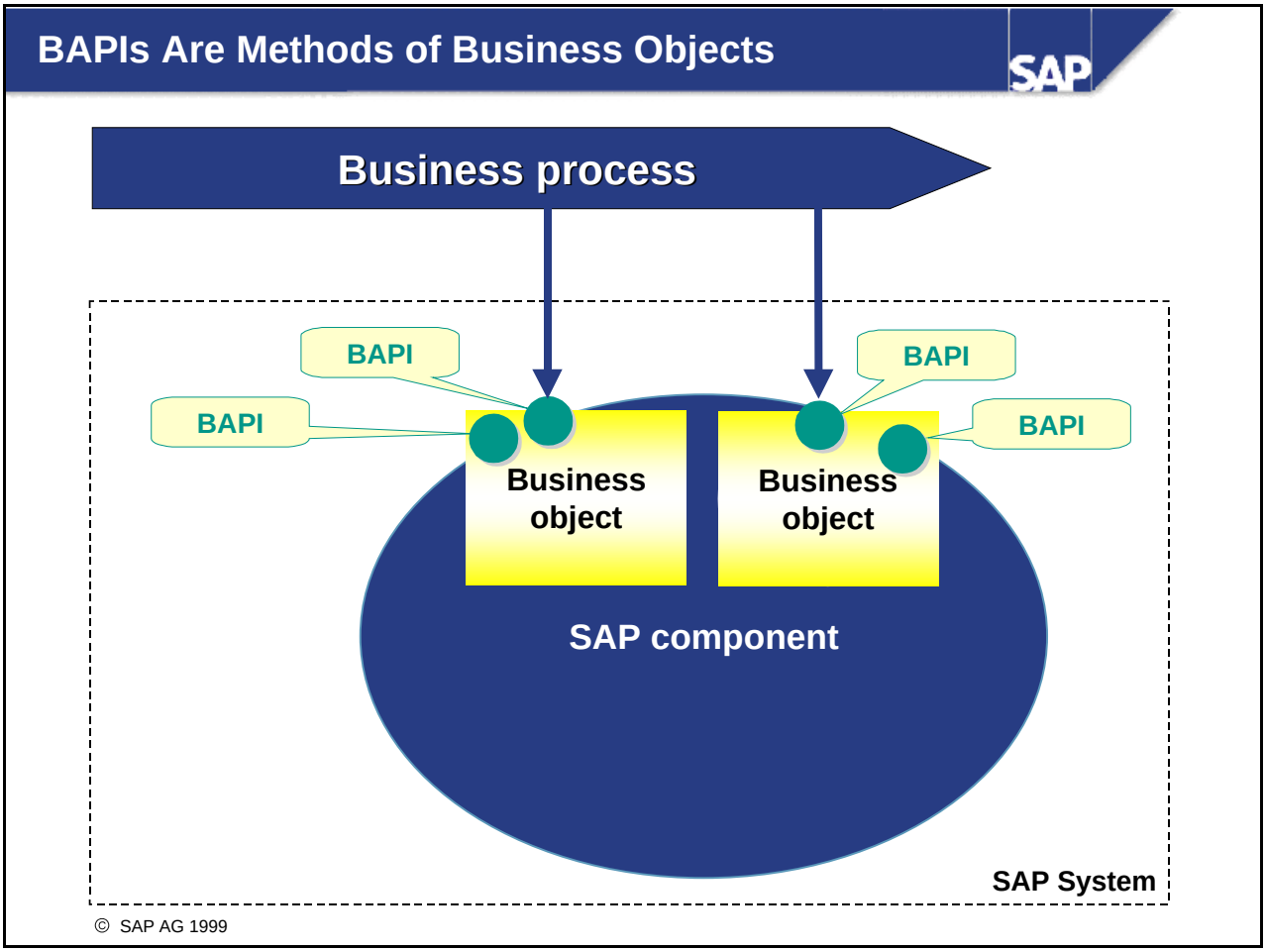

- A BAPI is an interface that can be used for various applications. For example:
  - Internet Application Components, which make individual R/3 functions available on the Internet or an intranet for users with no R/3 experience.
  - R/3 component composition, which allows communication between the business objects of different R/3 components (applications).
  - VisualBasic/JAVA/C++ - external clients (for example, alternative GUIs) that can access business data and processes directly.



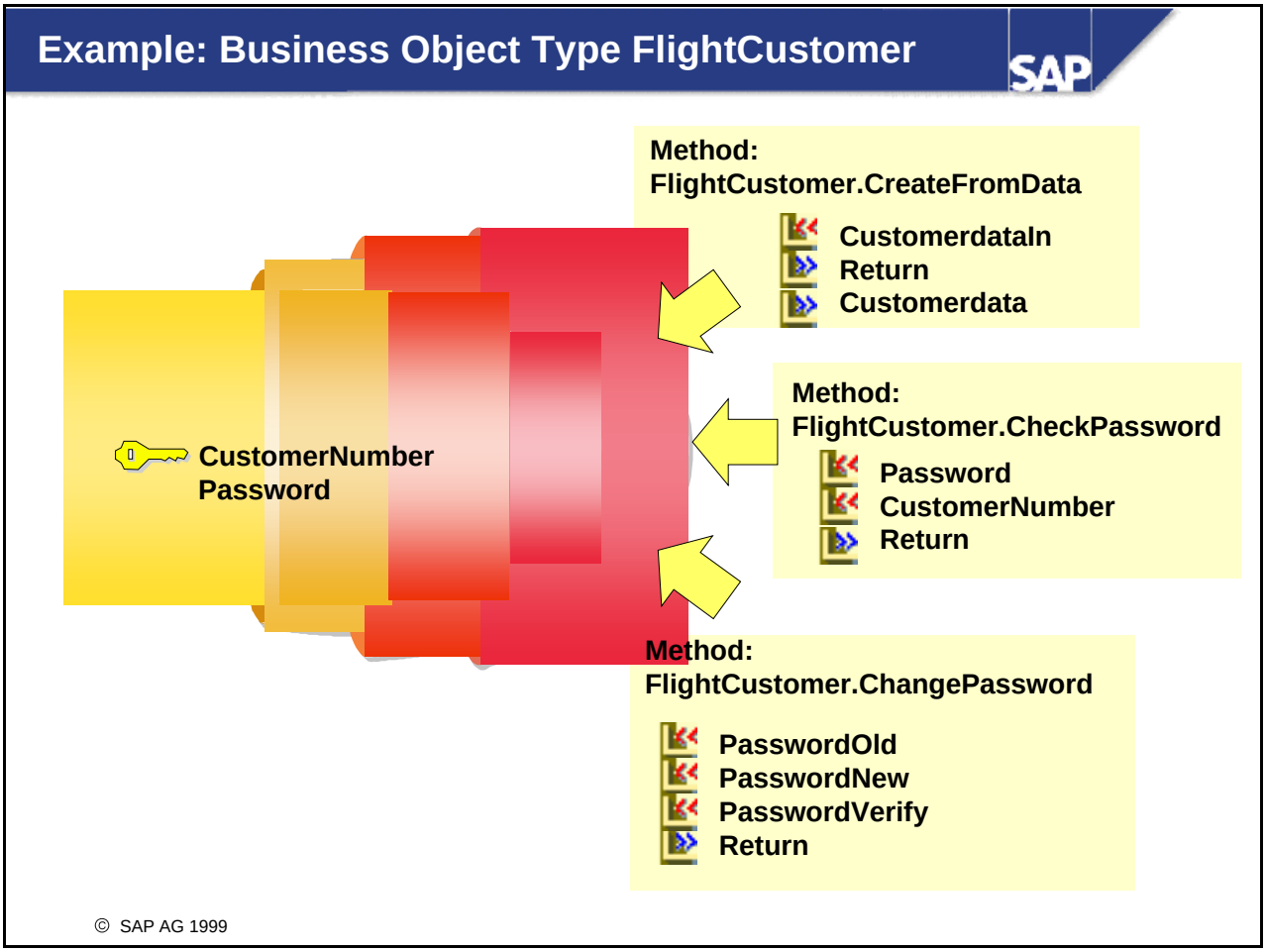

- In a mySAP.com system landscape there are usually several components used in the various systems. Because of this it is necessary to use a technique that can map business processes that extend across various systems.
- One example is the Workplace. The Workplace contains links to the functions of different components. The links can be:
  - Components of mySAP.com: Classic and new Web-based SAP transactions (R/3 standard system, New Dimensions, Industry Solutions), reports (for example, Business Warehouse reports with BW 2.0a), Knowledge Warehouse.
  - Non mySAP.com components:  External system used within open Internet standard.
  - mySAP.com Internet services: my.SAP.com Marketplace
  - All Internet and local intranet Web pages



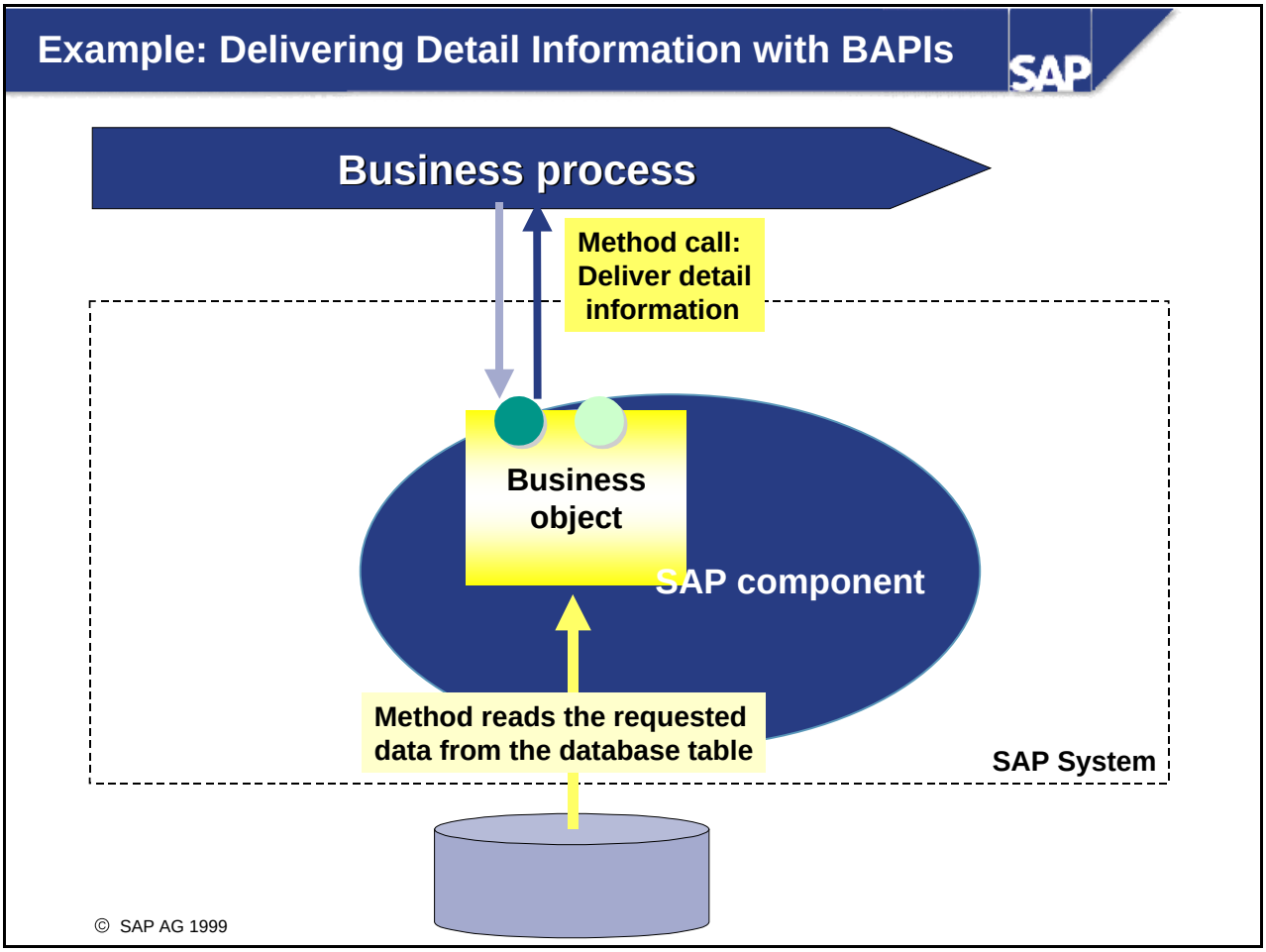

- BAPI stands for Business Application Programming Interface. A BAPI is a method that carries out a consistent process step of a business process in a SAP component. The scenario of a business process extends across several components. Therefore there must exist methods that can be accessed 'externally'. BAPI interfaces are designed so that you can use them without knowing all SAP details. For example, the names of interface parameters are in English. You do not need to know details of the database tables, consistency checks, and authorization checks to use BAPIs. The BAPI contains all necessary technical steps for executing a consistent process step. You can find details in the documentation of a BAPI.
- To simplify matters, the examples on BAPIs in this unit are based only on an SAP component. You can gain an overview of the different integration techniques in the course *BC095 Business Integration Technology*.



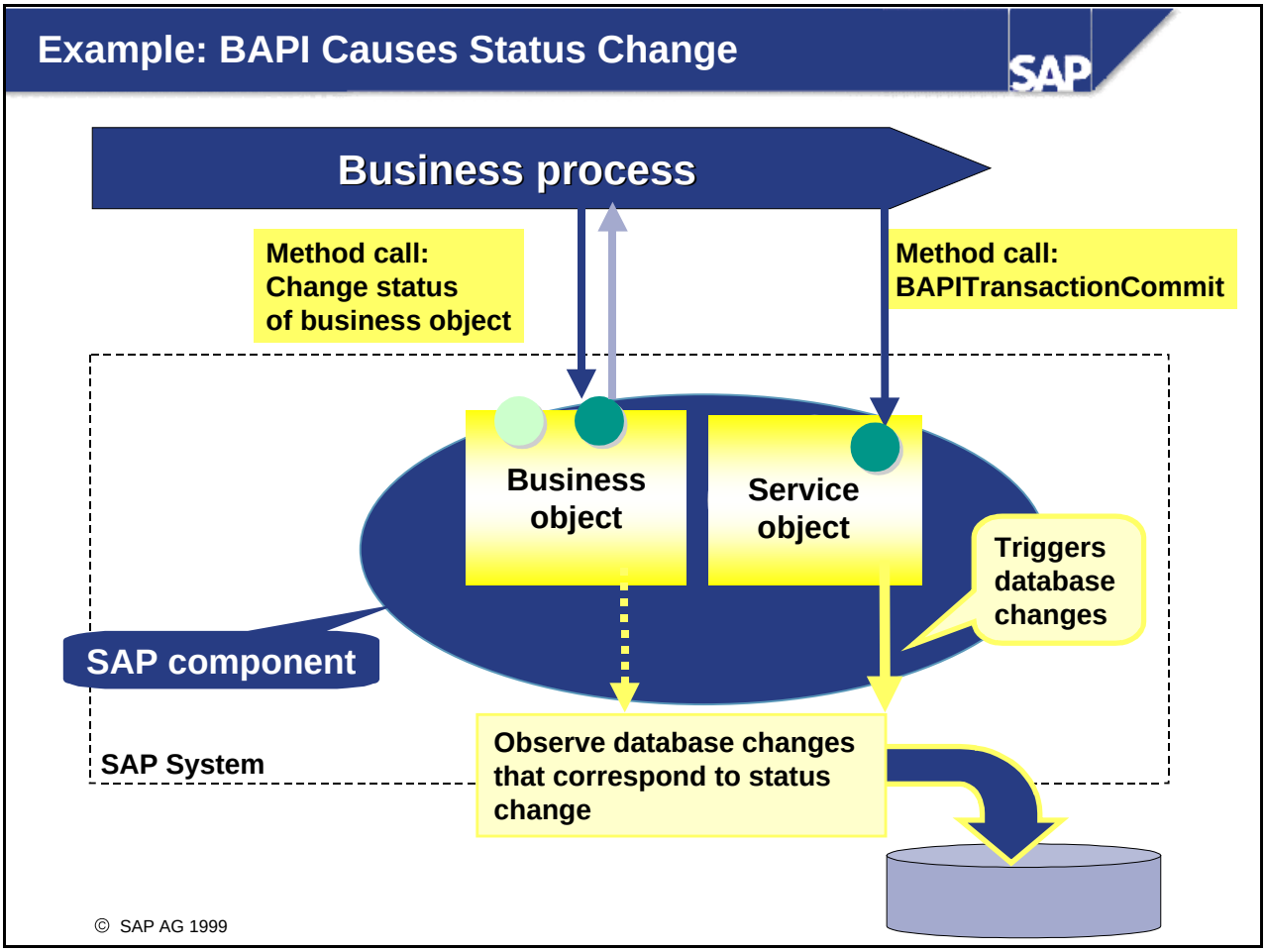

- BAPIs are defined as methods of a business object. The modeling of business objects corresponds to a business way of thinking rather than a numeration of available transactions.
- Example: When you think of a 'Flight booking', you think straight away of the activities that have an effect on the booking: Creating or canceling a booking, displaying details of a booking, and so on.
  The activities are mapped as methods that retrieve information or change the status. Technical details are, as far as possible, hidden inside a business object.



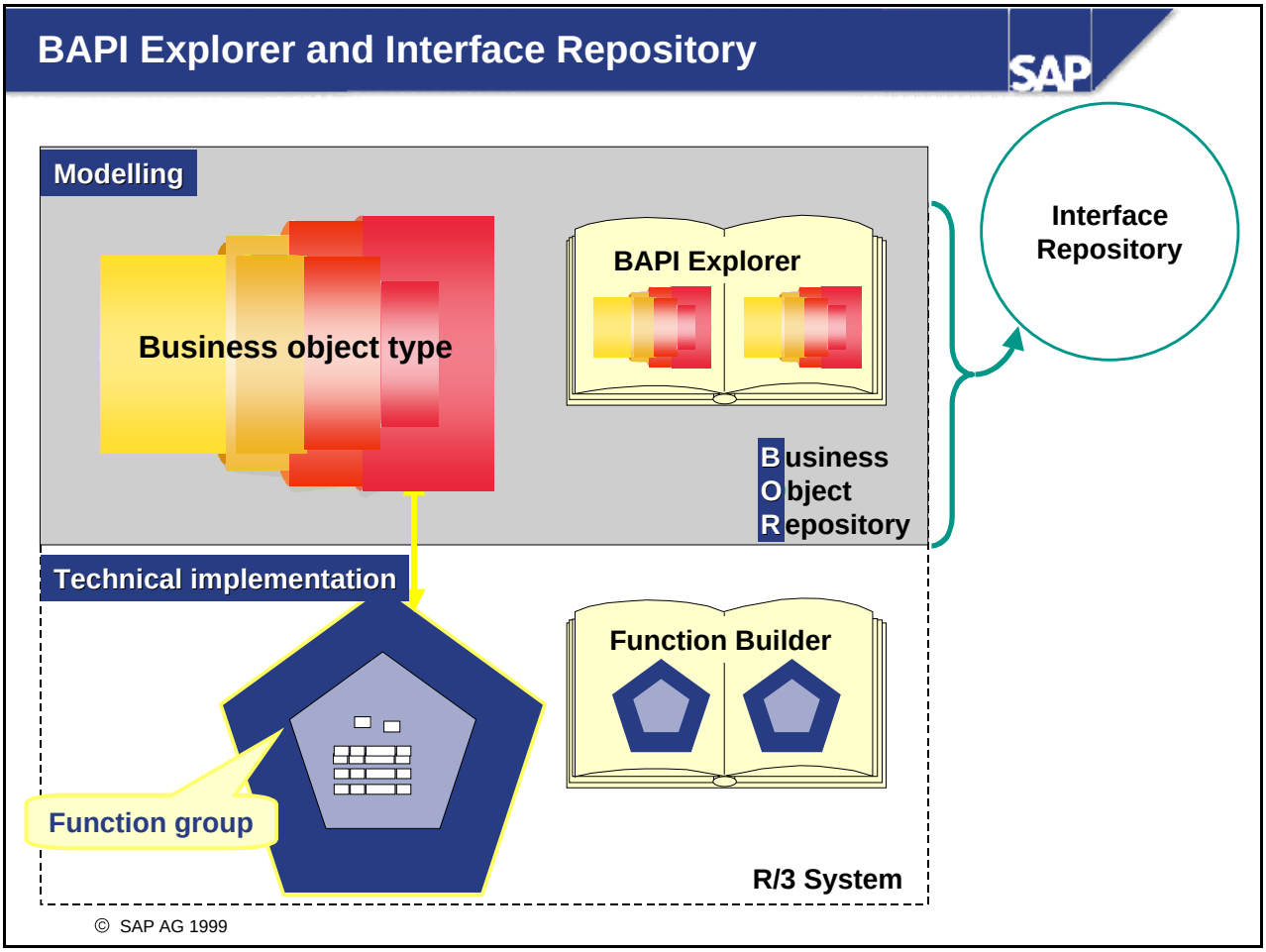

- An example business object from the training course data model is called FlightCustomer. It contains a customer for a flight. A customer can be uniquely identified using the key information **CustomerNumber.** The object also contains information about the current password of the customer. The following methods are available for this object:
  - **FlightCustomer.CreateFromData** Creates a booking.
  - **FlightCustomer.CheckPassword** Checks the password.
  - **FlightCustomer.ChangePassword** Changes the password.
- In this example no methods are given for reading the password. The password is a 'private' attribute that is 'hidden' in the object. (Secrecy principle of object orientation).




- The method GetDetail that exists for most business objects, provides detail information on a business object. The detail information is compiled at a business level. It is not important for the calling program to specify which database table the data will be read from.

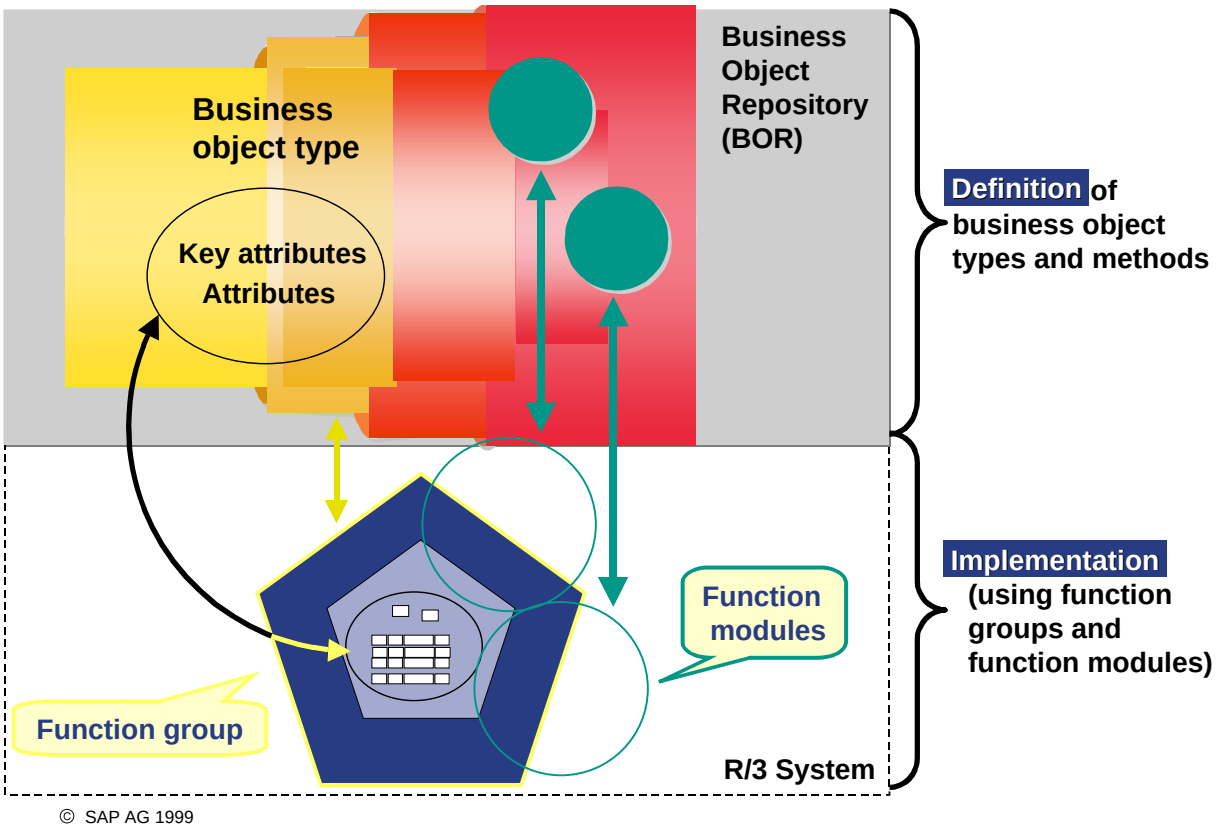



- Another example is a method that changes the status of business objects. In the case of business object FlightBooking this could be the creation of a booking. The method call itself causes only the change of the status of the (runtime) object. Within the BAPI, the steps that are necessary are already noted to save the status change persistently on the database.
- The database change can be triggered by calling a special BAPI (TransactionCommit) of a service business object.

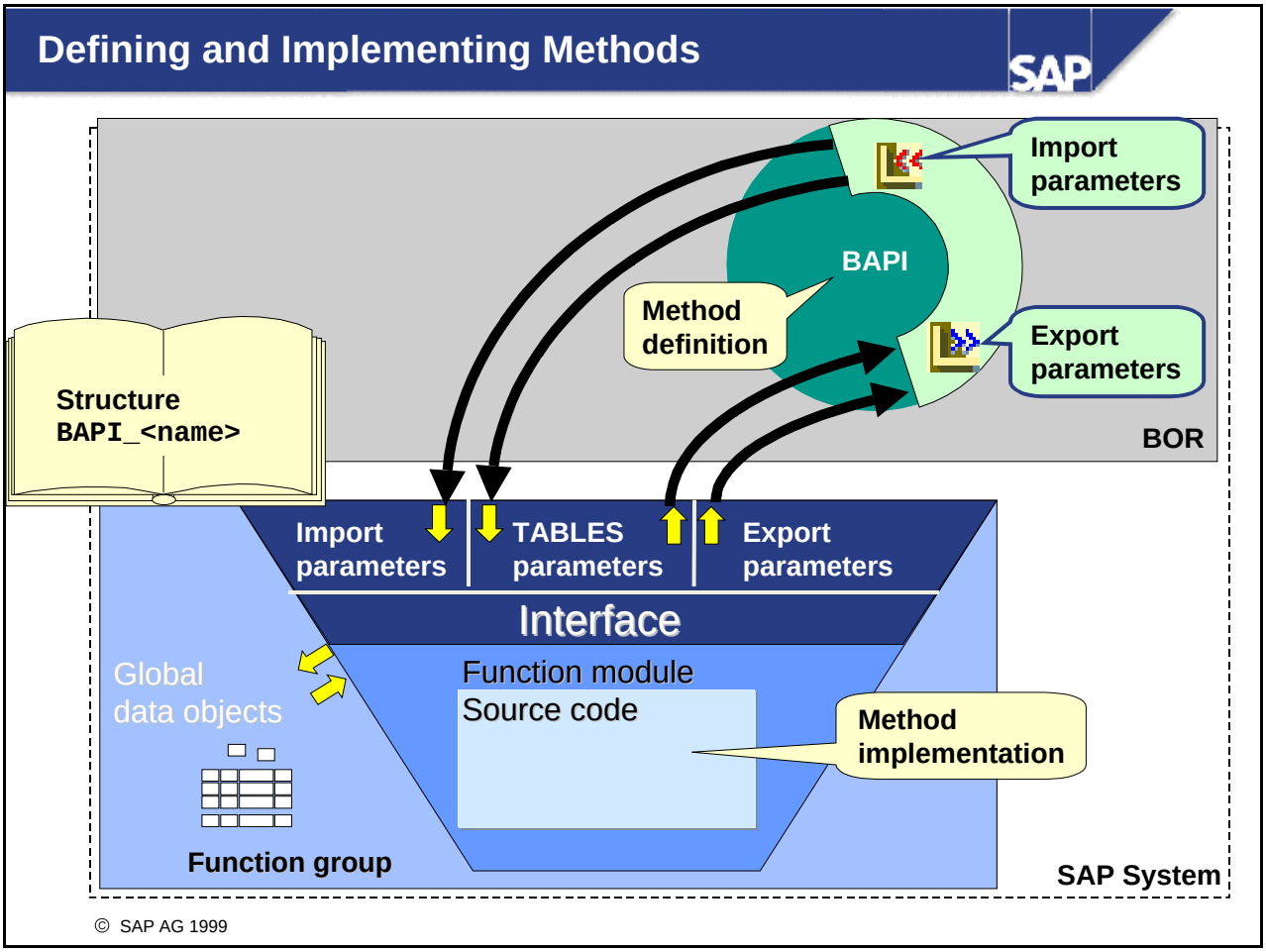



- The Business Object Repository (BOR) is a tool in which the modeling of business objects and their corresponding methods is stored.
- You can find information on existing business objects using the **BAPI Explorer** (transaction BAPI). You can navigate directly from this tool to other ABAP Workbench tools to find out details of technical implementations. The BAPI Explorer filters information from the Business Object Repository that is of interest to external users. For example, only released BAPIs appear.
- Outside of the SAP System you can look at information on BAPIs and their interfaces in XML in the Interface Repository. The Internet address for the Interface Repository is http://ifr.sap.com.
- The complete information on a business object is contained in the Business Object Builder.
- The technical implementation of business objects and methods is currently carried out using function groups and function modules. You can navigate directly from the BAPI Explorer to the display of a BAPI function module in the Function Builder.

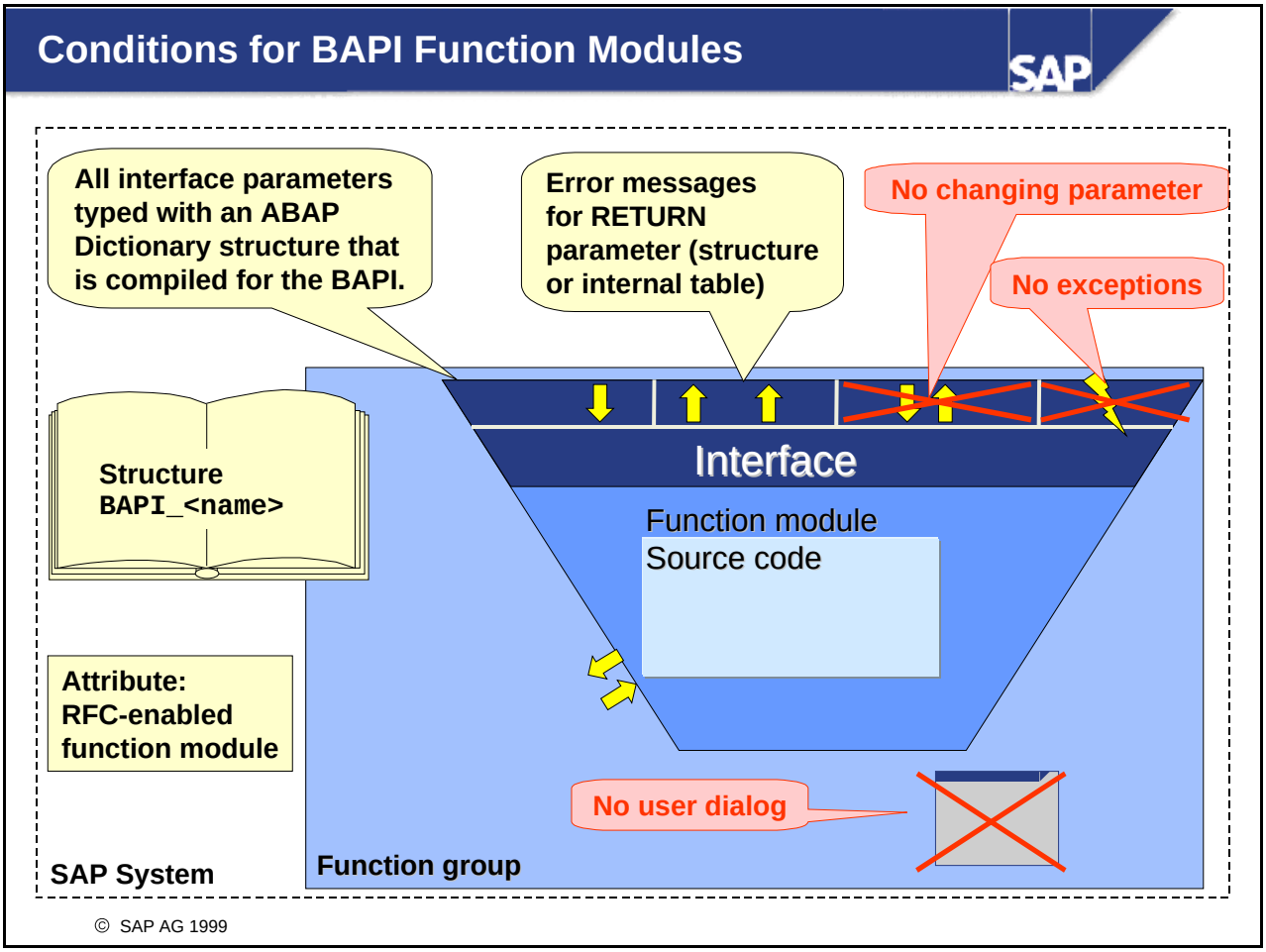
## Defining and Implementing Business Object Types



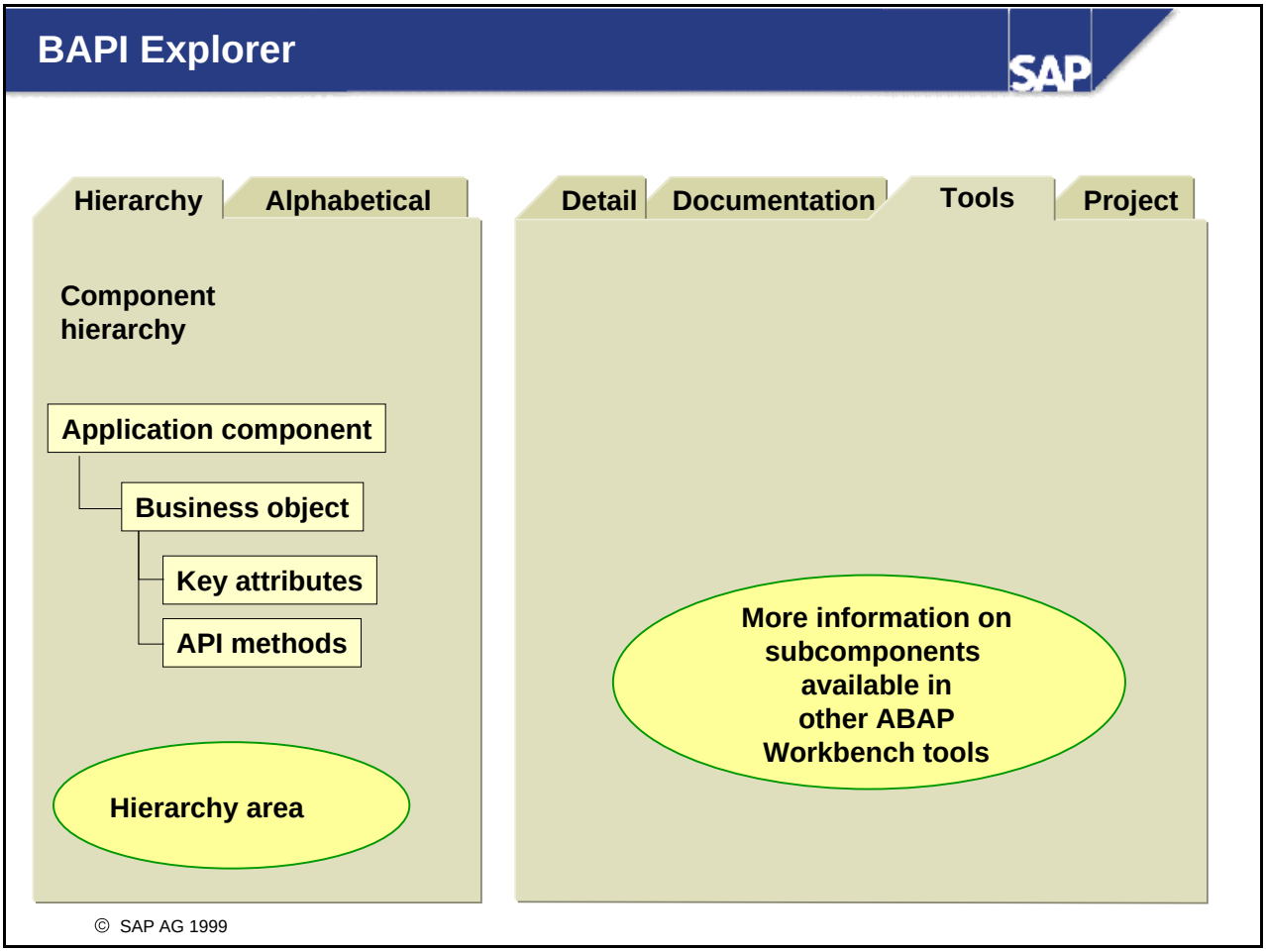

- Generally a function group exists for each business object type. The attributes are implemented using global data objects of the function group.
- Generally a function module of the function group exists for each business object type of the BAPI. The implementation of the method is carried out using a function module.



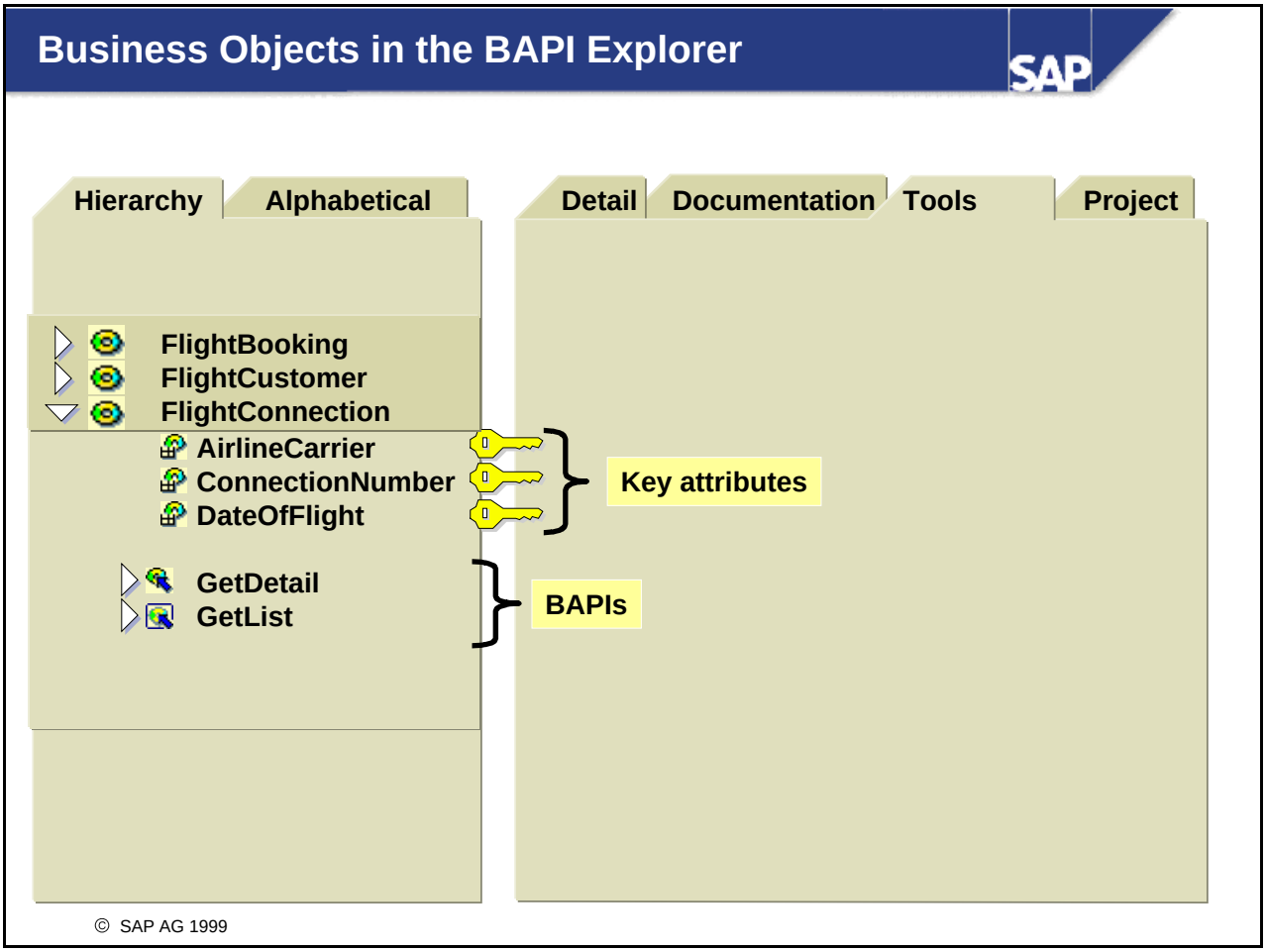

- The interface of the BAPI must be displayed in the BOR so that it corresponds to business logic. This is independent of whether technically defined attributes of the interface parameters can be completely hidden. In an ideal scenario the business object with the methods should remain as a 'modeling wrapper' without changing, even when the technical implementation changes.
- There are two types of interface parameters - import and export. Elementary parameters or flat structures are created using import or export parameters of the corresponding function module. Internal tables as parameters have a special characteristic: For RFC-enabled function modules the typing of interface parameters is not yet supported using complex dictionary types. For this reason **TABLES** parameters must be used. Parameters are classified in the Business Object Repository as an import or export parameter, even when technically **TABLES** parameters are always changing parameters.
- Note: RFC stands for Remote Function Call. A RFC-enabled function module can be called from outside the SAP System or from another SAP component.




- Function modules for BAPIs must fulfill the following requirements:
  - Naming convention: **<BAPI>_<Business Object>_<method>.**
  - Function module attributes: RFC-enabled
  - The function module must not contain user dialogs. This includes messages.
  - The function module must not contain exceptions. Errors are reported to the user with export parameter **RETURN.**
  - The function module must not contain **CHANGING** parameters.
  - All interface parameters are typed with an ABAP Dictionary structure that is compiled for the BAPI. The structures have the naming convention **BAPI_<structure name>.** These structures are 'frozen' and must not be changed incompatibly.




- You can display more information on business objects and the BAPIs that belong to them using BAPI Explorer Information. The screen is divided into two parts: a hierarchy area and a details window. The hierarchy area displays the component hierarchy. You can expand an application component to find out which business objects belong to it. If you expand a single business object, the system displays a sub-tree, showing you which key attributes and API methods belong to it. (API stands for Application Programming Interface).



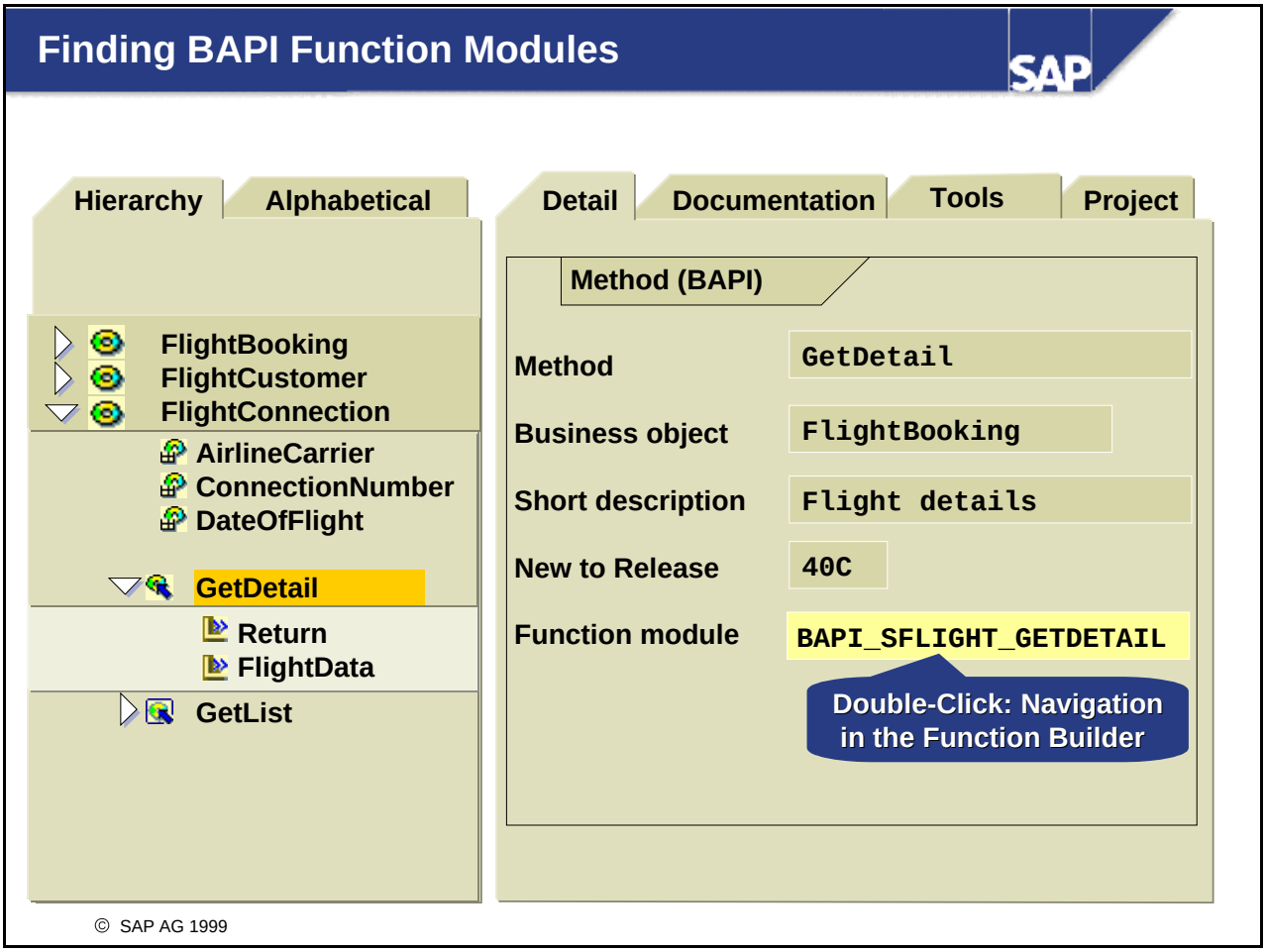

- If you expand a sub-tree for a business object in the BAPI Explorer, the system displays the following:
  - **Key attributes:** Provide a unique identifier for each business object
  - **Instance-specific methods:** Methods that are bound to the instance identified by the key attributes The business object type FlightBooking has one instance-specific method, **GetDetail** that returns a structure with booking details. Key attribute values must be passed to this method.
  - **Non-instance specific methods:** Can be called by all instances of an object type FlightBooking has one such method, **GetList** that returns a list of all bookings, for which a business object already exists at runtime.

## Standardized BAPIs

- **GetList**
  - Gets list of object key fields that satisfy selected criteria (search function)
- **GetDetail**
  - Retrieves details (attributes) about an object whose complete key is given
- **Create, Change, Delete, Cancel**
  - Creates and changes business objects in R/3
- **AddItem, RemoveItem**
  - Adds and removes subobjects (for example, item for an order)

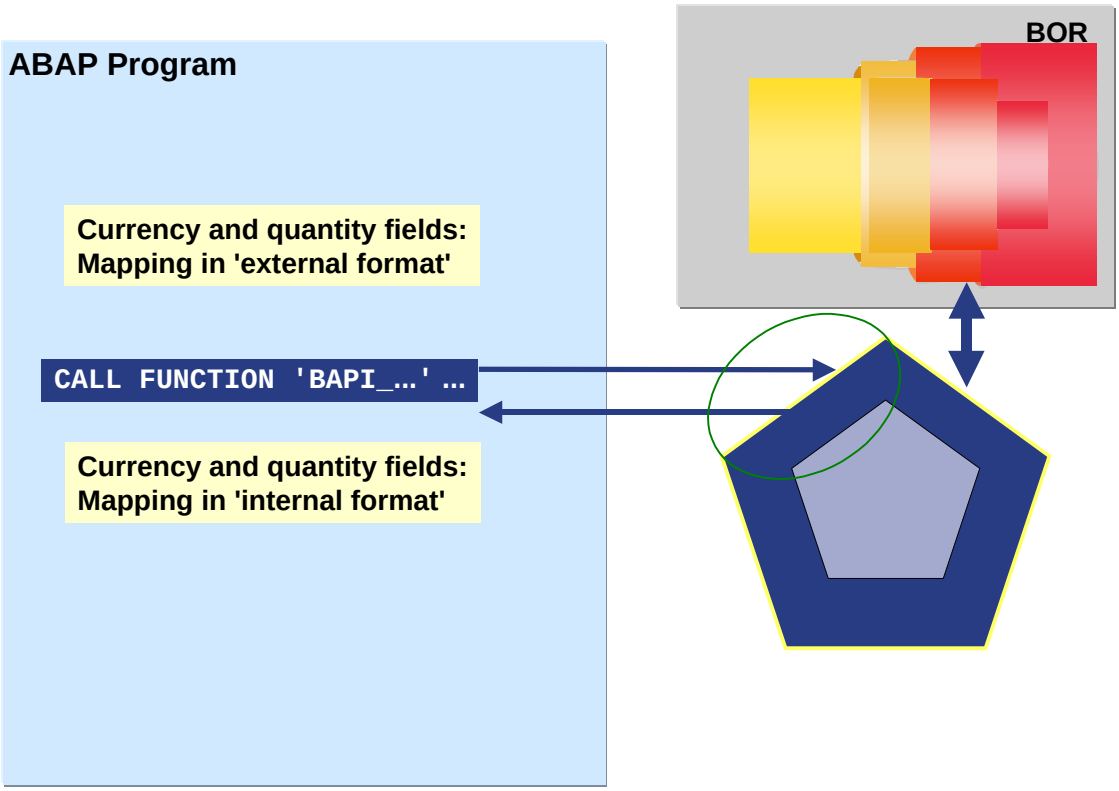


- BAPIs with standardized names contain standardized methods. These are a few of the most important standardized BAPIs.




- In Release 4.6 BAPIs are implemented using function modules. You can display the function module for the BAPI you have chosen using the BAPI Explorer.
  - Select the BAPI in the hierarchy area.
  - In the detail information display window, choose the *Detail* tab.
  - By double-clicking on the name of the function module you can display the function module in the Function Builder.

## Calling a BAPI Function Module from an ABAP Program






- If you would like to use a BAPI in an R/3 System, you can directly call the function module containing it. Note that information about any errors that occur are passed to the program using the interface parameter **RETURN**. BAPI function modules do not contain either exceptions or user dialogs. They exist only to encapsulate business logic .
- BAPI interfaces are created according to the needs for the 'external' call, a non R/3 System. Quantities are expected in an 'external compatible' format with 4 or 9 decimal places. The quantities must be transferred to the interface, even when the corresponding currency has no decimal places. When you call a BAPI function module from an ABAP program, you will generally get the values in internal (SAP) format, which is how the data is saved on the database. In this case you must execute mapping of the external format before the BAPI call. For mapping you can use function modules from function group **BACV** (development class **SBF_BAPI**).

# Overview: Objects and Methods

Function Groups and Function Modules

Business Objects and BAPIs

Objects and Methods

# Objects and Methods: Course Objectives

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

- **Find out important information on classes and their methods using the Class Builder**
- **Write a program that displays a simple list using an ALV grid control, and contains the following**
  - **Reference variables**
  - **A CREATE OBJECT statement**
  - **Calling of methods**
  - **A container area on a screen**

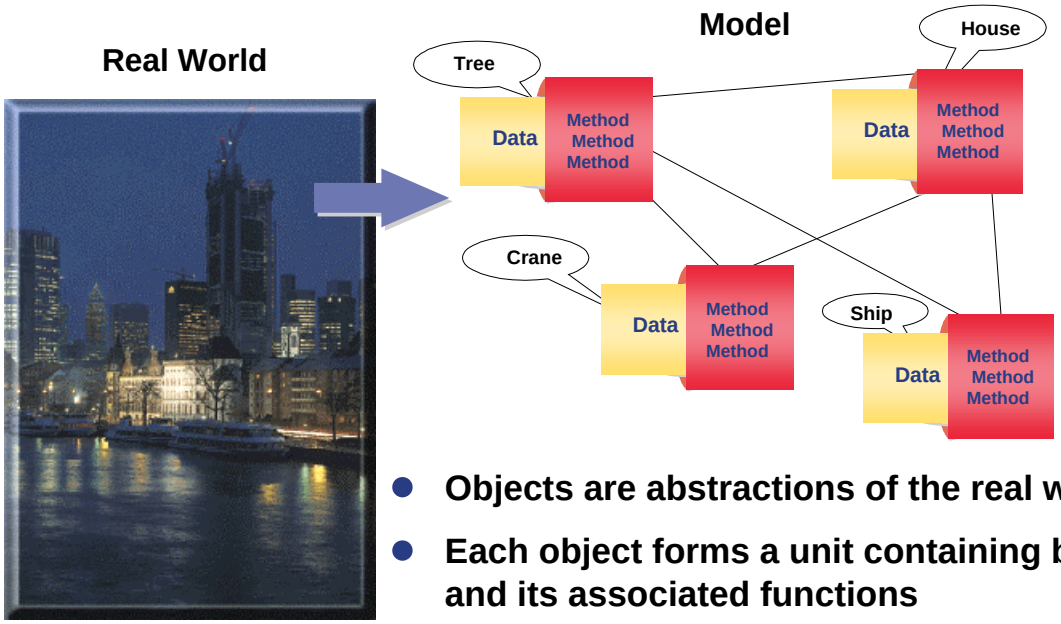
## Benefits of Object-Oriented Programming

- **Integrated software development process**
  - **Facilitates communication between users and developers**
- **Encapsulation**
  - **Programs are clearer and easier to maintain**
- **Polymorphism**
- **Inheritance**



- **Integrated Software development process:** Each phase in the development process (analysis, specification, design, and implementation) is described in the same "language." Ideally, this means that changes you make to the design during implementation can be applied retrospectively to the data model.
- **Encapsulation (information hiding):** The ability to hide the implementation of an object from other system components. The components cannot make assumptions about the internal status of the object, and do not depend on using a particular implementation to communicate with the object.
- **Polymorphism:** In object technology, the fact that objects of different classes react differently to the same message.
- **Inheritance:** Defines the implementation relationship between classes, such that one class (the subclass) shares the structure and behavior that have already been defined in one or more superclasses.

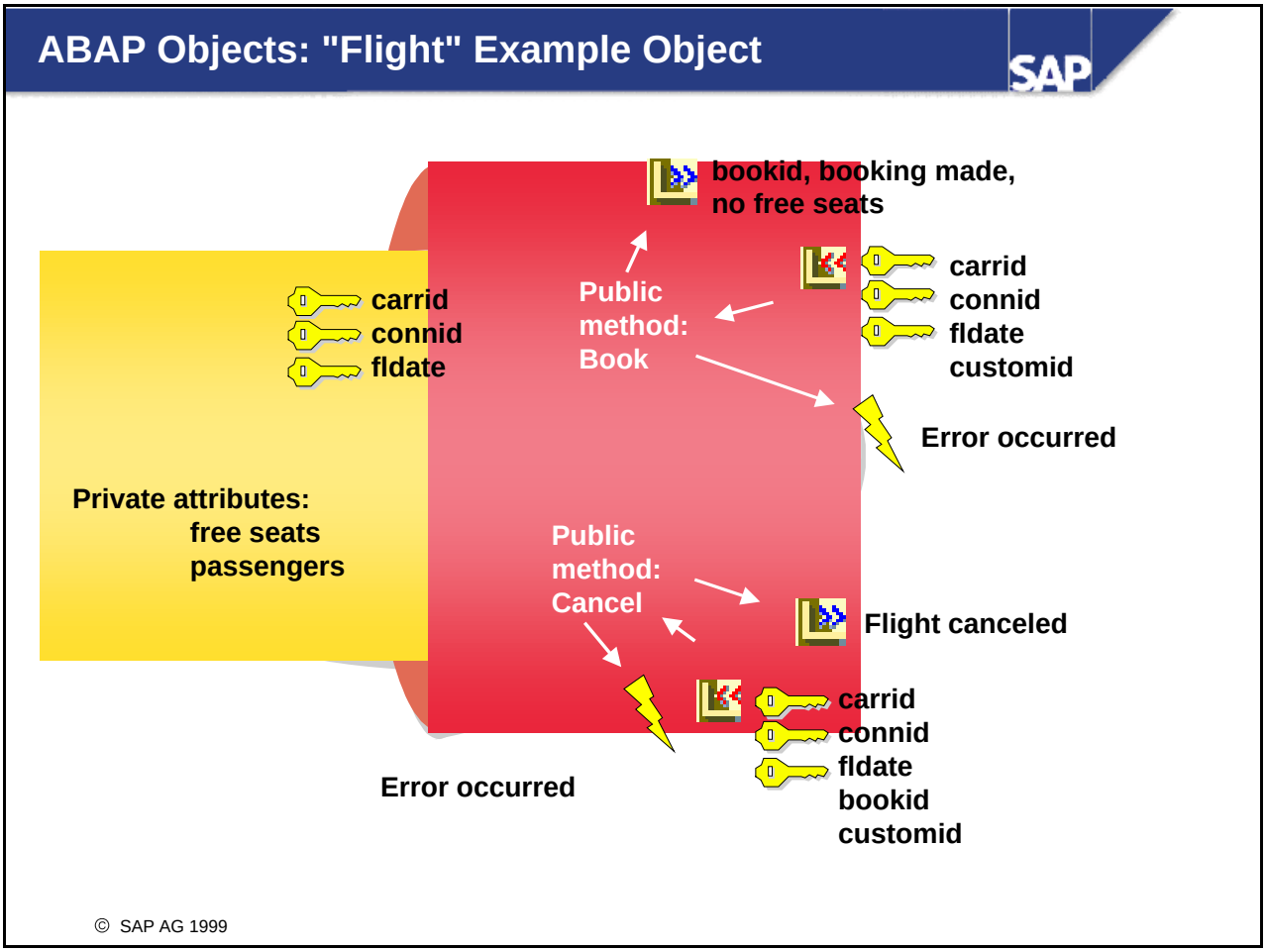
## Real World / Functions / Objects




- **Objects are abstractions of the real world**
- **Each object forms a unit containing both data and its associated functions**



- Objects are central to the object-oriented approach and represent concrete or abstract entities in the real world. They are defined according to their properties, which are depicted using their internal structure and attributes (data). Object behavior is described using methods and events (functions).
- Each object forms a capsule, which encompasses both its character and behavior. Objects should enable the model of a problem area to be reflected as closely as possible in the design model for its solution.



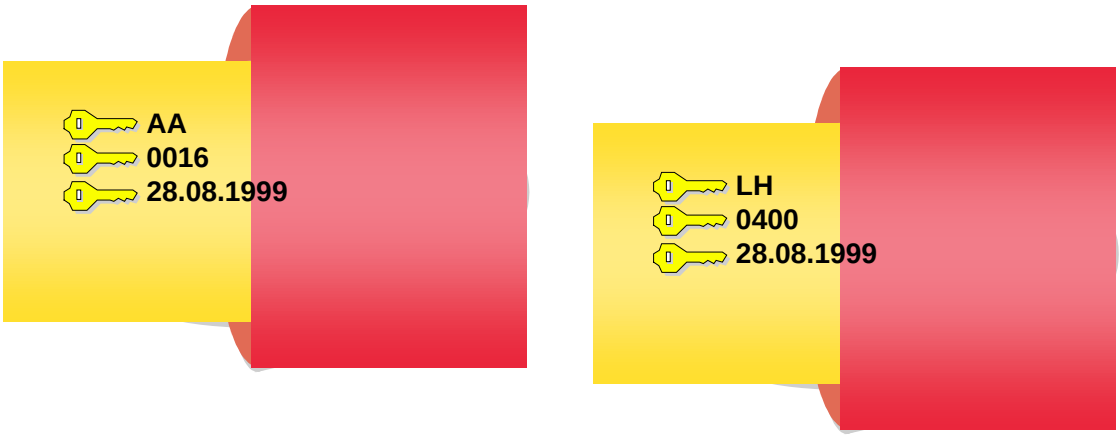

- As an example, consider the "flight" object displayed above.
- The object contains private attributes that pertain to flights.
  - Key attributes: The airline, the flight, and the departure date combined provide a unique identifier for each flight. Each flight number also contains: the airport from which the flight departs; the time of departure; and the destination airport.
  - Booking list: The list of people who have booked seats on the flight along with their booking numbers.
  - Flight information: Such as the airplane type and maximum number of seats.
- Local methods: The object can calculate the number of free seats from the private attributes "booking list" and "maximum number of free seats".
- The object contains an interface with two methods
  - "Book" method: If this method is called from outside the object, and provided the necessary data has been passed to the interface, the method uses the private attributes to determine whether or not there is a free seat on the flight. If there is, the new customer is included in the booking list and a success message is passed to the calling program. Otherwise, the system returns the information that the booking could not be made because the flight is already fully booked.
  - "Cancel" method: Again, if this method is called from outside the object, and provided the necessary data has been passed to the interface, the method uses the private attributes to determine whether or

not the customer is included in the booking list. If so, his or her booking is cancelled and a success message returned to the calling program. If the customer is not in the booking list, the system displays an error message to this effect.

# Example Scenario: Changing a Flight Booking

**Only cancel the first flight if the new booking is successful**

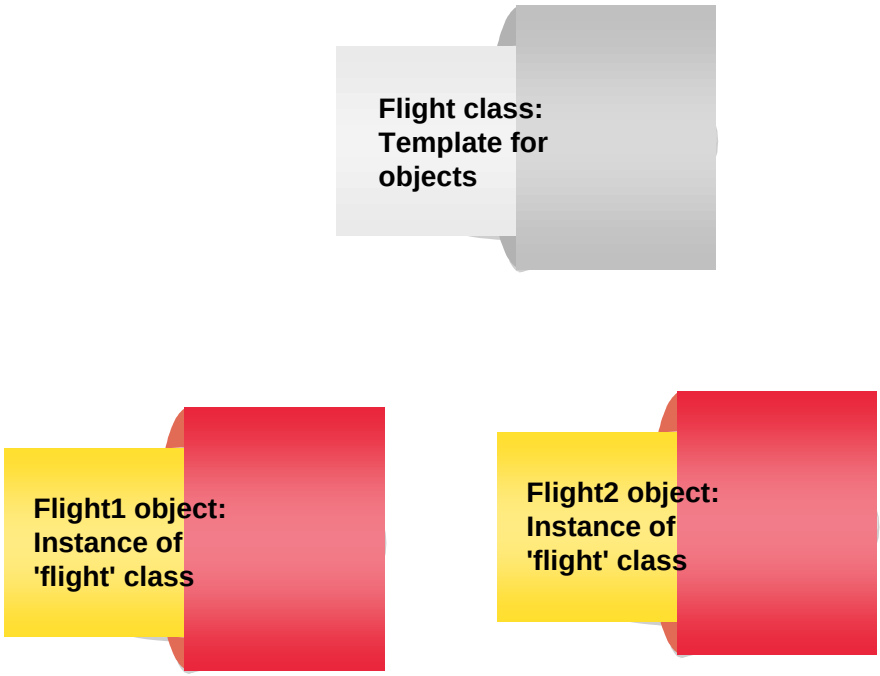
**=> Two instances are required, one for each flight**





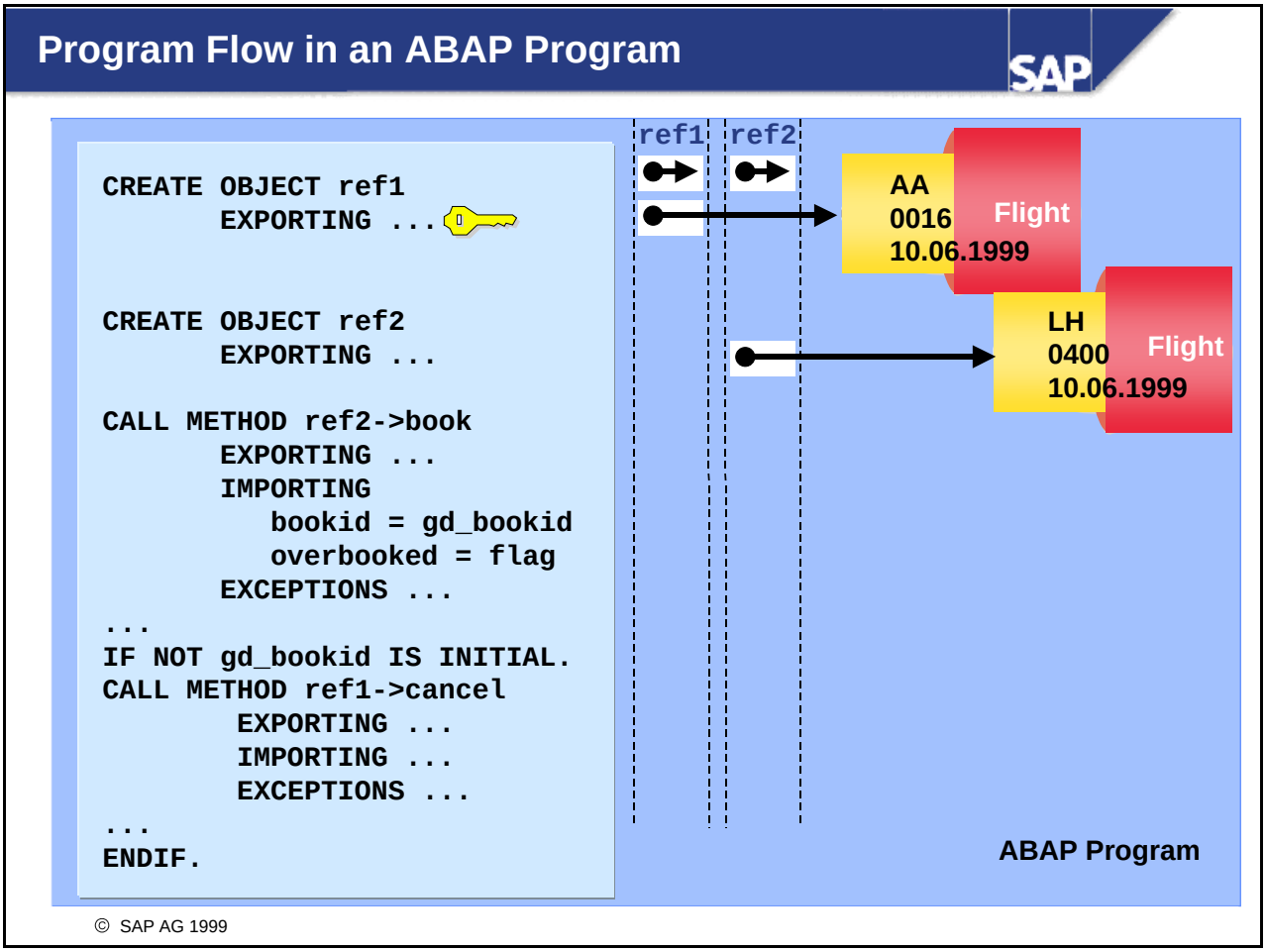

- Generally, when customers change a booking in a travel agency, they want to be sure that they have a seat on their new flight before they cancel the first.
- Technically, this means that there are two objects of the same type, but with different key attributes.

## Objects Are Instances of a Class






- In object-oriented programming, this is implemented such that each class is defined as an object type. Instances of this class are created at runtime. That is, the system creates objects of an object type (and thus, of the class).




- An ABAP program that changes bookings using objects has the following program flow:
- The program starts and the program context is loaded. Memory areas are made available for all the program's global data objects. Reference variables are also made available for each object. You can view a summary of the data objects that are made available when you run the program by expanding the *Fields* and *Dictionary structures* subtrees in the program object list. You can also navigate to the source text in which the data objects have been defined - for example, using a **DATA** or **TABLES** statement. The reference variables are defined using a **DATA: <ref> TYPE REF TO <class>** statement.
- The objects are generated at runtime, as soon as the **CREATE OBJECT** statement is processed. The system needs to tell the statement which reference variables the object should point to. The import parameters must be passed using a special method, the **CONSTRUCTOR.** In this example, only the key attributes need to be passed to the statement.
- As soon as the **CALL METHOD** statement is processed, the method is called. Unlike calling a function, when a method is called, the object in which the method is to be processed must be stated explicitly. The system specifies a reference variable pointing to the object. The reference variable name is followed by a -> and the method name.

## Application Areas of ABAP Objects

- **Office Integration**
- **Business Add-Ins (new enhancement concept)**
- **Controls**



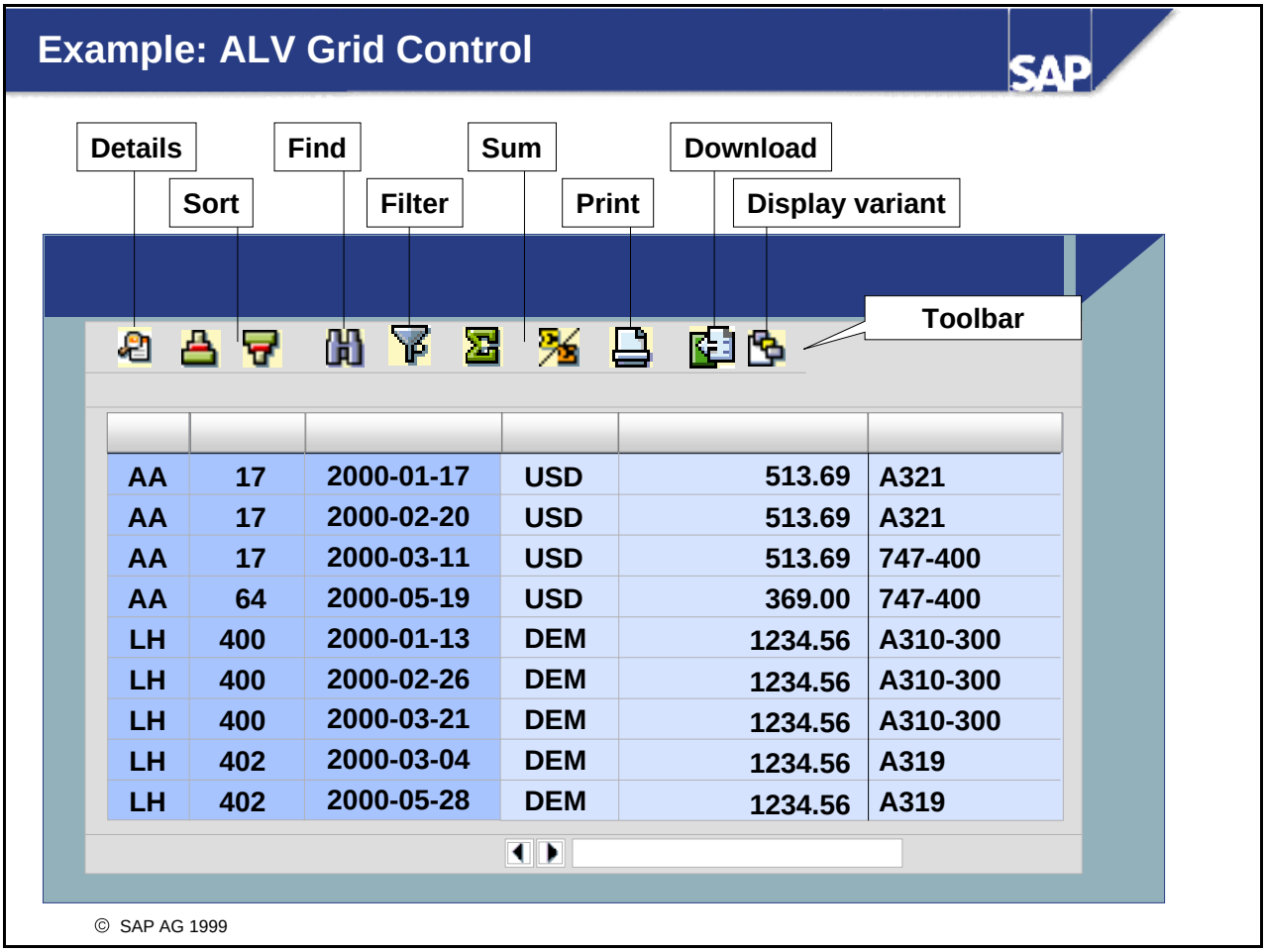
- In Release 4.6, the most important aspects of the system for object-oriented enhancements of the ABAP language are:
  - **Office Integration:**
    The system offers a new object-oriented interface, which will help you to make use of R/3 office product functions.
  - **Business Add-Ins:**
    An object-oriented enhancement technology, which combines the advantages of existing technologies. If Business Add-Ins are included in standard programs, you can enhance the program using special methods, without having to carry out a modification.
  - **Controls:**
    The R/3 System allows you to create Custom Controls using ABAP objects. The application server is the Automation Client, which drives the custom controls (automation server) at the front end. This task is performed by the Central Control Framework.
- Some parts of SAP's own aplications have been re-designed using object-oriented principles - for example, the new ABAP Workbench.

## ALV Grid Control: What are controls?

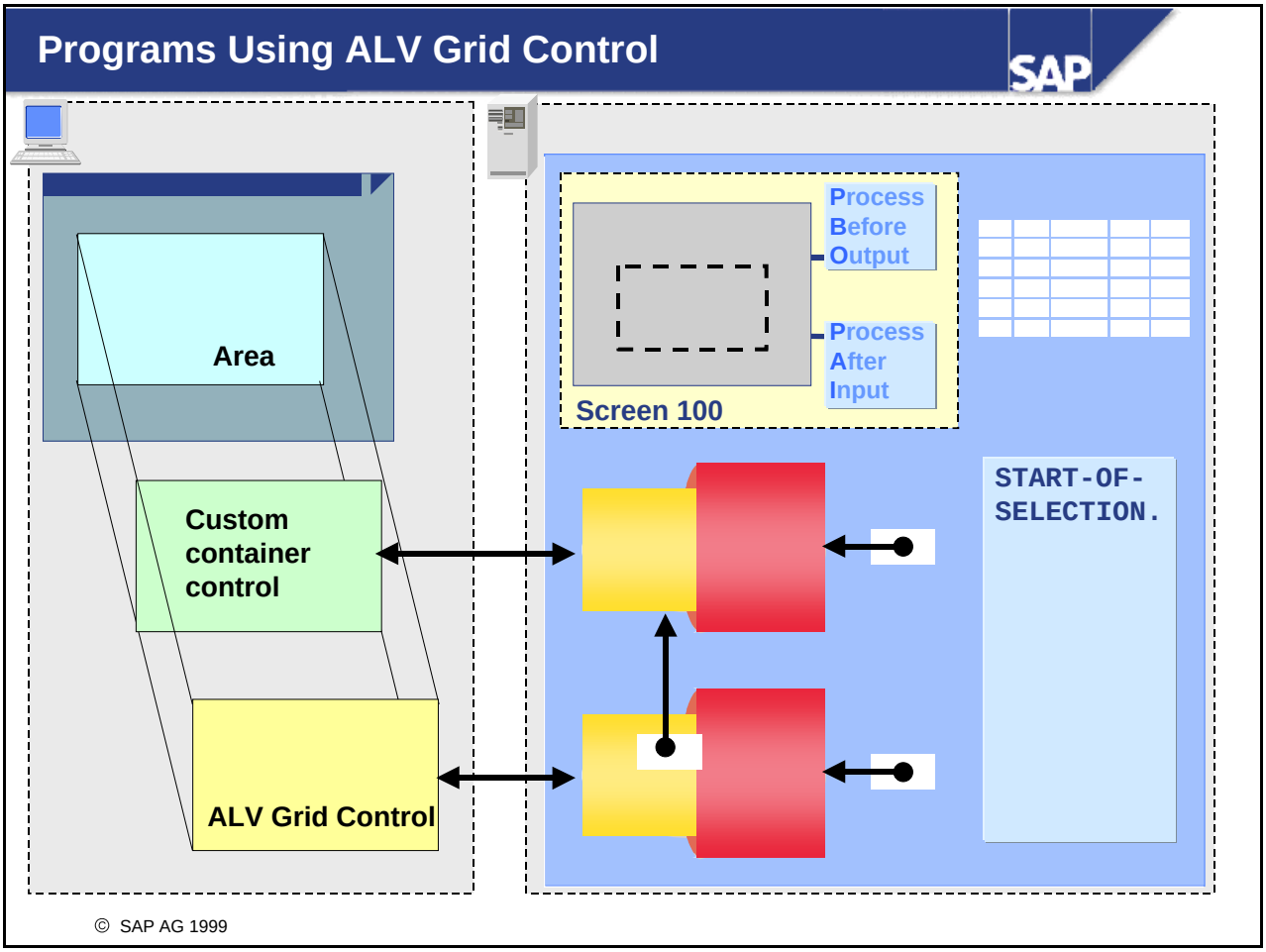
- Independent binary software components
- Installed locally on the front end using SAPGUI
- Move functions from the application server to the frontend
- Have a wrapper class in ABAP Objects
- Intended for reuse



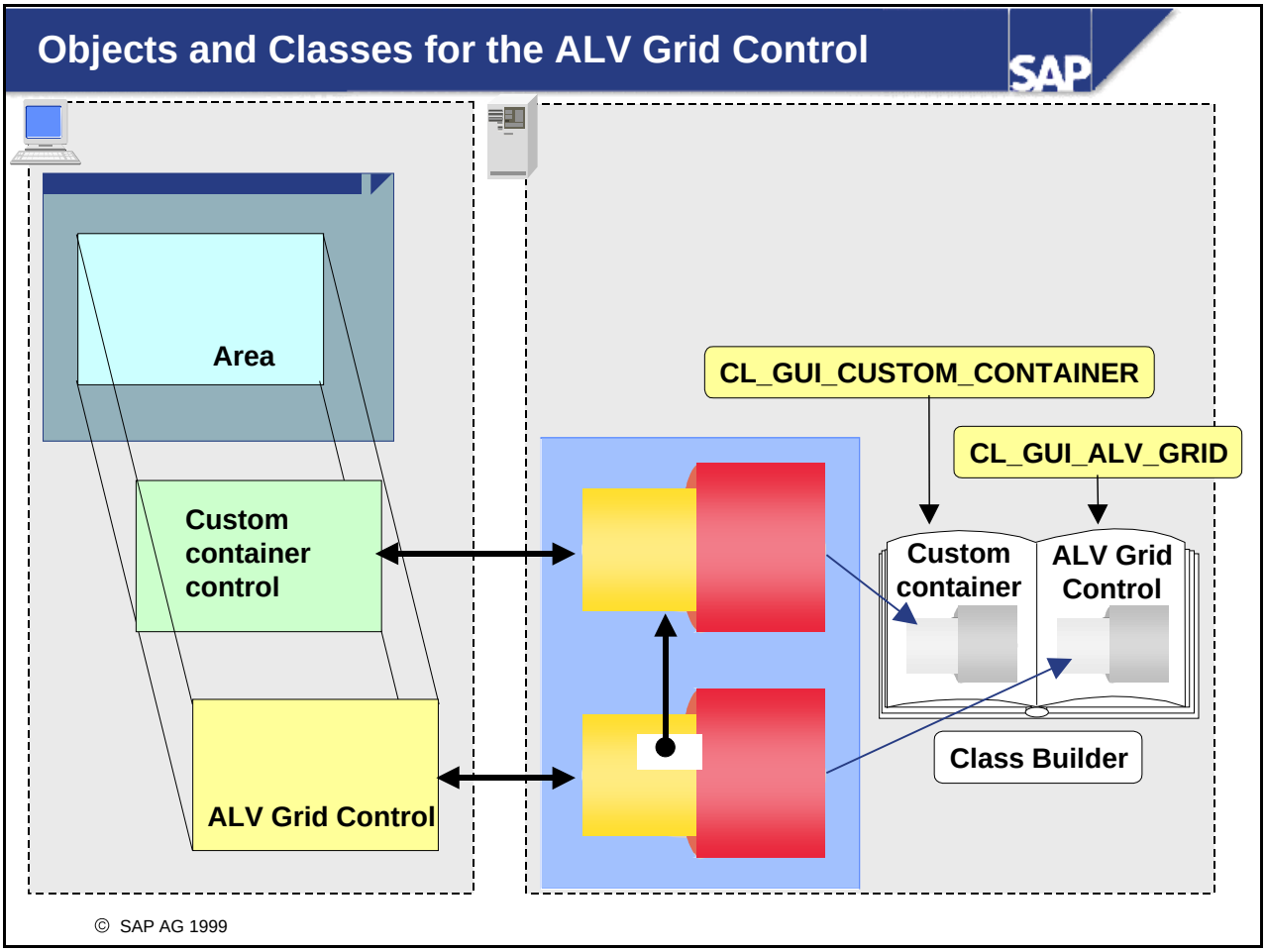
- This task is performed by the Central Control Framework.
- The R/3 System allows you to create custom controls using ABAP objects. The application server is the Automation Client, which drives the custom controls (automation server) at the front end.
- If custom controls are to be included on the front end, then the SAPGUI acts as a container for them. Custom controls can be ActiveX Controls and JavaBeans.
- The system has to use a Remote Function Call (RFC) to transfer methods for creating and using a control (ABAP OO) to the front end.



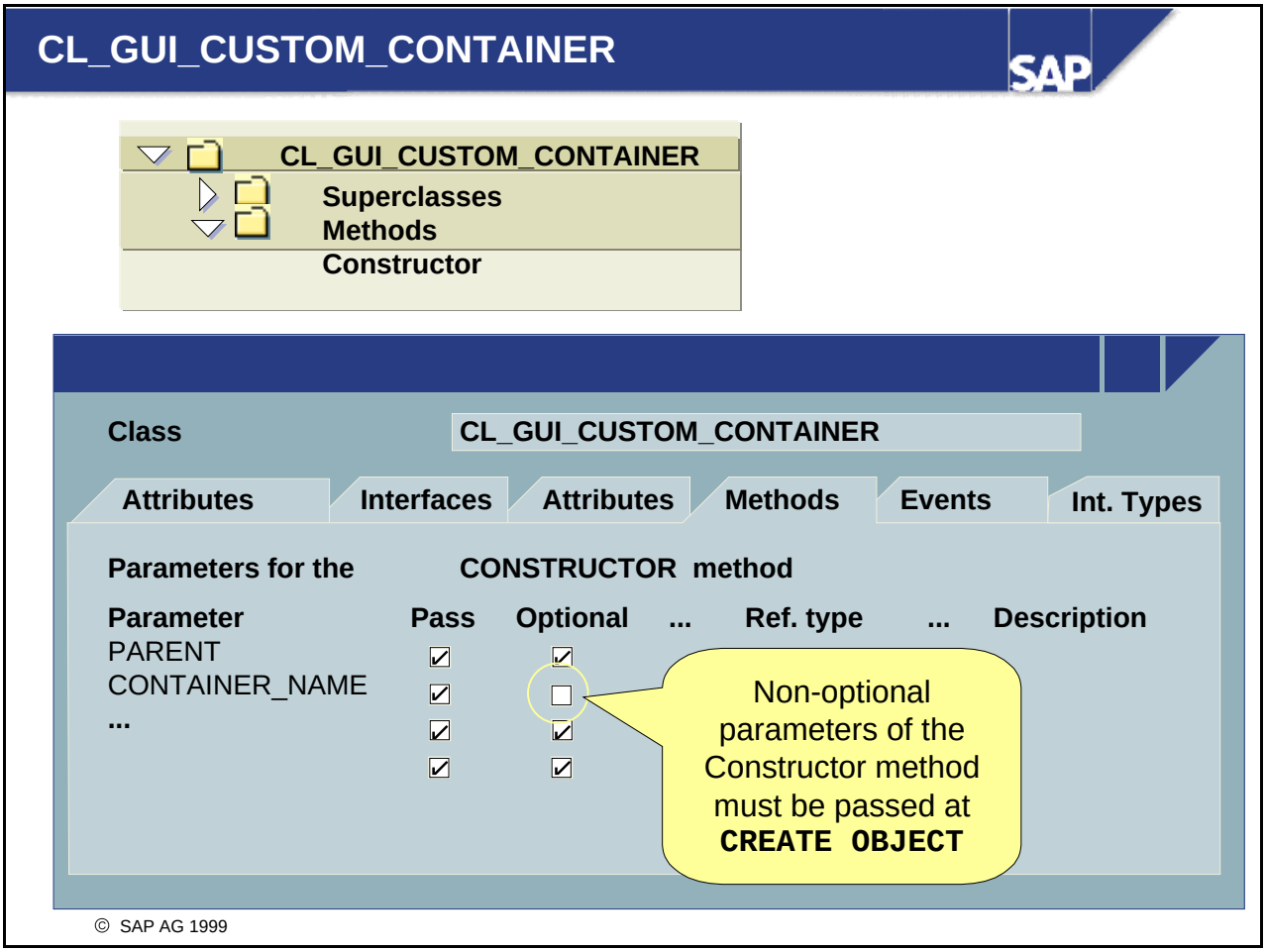

- In the control, you can adjust the column width by dragging, or use the 'Optimum width' function to adjust the column width to the data currently displayed. You can also change the column sequence by selecting a column and dragging it to a new position.
- Standard functions are available in the control toolbar. The details pushbutton displays the fields in the line on which the cursor is positioned in a modal dialog box.
- The sort function in the ALV Control is available for as many columns as required. You can set complex sort criteria and sort columns in either ascending or descending order.
- You can use the 'Search' function to search for a string (generic search without *) within a selected area by line or column.
- You can use the 'Sum' function to request totals for one or more numeric columns. You can then use the "Subtotals" function to set up control level lists. Select the columns (non-numeric columns only) that you want to use and the corresponding control level totals are displayed.
- For 'Print' and 'Download' the whole list is always processed, not just the sections displayed on the screen.
- You also have the option of setting display variants.



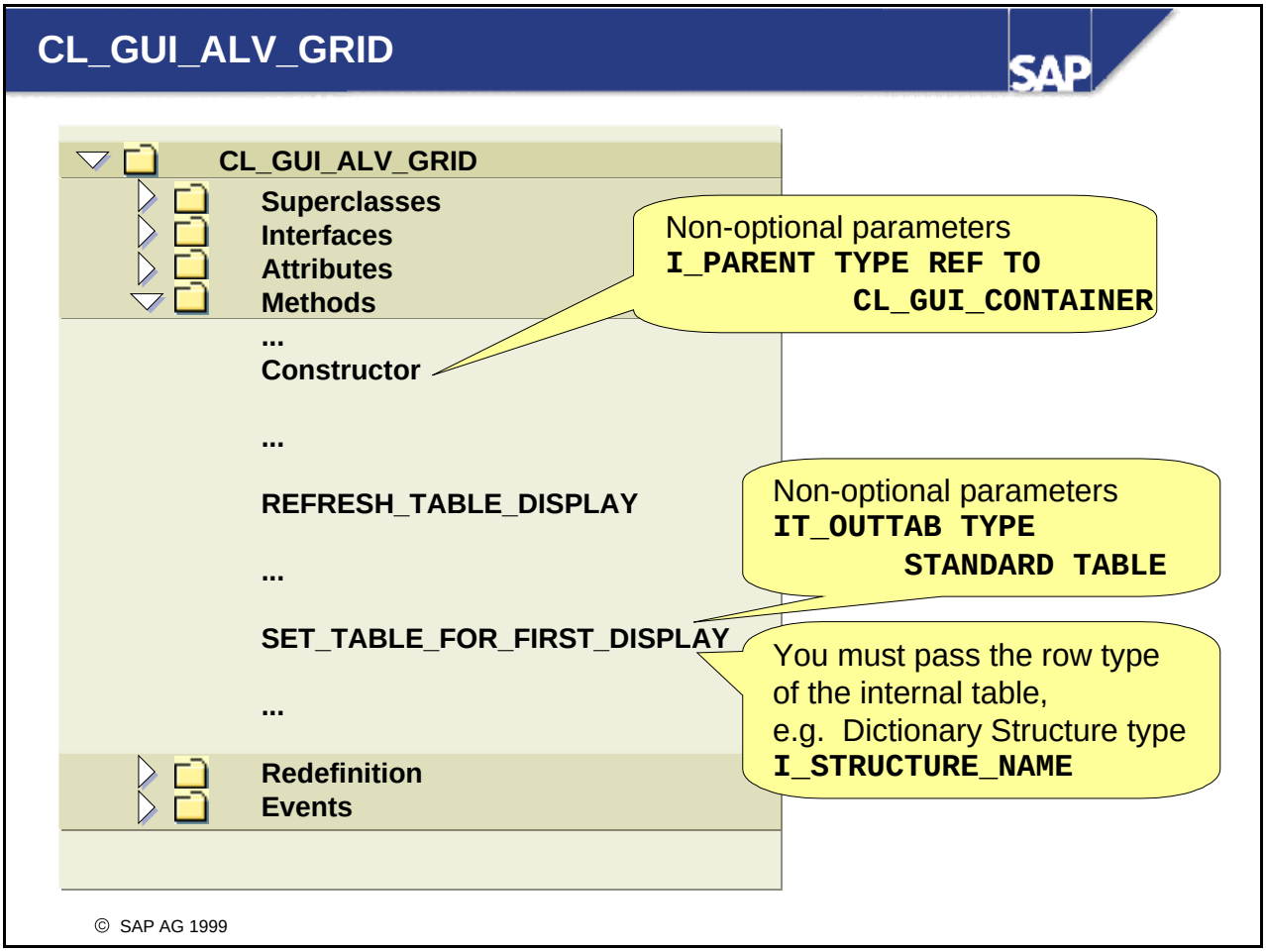

- An SAP Container can contain other controls (for example, SAP ALV Grid Control, Tree Control, SAP Picture Control, SAP Splitter Control, and so on). It administers these controls logically in one collection and provides a physical area for the display.
- Every control exists in a container. Since containers are themselves controls, they can be nested within one another. A container is its control's parent.




- There are object types available in the Class Builder for administering custom controls and the ALV Grid Control. At runtime, the system creates two objects - one of type **CL_GUI_CUSTOM_CONTAINER** and one of type **CL_GUI_ALV_GRID.** These objects contain the methods needed to administer the controls. You can find more information on object types (classes) and their associated methods in the Class Builder.

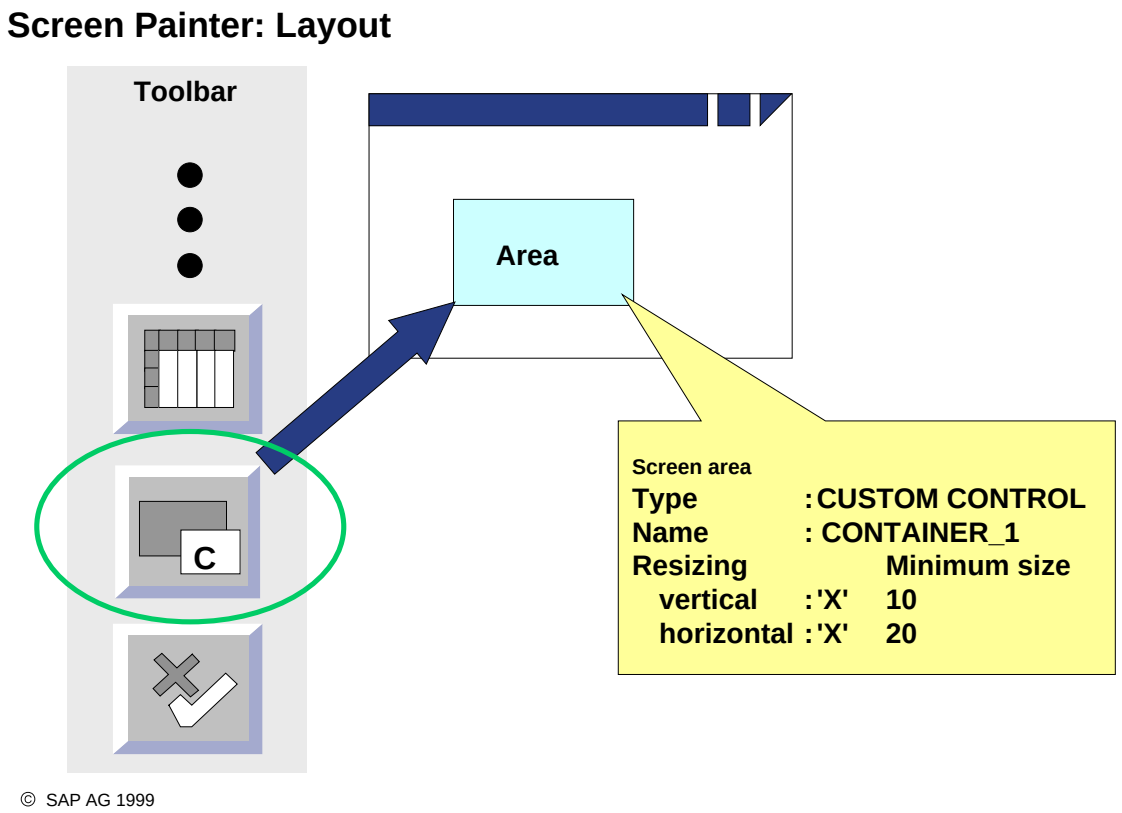



- You can navigate to the Class Builder by entering the name of a class in the *Class* input field on the *Object Navigator* initial screen and choosing *Display.* The system displays a tree structure for the class you have chosen. Double-click the root node to display the Class Builder work area. Choose the *Methods* tab and select the method for which you want more information. Choose the *Parameters* button, to display more information on the interface parameters.
- The class **CL_GUI_CUSTOM_CONTAINER** contains only the **CONSTRUCTOR** method. When you create an object in a program using **CREATE OBJECT** you must pass the non-optional parameter **CONTAINER_NAME**. The name of the container area on the screen must be passed to this parameter.




- Global class **CL_GUI_ALV_GRID** contains many methods. To display an internal table, which is of the ABAP Dictionary Structure row type, using an ALV Grid Control, you only need to know the details of three methods.
- **CONSTRUCTOR:** The reference variable pointing to the object (with which the container control communicates) must be passed to the constructor.
- The first time a table's contents are displayed using an ALV Grid Control, display is implemented using the **SET_TABLE_FOR_FIRST_DISPLAY** method. The internal table is passed to the parameter **it_outtab**. In this case, it is not enough simply to pass the non-optional parameter **it_outtab**. In terms of content, information about the row structure must also be passed to the object. In the case of numeric fields containing a unit, the relationships between fields must be passed - either explicitly using a field list, or implicitly, provided the internal table is of the ABAP Dictionary Structure type. In the latter case, the name of the Dictionary Structure is passed to the **I_STRUCTURE_NAME** parameter.
- **REFRESH_TABLE_DISPLAY** can be called if the internal table has already been displayed using the Grid Control, and if the content of the internal table differs from that shown on the screen. In this case, the front end control already knows the row type of the internal table and reference fields.

# Creating a Custom Control Screen Element

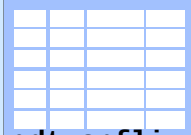





- To reserve an area of the screen for an EnjoySAP control, open the Screen Painter and choose the *Layout* button.
- In the toolbar to the left of the editing area, choose the *Custom control* button. (This works similarly to the *Subscreen* button). :
  - On the editing area of the screen, specify the size and position of the screen area 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 down and right to where you want the bottom right corner. Once you release the mouse key, the bottom right corner is fixed in position.
  - You can change the size and position of the area at any time by dragging and dropping the handles. Again, these areas are similar to subscreen areas.
- Enter a new name for the screen element (CONTAINER_1 in the example above).
- Use the *Resizing vertical* and *Resizing horizontal* to specify whether or not the area of the custom control should be resized when the main screen is resized. You can also set minimum values for these attributes using *Min. row* and *Min. column*. You determine the maximum size of the area when you create it.

## Syntax Example: Defining Reference Variables

```
DATA: gdt_spfli TYPE sbc400_t_spfli.



DATA: container_r TYPE REF TO cl_gui_custom_container,
      grid_r      TYPE REF TO cl_gui_alv_grid,

DATA: ok_code TYPE sy-ucomm.



START-OF-SELECTION.
perform fill_itab USING gdt_spfli.

CALL SCREEN 100.
```



gdt_spfli

container _r

grid_r

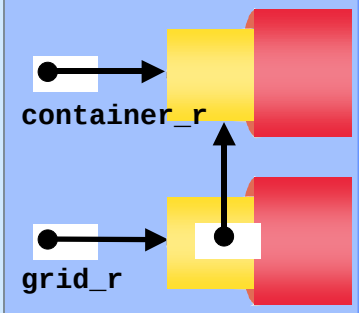
ok_code



- The program requires two reference variables.
- The first reference variable, **container_r** points to the object that communicates with the container control. It is typed with the global class **cl_gui_custom_container**.
- The second, **grid_r** points to the object that communicates with the ALV Grid control. It is typed with the global class **cl_gui_alv_grid**.

# Syntax Example: CREATE OBJECT

```
MODULE create_control OUTPUT.
  IF container_r IS INITIAL.
     CREATE OBJECT container_r
       EXPORTING container_name = 'CONTAINER_1'.

    CREATE OBJECT grid_r
       EXPORTING  i_parent =  container_r.

    CALL METHOD
      grid_r->set_table_for_first_display
        EXPORTING i_structure_name = 'SPFLI'
        CHANGING  it_outtab        = gdt_spfli.
  ELSE.
    CALL METHOD
       grid_r->refresh_table_display
         EXPORTING i_soft_refresh = 'X'.
  ENDIF.

ENDMODULE.
```

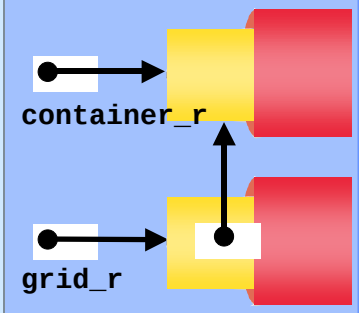





- The **CREATE OBJECT** statement creates an object at runtime. You only need to enter the reference variable, since it already has the same object type as the class.
- To generate the object that communicates with the container control, you only need to include the name of the container area on the screen, provided this occurs in a **PBO** module of the screen on which the container area has been defined. If the **CREATE OBJECT** statement has been implemented in another ABAP processing block, you must include the screen container number and the program number.
- To generate the object that communicates with the ALV grid control, you must pass the reference variable that points to the custom container object. This "tells" the object the container in which it is to be included.

## Syntax Example: Calling Methods

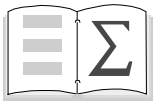
```
MODULE create_control OUTPUT.
  IF container_r IS INITIAL.
    CREATE OBJECT container_r
      EXPORTING container_name = 'CONTAINER_1'.



    CREATE OBJECT grid_r
      EXPORTING i_parent = container_ref.

    CALL METHOD
       grid_r->set_table_for_first_display
         EXPORTING i_structure_name = 'SPFLI'
         CHANGING  it_outtab        = gdt_spfli.
  ELSE.
    CALL METHOD
       grid_r->refresh_table_display
         EXPORTING i_soft_refresh = 'X'.
  ENDIF.

ENDMODULE.
```






- To display data in an ALV grid control, you must make them available in an internal table. The system then calls the method that receives the content and structure of the internal table. The method is called **set_table_for_first_display**. Provided the internal table has the type ABAP Dictionary Structure, the name of the structure is passed to the **i_structure_name** parameter. The method then gets the information it needs - column names, column types, and column links for currency fields - directly from the ABAP Dictionary.
- If only the content of the internal table changes while the program is running, the program must call the **refresh_table_display** method before sending the screen with the container area again.

# Reuse Components



**You are now able to:**

- **Find and use function modules**
- **Display a simple list using the ALV grid control (objects from global classes in the Class Builder)**
- **Use a BAPI and find BAPIs using the BAPI Browser**

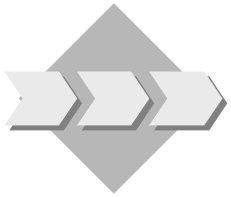
Exercises

**Unit: Reuse Components**

**Topic: Function Modules**

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

- Search for a function module
- Insert a function module call in a program



Extend your program **ZBC400_##_SELECT_SFLIGHT** or the corresponding model solution as follows:
If the *Cancel* function is chosen on the screen, the system should process a standard dialog box that is encapsulated in a function module.

**Program:** **ZBC400_##_DYNPRO**

**Model solution:** **SAPBC400UDS_DYNPRO_E**

1-1 Extend your program, **ZBC400_##_DYNPRO**, or copy the relevant model solution **SAPBC400UDS_DYNPRO_D** and give it the name **ZBC400_##_DYNPRO_D**. Assign your program to the development class **ZBC400_##** and to the transport request for this project, BC400… (replacing ## with your group number).

1-2 Using the method outlined during the course, search for the function module that encapsulates the standard dialog, which is usually triggered when the user chooses *Cancel*.

1-3 Find out about the function module interfaces, read the documentation, and test the function module using the test environment.

1-4 In the GUI status of the screen, activate the ‘Cancel’ function.

1-5 Extend the **USER_COMMAND_0100** module to evaluate the function code for the *Cancel* function. Then insert the function module call using the “pattern” function of the ABAP Editor. React to the user’s input that gets the function module, as follows:

- If the user would like to cancel, set the next screen dynamically to 0.

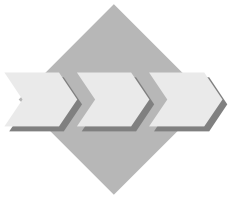
- If the user does not want to cancel, set the next screen dynamically to 100.

**Unit: Reuse Components**

**Topic: ABAP Objects and the ALV Grid Control**

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

- Output a simple list using an ALV grid control



Write a program that outputs the contents of the database table **SPFLI** using an ALV grid control.

**Program:** **ZBC400_##_ALV_GRID**

**Model solution:** **SAPBC400RUS_ALV_GRID**

2-1 Copy the program, **SAPBC400RUT_ALV_GRID** giving it the name **ZBC400_##_ALV_GRID**. Assign your program to **development class ZBC400_##** and the change request for your project "BC400…" (replacing ## with your group number). The template program contains the definition of an internal table with the same line type as the database table **SPFLI** and a user dialog (screen 100).

2-2 Become familiar with the program.

2-3 Fill the internal table with data records from the data table **SPFLI** using the Array-Fetch**.**

2-4 Navigate to the Class Builder and find out the following:

2-4-1 Which parameters of method **CONSTRUCTOR** for class **CL_GUI_CUSTOM_CONTAINER** are compulsory?

2-4-2 Which parameters of method **CONSTRUCTOR** for class **CL_GUI_ALV_GRID** are compulsory?

2-5 Create a container control area on the screen. Make sure you give the area a name.

2-6 Define two reference variables, one for the **CL_GUI_CUSTOM_CONTAINER** class and one for the **CL_GUI_ALV_GRID** class.

2-7 Make sure that a module is called from the event **PROCESS BEFORE OUTPUT** from screen 100, to generate the objects. Generate the object for the custom container using **CREATE OBJECT.** Pass the name of the container area for screen 100 to the constructor's mandatory parameter. Create the object for the **ALV GRID CONTROL** using **CREATE OBJECT**. Pass the reference variable for the custom container to the mandatory parameter. Use a query to ensure that the object is only generated when **PROCESS BEFORE OUTPUT** runs for the first time.

2-8 When **PROCESS BEFORE OUTPUT** runs for the first time, call the method **SET_TABLE_FOR_FIRST_DISPLAY**; pass the name of the line type of the internal table to the parameter **I_STRUCTURE_NAME**; pass the internal table to the parameter **IT_OUTTAB**.

2-9 If **PBO** runs more than once, the method **REFRESH_TABLE_DISPLAY** should be called. Pass 'X' to the parameter **I_SOFT_REFRESH**.

**Unit: Reuse Components**

**Topic: Function Modules**

1-2 The function module is called '**POPUP_TO_CONFIRM_LOSS_OF_DATA**'.

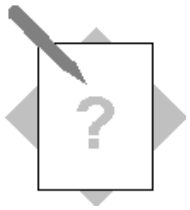
1-3 The following interface parameters exist:
Mandatory import parameters:
**TEXTLINE1** (max 70 char.) : first line of the dialog window
**TITEL** (max 35 char.) : title of the dialog window
Optional import parameters:
**TEXTLINE2** (max 70 char.) : first line of the dialog window
**START_COLUMN** (Type SY-CUCOL): First column of the dialog window
**START_ROW** (Type SY-CUCOL): First line of the dialog box
Export parameters:
**ANSWER** (Type C), : user's input
"Y" = user has confirmed the processing step
"N" = user has canceled the processing step

1-4 The function code for the *Cancel* function is **RW**.

1-5

```
*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  save_ok = ok_code .
  CLEAR ok_code .
  CASE save_ok.
    WHEN 'BACK'.
      SET SCREEN 0.
    WHEN 'RW'.

      CALL FUNCTION 'POPUP_TO_CONFIRM_LOSS_OF_DATA'

        EXPORTING
          textline1 = text-001
          titel     = text-002
        IMPORTING
```

```
            answer    = answer.
      case answer.
        when 'N'.
        leave to screen 100.
        when 'J'.
        leave to screen 0.
      endcase.
    WHEN 'SAVE'.
      MESSAGE ID 'BC400' TYPE 'I' NUMBER '060'.
      SET SCREEN 0.
    WHEN OTHERS.
      SET SCREEN 100.
  ENDCASE.
ENDMODULE.                    " USER_COMMAND_0100  INPUT
```



**Unit: Reuse Components**

**Topic: ABAP Objects and the ALV Grid Control**

**Program:** **ZBC400_##_ALV_GRID**

**Model solution:** **SAPBC400RUS_ALV_GRID**

2-3 START-OF-SELECTION.

```
* fill internal table
  SELECT * FROM spfli
        INTO TABLE gdt_spfli.
*       WHERE ...
  CALL SCREEN 100.
```

2-4-1 The following parameter of the method **CONSTRUCTOR** (for the class: **CL_GUI_CUSTOM_CONTAINER)**is mandatory.

CONTAINER_NAME: the name of the control container on the screen

2-4-2 The following parameter of the **CONSTRUCTOR** method (for the **CL_GUI_CUSTOM_CONTAINER** class**)** is mandatory:

**I_PARENT**: parent-container: The name of the reference variable that points to the object for the **CL_GUI_CUSTOM_CONTAINER** class must be passed to this parameter.

2-5 Create a container control area on the screen. Name of container area: **CONTAINER_1**

2-6 Enter the following in the data declarations section:

```
DATA:
  container_r TYPE REF TO CL_GUI_CUSTOM_CONTAINER,
  grid_r TYPE REF TO CL_GUI_ALV_GRID.
```

2-7 to 2-9:

**Flow logic:**

```
PROCESS BEFORE OUTPUT.
 MODULE STATUS_0100.
 module create_control.
*
PROCESS AFTER INPUT.
module copy_ok_code.
MODULE USER_COMMAND_0100.
```

**PBO module in the program:**

```
*&---------------------------------------------------------------------*
*&      Module  CREATE_CONTROL  OUTPUT
*&---------------------------------------------------------------------*

MODULE create_control OUTPUT.
  IF container_r IS INITIAL.
    CREATE OBJECT container_r
         EXPORTING container_name = 'CONTAINER_1'.

    CREATE OBJECT grid_r
          EXPORTING  i_parent =  container_r.

    CALL METHOD    grid_r->set_table_for_first_display
        EXPORTING i_structure_name = 'SPFLI'
        CHANGING  it_outtab        = gdt_spfli.
  ELSE.
    CALL METHOD grid_r->refresh_table_display
        EXPORTING i_soft_refresh = 'X'.
  ENDIF.
ENDMODULE.                    " CREATE_CONTROL  OUTPUT
```

# Software Logistics and Software Adjustment: Contents

- Team and project-oriented software development using the Transport Organizer
- Ways of changing the SAP standard software

# Software Logistics and Software Adjustment: Unit Objectives

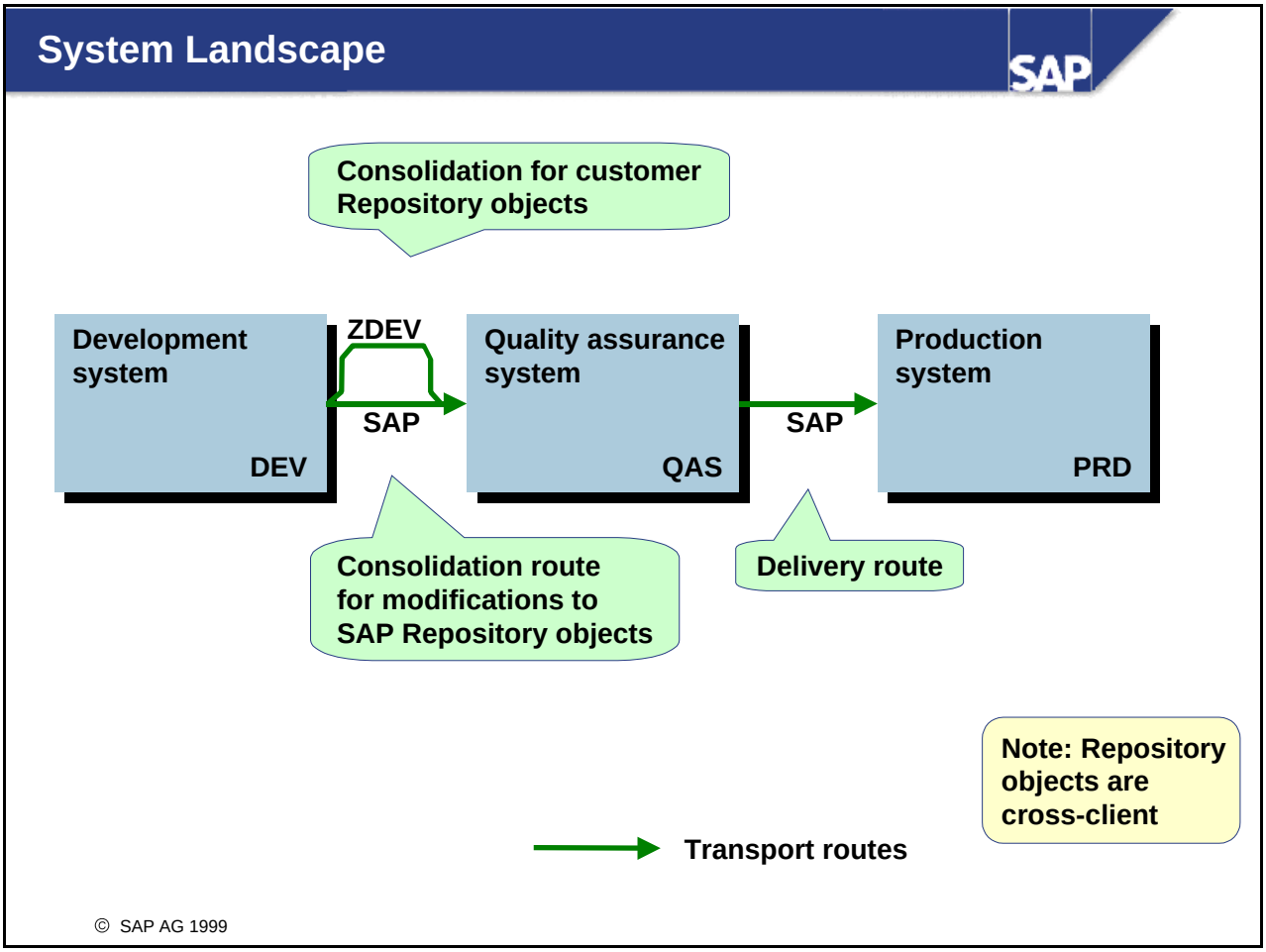
**At the conclusion of this unit, you will be able to:**

- **Map a project in the R/3 System using the Transport Organizer**
- **Describe the options for enhancing or changing the functions of existing programs**

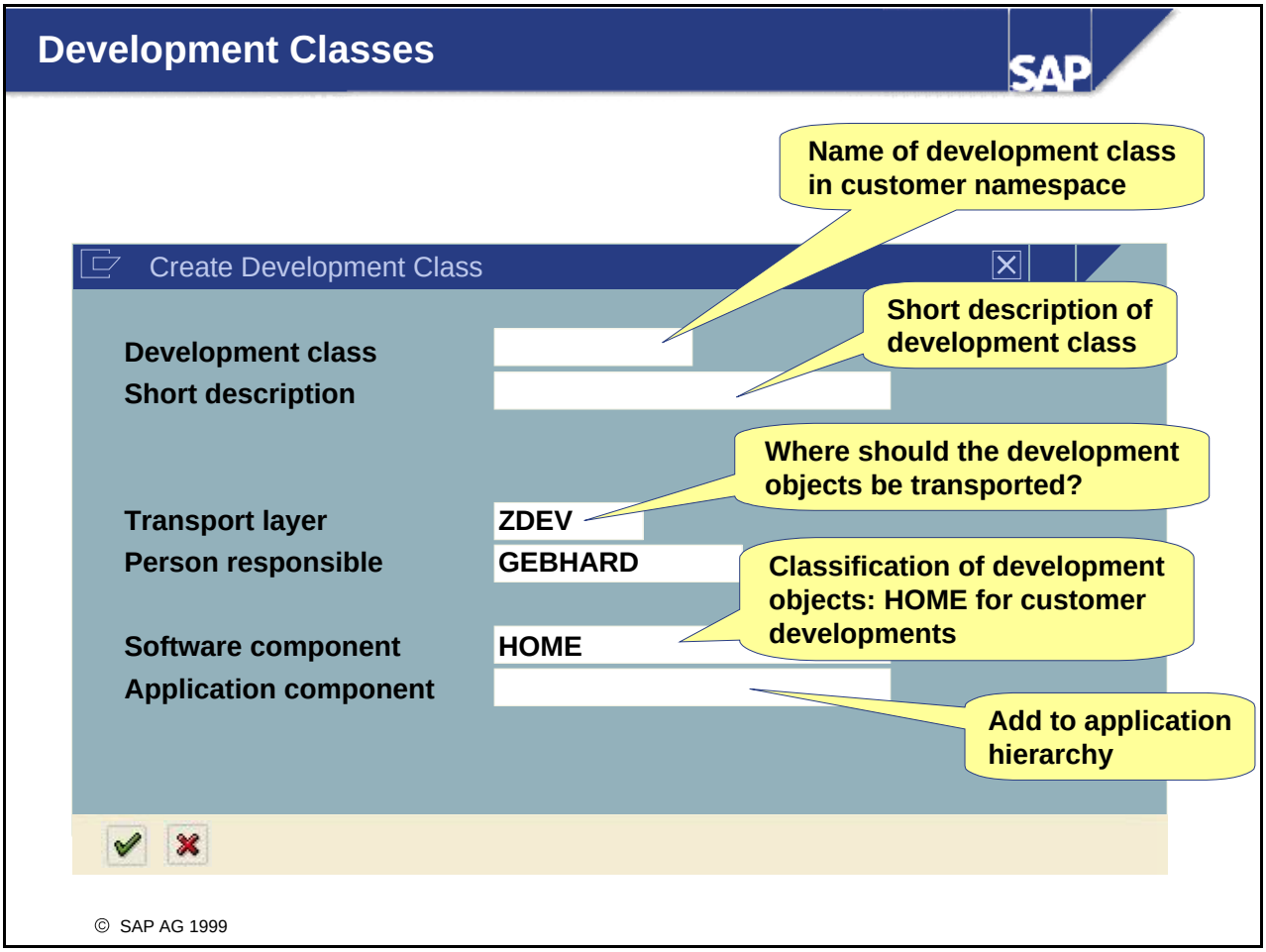
# Software Logistics and R/3 Adjustment

**Organization of Software Development**

**Customer development, enhancement, or modification?**

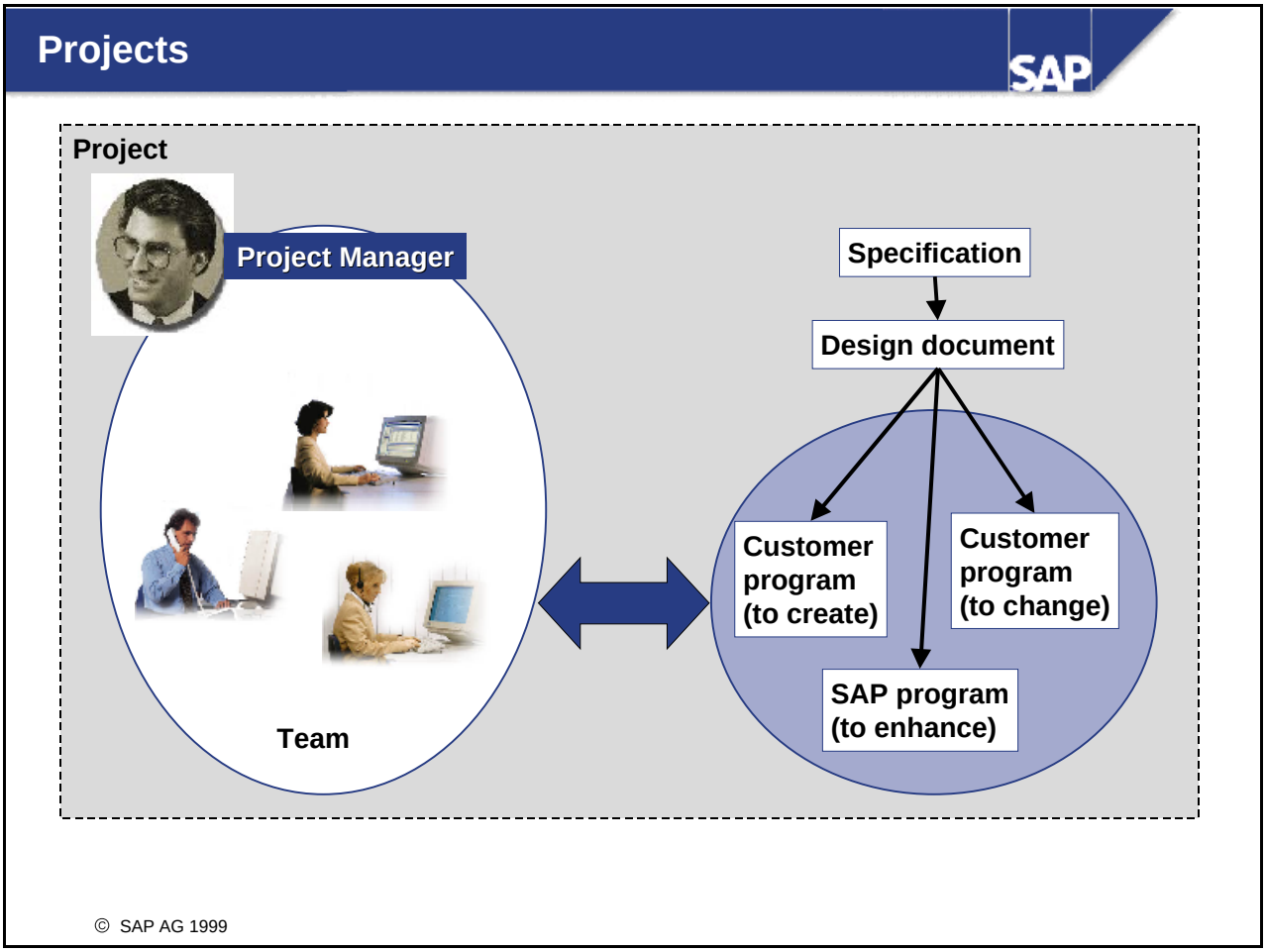



- You can see the current system configuration in the Transport Management System. From the SAP Easy Access menu you can reach the Transport Management System by choosing the menu path *Tools --> Administration --> Transports --> Transport Management System* or by using transaction **STMS**. By choosing the *Transport routes* icon you can display a diagram of the current system and corresponding transport routes.

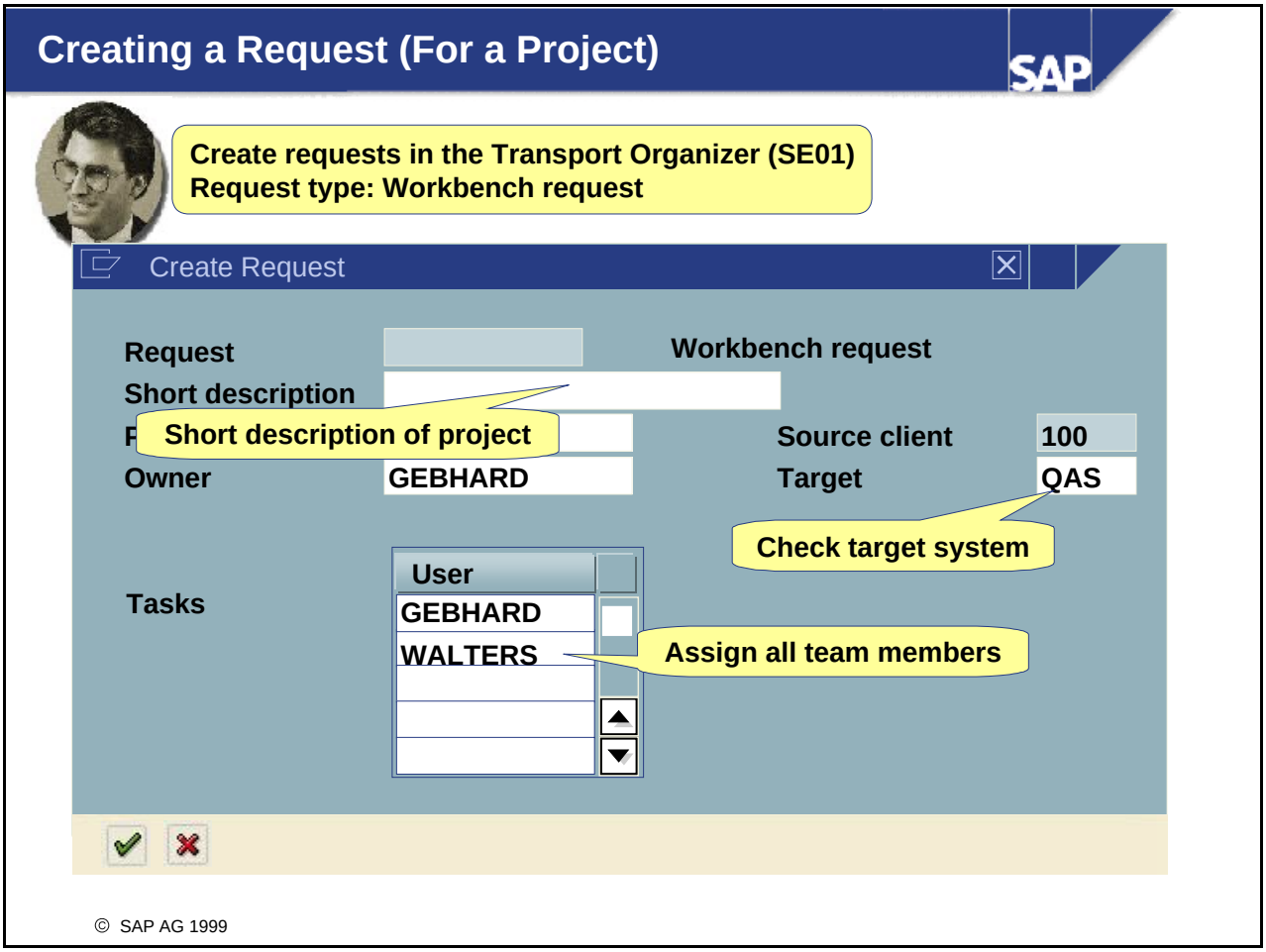



- You can create a development class in the Object Navigator. Observe the customer namespace conventions when naming your development class.
  - Choose *Edit object* from the initial screen or *Other object*. Enter the name of the development class in the input field on the tab page *Other*.
  - In Release 4.6C you can create a development class by selecting object type D*evelopment class* and entering the name of the development class. When you press *Enter* or choose *Display,* the system checks whether a development class with the same name already exists. If it does, then the object list of development classes appears in the navigation area. If the development class does not already exist, the system opens the *Create Development Class* dialog box.
- As well as the name and short description, you must specify the following information:
  - Transport layer: If you carry out your own developments, you must set up a transport layer for customer developments.
  - Enter the name of the person responsible for the object of the development in the corresponding input field. The system automatically uses your user name as a proposal.
  - For customer developments you should enter HOME as the software component. You can get detailed information on the field from the F1 help function.

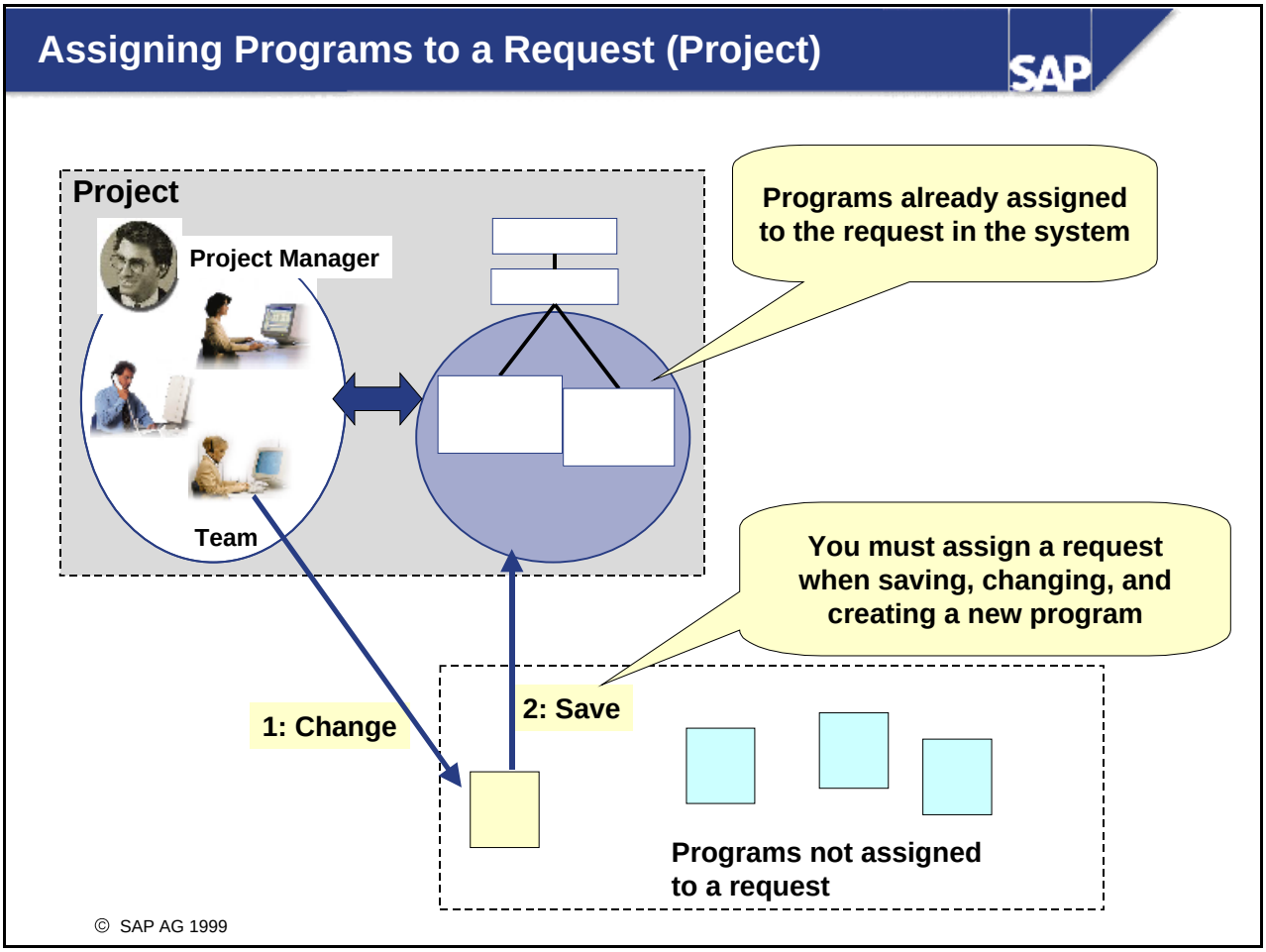
- In the *Application component* field you can determine the assignment of development classes in the application hierarchy.



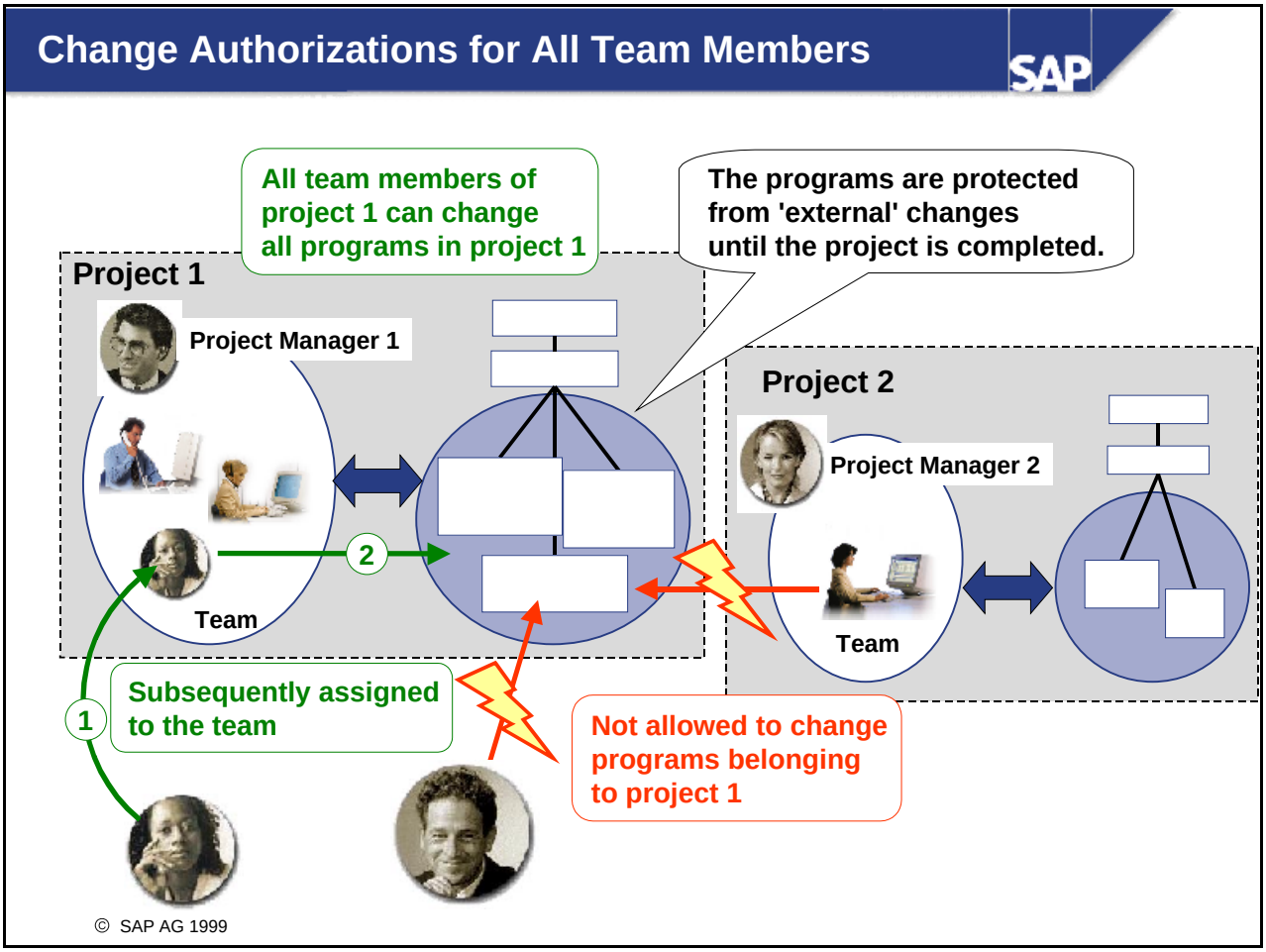

- Each development project requires the following information:
  - Project Manager and team
  - Selection of the contents of the project (specification or blueprint) and the concept for conversion in the system (design document or technical design). In this context you can determine which programs are created, which customer developments are changed, and which SAP programs are enhanced.
  - Time frame and development deadline
- In the system a change request is created for a development project. The project manager and team members are determined in the change request.
- The program, which is to be created or changed, is assigned to the request when saving the first change. A program only be assigned to a single project at any one time.
- The development deadline is not a direct attribute of a change request. At the end of development the request is released and the changes are exported from the system. After their release, the programs, which are assigned to the request, can be assigned to another change request and a subsequent development project.




- As the project manager you can create a change request in the Transport Organizer:
  - From the SAP Easy Access menu choose: *Tools --> Administration --> Transports --> Transport Organizer* or transaction SE01.
- Choose *Create* and select *Workbench request* as the request type.
- Enter a short text for the development project. As far as possible, you should formulate the short text so that the team members can identify the request with the project, if they are working on multiple projects.
- Your user name is used for the owner name. If you create the change request on behalf of the project manager, you can change the owner name.
- Check the target system. The administrator can transport all programs of the change request after release in the selected target system.
- Enter the user names of all team members.

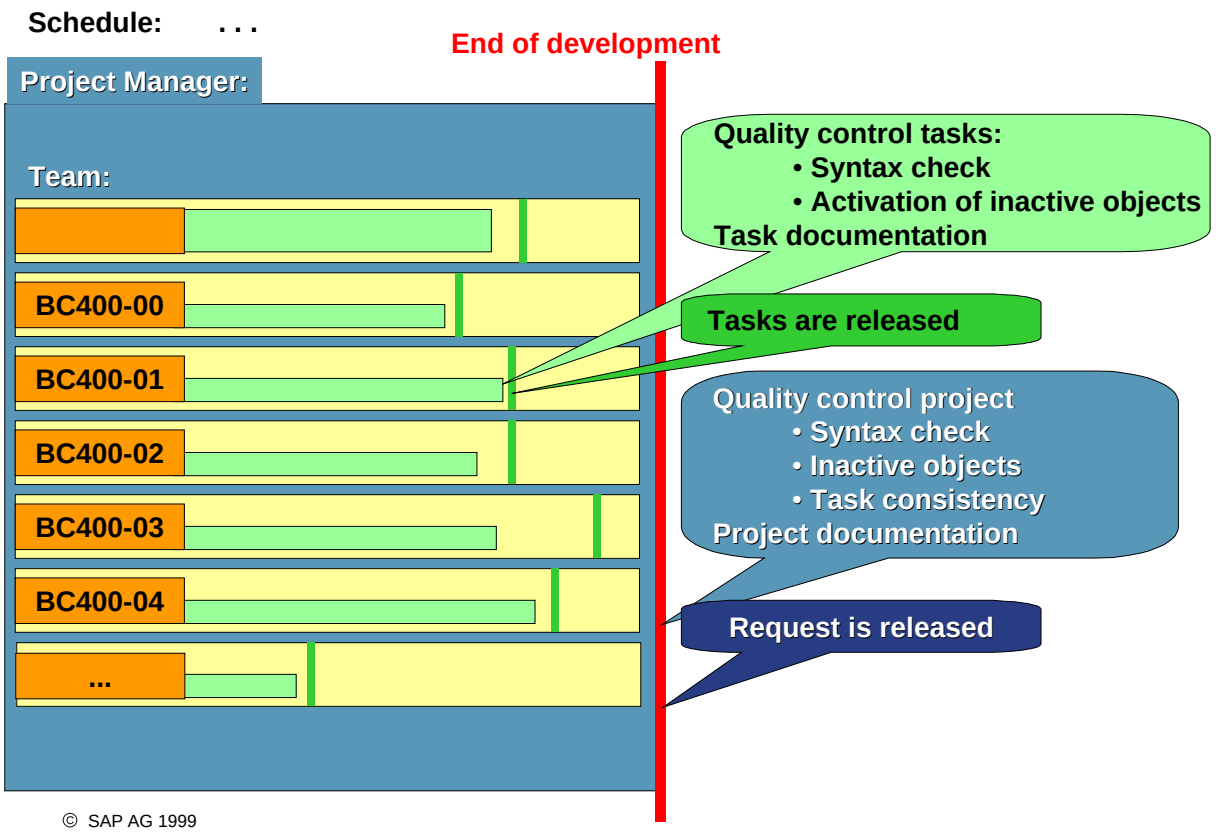


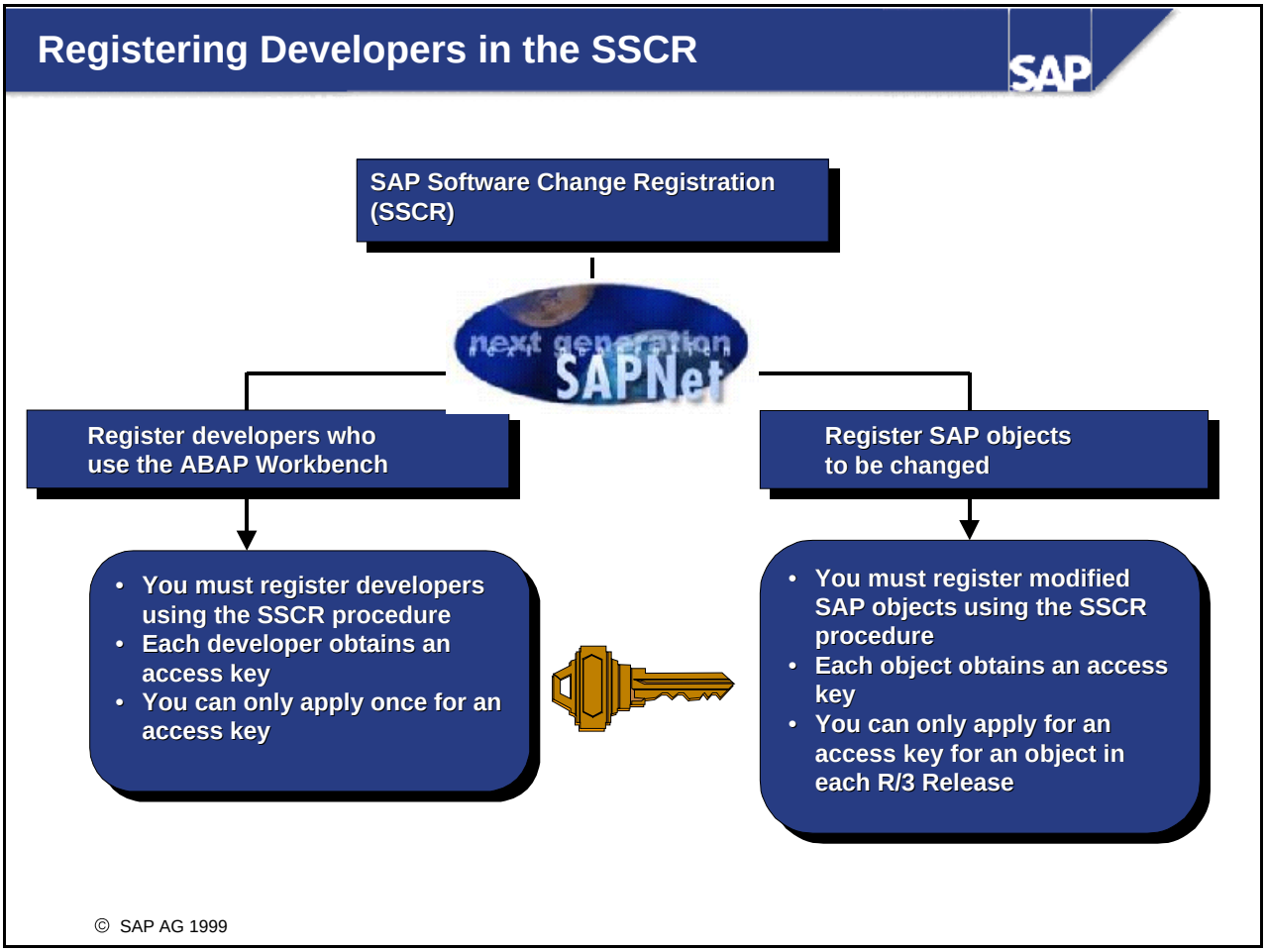

- When a developer wants to save a change or a new program, the developer must assign it to a change request and a logical development project. The program is then linked to the change request until the request is released.
- A program can be linked to a maximum of one change request at any one time. If a program that you need to change for your development project A is already being changed within another development project B, then project B must be completed first. After you have released the change request for project B you can edit the program.
- Note: All programs of a request are transported at the same time. Therefore, when assigning programs to a request you must consider the dependencies of other programs. When releasing a request it should create a consistent state. If changes are carried out in another development project, and you need these changes, you must ensure that the other change request is imported into the system before your request.




- All team members can access all programs of the change request. This allows for flexible team work.
- The developers who are not assigned to the change request can only display the programs that are assigned to the change request.
- If a developer subsequently becomes a team member, he or she is then assigned to the change request. Afterwards the developer can change all programs in the request.
- The assignment of members to a request is valid only until the project is released. For following projects members must be reassigned to a request.

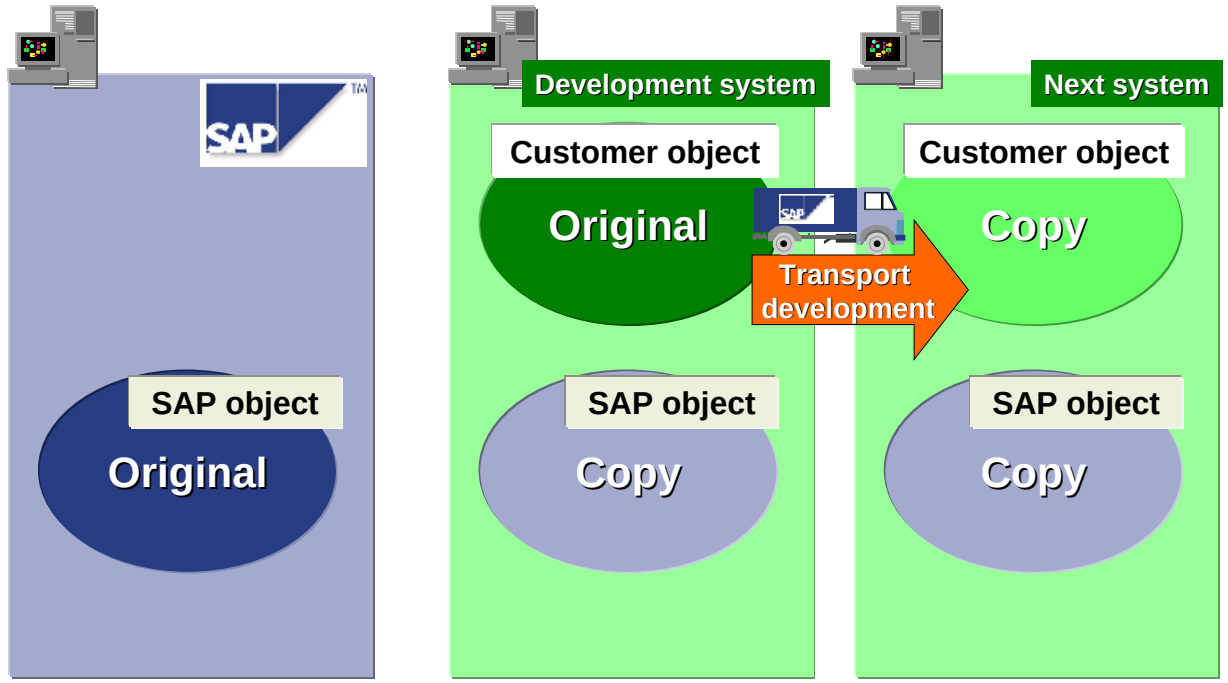
# At the End of Development





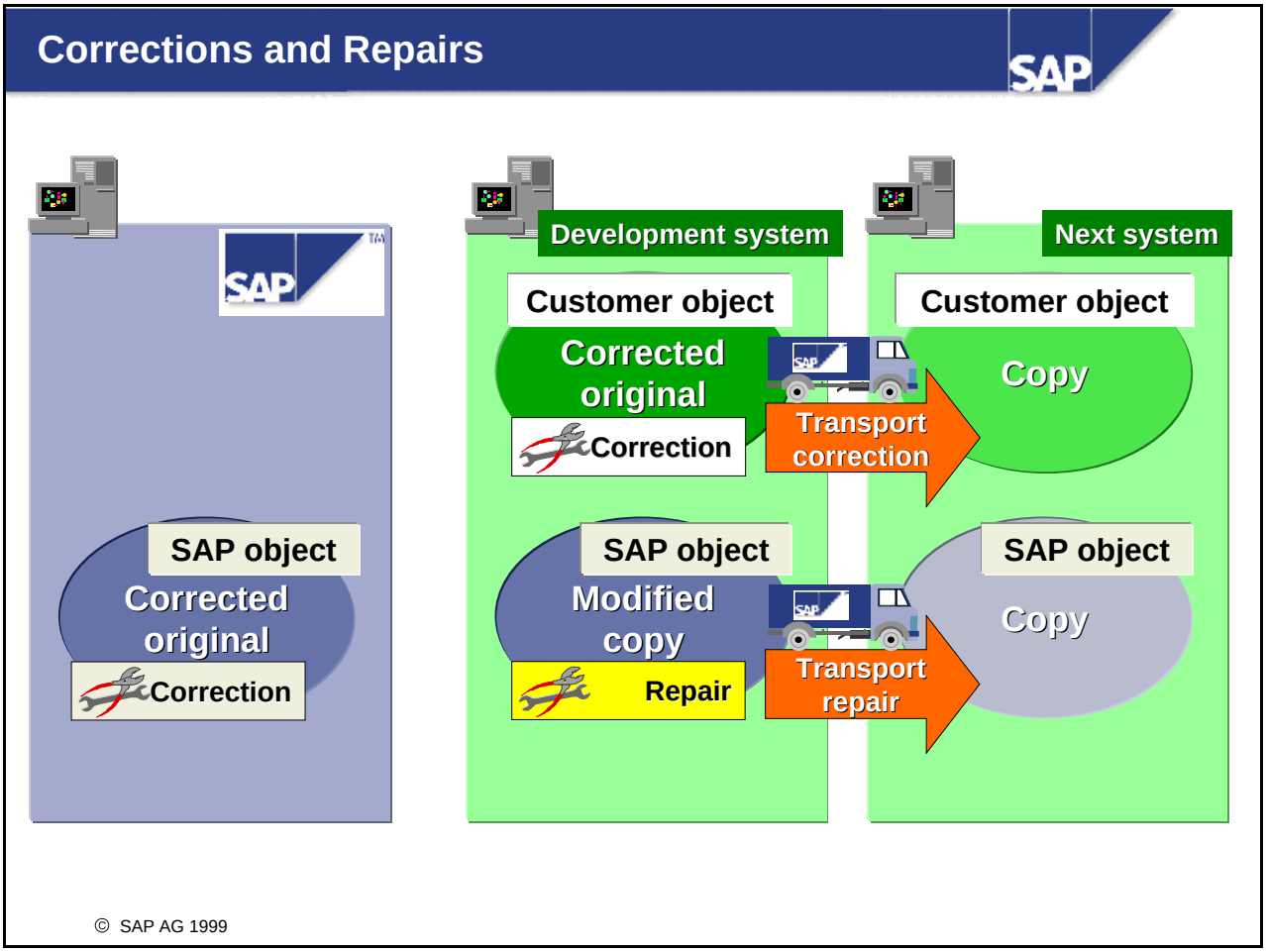

- All R/3 user who wish to use the ABAP Workbench to create or change a Repository object in their system (including customer developments), must request a key using the SAP Software Change Registration (SSCR) procedure.
- After the registration process all development users receive a key. The key is linked to the developer's user ID and the license number of the R/3 System. The system prompts development users for their key the first time they attempt to create or change a Repository object.
- You must also register all SAP Repository objects that you wish to modify. To register you must enter the name of the object, the object type, the license number of the R/3 System, and the number of the Release. You register each Repository object once. The registration is valid until the next upgrade.

## Originals and Copies

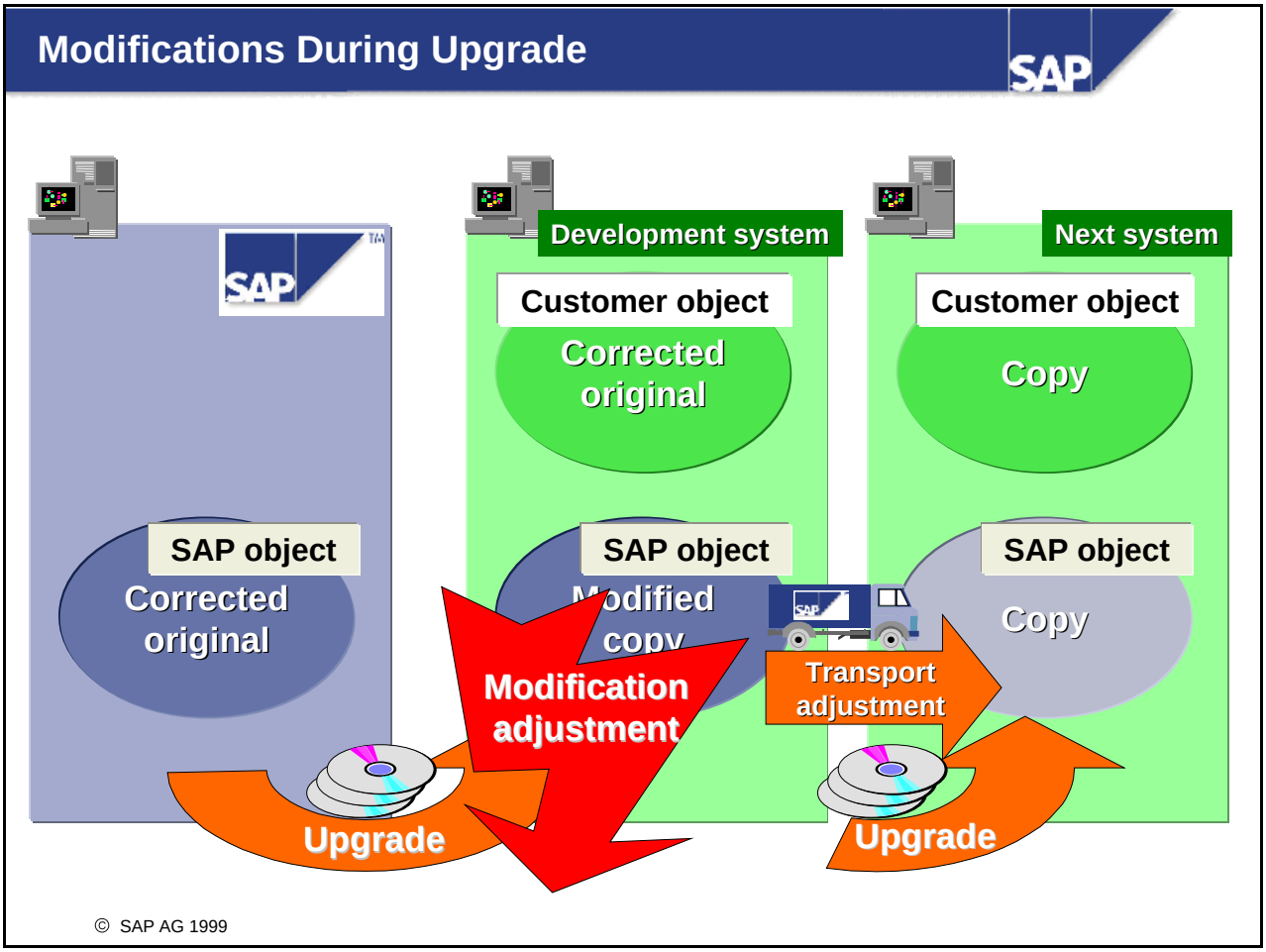




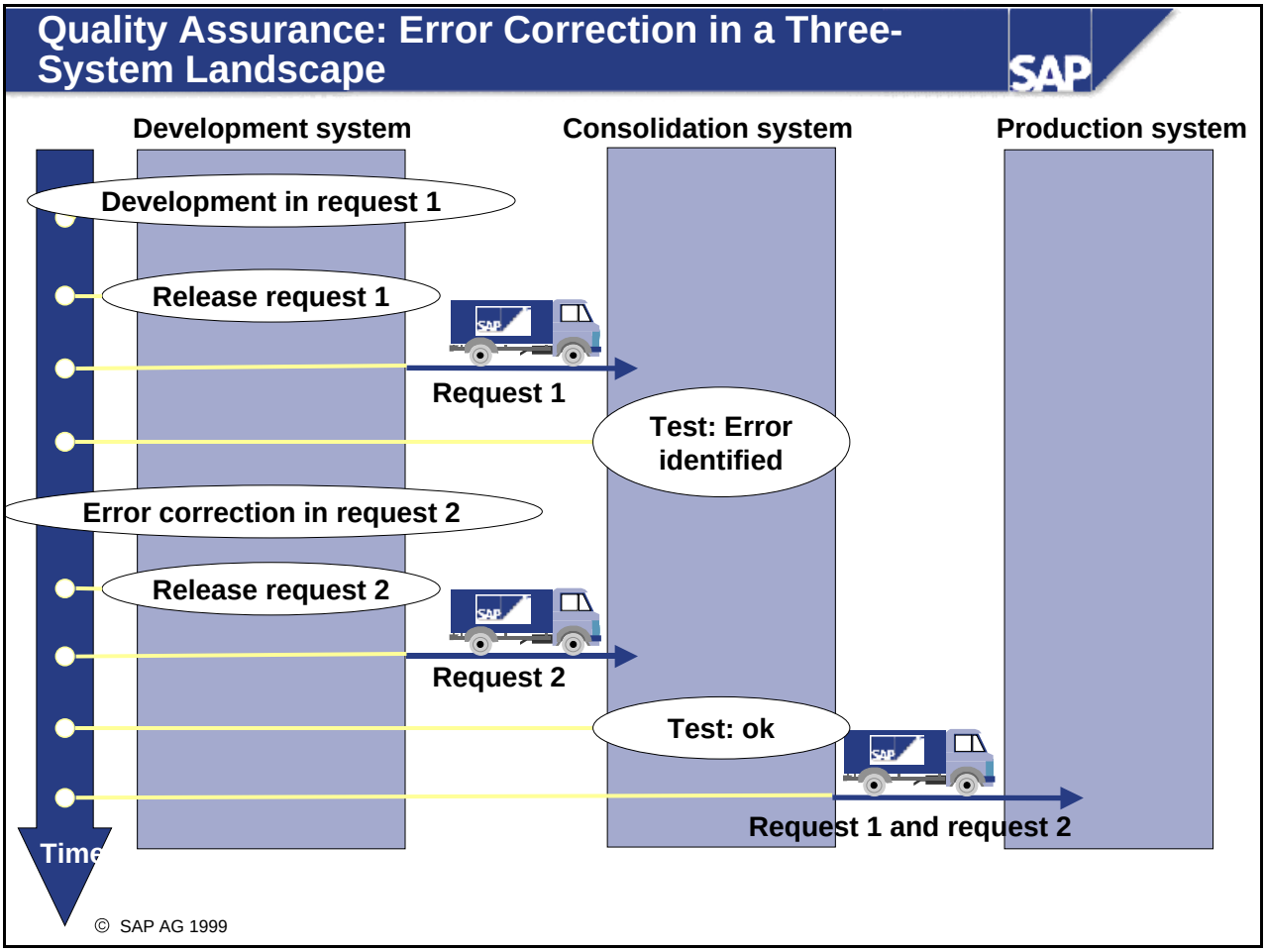

- When you create a Repository object, the Transport Organizer automatically notes the system in which you created it. We say that the original of a Repository object is in a certain system.
- The original version of an object can only exist in one system. All other systems contain copies of the object.
- The idea of having an original system is to ensure that Repository objects can only be changed in the integration system. The integration system is where you carry out your development work, so all of the objects in it are originals. This means that there is one central location for changing Repository objects.
- If you now transport your Repository objects into a consolidation system, it exists there as a copy. Although it is possible to change copies in exceptional cases, you should always try to make the changes in the integration system and transport the new version to other systems. This ensures that the state of objects remains consistent across systems.
- Originals are never overwritten in transports.




- Changing an original is called a correction. The system records these changes in a request containing tasks of type "development/correction".
- If a copy is changed (that is, if an object is not changed in its original system), this change is recorded in a task with type "repair". A repair of an SAP object is called a modification.
- The changes made to your own objects (e.g. due to an emergency in the production system) can also be made immediately to the originals in the development system. **It is imperative that you immediately make the changes you made to the copies to the original as well.**
- This is not possible for SAP objects because the originals are not in any of your systems.



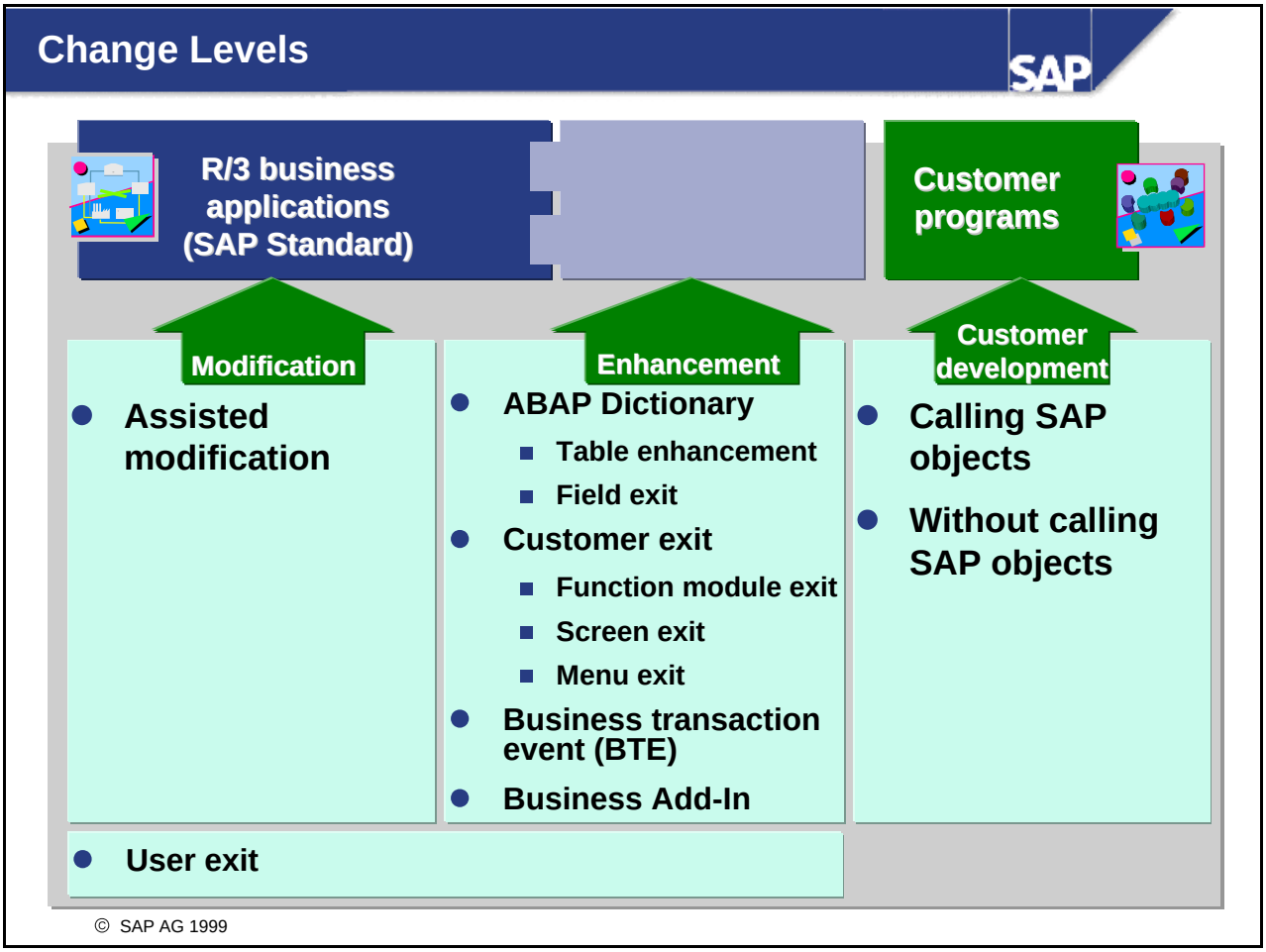

- A conflict could occur when you apply an upgrade, a support package, or some other transport request from SAP to your system.
- A conflict occurs if you change an SAP object and a new copy is being delivered by SAP in an upgrade. The object delivered by SAP becomes the active object in the repository of your R/3 System.
- If you want to save your changes, you have to make a **modification adjustment** for the corresponding objects. Modifying a number of SAP objects can cause a substantial delay when performing an upgrade.
- To ensure that the development system and the next system are consistent, you should only make the modification adjustment in the development system. The objects of the adjustment are then transported to the subsequent systems.




- In a three-system landscape the development cycle can be illustrated in the following way:
- All new and changed programs are assigned to a change request. Immediately before the start of the test phase in the consolidation system, the request is released and transported to the consolidation system.
- Tests are carried out in the consolidation system. Identified fields are passed to the development. This carries out the necessary corrections and assigns the corrected program to a new change request. The new request is released and transported to the consolidation system. It may be necessary to repeat this part of the process several times.
- As soon as the test has been successfully completed, all requests are transported into the production system.
- If you simultaneously carry out multiple independent development projects that you want to transport separately into the production system, you can create a so-called *'Project'* for each development project in the development system. You can then assign the change requests to the corresponding '*Project*'. You can find additional information in the SAP Library under *Basis Components --> Change and Transport System --> Transport Organizer --> Working with Projects* or in the training course *BC325 Software Logistics*.

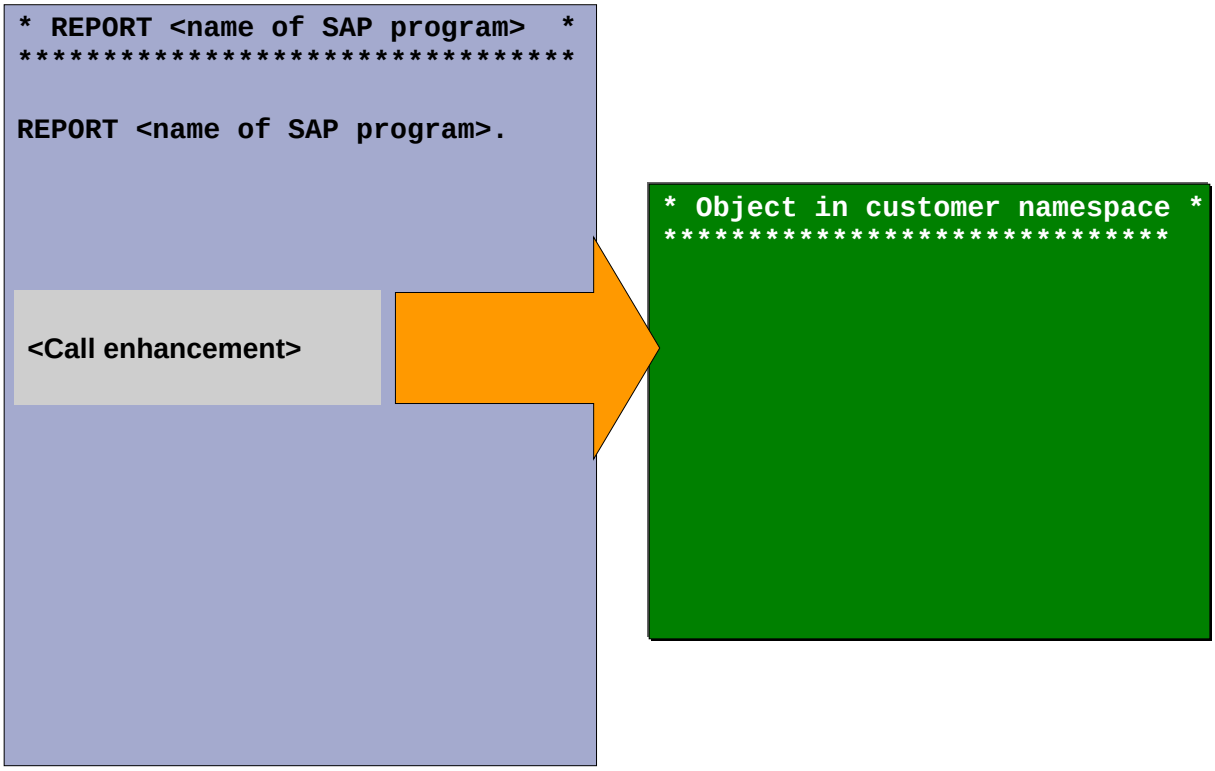
Organization of Software Development

Customer development, enhancement, or modification?



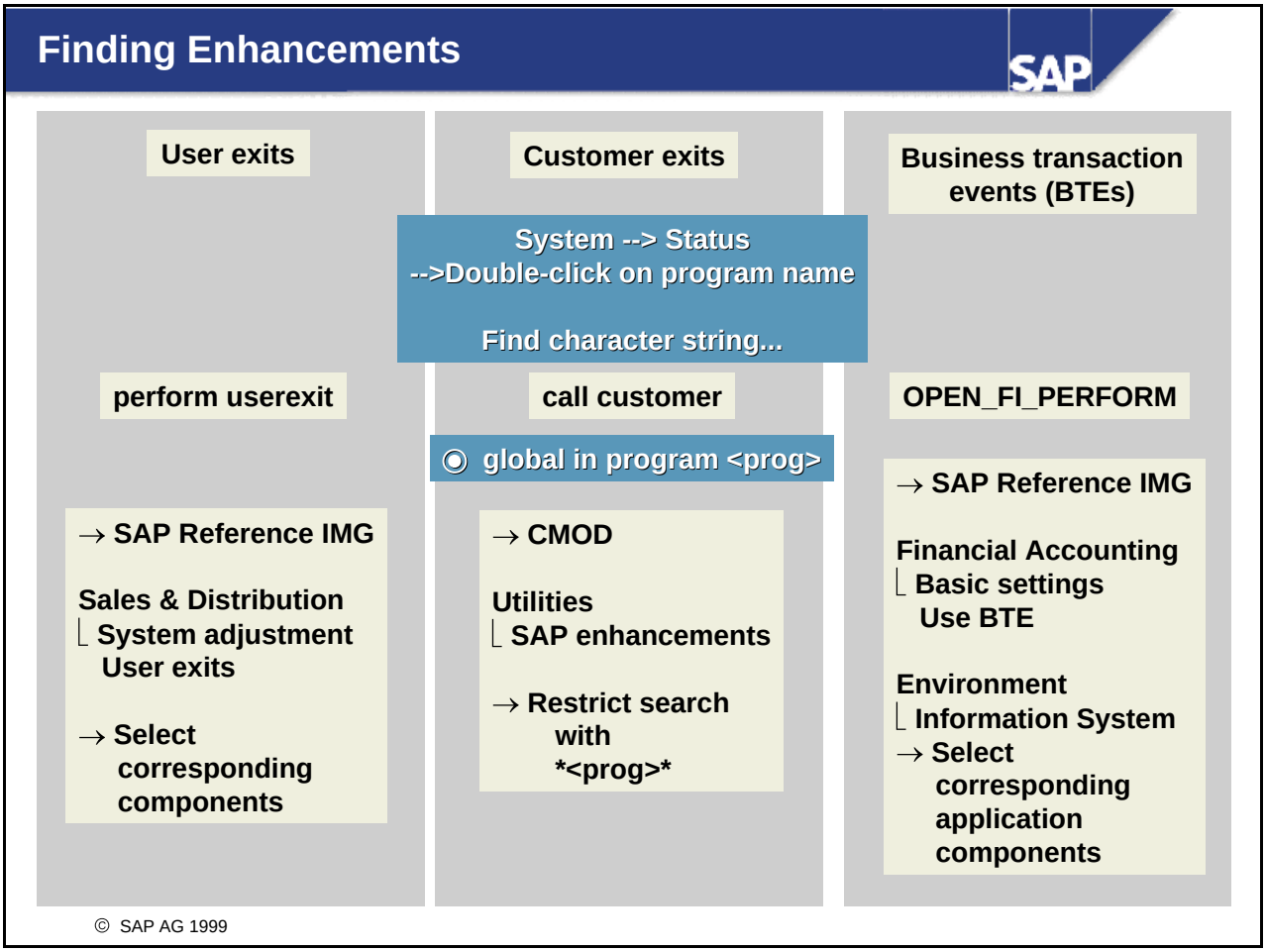

- There are four different ways of changing the system to meet your requirements:
  - **Customizing:** Allows you to change system parameters using a special interface. All possible changes have been thought of and organized. Customizing is a mandatory part of setting up a system.
  - **Enhancement concept:** Allows you to change SAP Repository objects without modifications.
  - **Customer development:** Creating customer-specific objects in accordance with the customer namespace conventions.
  - **Assisted modification:** You can carry out modifications to SAP Repository objects using the Modification Assistant. Modifications can lead to a considerably increased workload for an upgrade. If there are changes on the same Repository object in the new Release, you have to adjust the versions manually. The Modification Assistant can help you carry out the adjustment automatically in parts. You can find additional information on the Modification Assistant in the SAP Library under *Basis Components --> ABAP Workbench --> Changing the SAP Standard --> Modification Assistant.*

## How Enhancements Function

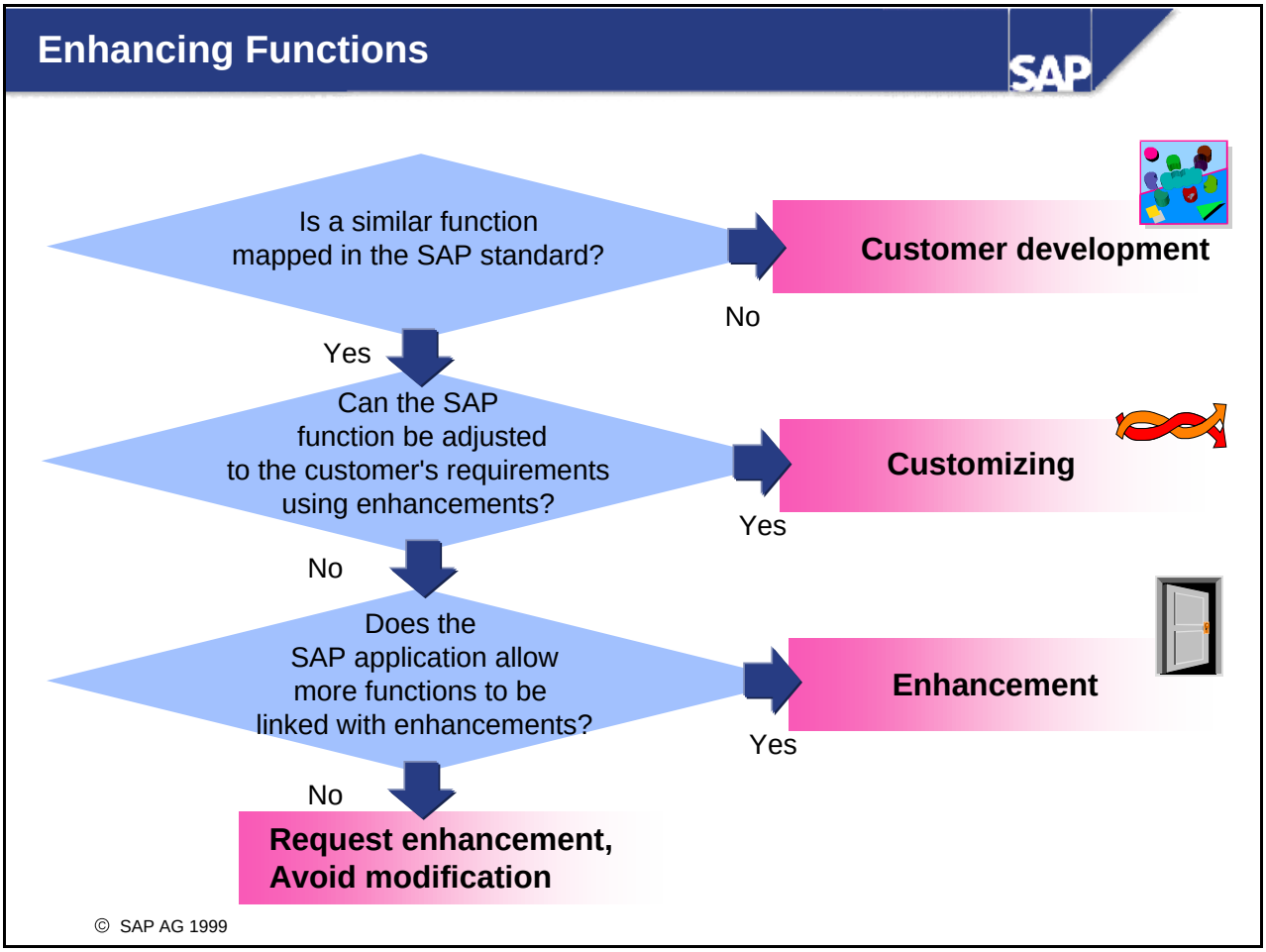





- SAP objects are usually enhanced as shown.
- The SAP object enables you to go to a customer object. Since this object usually does not exist, you have to implement it in the specified manner.
- Different enhancement techniques are implemented differently. Depending on the enhancement technique, you have to use a different maintenance transaction to use the enhancement and to create the corresponding objects.
- The corresponding maintenance transaction has a search function for finding a suitable enhancement. You can also find documentation about the corresponding enhancement here.

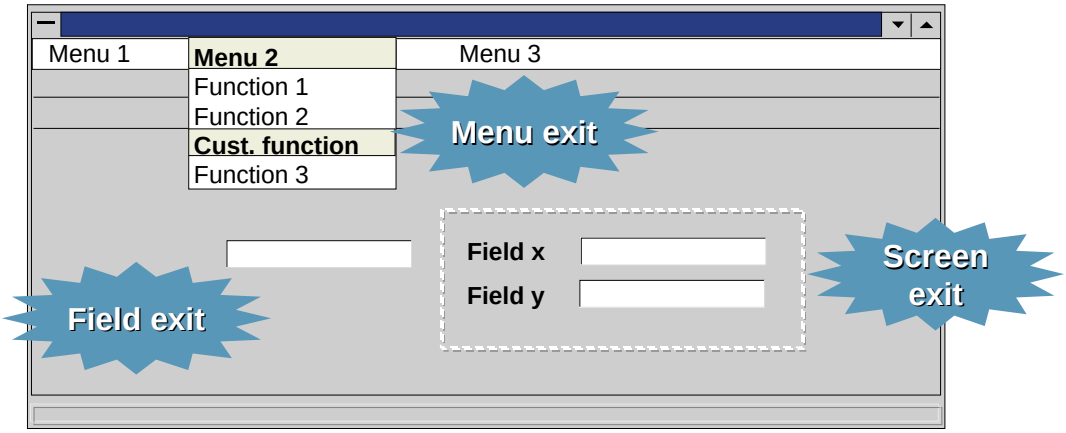



- With Release 4.0, R/3 offers three options for enhancing the delivered standard. You have now learned the techniques for implementing the enhancements. The options for finding out whether a program offers an enhancement directly from the running ABAP program are here. The strategy is always the same. Find out the program name with the menu path ***System --> Status***. Navigate forwards to the source text of the main program. Use the search function to find specific character strings in the *entire* source text of the program.
- Once you have found that the program has an enhancement option, you have to find the documentation belonging to the enhancement. This documentation tells you about uses of the enhancement and its implementation methods.




- Before starting a modification, check whether it is possible to meet your requirements using Customizing or the enhancement concept instead.
- Enhancements are a method of changing SAP software while avoiding a classic modification. You can change or extend functions without having to adjust the software manually during upgrades. The different types of enhancement are mentioned later in this unit.
- You can request enhancements in the SAP Service Marketplace (http://service.sap.com).

# Enhancing User Dialogs






- You can enhance the R/3 System at the following levels:
  - **Menu exit:** The system contains various points at which you can include menu items that start customer programs.
  - **Screen exit:** Some screens contain areas (subscreens) in which you can display your own screens.
  - **Field exits:** Field exits allow you to incorporate extra field checks.

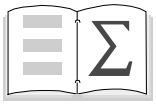
## Enhancements: Examples

- **Program exits:**
  - **Predefined exits to application programs**
- **Dictionary enhancements:**
  - **Table appends: Allow you to add extra fields to standard tables**
  - **Semantic information for data elements: You can change the field documentation (F1 help) and short texts using an enhancement**



- Customers can use the following types of programming enhancement:
- **Enhancing application programs**: SAP developers have included exits at certain points in application programs to allow you to call sections of your own coding.
- **Enhancing Dictionary objects:**
  - Table appends: Allow you to add extra fields to standard tables
  - Field documentation: You can replace the field documentation that is displayed when the user presses F1 with your own texts. The help texts are stored with the corresponding data element.
  - You can replace the field labels (short, medium, and long texts for the field) with your own texts.

# Software Logistics and Software Adjustment: Unit Summary



**You are now able to:**

- **Map a project in the R/3 System using the Transport Organizer**
- **Describe the options for enhancing or changing the functions of existing programs**

# Database Dialogs II (Making Changes to the Database)

## Contents:

- Basic Business Process
- Database LUW
- Bundling Database Changes
- Lock Concept

# Database Updates: Unit Objectives

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

- **Explain why you have to make database changes to a business unit in a database LUW,**
- **Describe the SAP lock concept**

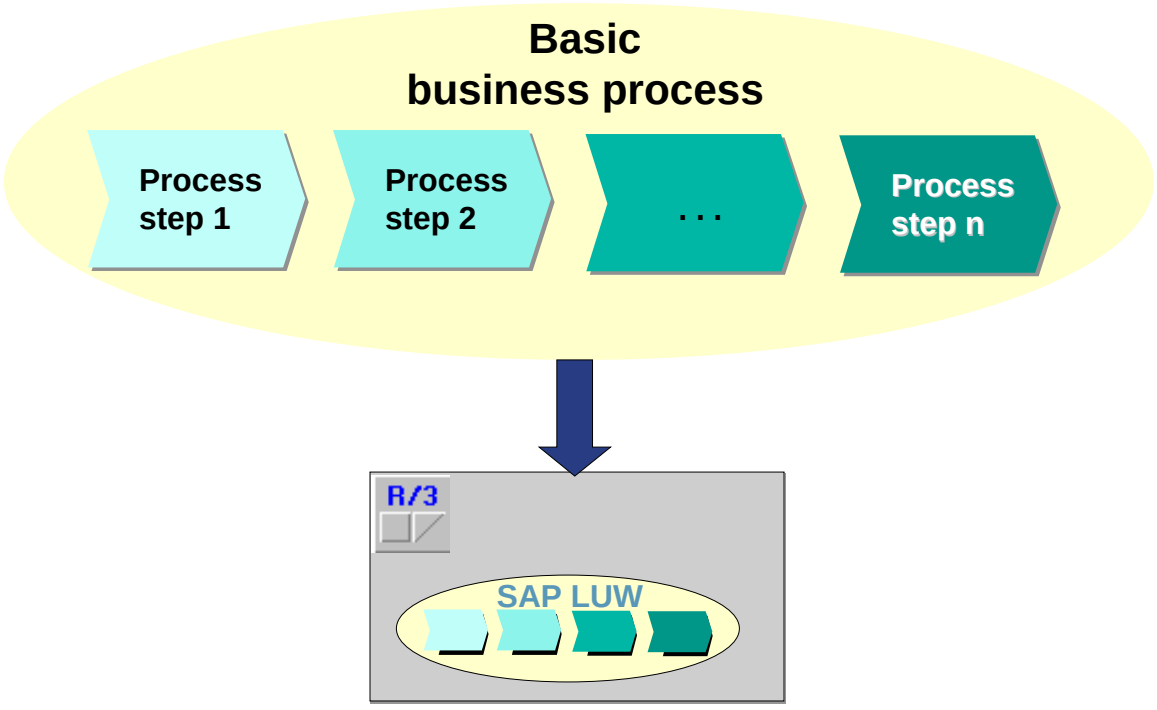
# SAP LUW and Database LUW

SAP LUW and Database LUW

Database Updates

Lock Concept

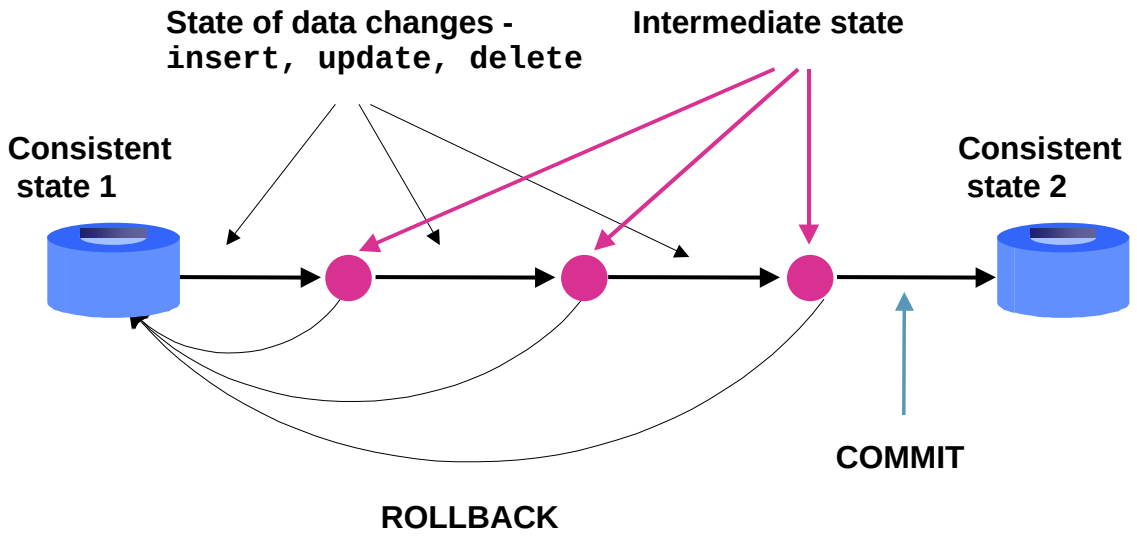
## Basic Business Process





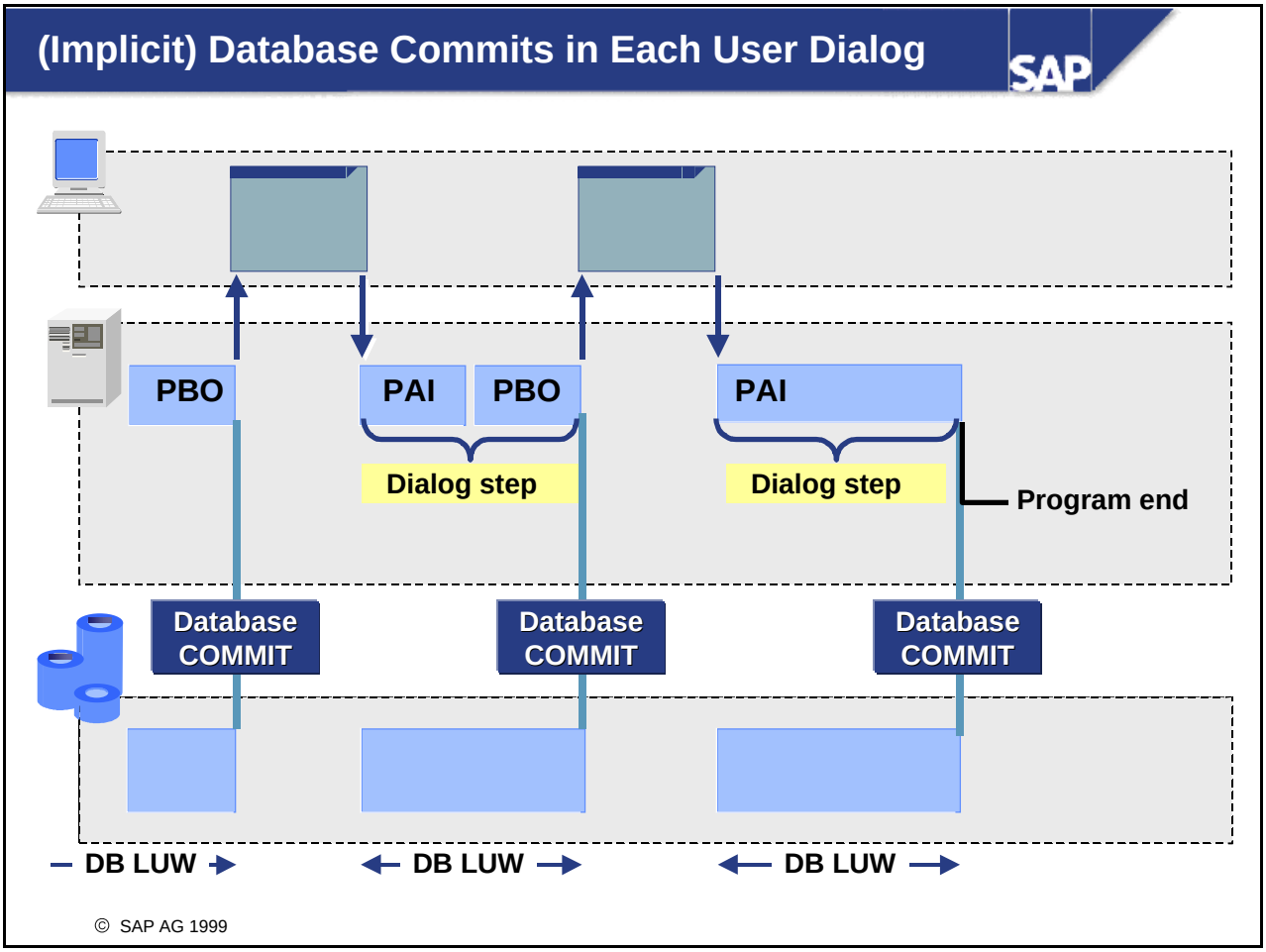

- An **SAP Logical Unit of Work (LUW)** contains a series of dialog steps for a business process in the R/3 System that form a logical unit.
- The steps in the process chain of the business process must be logically related.
- SAP LUWs work on an all-or-nothing principle: Either the system processes all of the steps, or none of them at all.
- The business process represented in the LUW must be basic, that is it must not be too big. For example, the entire process from customer order to billing is too big to be included in a single LUW. Instead, you would split the process up into smaller, independent sections, each of which would form a transaction in the R/3 System. Exactly what constitutes a "basic" process depends on the business process and the way in which you have modeled it.
- **Note:** A business LUW is often refered to as a **transaction**. The term transaction has several meanings. In an SAP environment, transaction is often understood to mean an application that you start using a transaction code. A program can include several SAP LUWs. In this training course we will use the term SAP LUW.

## Database LUW

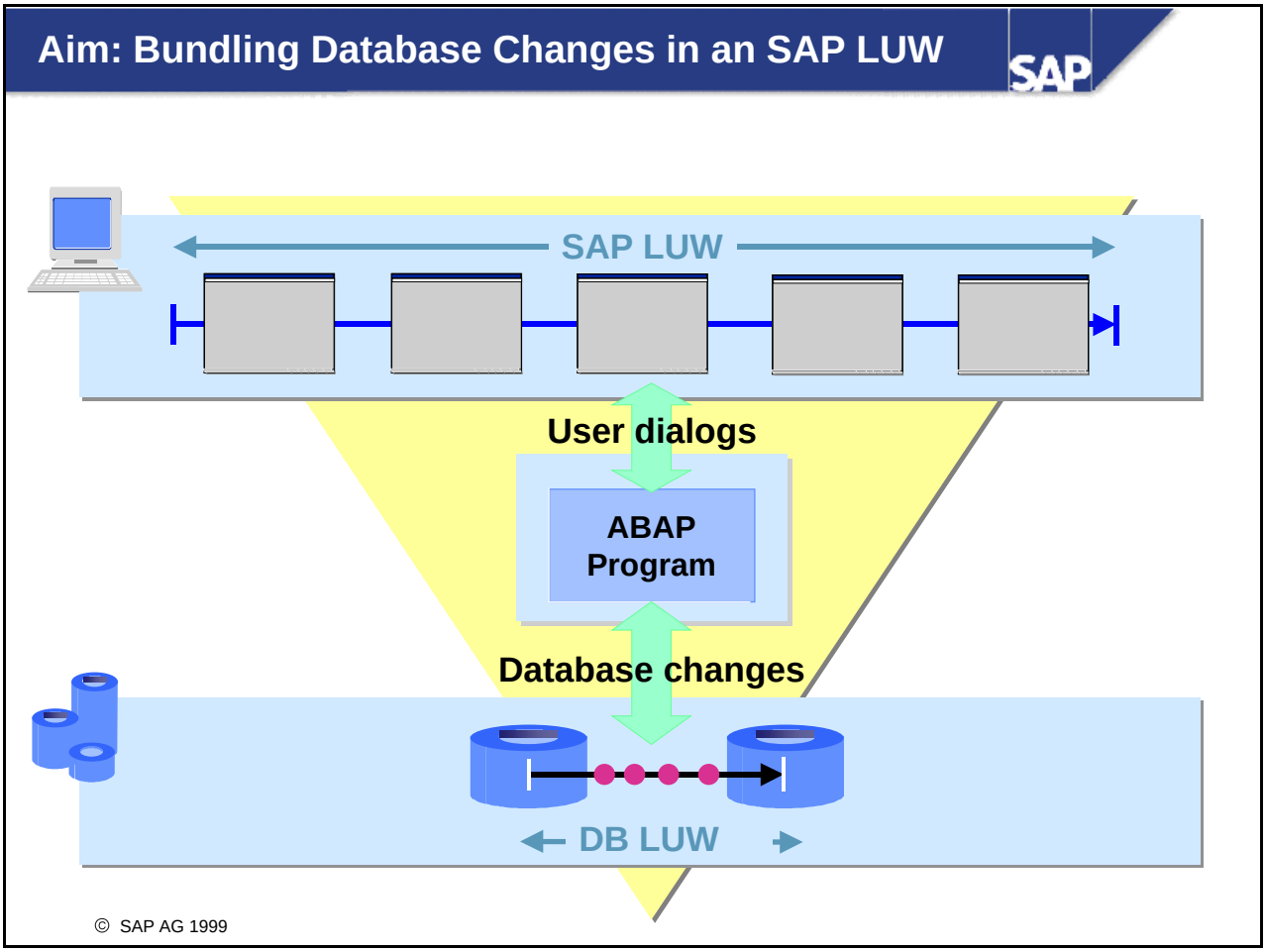





- A **database LUW** (DB Logical Unit of Work) is an inseparable sequence of database operations that takes the database from one consistent state to another.
- Database LUWs are either completely executed by the database system, or not at all.
- Database LUWs close with a database commit. It is only in the commit that the changes are firmly written in the database. Until the commit occurs, you can undo your changes using a database rollback.




- Implicit database commits are triggered when:
  - A dialog step is completed
  - An error dialog occurs
  - You call a function module in another work process (RFC).
- The SAP LUW can span several dialog steps and contains consistency checks, which are processed as closely as possible to the user action, so that the user can be informed of the error status. Since the user processing time of a screen is generally much longer than a dialog step of the application server, an SAP LUW requires a much larger time frame than a database LUW.




- Using an SAP LUW to represent a business process in the R/3 System involves both user dialogs and a database dialog. The purpose of a transaction is to make sure that the data exchanged between program and user in the user dialogs is processed on an all-or-nothing basis in the database. This means that all of the changes from the SAP LUW must be processed in a **single** database LUW.
- Usually an SAP LUW is processed in more than one DB LUW.
- The aim when programming a transaction is to bundle the segments of the database dialog in a DB LUW.
- You should aim to process the database dialogs as late as possible within the database LUW, and to keep the database locks set for as short a time as possible.

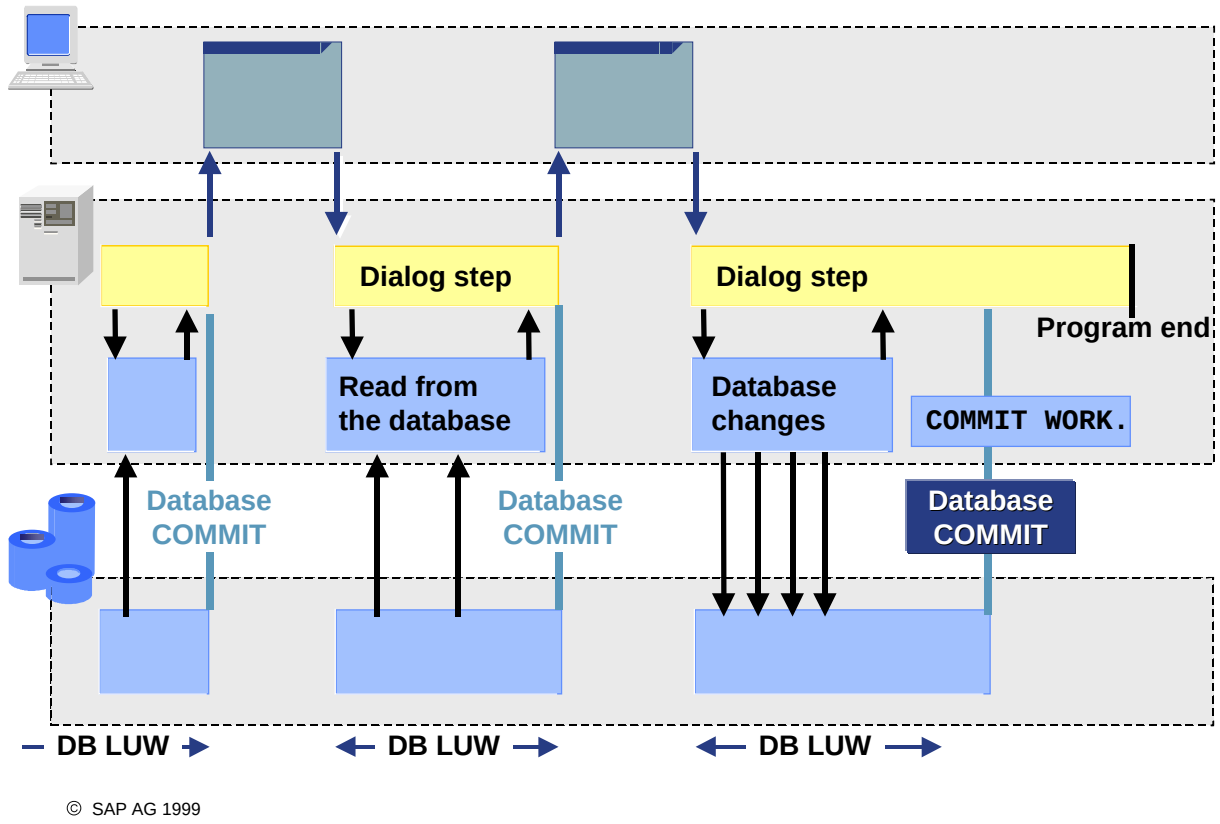
# Database Updates

SAP LUW and Database LUW

Database Updates

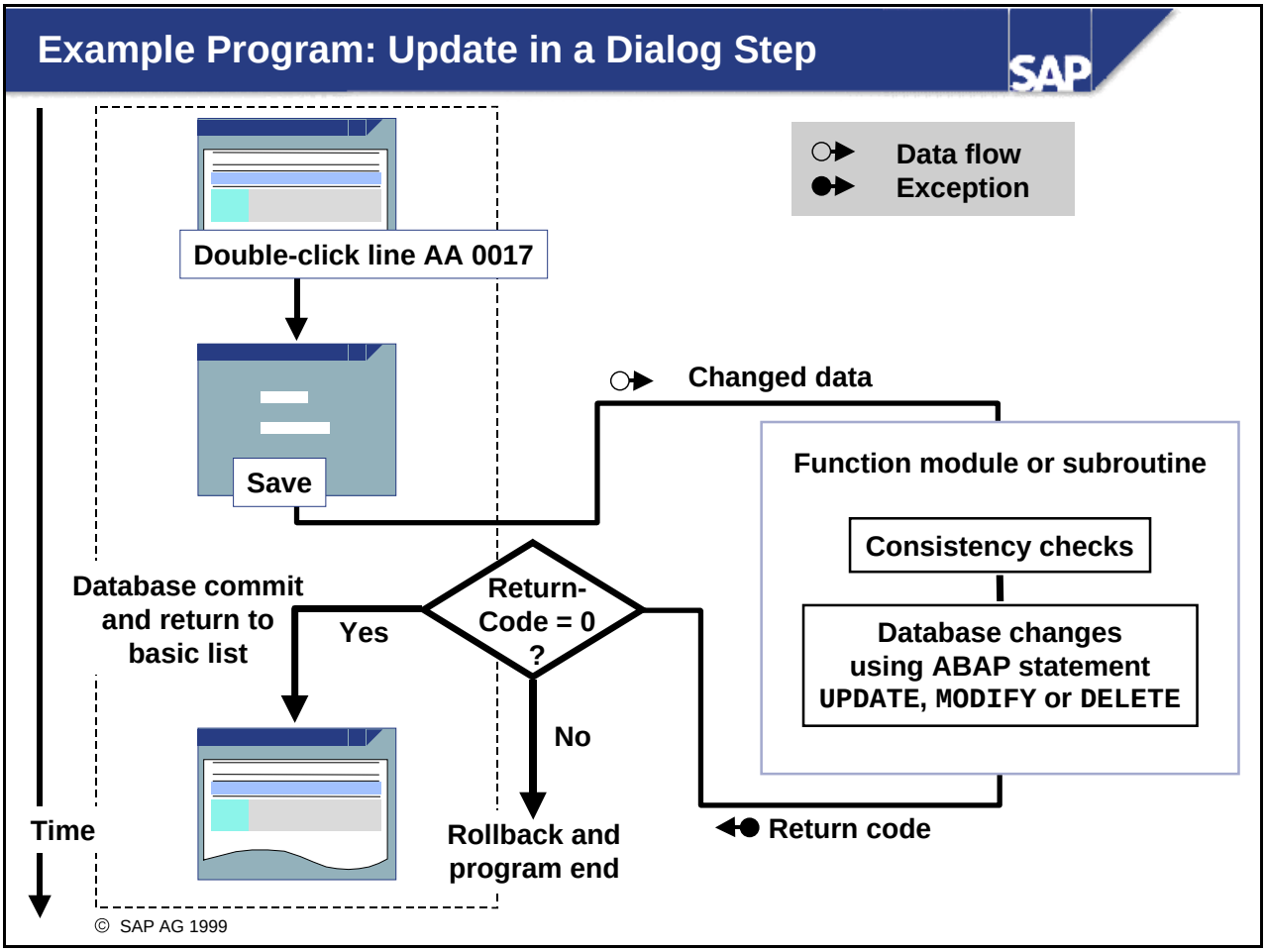
Lock Concept

## Solution: Database Updates in a Single Dialog Step

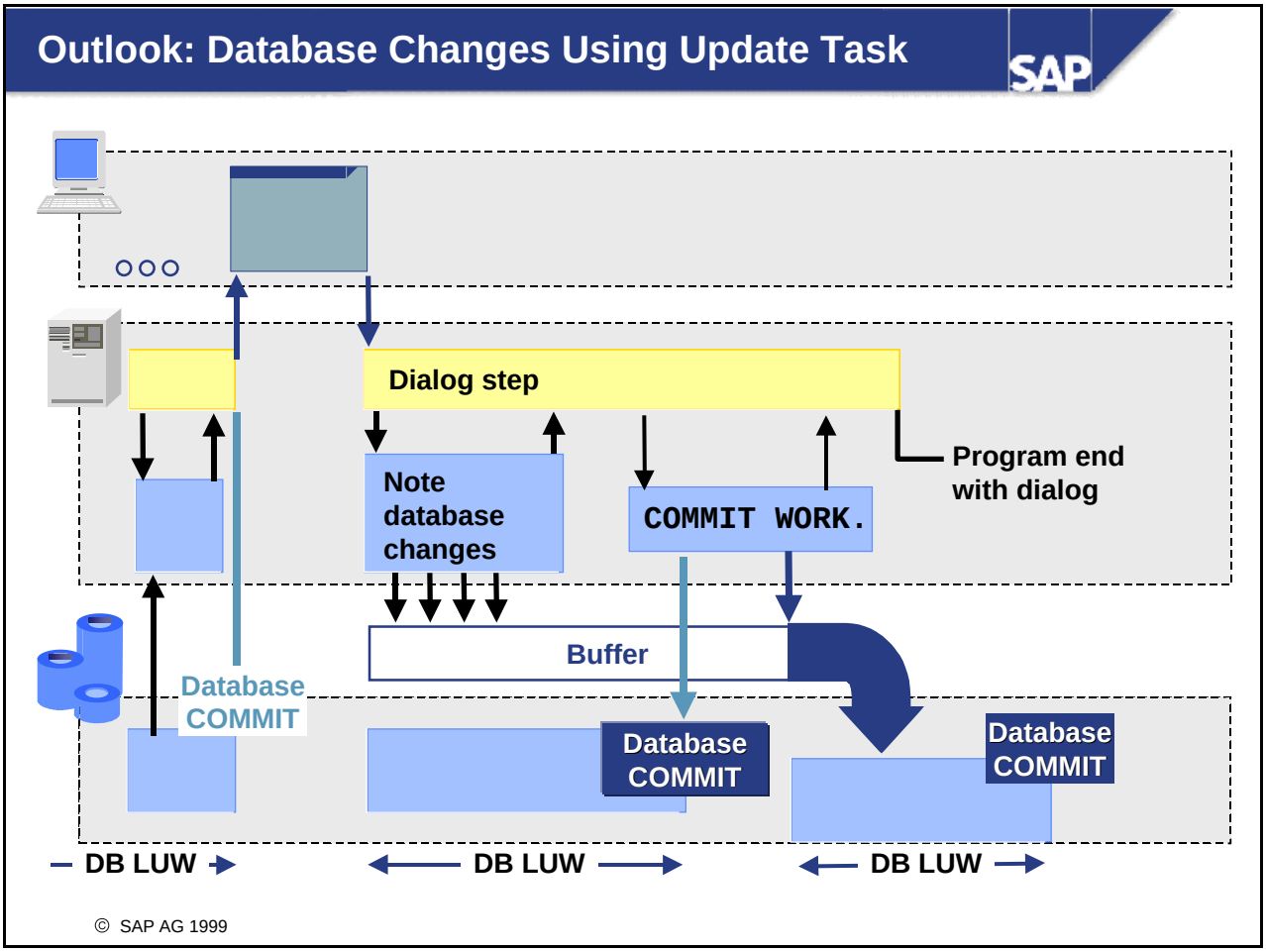





- For simple database changes you can ensure the bundling of database changes by grouping together the ABAP statements for all database changes in a processing block. You should be careful not to trigger a user dialog in the processing block. In this context, note that a message of type I, W, or E is triggered in the user dialog.
- After the database update, trigger an explicit database commit using the **COMMIT WORK** statement. Ensure that the database changes are written to the database, when an implicit database commit is triggered.




- This diagram shows one method of carrying out database changes in our example program.
- By choosing *Save* you trigger the database changes on the screen. This calls a function module in a PAI module that executes consistency checks and calculates the data. If no errors occur, the database changes are executed and the function module terminates normally (with return code 0). If errors occur, the function module is terminated and return code <> 0 is given.
- In the program the return code can have the following values:
  - 0 A database commit is triggered and processing continues
  - <> 0 A database rollback is triggered and the program is terminated




- Generally, database changes are more complex for business programs. For this you usually use a technique that notes the database changes first. The ABAP statement **COMMIT WORK** triggers an update work process in which the database changes are carried out.
- If you choose different attributes the database update is triggered asynchronously, so that the user can continue processing without waiting for the update to finish. This logs errors simultaneously. A database update that is terminated because of errors can be restarted.
- Before you program extensive database changes, find out about the available techniques from the online documentation or training course ***BC414: Programming Database Updates.***

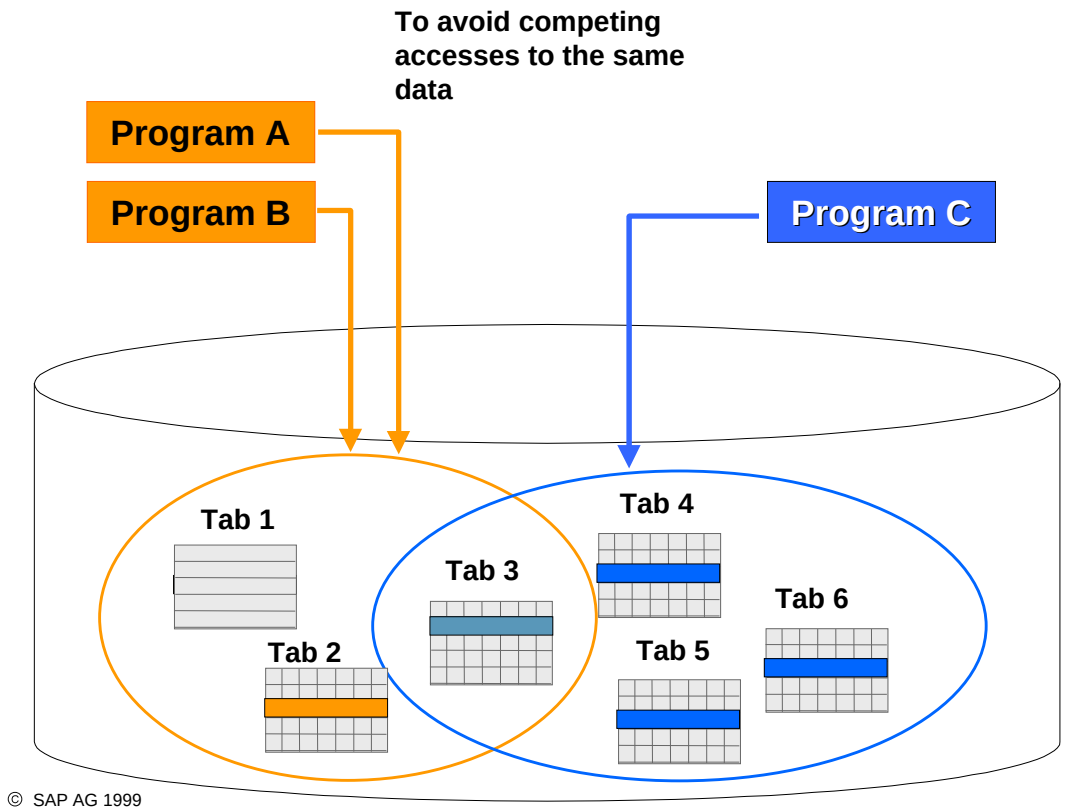
# Lock Concept

SAP LUW and Database LUW

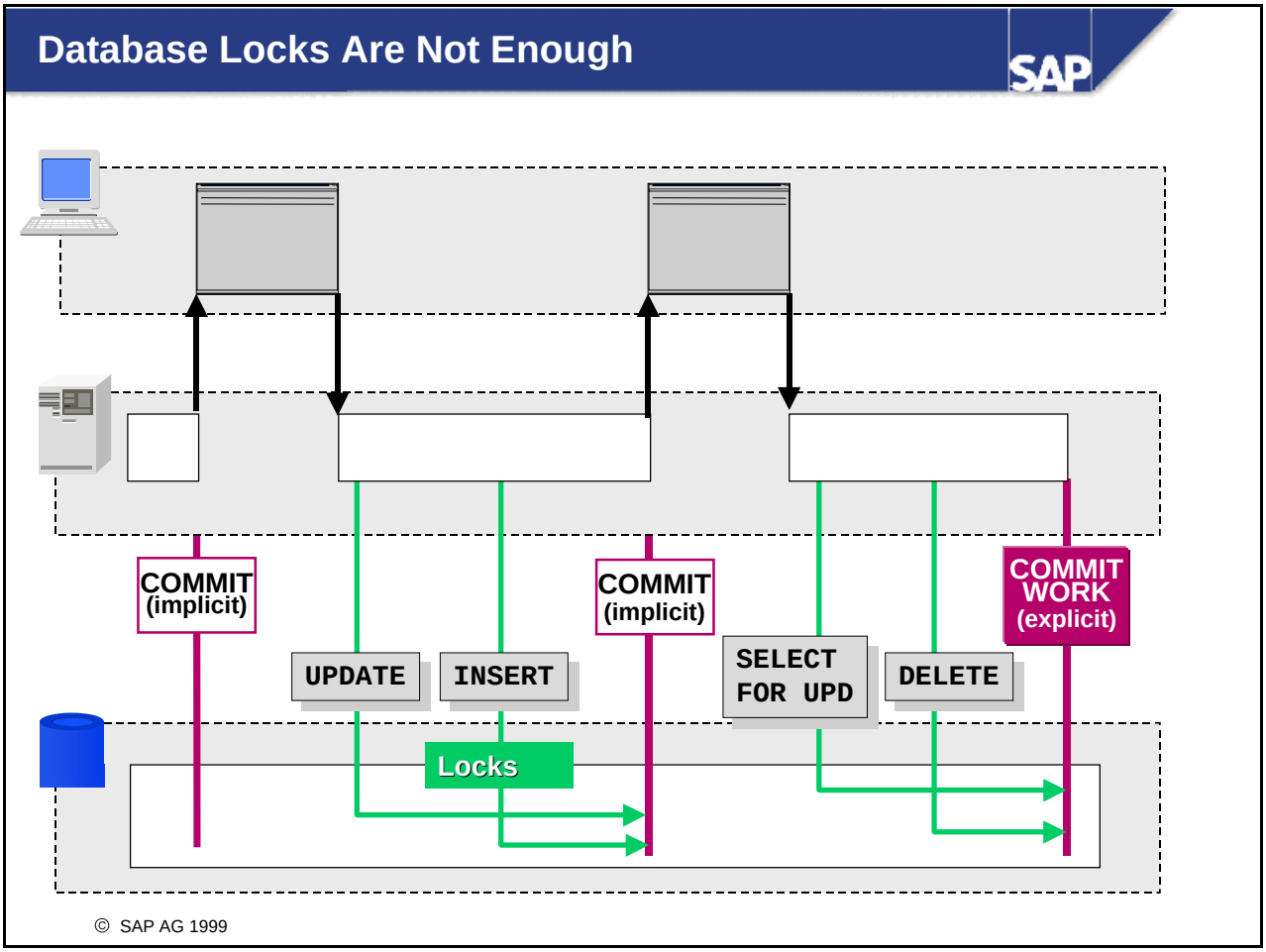
Database Updates

Lock Concept

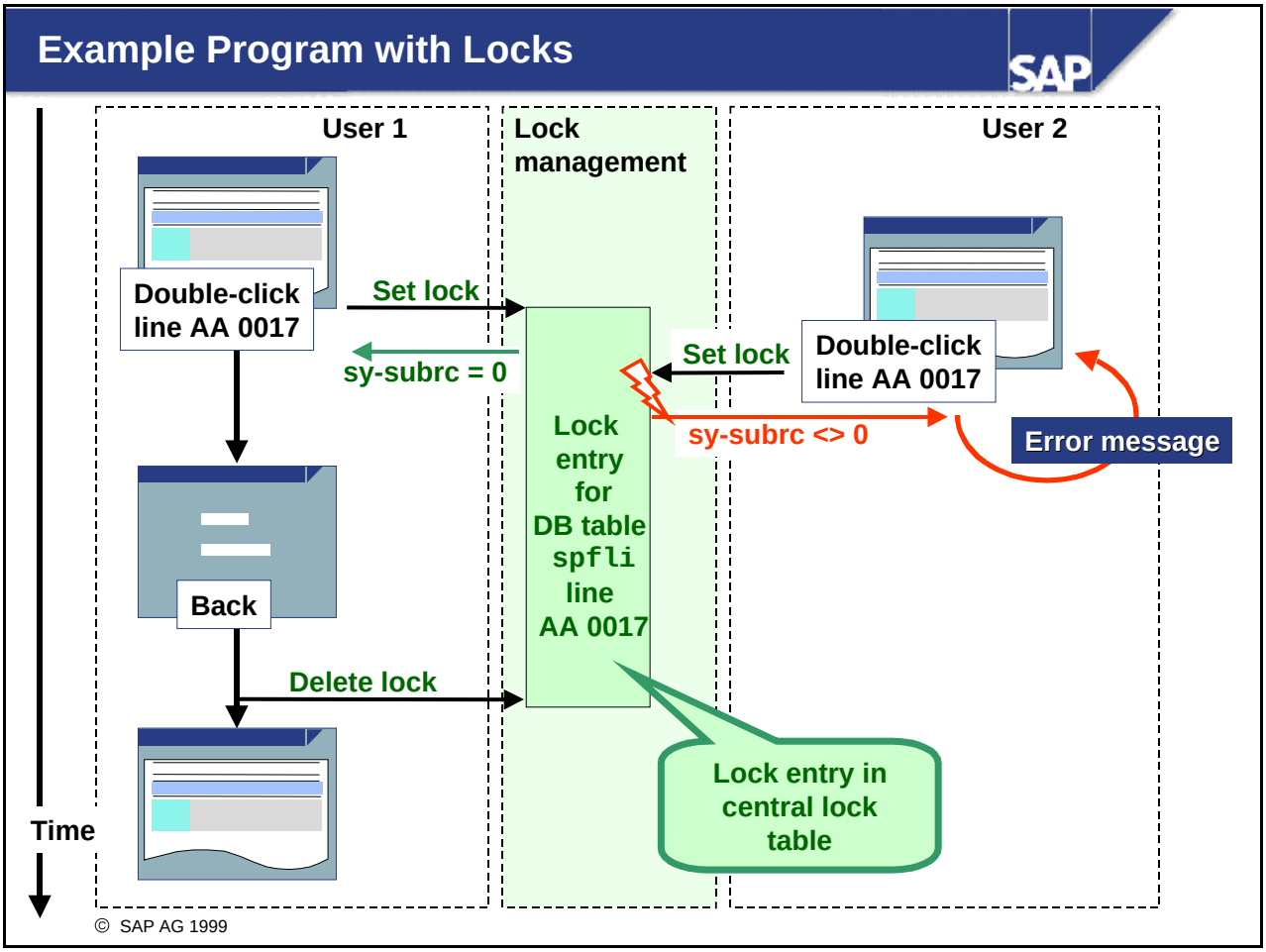
## Why Set Locks?




- If several users are competing to access the same resource or resources, you need to find a way of synchronizing the access in order to protect the consistency of your data.
- Example: In a flight booking system, you would need to check whether seats were still free before making a reservation. You also need a guarantee that critical data (the number of free seats in this case) cannot be changed while you are working with the program.
- Locks are a way of coordinating competing accesses to a resource. Each user requests a lock before accessing critical data.
- It is important to release the lock as soon as possible, so as not to hinder other users unnecessarily.




- Whenever you make direct changes to data on the database in a transaction, the database system sets corresponding locks.
- The database management system (DBMS) physically locks the table entries that you want to change (INSERT; UPDATE, MODIFY), and those that you read from the database and intend to change (SELECT SINGLE <f> FROM <dbtab> FOR UPDATE). Other users who want to access the locked record or records must wait until the physical lock has been released. In such a case, the ABAP program waits until the lock has been released again.
- At the end of the database transaction, the database releases all of the locks that it has set during the transaction.
- In the R/3 System, this means that each database lock is released when a new screen is displayed, since a change of screen triggers an implicit database commit.




- Locks are maintained in a central lock table. This ensures that also programs that run on a different application server of the same SAP System, are informed of the locks.
- Before a database update is triggered, the program requests a lock using a special function module. You can set a lock for a data record in a database table, or even a set of records, according to you requirements. The function module first checks whether there is an existing lock that will obstruct the lock request. If a lock does not already exist then the lock is set.
- If another program tries to set the same lock, the function module sends a message to say that the record is already locked. This is carried out using an exception of the function module. Afterwards the return code is set to the value <> 0. The return code must be supplied with values in the program. You can inform the user of a corresponding message.
- If the database change is successful, then the lock entry in the central lock table is deleted using a different function module.

## Example Program: Locking and Unlocking

```
REPORT SAPBC400TCD_ENQUEUE_DEQUEUE.
...
AT LINE-SELECTION.
 PERFORM authorization_check USING wa_spfli-carrid '02'
                             CHANGING subrc.
  IF subrc <> 0. MESSAGE e047(BC400) WITH wa_spfli-carrid. ENDIF.

  CALL FUNCTION 'ENQUEUE_ESSPFLI'
              EXPORTING carrid = wa_spfli-carrid
                        connid = wa_spfli-connid
              EXCEPTIONS
                       foreign_lock   = 1
                       system_failure = 2
                       others         = 3.
  IF sy-subrc <> 0.
   MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO
           WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
  ENDIF.
  SELECT SINGLE * FROM spfli INTO wa_spfli
                 WHERE carrid = wa_spfli-carrid
                 AND  connid = wa_spfli-connid.
   MOVE-CORRESPONDING wa_spfli TO sdyn_conn.
CALL SCREEN 100.
CALL FUNCTION 'DEQUEUE_ ESSPFLI' ...
```



- In the example program it makes sense to set a lock entry, before the data record has been read from the database and the screen has been processed.
- You set a lock entry by calling an **ENQUEUE** function module for an appropriate lock object. You can find out which lock objects are available for a database table from the table's where-used list in the ABAP Dictionary. The "lock" and "unlock" function modules for the selected lock object require only the **ENQUEUE_<Name of lock object>** and **DEQUEUE_<Name of lock object>** naming conventions. In general, you need only pass the interface key fields. Default values are passed to all the other parameters.

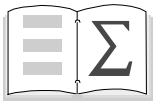
## Example Program: Database Updates

```
REPORT SAPBC400TCD_ENQUEUE_DEQUEUE.
...
MODULE user_command_0100 INPUT.
  CASE ok_code.
    ...
    WHEN 'SAVE'.
      MOVE-CORRESPONDING sdyn_conn TO wa_spfli.
      CALL FUNCTION 'BC400_UPDATE_FLTIME'
           EXPORTING
                iv_carrid             = wa_spfli-carrid
                iv_connid             = wa_spfli-connid
                iv_fltime             = wa_spfli-fltime
                iv_deptime            = wa_spfli-deptime
           EXCEPTIONS
                    OTHERS                 = 1.
      IF sy-subrc NE 0.
        MESSAGE a149.  " implicit database rollback
      ELSE.
        COMMIT WORK.   " explicit database commit
        MESSAGE s148.
        LEAVE TO SCREEN 0.
      ENDIF.
  ENDCASE.
ENDMODULE.                              " USER_COMMAND_0100  INPUT
```



- The example program shows a simple database update, which affects only one database table. This allows us to update the database directly from a dialog - that is, using the simplest technique. The ABAP statements that update the database are all executed within a DB LUW. You must not include a user dialog between two ABAP statements that update the database. NOTE: This includes error messages. If an error occurs, the program should be canceled with an "abnormal end" message, so that a rollback automatically occurs. All the necessary calculations, consistency checks, and database updates are encapsulated here in a function module. A function module call (without a **DESTINATION...**addition) does not affect the DB LUW.
- More complex database updates are performed using update modules. For more information, see ***BC414 Programming Database Updates.***

# Database Updates: Unit Summary



**You are now able to:**

- **Explain why you have to make database changes to a business unit in a database LUW,**
- **Describe the SAP lock concept**

# Developing Internet Applications

## Contents:

- SAPGUI for HTML
- Making Selected Transactions Available for the Web
- Outlook: Making Functions Available in a Web-Specific Layout

# Developing Internet Applications

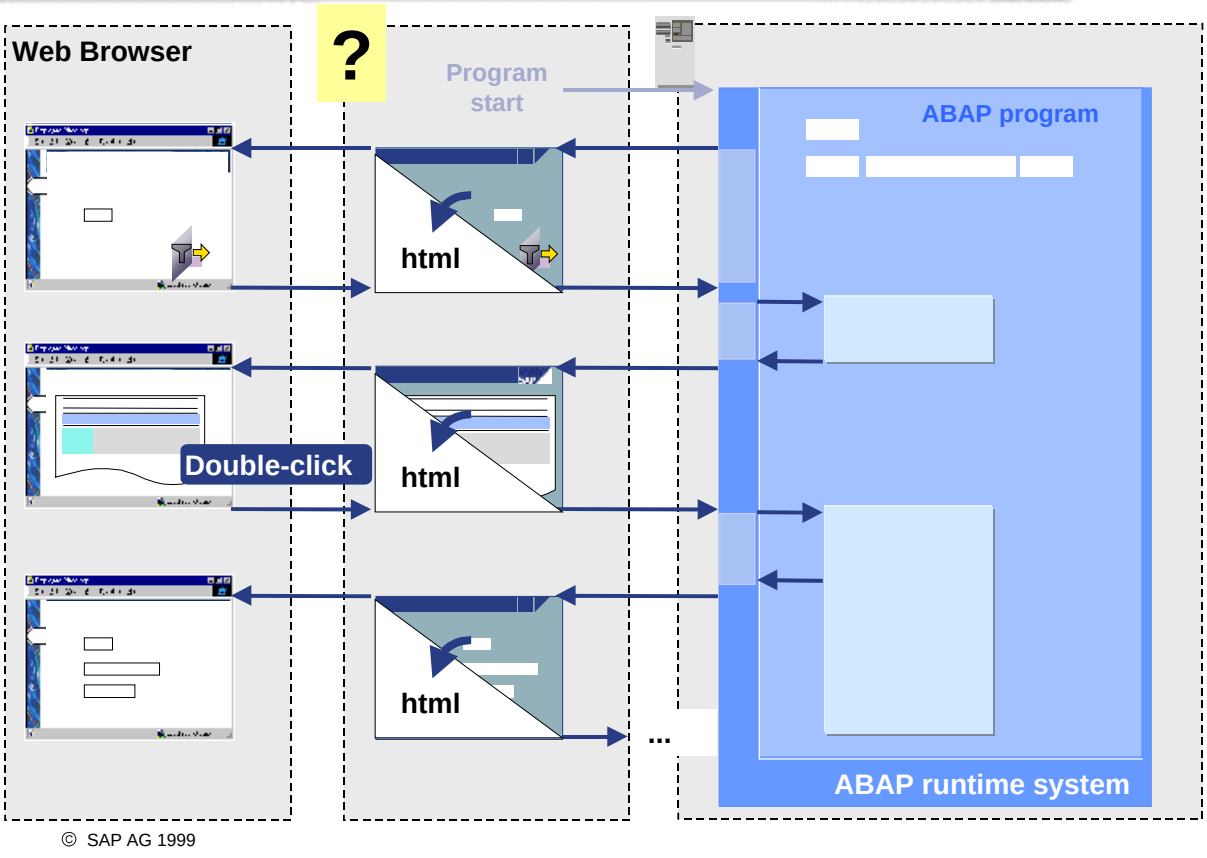
**At the conclusion of this unit, you will be able to:**

- **Create a transaction code for an Easy Web Transaction**
- **Create an Internet service**
- **Publish an Internet service**
- **Test an Easy Web Transaction using a Web Browser**

# Overview of SAPGUI for HTML

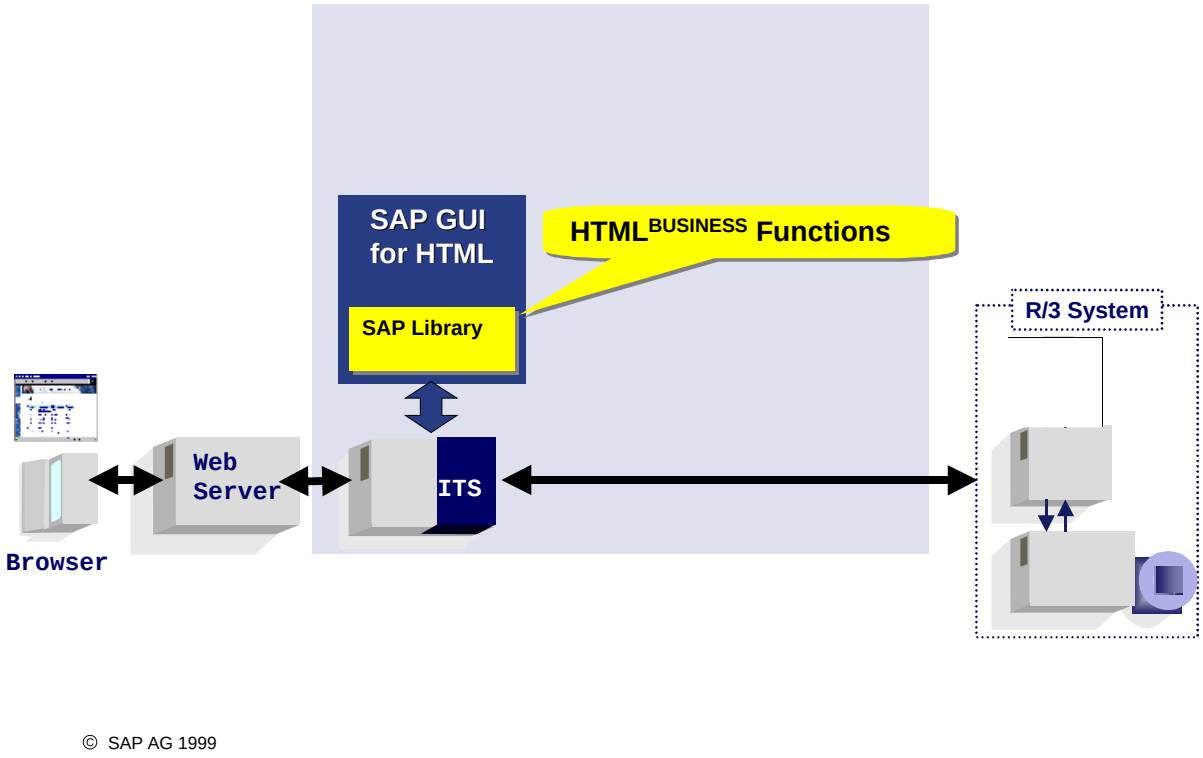
- SAPGUI for HTML
- Easy Web Transaction
- Outlook: ITS Flow Logic

# Objective: Representing Screens Using HTML Pages

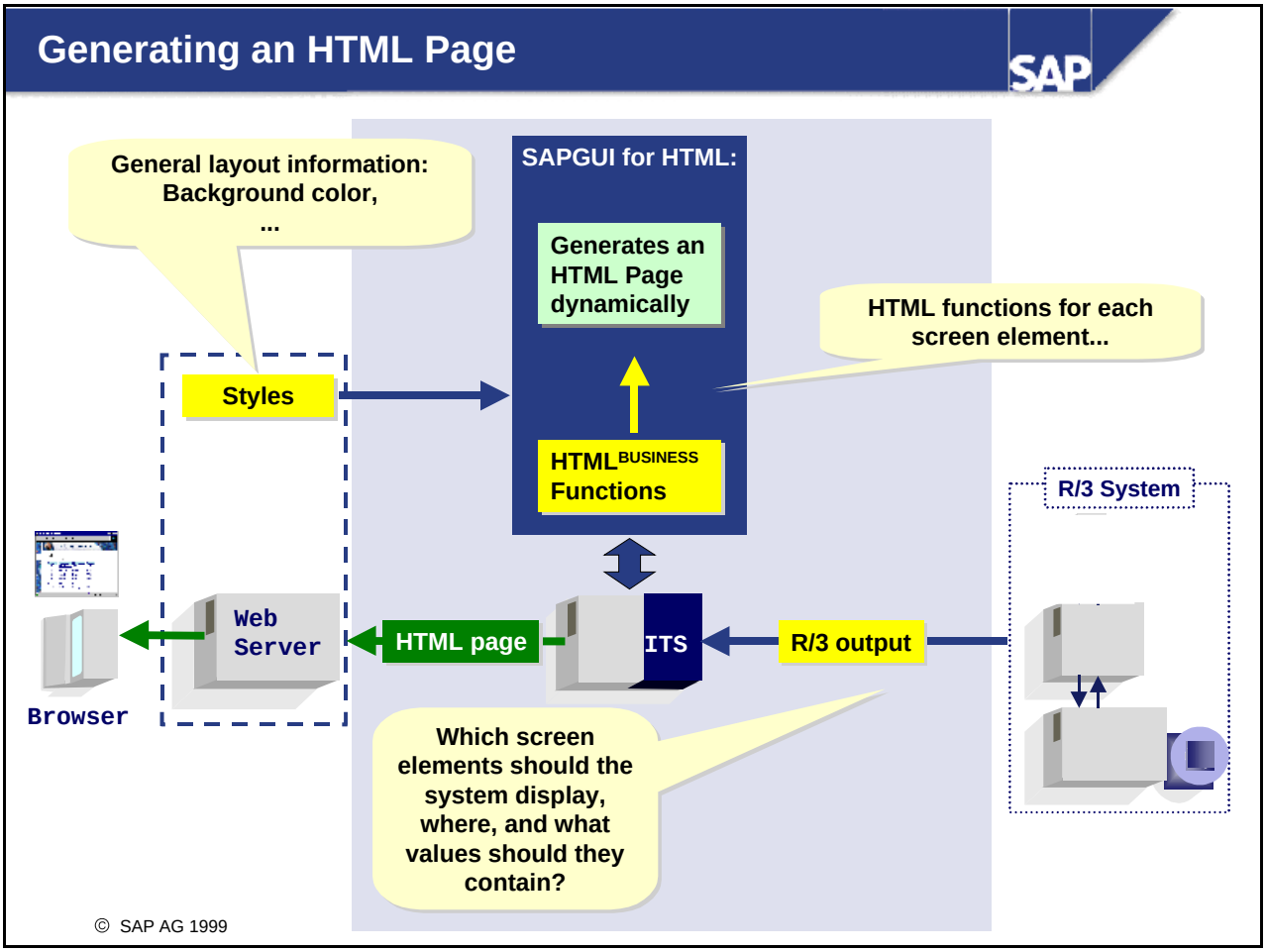



- To launch an SAP program from a Web browser, you must first generate HTML pages for the screens.
- You can generate these automatically.

# SAPGUI for HTML: Architecture




- You can start an SAP transaction using any Web Browser, provided there is an Internet Transaction Server (ITS) set up for the SAP System. Users do not need to install a GUI at the front end. Instead, a ***SAPGUI for HTML*** is installed on an ITS. This installation includes a library of HTML BUSINESS functions that you need to generate HTML templates dynamically. This generated HTML page is then sent to the Web Browser.
- A query-answer cycle between an SAP System and a Web Browser is structured as follows: A query comes from a Web Browser to the Web Server, which sends the query to the ITS. The ITS makes sure that the user is logged on to an SAP System. The data that would normally be sent to the presentation server when a screen is sent is sent to the ITS. The ***SAPGUI for HTML*** generates an HTML page and sends it to the Web Server. The Web Server sends the HTML page to the Web Browser.
- **Note:** Communication between the Web Browser and the Web Server is implemented using a **stateless** HTTP protocol. This means that the Web Server does not retain any information once the query-answer cycle has been completed. In the SAP System, a user context is created when the user logs on. It is retained until the user logs off. If a Web transaction includes several query-answer cycles, a connection must be made to the user context. For this reason, the ITS maintains its connection to the R/3 System during the entire SAP GUI for HTML session (**stateful**). The connection between the Web Browser and the Web Server is closed after every query-answer cycle (stateless). To guarantee the assignment to the user context, the Web Browser uses cookies to send an ID number with each HTTP query. This ensures that there is a unique identification with the R/3 user context.

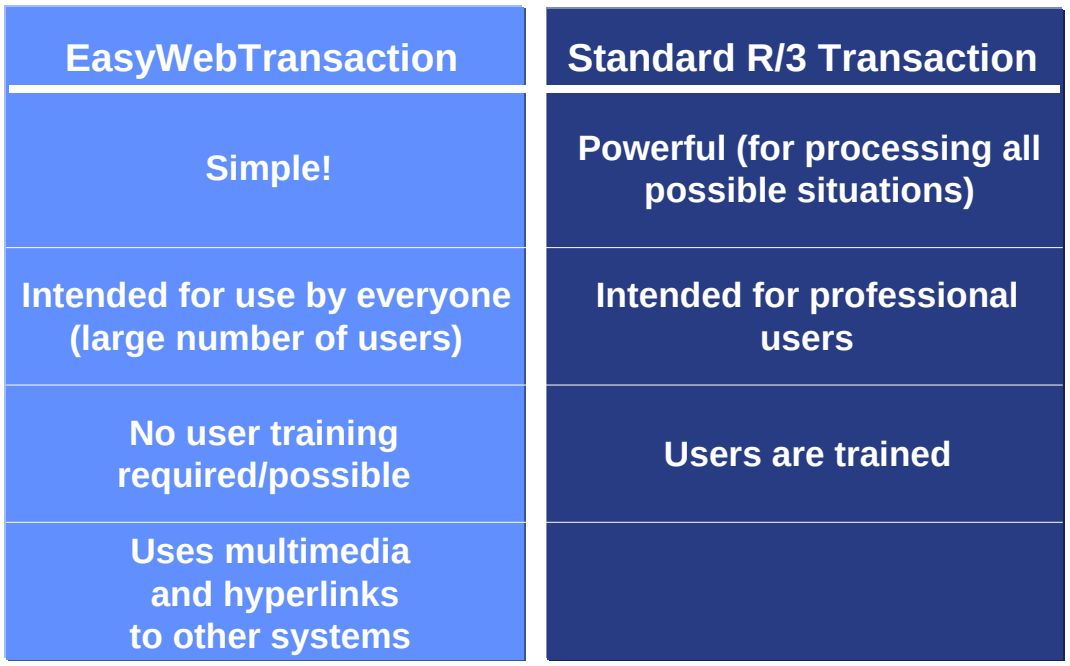



- The query-answer cycle between an SAP System and a Web Browser is structured as follows: A query comes from a Web Browser to the Web Server, which sends the query to the ITS. The ITS makes sure that the user is logged on to an SAP System. The data that would normally be sent to the presentation server when a screen is sent is sent to the ITS. The ***SAPGUI for HTML*** generates an HTML page and sends it to the Web Server. The Web Server sends the HTML page to the Web Browser.
- To generate the HTML page, the system needs the following information:
  - **R/3 output:** The information that would normally be sent to the presentation server are sent to the ITS.
  - **General layout information:** Stored on the Web Server as *styles*.
  - **SAP Library:** The HTML BUSINESS functions needed for the functions of the different screen elements are stored in a library on the ITS.

# Overview of Easy Web Transaction

- SAPGUI for HTML
- Easy Web Transaction
- Outlook: SAP Transactions with a Web Layout
- Outlook: ITS Flow Logic

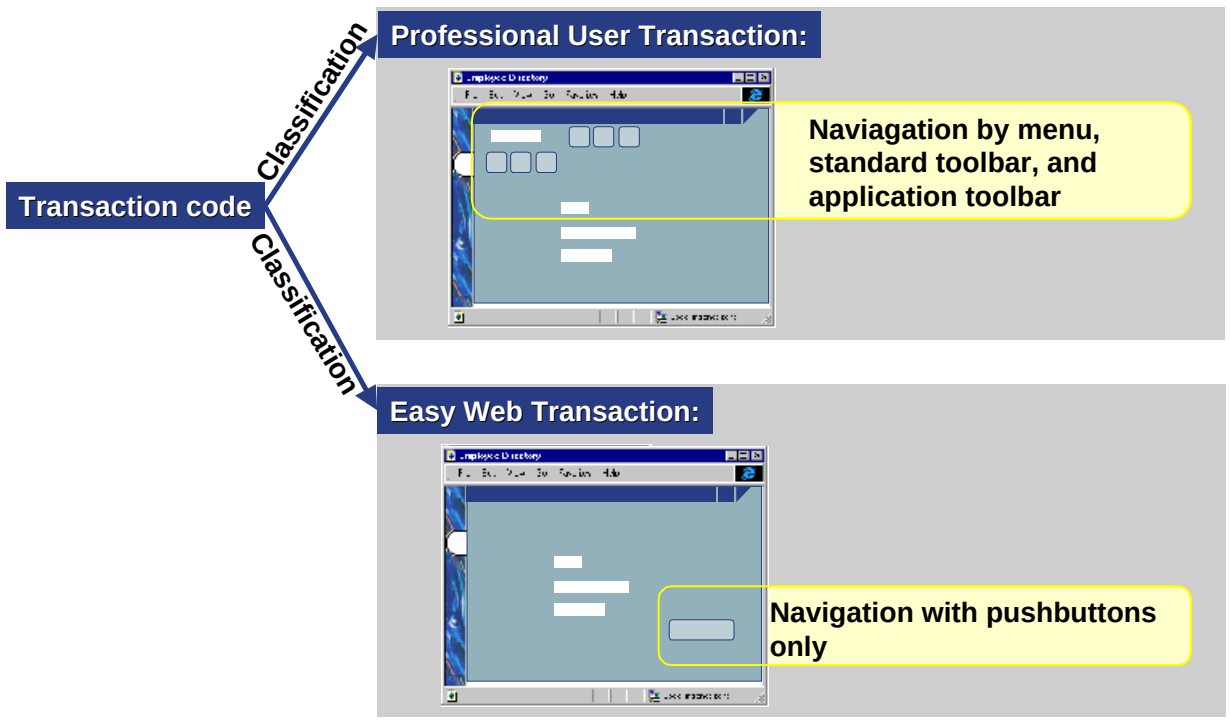
## Transaction Features





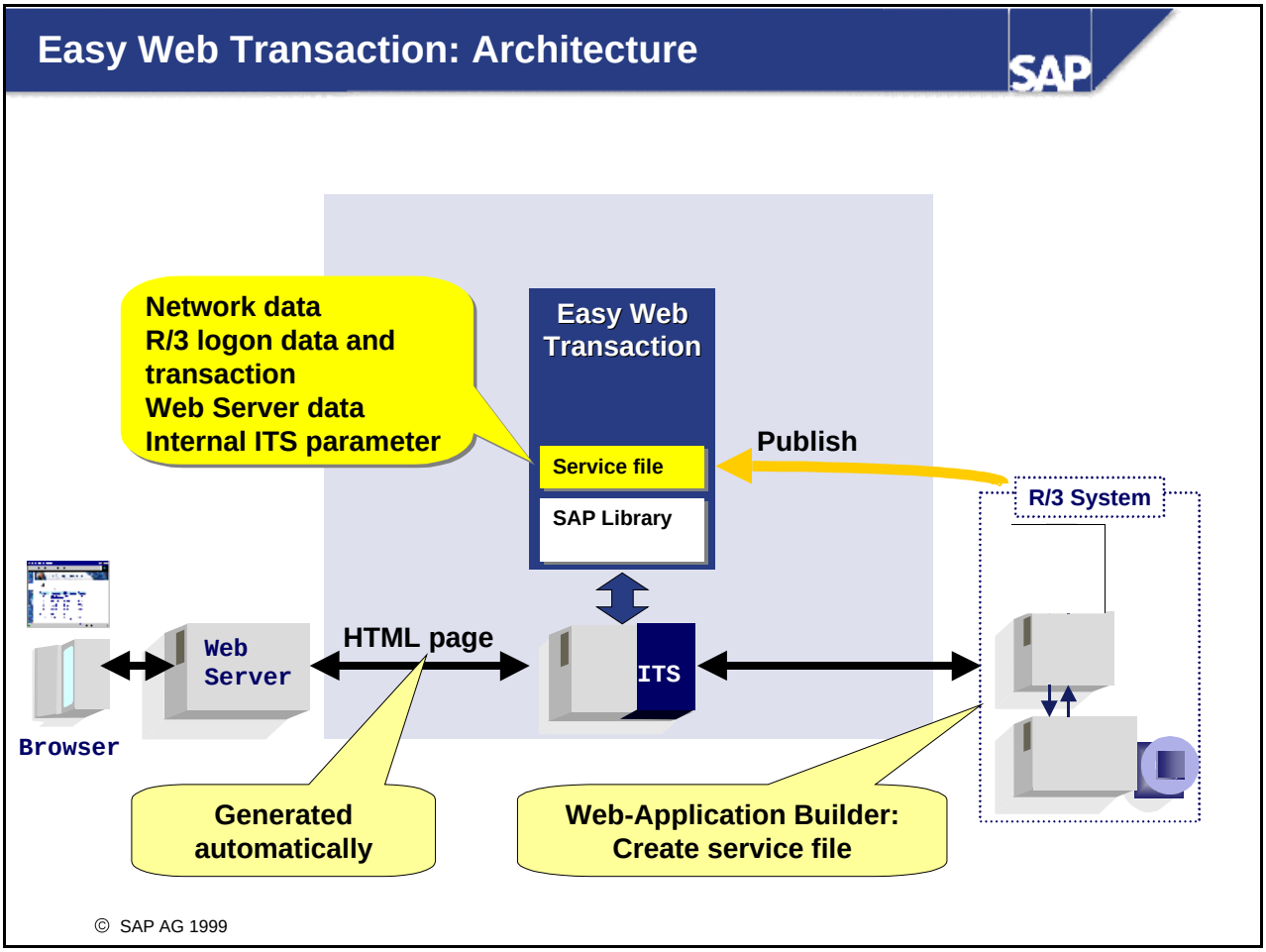

- An EasyWebTransaction makes a small part of the entire R/3 functions available to Internet users. This means that untrained Internet users can also use this functionality.

## Transaction Classification

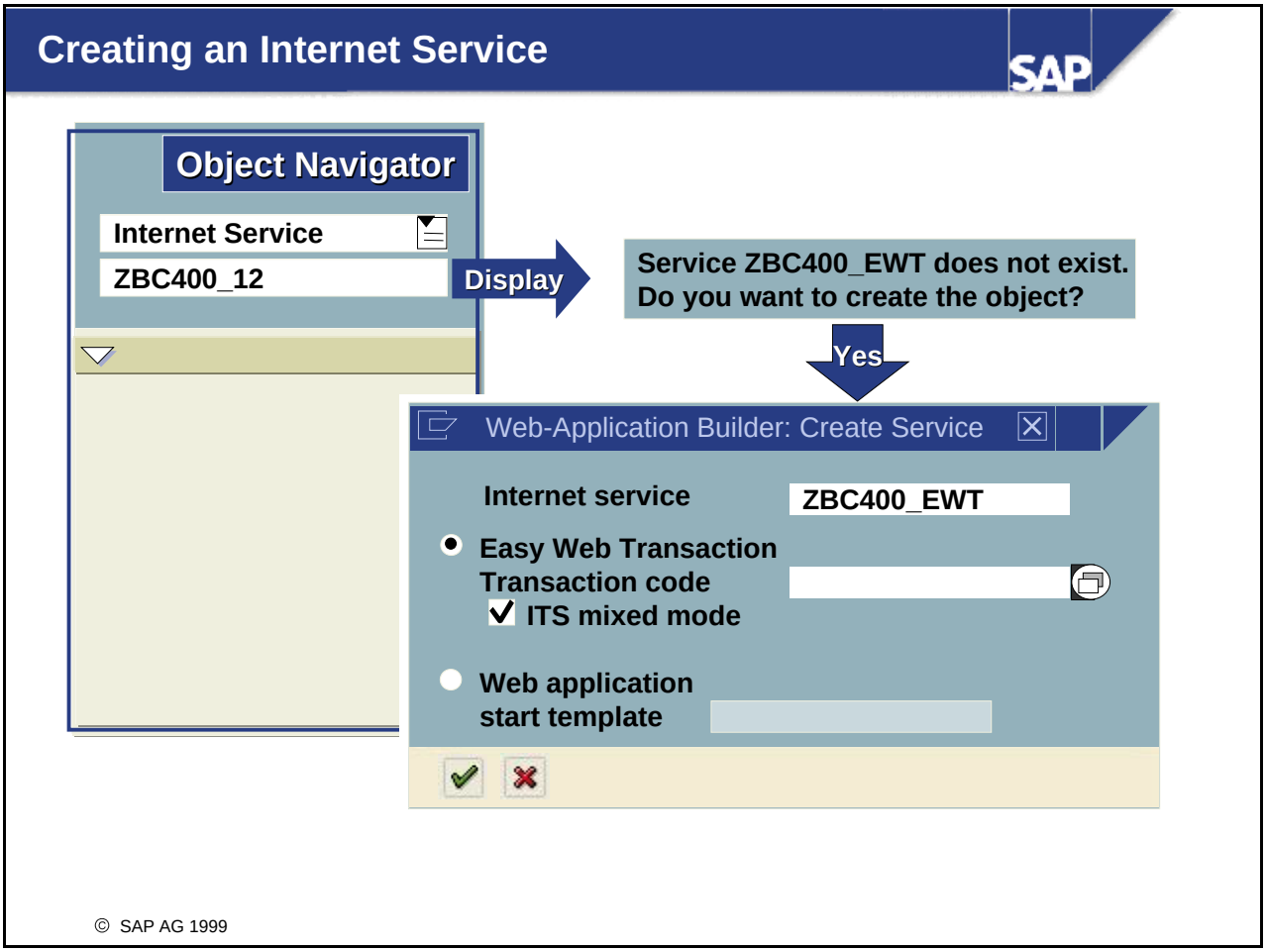





- When you create a transaction, you must specify whether it is a Professional User Transaction or a Easy Web Transaction.
- The default is ***Professional User Transaction***. You should classify a transaction as a ***Professional User Transaction*** either where users can only reasonably launch from the SAP System, or if it requires complex navigation functions. These transactions can also be launched using the *SAPGUI for HTML* - albeit with some restrictions. If the transaction can be launched with the *SAPGUI for HTML* without restrictions, you should set the *SAPGUI for HTML* flag. The *SAPGUI for HTML* generates an HTML page including all the elements of the screen including the interface (menu, standard toolbar, and application toolbar). Thus, Professional User Transactions are suitable for experienced SAP users. However, navigation is different from what an experienced Web user would expect.
- ***Easy Web Transactions*** suit the requirements of a casual user calling transactions on the Web. You should make navigation as simple as possible. The user must be able to trigger every function without using a menu, standard toolbar, or application toolbar. The system ignores interface elements when it generates the HTML page.



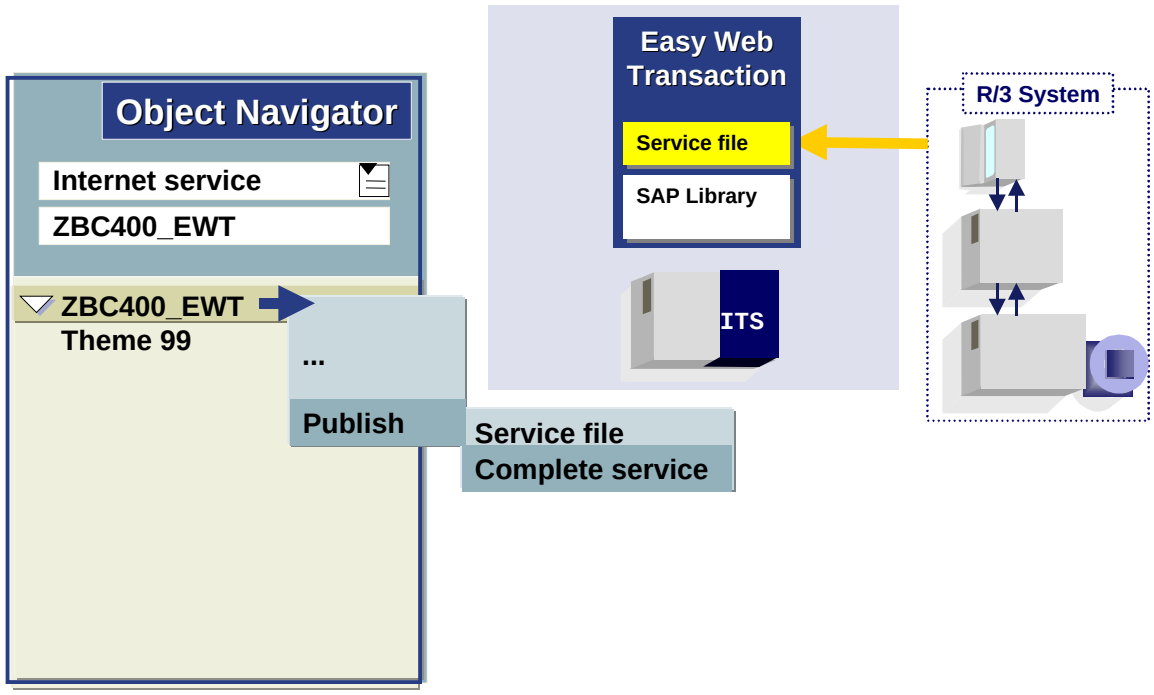

- To launch an Easy Web Transaction directly from a Web Browser without knowing the transaction code or SAP navigation path, you can create service files. These service files contain information on
  - **The network:** "how can the user reach the R/3 application server?" and other information
  - **R/3 logon data:** The name of the R/3 System, client, user name, and logon language
  - **Transaction:** What is the code for the transaction that is to be opened in the R/3 System?
  - **Web Server data:** such as the time-out parameter
  - **Internal ITS parameters**
- You can create a service file in the R/3 System in the Web Application Builder. It then needs to be published on the ITS.
- Note: Web Application Builder in the ABAP Workbench is new in Release 4.6 C. It replaces the SAP@Web Studio. You can create a service file outside an R/3 System using the SAP@Web Studio. For more information, refer to the SAP Library under *Basis->Frontend Services->ITS/SAP@Web Studio.*




- One way of making a transaction available over the ITS directly is to create a service file, which contains the following information:
  - The transaction that is to be started
  - The system in which the transaction is to be started
  - The logon client, user, and password
  - The logon language that is to be used
- You can create a service for a transaction in the Object Navigator. Choose *Internet Services* and enter a name for your service. Comply with the customer namespace conventions. Internet services are often given the same name as the transaction associated with them. You could, however, use a different name, particularly if you are creating several services for the same transaction.
  - Choose *Enter* or the *Display* icon (a pair of glasses). If there is no service saved under the name you have entered, the system displays a dialog box and asks: Do you want to create the object? Choose *Yes.*
  - In the dialog box that appears, choose *Easy Web Transaction* and enter the transaction code. Choose *ITS mixed mode.* Choose *Enter* to confirm.

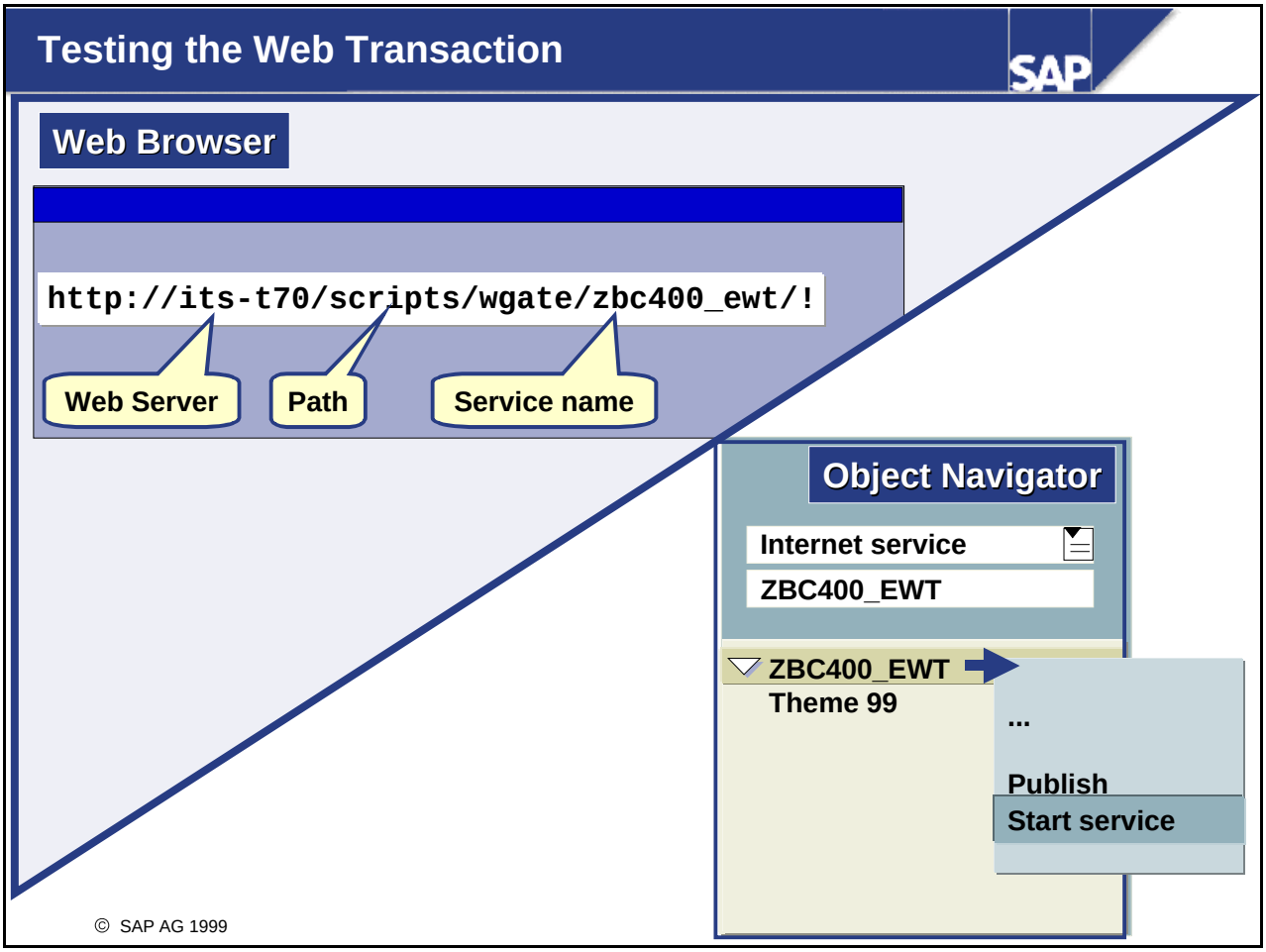
- Note: If you choose *ITS mixed mode,* you do not need to generate a static HTML page for each screen. If a screen is not associated with an HTML page, the SAPGUI for HTML generates the page dynamically at runtime.

# Publishing an Internet Service






- You must save the Internet service in a file that the ITS can access. You can publish the service in the Web Application Builder in the Object Navigator.
  - In the Object Navigator, open the object list for the Internet service.
  - In the service context menu, choose *Publish->Complete service*
- Note: Your user-specific ITS settings must be correct before you can publish a service. You can check these settings from the Object Navigator by choosing *Utilities->Settings* followed by the *ITS* tab.




- To test a transaction after publishing it:
  - Choose the *Start service* from the context menu (from the Internet service object list)
  - Alternatively, enter the path **http://<server>:<port>/<path>/wgate/<service_name>/!** in a Web Browser.

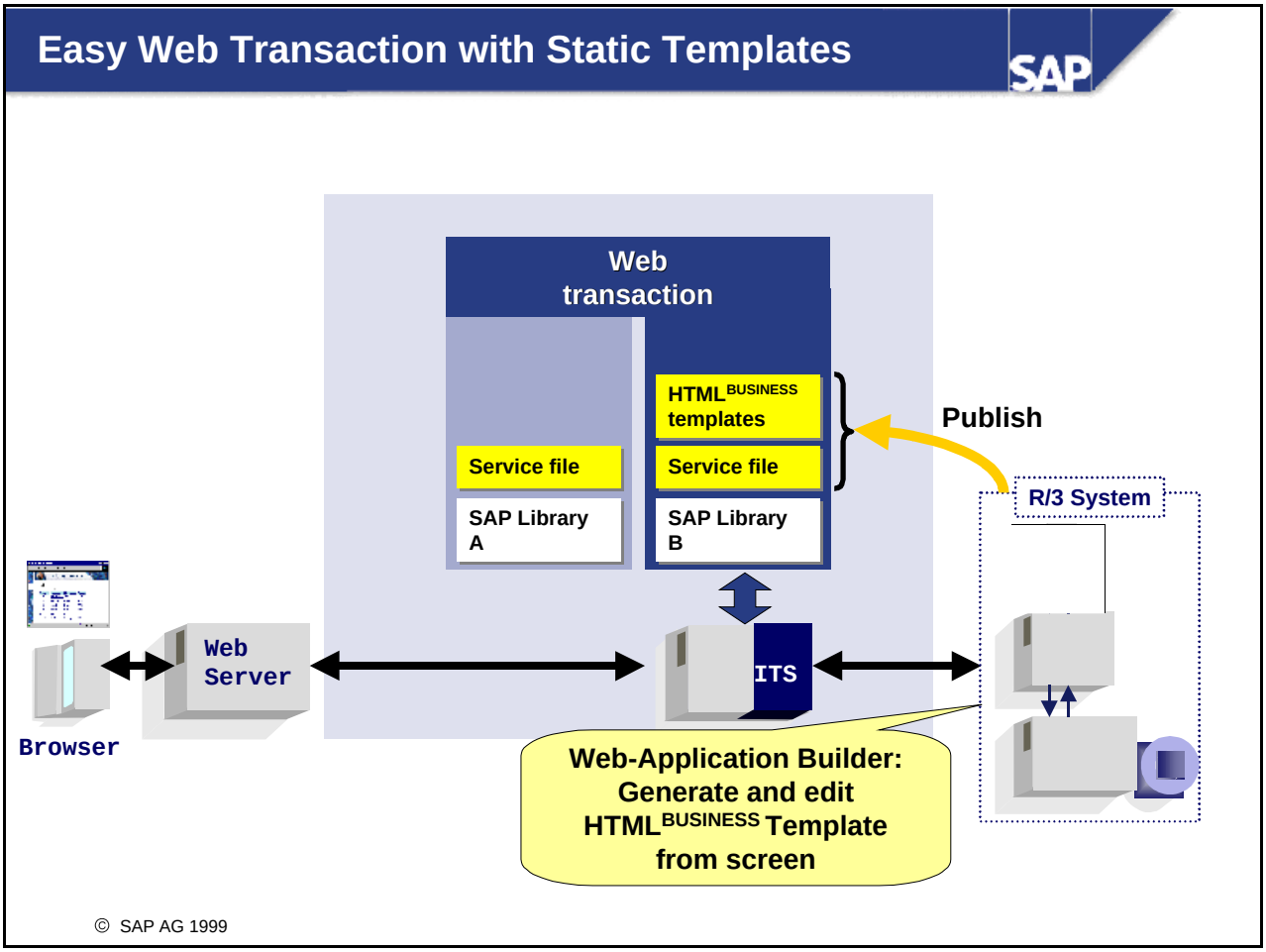
# Overview: Transactions with a Web Layout

SAPGUI for HTML

Easy Web Transaction

Transactions with a Web Layout

Outlook: ITS Flow Logic

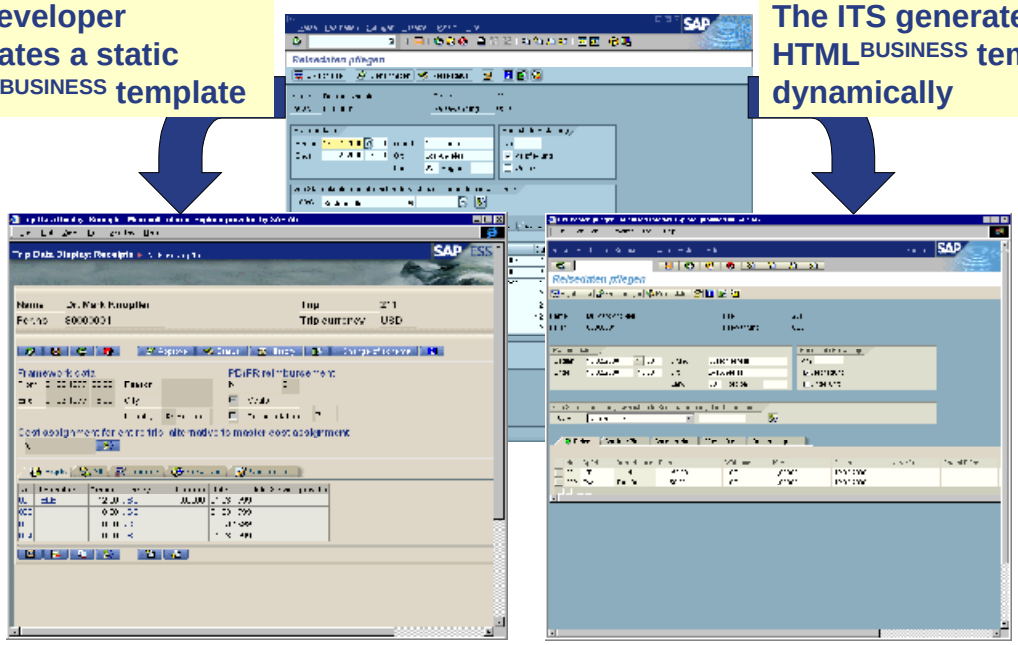



- You can generate static HTML templates for web transactions from SAP screens and then enhance them manually.
- Note: Navigation on the Internet differs from SAP System navigation. Users can navigate back to previous pages using the *Back* button. If they send a new query to the ITS, the screen displayed no longer matches the SAP transaction screen. The ITS notices this inconsistency and returns a catchable error message.

## HTML Pages for SAP Screens

**The developer generates a static HTML$^{BUSINESS}$ template**

**The ITS generates a HTML$^{BUSINESS}$ template dynamically**



**With HTML template:**
**Developer can change layout or add other HTML elements**

**or**

**Without HTML template:**
**Page generated automatically, cannot be edited afterwards**



- ***SAP GUI for HTML* generates HTML dynamically** from the relevant SAP screens:
  - Simple, SAP screen-based layout
    You can choose some graphic layout attributes using styles
    (such as font, font size, background color)
  - SAP (such as text or input fields) screen elements are mapped 1:1 to the fields on the generated HTML page
  - No other HTML elements are available
  - No further development is necessary
- **You can adapt the layout of your Web applications** using **static HTML$^{BUSINESS}$ templates**
  - Flexible layout: Using static HTML templates and MIME objects allows you to enhance the layout according to your own needs These techniques allow you to add more pushbuttons and pictures, even if there are no placeholders for them in the template.
  - Flexible field mapping:
    You can hide fields that contain default values
  - You can add additional functions

- However, you must carry out further development:
  You need to create, edit, and publish HTML templates. You also need to implement additional functions yourself.

# Reference Model

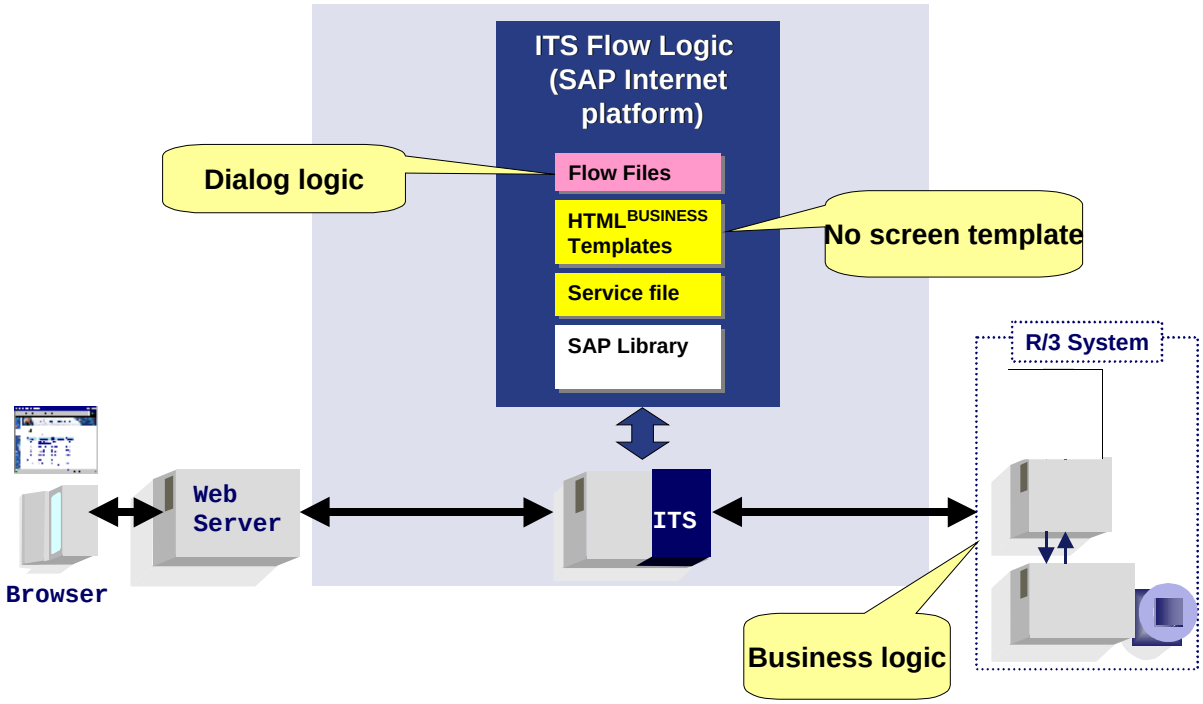
- Create a service for the transaction
- Generate an HTML$^{Business}$ template for each screen
- Edit HTML$^{Business}$ template using
  - HTML$^{Business}$
  - HTML$^{BUSINESS}$ functions
  - HTML
  - JavaScript

**The Web Application Builder in the Object Navigator offers all of the above functions**

# Overview: ITS Flow Logic

- SAPGUI for HTML
- Easy Web Transaction
- Outlook: SAP Transactions with a Web Layout
- **Outlook: ITS Flow Logic**

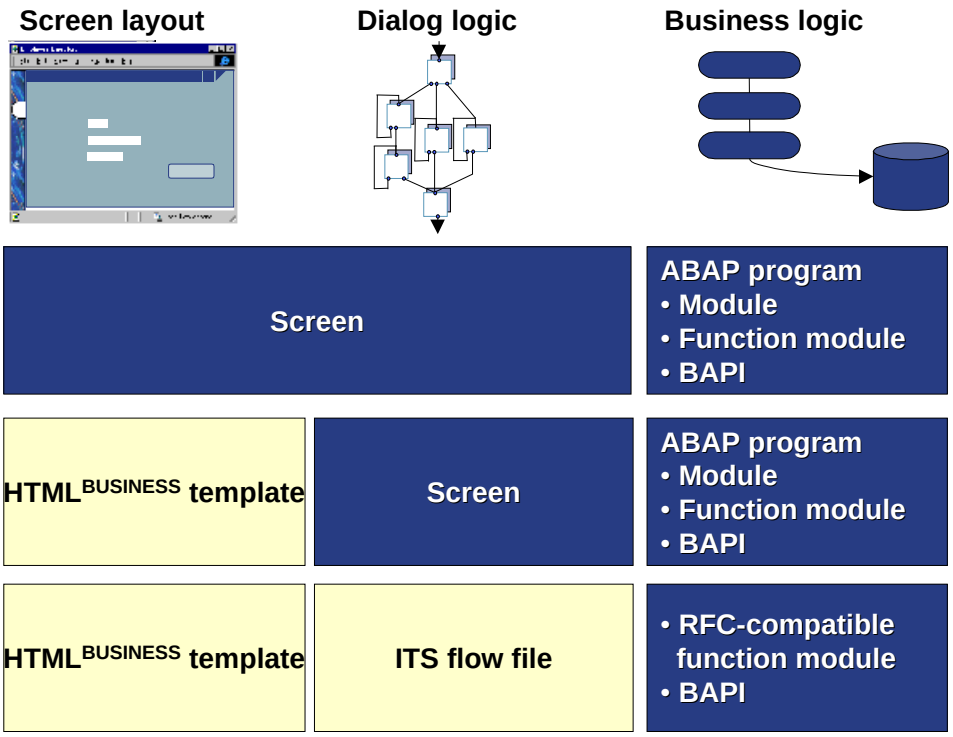
# ITS Flow Logic: Development Outside the R/3 System






- In the ITS Flow Logic programming model, the flow logic for the screens is processed on the ITS. The connection to the SAP System is used only for business requirements (such as retrieving data from or changing it in the R/3 System).
- Note: In this programming model, the entire user dialog logic is stored on the ITS, so there are no associated screens in the SAP System. This means that you **must** create templates manually - you cannot generate a raw version of the page from the R/3 screen.
- The ITS Flow Logic programming model uses both stateless and stateful queries to the ITS:
  - Stateful call: Uses the existing connection - that is, the user's logon. If necessary, the user has to log on again. After calling the page, users can log out themselves. If they do not, an automatic time out occurs.
  - Stateless call: uses a connection made solely for one query-answer cycle. After the RFC or BAPI has been called, the connection is closed.

## ITS Programming Models

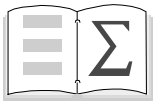





- Each business application consists of three logical levels:
  - The **business logic:** The system can then carry out the necessary authorization and consistency checks. After these checks have been made, the database changes can be triggered.
  - **Dialog logic**: Which user dialogs must be carried out, and when, to obtain the data you need to execute the business logic? This logic includes sending error messages, if an inconsistency has occurred. Some of these checks must be carried out in the dialog logic to prevent the user from entering a false value.
  - **Display:** How are the user dialogs stored - what background color is used, how does each element appear, and so on?
- The different programming models used to create Internet applications implement these different levels differently:
  - **SAPGUI for HTML:** An ABAP program with user dialogs is used as a basis for all three levels. The layout is generated automatically from the screen information. The appearance in the Web Browser contains exactly the same elements as the SAP System screen.
  - **Easy Web Transaction:** The business logic and the dialog logic are copied from the ABAP program. You can then change the appearance of the page.

- **ITS Flow Logic:** Only the pure business logic is executed in an ABAP program, which can be called externally - generally a BAPI or an RFC-compatible function module. The dialog logic and screen layout are defined on the ITS.

# Developing Internet Applications



**You are now able to:**

- **Create a transaction code for an Easy Web Transaction**
- **Create an Internet service**
- **Publish an Internet service**
- **Test an Easy Web Transaction using a Web Browser**

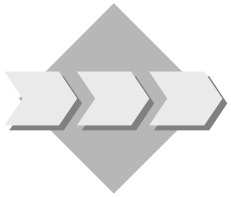
Exercises

**Unit: Developing Internet Applications**

**Topic: Creating and Publishing an Internet Service**

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

- Create an Internet service for an available transaction
- Publish this Internet Service on an ITS



Edit the program that displays a list using the ALV Grid Control, so that the screen displaying the data appears as soon as the user enters a transaction code You should make this transaction as an Internet service on the ITS.

1-1 Copy your program **ZBC400_##_ALV_GRID** or the template **SAPBC400IAT_EWT** and give it the name **ZBC400_##_EWT**. Edit **ZBC400_##_EWT** so that the user can display the screen with all the data from **SPFLI** by entering a transaction code.

1-1-1 The program accesses the database in the **START-OF-SELECTION** block. Change the program so that it accesses the database in a PBO module instead. You can then call the screen directly using a transaction code. Make sure that the database is accessed once only.

1-1-2 Create a transaction code with the name **ZBC400_##_EWT** for your program, **ZBC400_##_EWT**. As a start object, choose *Program and screen* (*dialog transaction*) Enter the program name **ZBC400_##_EWT** and screen *100*. Choose the transaction classification *Easy Web Transaction*. Save the transaction code. Assign the transaction code when saving your development class.

1-2 Create a Internet service with the name **ZBC400_##_SRV**.

1-2-1 In the Object Navigator, choose *Internet Services* and enter a name for your service. When you choose *Enter*, the system checks to see if there is an existing Internet service saved under the name you entered. If not, the *Create Internet Service* dialog box appears.

1-2-2 In the dialog box that appears, choose *Easy Web Transaction* and *ITS-Mixed-Mode* and enter the transaction code **SAPBC400IAS_EWT**.

1-3 Publish the service

1-3-1 Display the Internet service, **ZBC400_##_SRV**, in the Web Application Builder. Publish the entire service using the Internet service context menu.

1-4 Test your Internet application.

1-4-1 (Choose *Start service* from the context menu.

## Developing Internet Applications: Solutions

**Unit: Developing Internet Applications**

**Topic: Creating and Publishing an Internet Service**

1-1-1 You can program the database access in the **CREATE_CONTROL** module. The source code would then look like this:

```
MODULE create_control OUTPUT.
  IF container_r IS INITIAL.
* fill internal table
  SELECT * FROM spfli
         INTO TABLE gdt_spfli.
*        WHERE ...
    CREATE OBJECT container_r
         EXPORTING container_name = 'CONTAINER_1'.

    CREATE OBJECT grid_r
          EXPORTING  i_parent =  container_r.

    CALL METHOD    grid_r->set_table_for_first_display
       EXPORTING i_structure_name = 'SPFLI'
       CHANGING  it_outtab        = gdt_spfli.
  ELSE.
    CALL METHOD grid_r->refresh_table_display
       EXPORTING i_soft_refresh = 'X'.
  ENDIF.

ENDMODULE.                    " CREATE_CONTROL  OUTPUT
```

The implementation takes advantage of the fact that the Control is only created once. Use the **IF** statement is used to ensure that the database is only accessed once. Delete the **SELECT** statement from the **START-OF-SELECTION** event block.

**Alternatively**, create a new PBO module. You then need to:

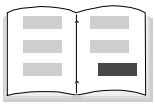
- Add a **MODULE** statement to the **PROCESS BEFORE OUTPUT** event in the screen flow logic.
- Create a screen using forward navigation
- In the module (provided the internal table is empty (**IF gdt_spfli IS INITIAL**)) program the database access and fill the internal table.

1-2 Follow the instructions in the *Creating an Internet Service* slide.

1-3 Follow the instructions in the *Publishing an Internet Service* slide.

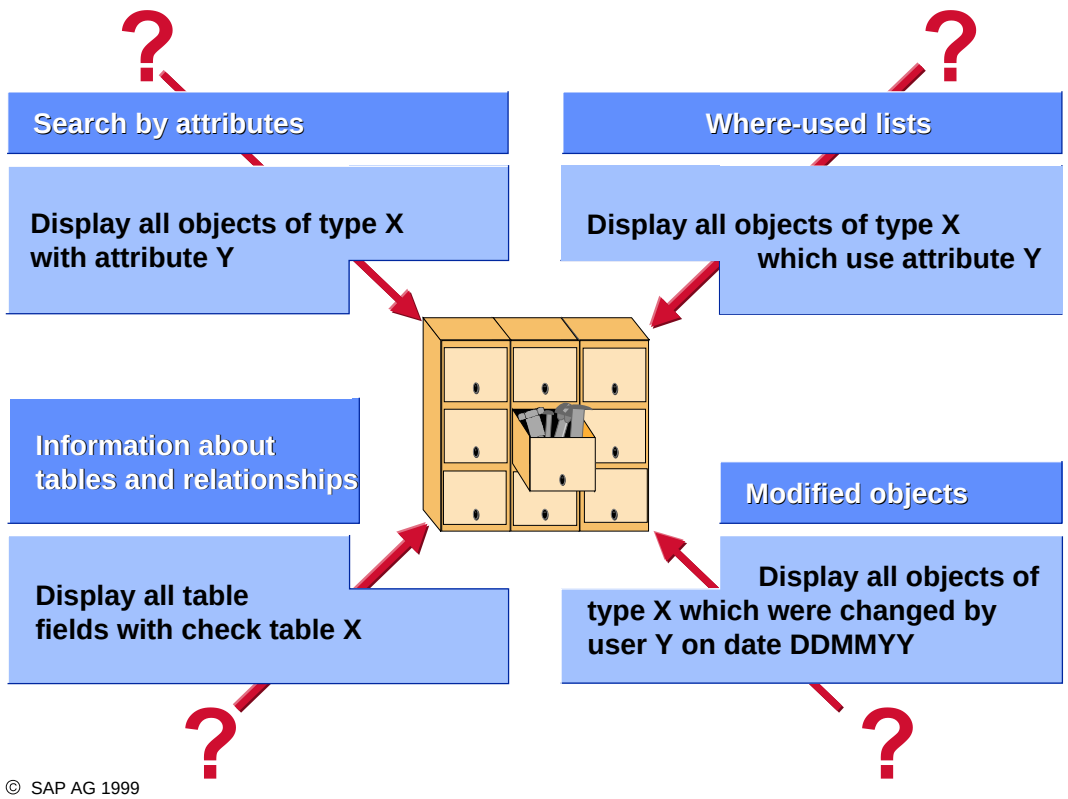
1-4 Follow the instructions in the *Testing the Web Transaction* slide.

# Appendix

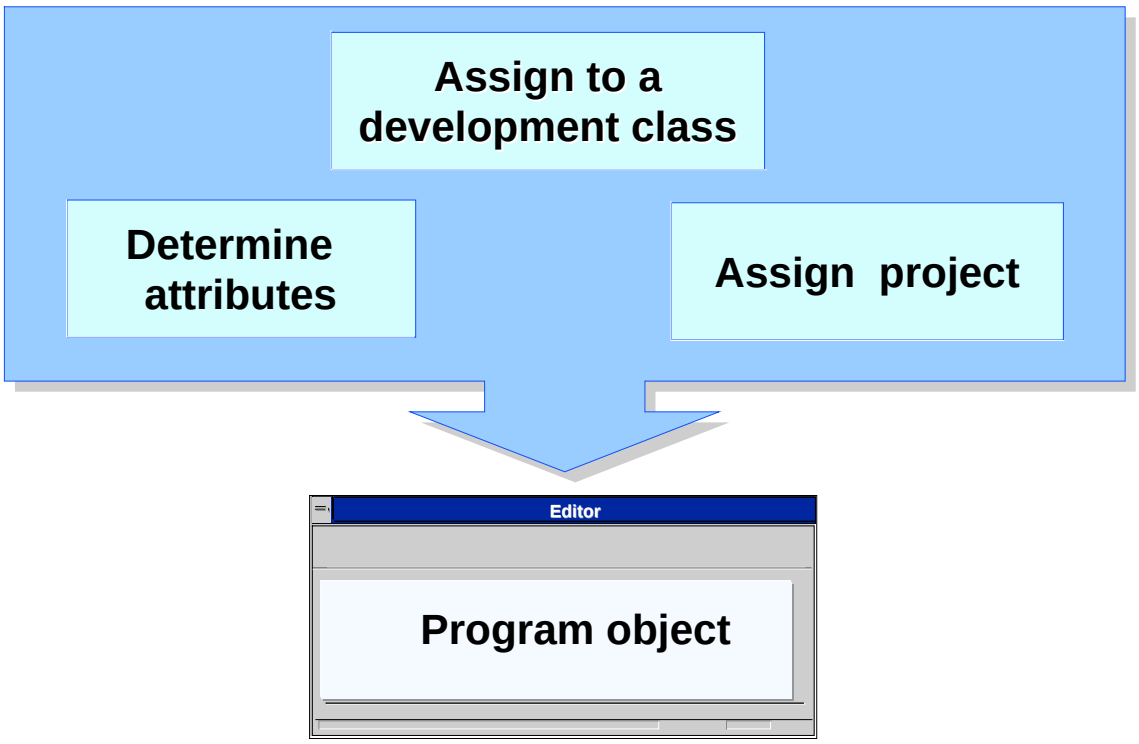


- This section contains supplementary material to be used for reference
- This material is not part of the standard course
- Therefore, the instructor might not cover this during the course presentation

# Typical Information System Requests



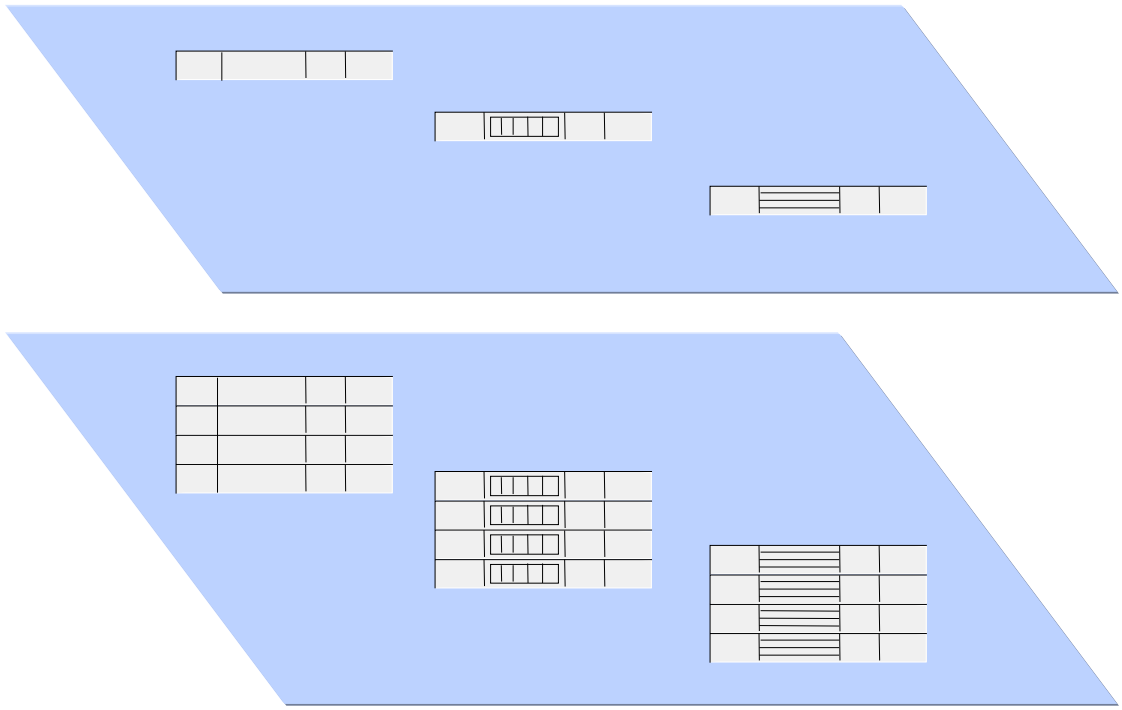
## Overview: Creating Programs






- When you create a new program you must first enter various administrative details.
- Programs must be assigned to a particular **development class.** . This classifies the program **logically**. Such designation only takes place once, when the repository object is initially created (development classes are in turn assigned to the component to which they logically belong).
- Additionally, several general **program attributes** must be determined. One important attribute is the **program type**. The program type determines how a program is executed in the R/3 client/server environment, for example if subsequent source text is an executable program or a reusable piece of code.
- Finally, a program must be assigned to a Workbench Organizer **project** (**change request**). This classifies the repository object **chronologically**. Once the current project is finished, a program can then be assigned to a new project.
- A program can only be edited after these initial designations have been made.

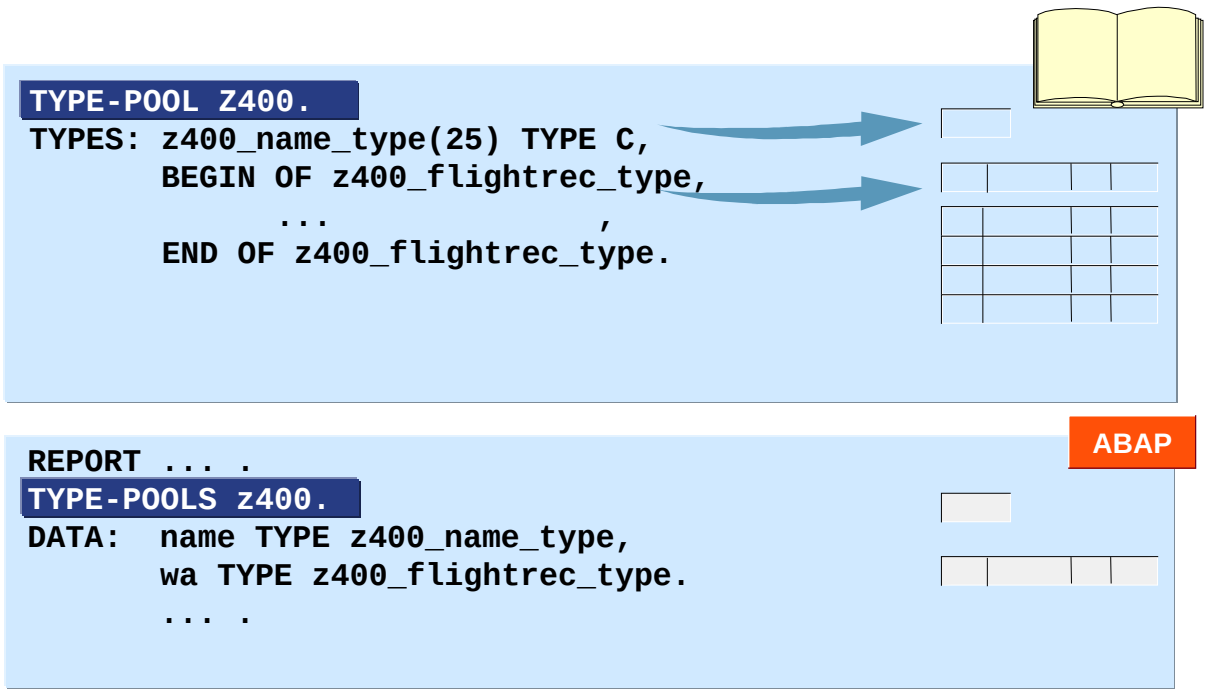
## Structures and internal Tables can be Nested





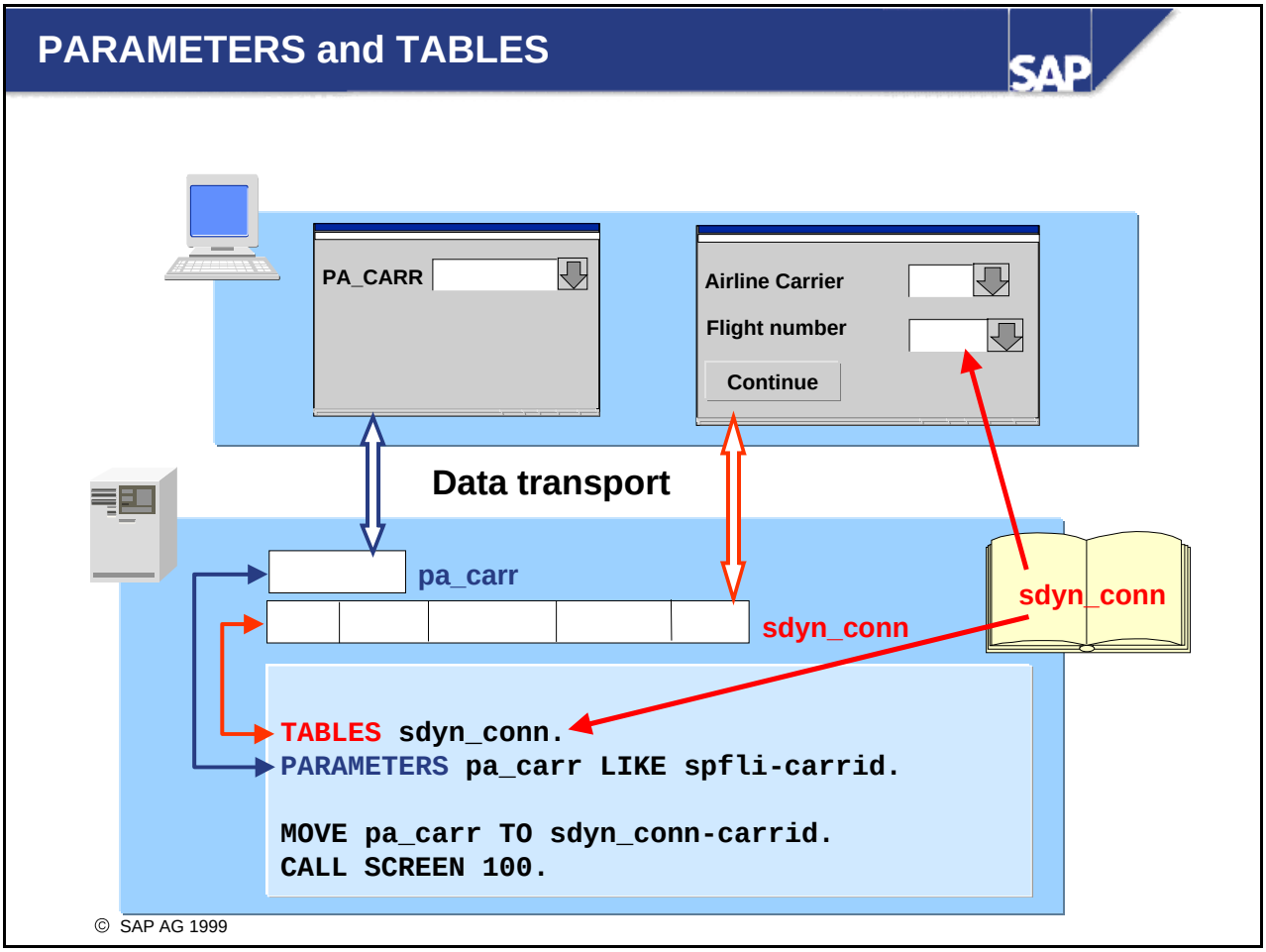
- Structure types can contain other structure types or table types.
- Table types can contain other table types or structure types.
- You can find out how to define table types in the keyword documentation under the key word TYPES.

## Type Groups in the ABAP Dictionary



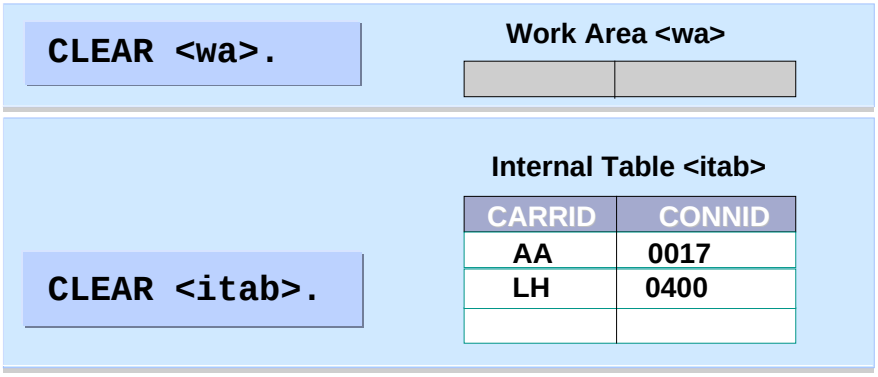



- You can define data types in a type group in the ABAP Dictionary instead of defining them within an ABAP program.
- The type group name in the ABAP Dictionary has a maximum of 5 characters. Type names within type group <typepool> must begin with <typepool> followed by an underscore.
- The types in a type group must be declared in ABAP programs with the TYPE-POOLS command.



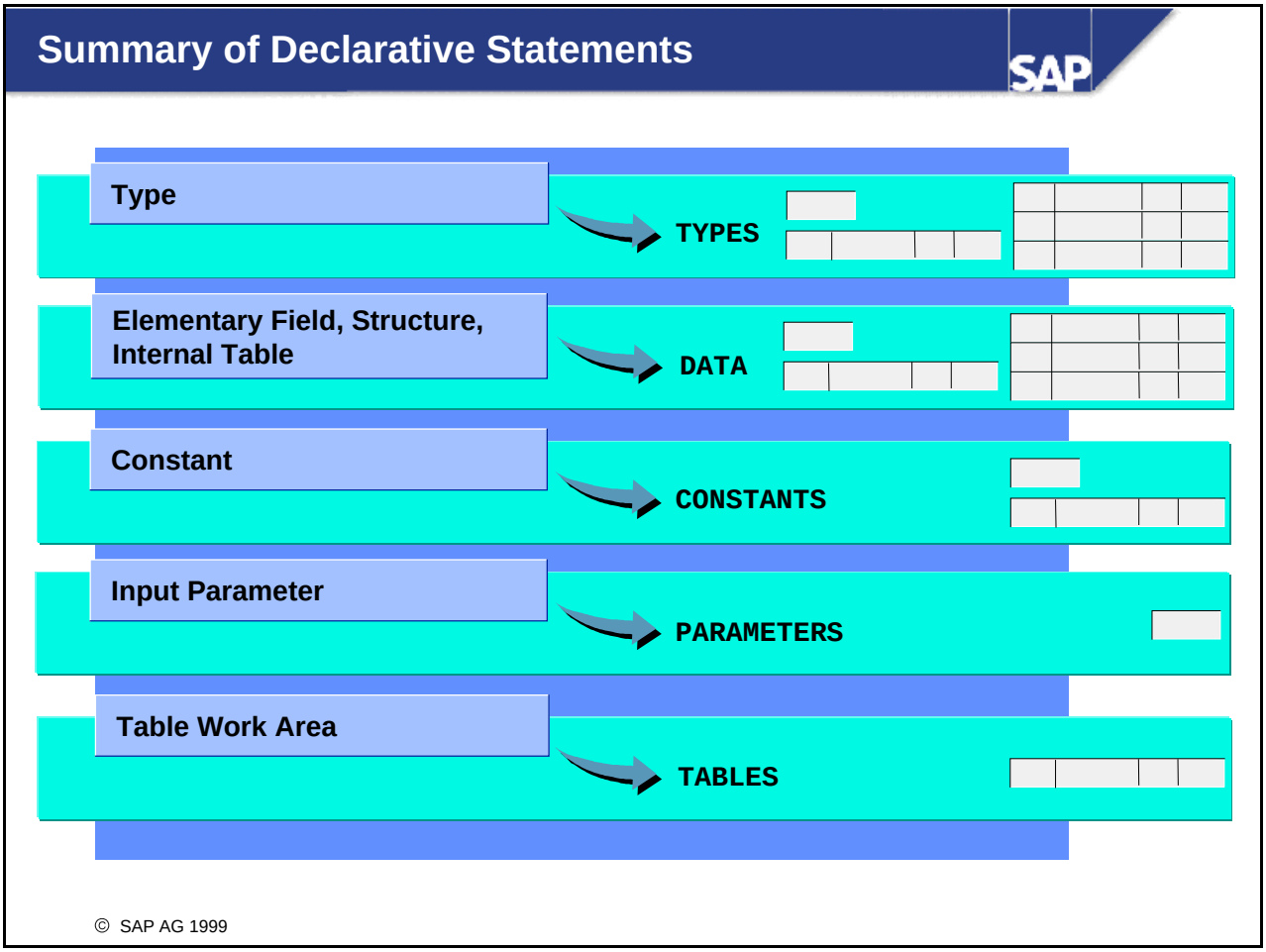

- The PARAMETERS statement is a declarative language element for establishing internal fields within the report. The difference between the PARAMETERS and DATA statements is that fields declared using PARAMETERS are presented for input on the selection screen.
- When you use the PARAMETERS statement, you can use the TYPE and LIKE additions as you would when using the DATA statement.
- Analogous to the way in which you can use VALUE with DATA to assign an initial value, you can use the addition DEFAULT with the PARAMETERS statement to set a default value for the field. This value can be a literal, a constant, or a system field which takes its value from the system when the report is processed (for example sy-datum).
- The TABLES statement declares an internal data object that serves as a screen interface whenever screen fields refer to the same Dictionary object.
- Use the TABLES statement to define an appropriate work area in your ABAP program for data that the user enters on a screen or that is passed to the screen from the program.

# Deleting an Internal Table






- Use CLEAR to reset the contents of a data object to the initial value **for its type**.
- Since internal table entries are always of a single type, only one CLEAR statement is necessary to delete the entire table.
- CLEAR <wa> initializes work area <wa>.




- We have encountered the following declarative statements so far:

| | |
|---|---|
| TYPES | Definition of types |
| DATA | Definition of elementary fields, structures, and internal tables |
| CONSTANTS | Definition of constants |
| PARAMETERS | Definition of input parameters |
| TABLES | Definition of table work areas |

- In the unit *Internal Tables* as well as in the sub-unit on *Selection Screens* in the unit on *Dialogs* you will learn about the following declarative statement:

| | |
|---|---|
| SELECT-OPTIONS | Definition of selection possibilities |

- You can display a comprehensive overview of the declarative statements in ABAP by pressing the Editor pushbutton 'i' and then choosing *ABAP Overview ->Overview of the ABAP programming language -> Classification of keywords by type*.

## Type Conversion

**Compatible types can be assigned without conversion**

**Two elementary types are compatible if they are identical in type and length (and decimal places) , in the case of type P).**

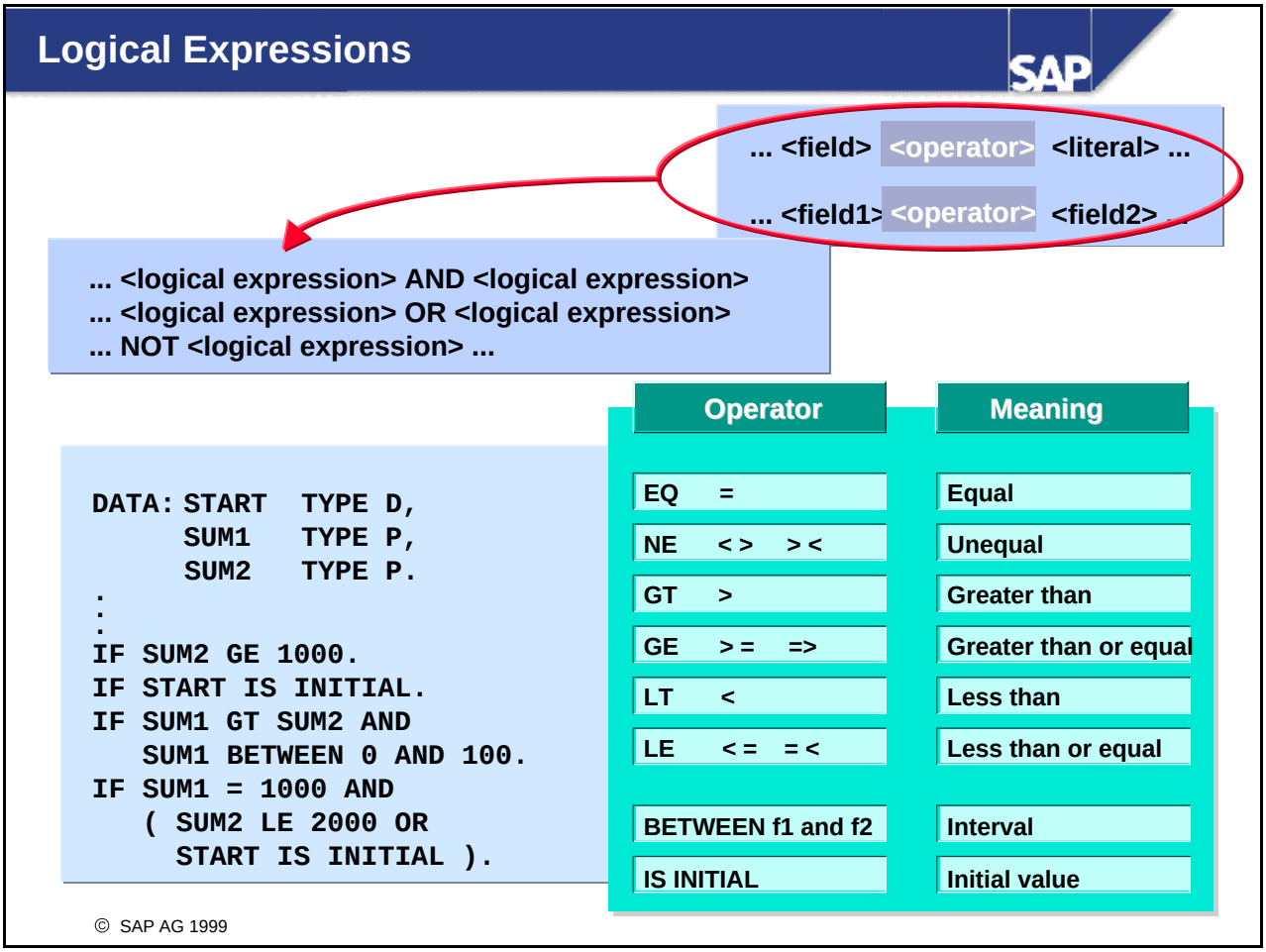
**Two structure types are compatible if they have the same structure and their components are compatible.**

**Two tables are compatible if their line types are compatible.**

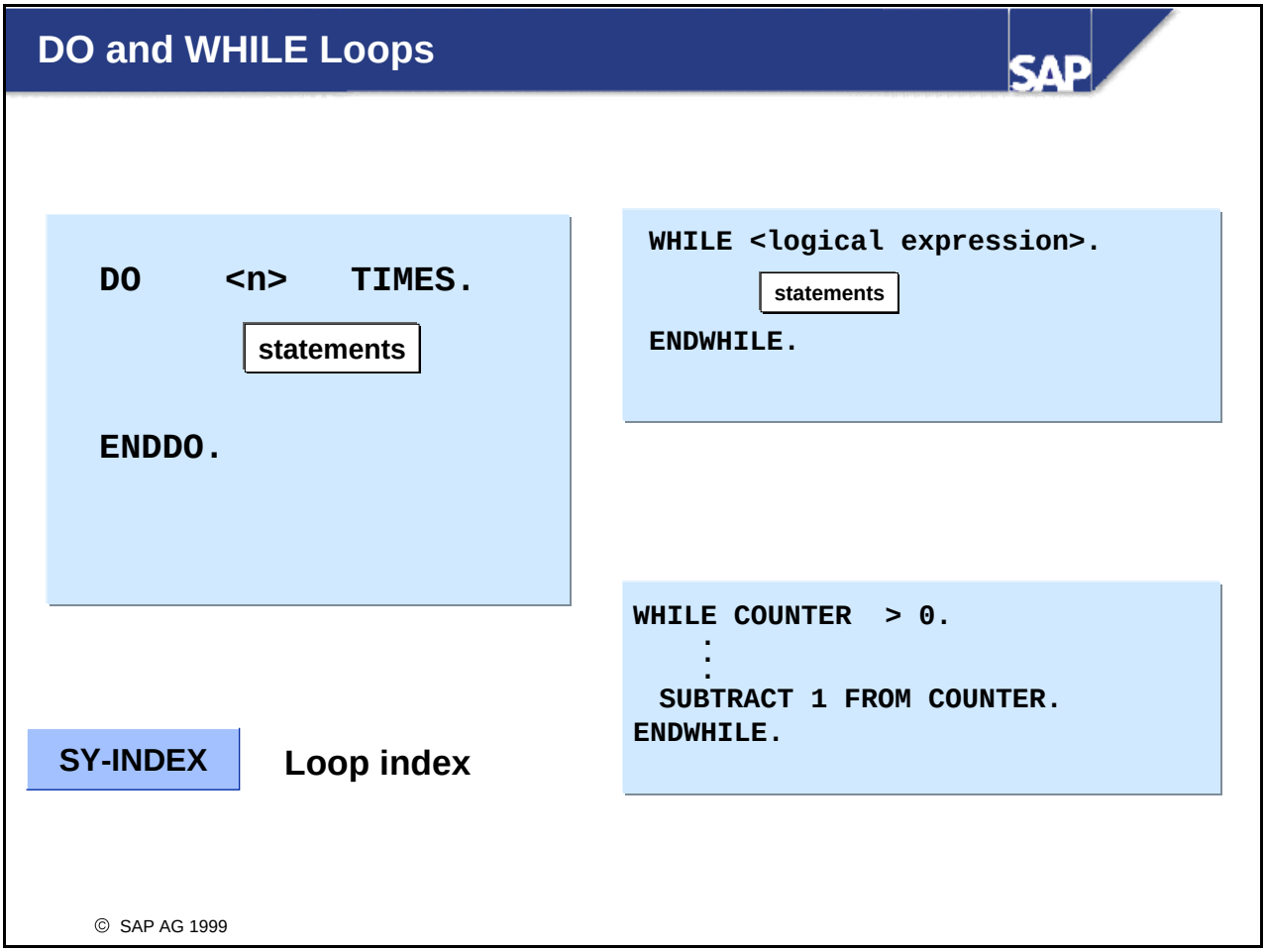
**Non-compatible types can be converted if a conversion rule has been defined**



- If you assign data objects of different types to one another, ABAP carries out a type conversion as long as a conversion rule has been defined for the types concerned.

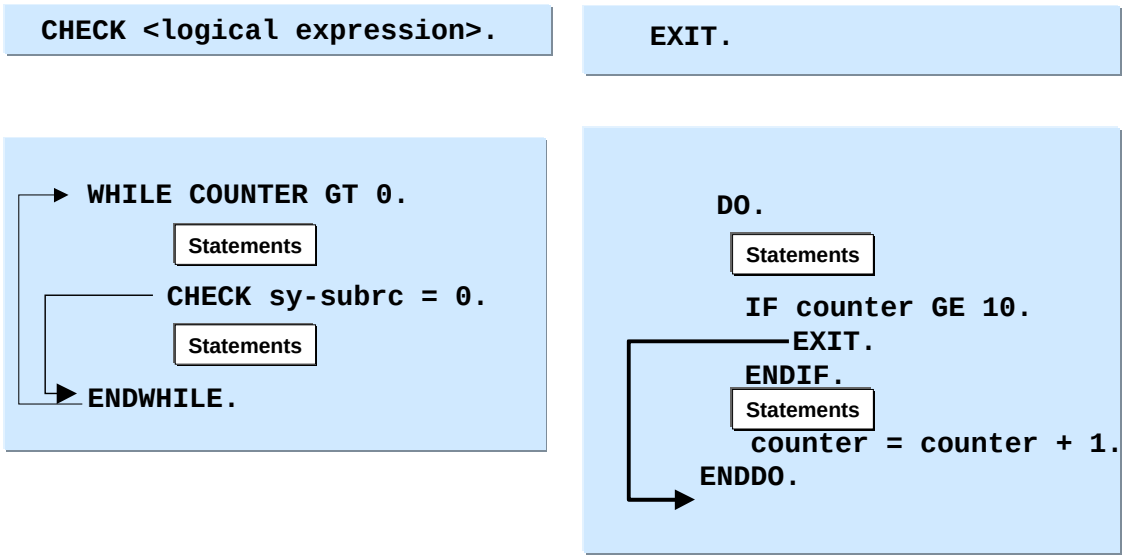



- Logical expressions can be linked with NOT, AND, and OR.
- You can nest parenthetical expressions as deeply as you want. The parentheses which denote sub-expressions always count as one word. They must therefore be separated by spaces.
- If you compare two type C fields with unequal length, the shorter field is lengthened to match the length of the longer one when the comparison is made. It is filled from the right-hand end with spaces.
- There is a whole range of further comparative operators which you can use to compare strings and bit comparisons. (See the online documentation for the IF statement).




- DO and WHILE loops are also used in ABAP.
- SY-INDEX is the loop counter for the loop commands DO and WHILE. SY-INDEX has the value 1 during the first loop pass and is increased by 1 by the system for each loop pass.
- The following is true for DO loops:
- The <n> TIMES parameter is optional. If you do not specify it, you need to build a termination condition into the loop (see EXIT statement).
  - The number of loop passes cannot be altered via the sy-index field or the loop counter within DO ... ENDDO.
- The following applies to WHILE loops:
- Provided the logical expression is fulfilled, the sequence of statements is executed.
  - The number of loop passes cannot be altered via the sy-index field within the WHILE... ENDWHILE.

## CHECK and EXIT






- Two statements are available for loop processing:
- CHECK <logical expression> : If the logical expression has not been fulfilled, the system jumps to the next loop pass. All statements between CHECK and ENDDO|ENDWHILE are ignored. If the logical expression has been fulfilled, the CHECK statement has no effect.
- See also: Keyword documentation for CONTINUE.
- EXIT statements within a loop structure cause the system to leave the current loop.
- For information about how these two commands work outside of loop processing, refer to the keyword documentation for CHECK and EXIT, or the appendix.

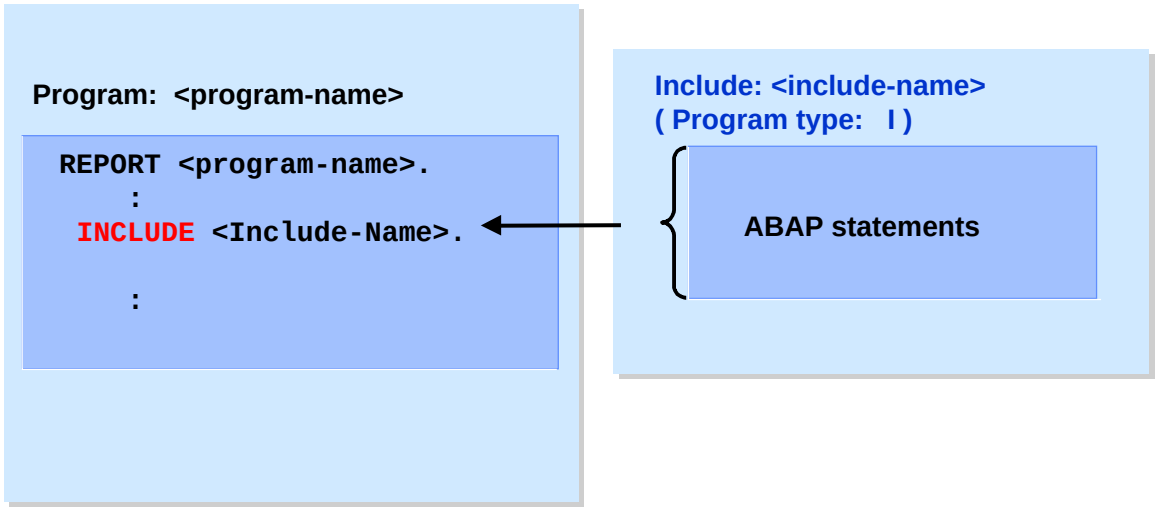
## Termination Conditions 2

| | EXIT | CHECK: If logical condition not met, then... |
|---|---|---|
| **Loops:**<br>`WHILE, DO,`<br>`SELECT, LOOP` | Exit current loop | Jump to next loop pass |
| **Events:**<br>`START-OF-SELECTION`<br>`GET`<br>`END-OF-SELECTION` | End of program, list displayed | |
| **Events:**<br>`INITIALIZATION`<br>`AT SELECTION-SCREEN ...`<br>`...` | Jump to end of processing block | |
| **FORM routines** | | |



- Use the ABAP statement CHECK <logical condition> outside of loops to end a processing block prematurely whenever the logical condition following the keyword is not fulfilled.
- EXIT outside of a loop also ends the current processing block. There are several events that are an exception to this rule. An EXIT statement in their event blocks leads to program termination. In this case a list is displayed immediately after the EXIT statement has been processed.

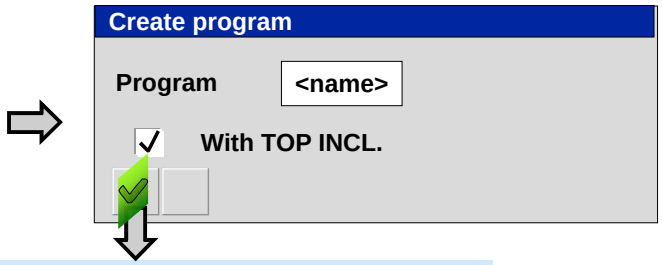
# Includes: Type I Programs



# TOP Includes

Repository Browser:

Create program




**Program: <name>TOP**

**Program type: I**

```
REPORT <name>.

 TYPES:...
 DATA: ...
```

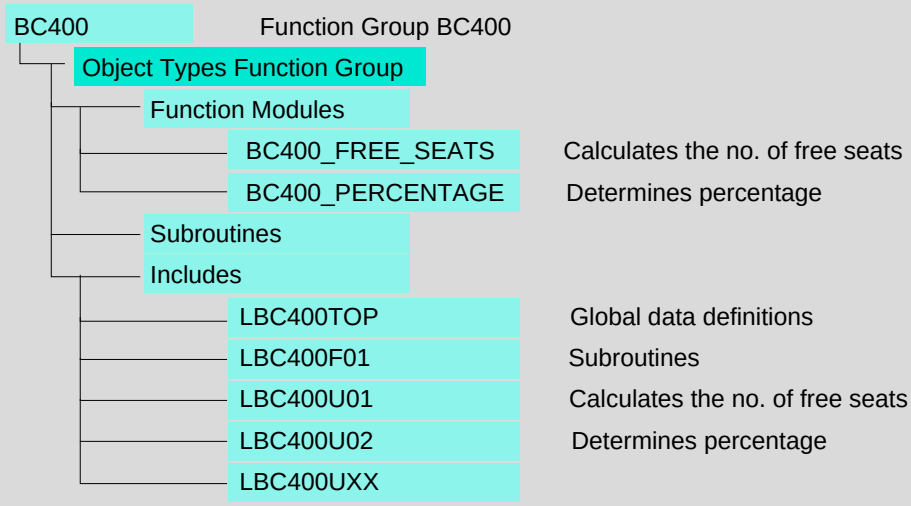
**Program: <name>**

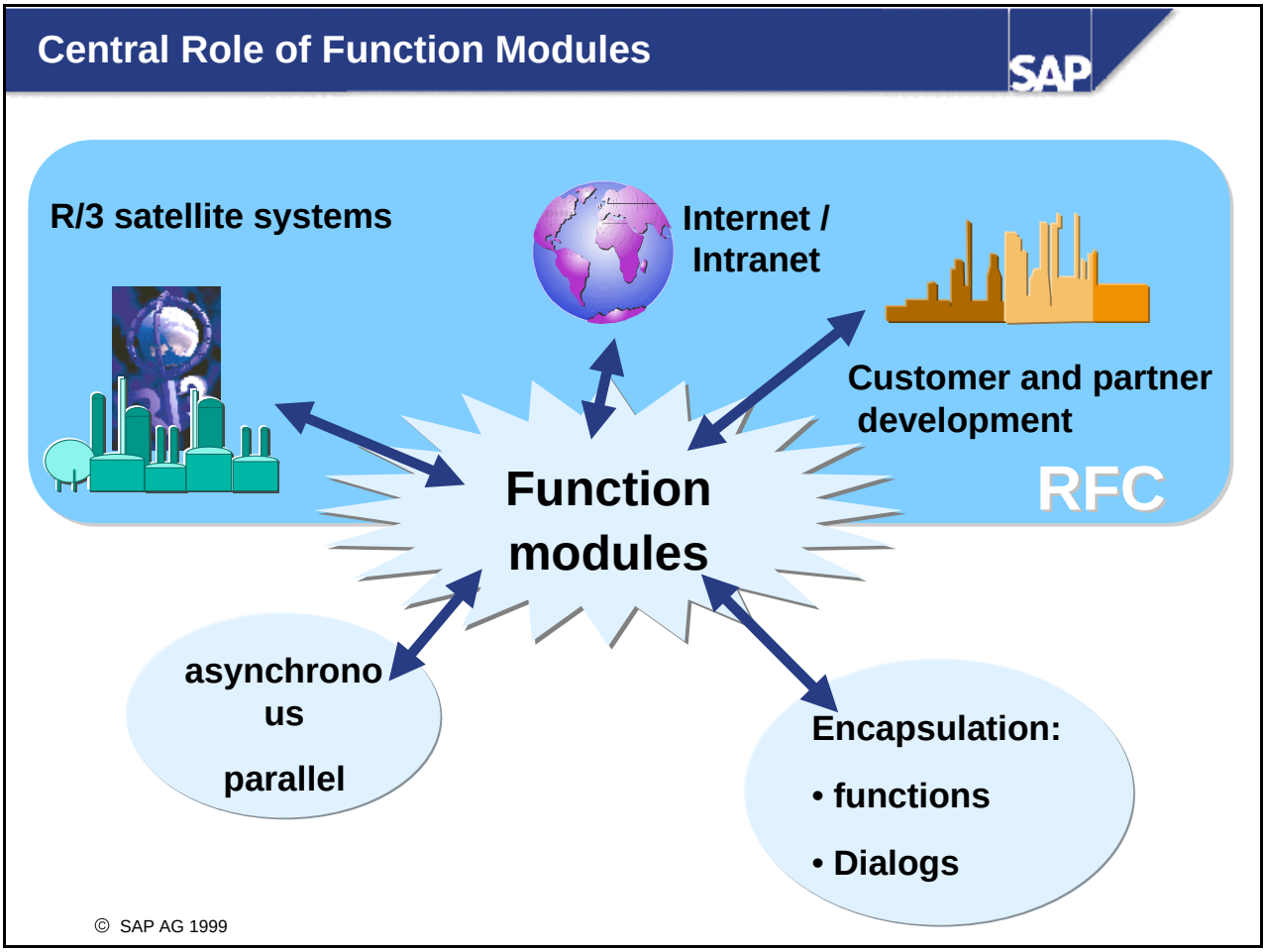
**Program type: 1**

```
 INCLUDE <name>TOP.

START-OF-SELECTION.
    :
```

# Standard Includes for Function Groups

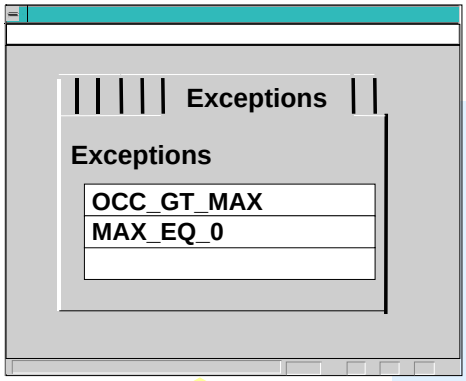
**Function group**






- You can start function modules either asynchronously or parallel.
- You can also encapsulate user dialogs.
- You can create function modules that can be started using Remote Function Call. These can then be started externally:
- From the World Wide Web, to allow you to access an R/3 System
- From another R/3 System
- From your own programs (for example, in Visual Basic, JAVA or C++).

# Exception Handling




Define exception

Raise exception

```
FUNCTION bc400_free_seats.

                .
                .
                .


  IF seatsmax = 0.
    RAISE max_eq_0.
  ELSEIF seatsocc > seatsmax.
    RAISE occ_gt_max.
  ENDIF.
                .
                .
                .

ENDFUNCTION.
```



- You can anticipate possible errors and write your program so that these errors do not cause runtime errors. You give these possible exceptions names in the function module interface, and trigger them from the program code using the RAISE statement. Each program that calls the function module can then interpret the exceptions by querying sy-subrc.
- In the function module, you can also ensure that the system displays an error message if the exception occurs, even if it is not explicitly handled by the calling program. For further information, see the key word documentation for MESSAGE ... RAISING.

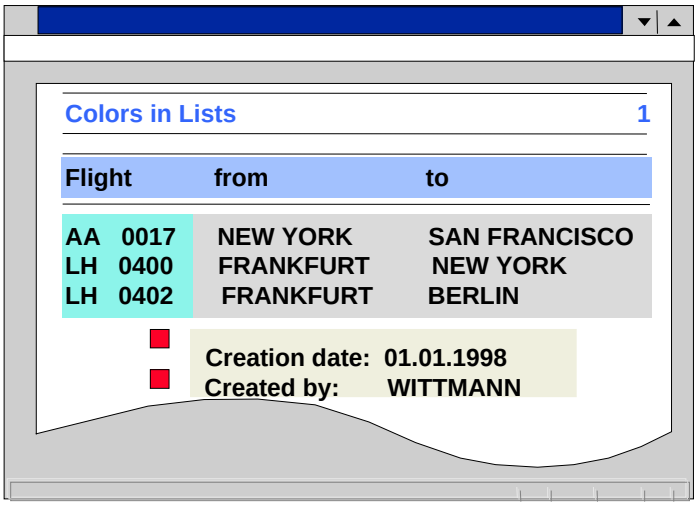
# Catching Exceptions

```
REPORT.
DATA:       free_seats LIKE sflight-seatsmax.
PARAMETERS: pa_occ LIKE sflight-seatsocc,
            pa_max LIKE sflight-seatsmax.
START-OF-SELECTION.
  CALL FUNCTION 'BC400_FREE_SEATS'
    EXPORTING
       seatsmax   = pa_max
       seatsocc   = pa_occ
    IMPORTING
       seatsfree  = free_seats
    EXCEPTIONS
       occ_gt_max       = 1
       max_eq_0         = 2
       others           = 3.
  CASE sy-subrc.
    WHEN 1.
       WRITE text-ex1.
    WHEN 2.
       WRITE text-ex2.
    WHEN 3.
       WRITE text-oth.
  ENDCASE.
```

# Colors/Icons/Symbols in Lists

**WRITE <data object> <option> .**

```
REPORT sapbc400udd_example_1a.
INCLUDE <LIST>.
 :
WRITE: / wa_spfli-carrid COLOR col_key,
         icon_date AS ICON,
```



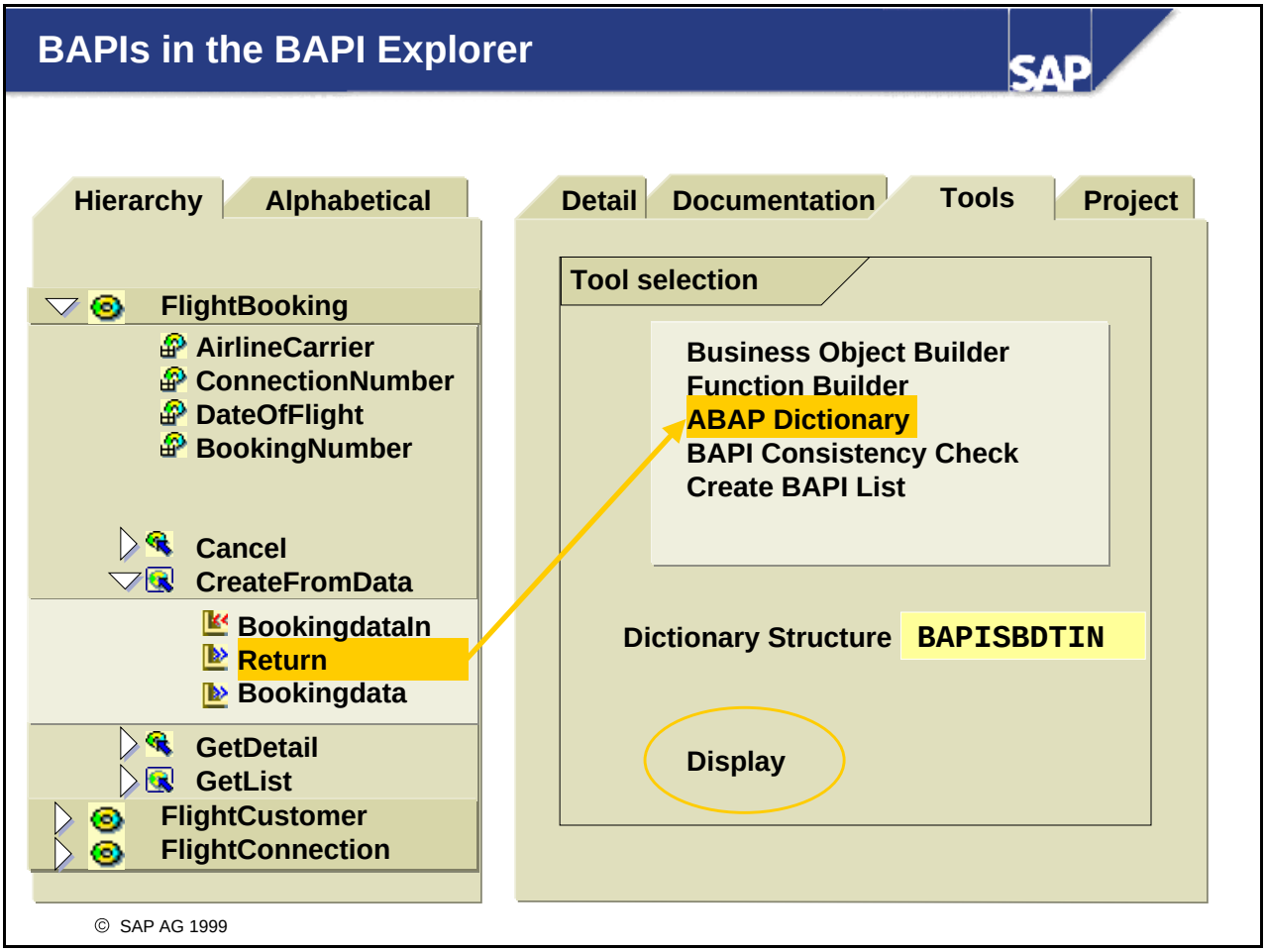



- You can set several list display attributes within a WRITE statement. One such attribute is **color**, which can be adjusted using the formatting option COLOR <n>. You can choose from seven background colors that are activated by either a numeric value or a symbolic name based on where they appear on a normal list.

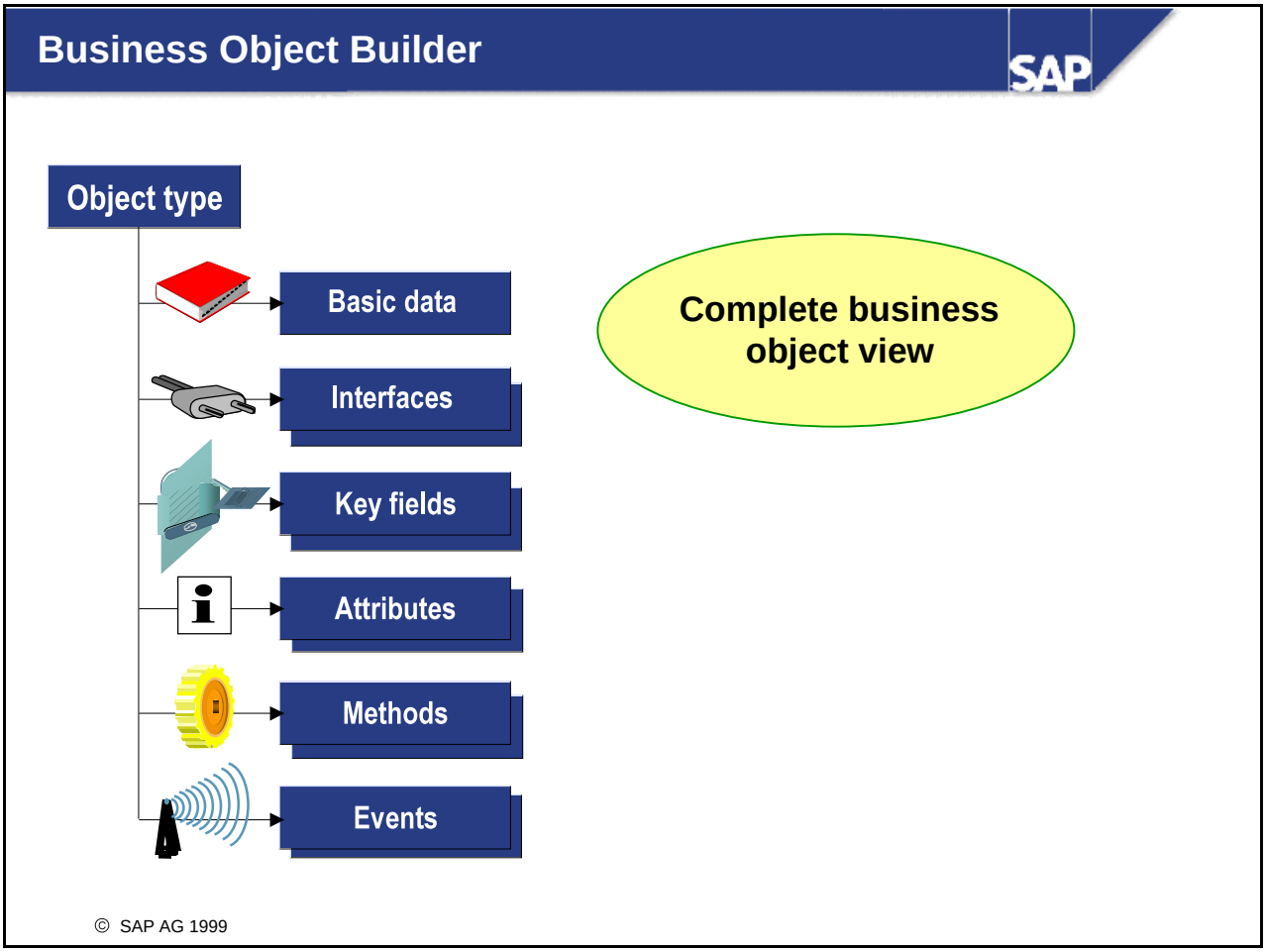
| | | |
|---|---|---|
| 0 | col_background | Background |
| 1 | col_heading | Headers |
| 2 | col_normal | List entries |
| 3 | col_total | Totals |
| 4 | col_key | Key columns |
| 5 | col_positive | Positive threshold values |
| 6 | col_negative | Negative threshold values |
| 7 | col_group | Control levels |

- SAP provides guidelines for creating lists in color. Please consult transaction LIBS for examples.
- The addition AS ICON within a WRITE statement allows you to add icons to your list. In order to be able to use AS ICON you must, however, add the include <LIST> to your program.

- You can find an overview of all available icons in either the keyword documentation under WRITE, or in the WRITE statement structure.




- If you expand a substructure for a method, the system returns the names of its import and export parameters. You can obtain more detailed information on the typing of interface parameters by choosing the *Tools* tab, then choosing the ABAP Dictionary. BAPI interface parameters are always typed using ABAP Dictionary types.
- BAPIs usually have an export parameter called RETURN. This can be a structure or internal table. The Return Parameter contains information on errors that occurred while the BAPI was being processed. There are no exceptions for BAPIs.



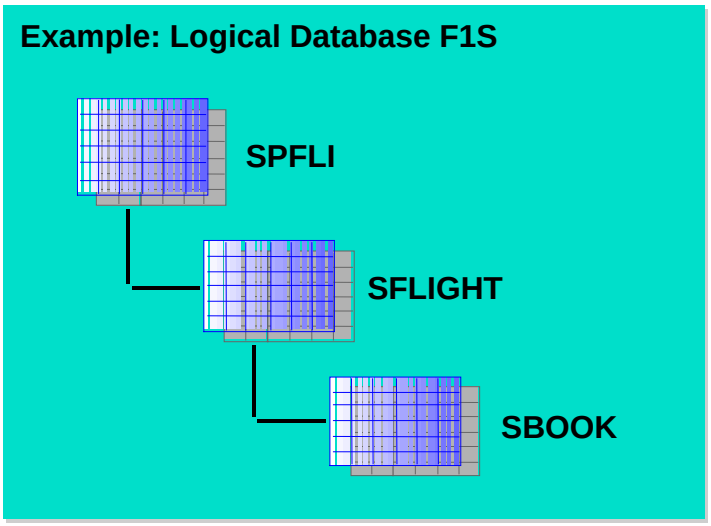

- To display complete information on a business object type, use the Business Object Builder tool. The system displays a tree structure for the business object type, including non-API methods.
- To search for a business object, use the Business Object Repository (BOR) tool. This tool displays the component hierarchy with all the business objects that belong to it. You can navigate from this tree structure to the Business Object Builder. The system displays the relevant business object automatically.

# Logical Databases Course Objectives

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

- **Describe the runtime behaviour of an executable program that uses a simple logical database, including the effects of:**
  - **The structure of the logical database**
  - **The NODES statement(s)**
  - **The GET event blocks**

# Reading Logically Dependent Data






- Use logical databases to read logically consistent data from databases. Each logical database has a structure containing a hierarchy of those tables and views that are to be read.
- You can attach exactly **one** logical database to each type 1 program. The logical database then supplies your program with entries from tables and views. This means that you only need to program the data processing statements.

## Logical Databases

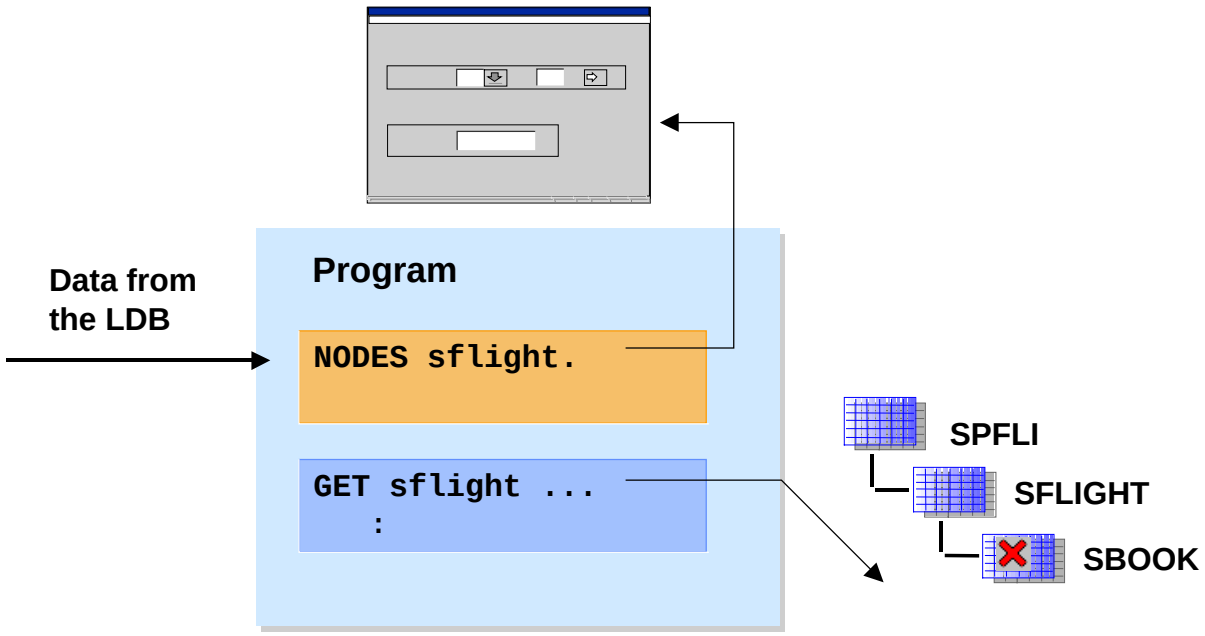
- **Are special data collection programs data delivered by SAP**
- **Provide your program with data in a hierarchically logical sequence**
- **Contain data base accesses that have been optimized for performance**
- **Supply a dynamic selection screen**
- **Contain all necessary authorization checks**

**You can attach a logical database to each type 1 program using the program attributes.**
**Special event blocks are also available for processing individual records.**



- Every logical database is an encapsulated data collection program for frequent database access.
- The database access has been optimized using Open SQL.
- If you are working with a logical database, you do not need to program a selection screen for user entry, since this is created automatically.
- The system performs authorization checks according to the SAP authorization concept.

## Controlling an LDB from within a Program



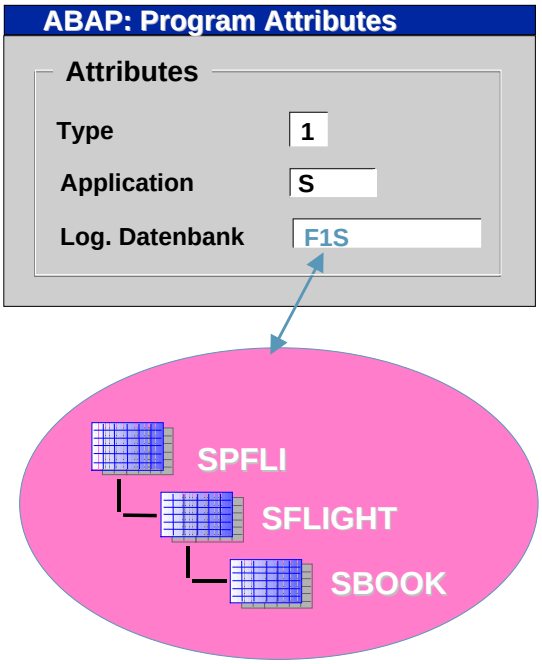



- The **NODES <node>** statement performs two functions:
  - It **defines a data object** (a structure) as a table work area that has the same structure as the ABAP Dictionary Structure <node>, that is a node of the hierarchical structure of the logical database. This structure is then filled at runtime with data records that the logical database has read from the database and made available to the program.
  - It determines **how detailed the selection screen is**: The selection screen that has been defined in the logical database should contain only those key information input fields that the program needs. The **NODES** statement allows you to ensure only information from relevant tables is available to the selection screen.
- Logical databases read according to their structure from top to bottom. The depth of data read depends on a program's **GET** statements. The level is determined by the deepest **GET** statement (from the logical database's structural view).

# Logical Databases

- Function Groups and Function Modules
- Objects and Methods
- Business Objects and BAPIs
- Logical Databases

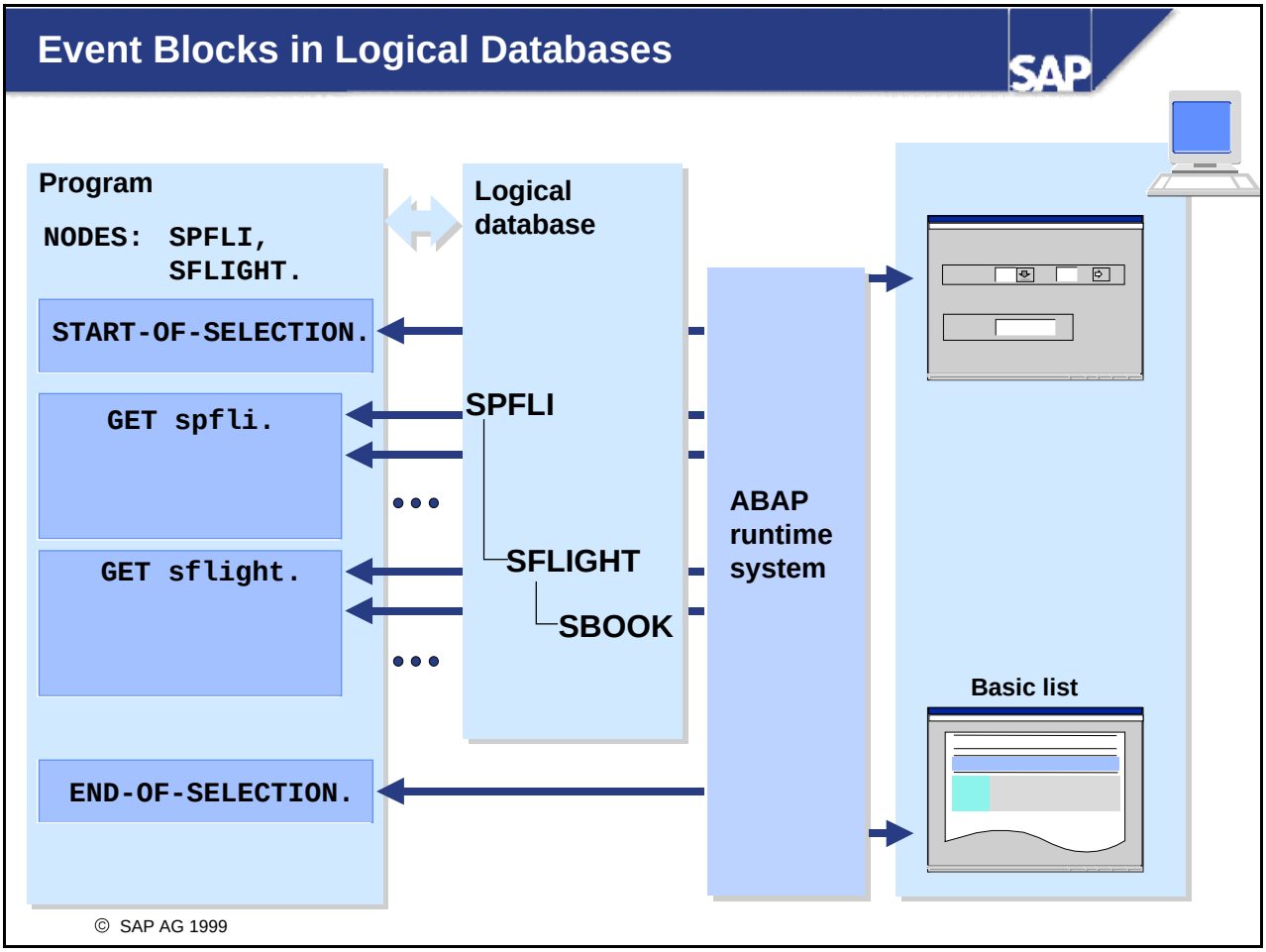
## Summary




```
NODES :spfli, sflight.

START-OF-SELECTION.
  WRITE: / 'START-OF-SELECTION'
             color 3.

GET spfli FIELDS carrid connid.
  WRITE: / 'GET SPFLI' color 1,
           spfli-carrid,
           spfli-connid.

GET sflight FIELDS fldate.
  WRITE: / 'GET SFLIGHT' color 2,
           sflight-fldate.

GET spfli LATE.
  WRITE :/ 'GET SPFLI LATE'.

END-OF-SELECTION.
  WRITE: 'END-OF-SELECTION'
             color 3.
```



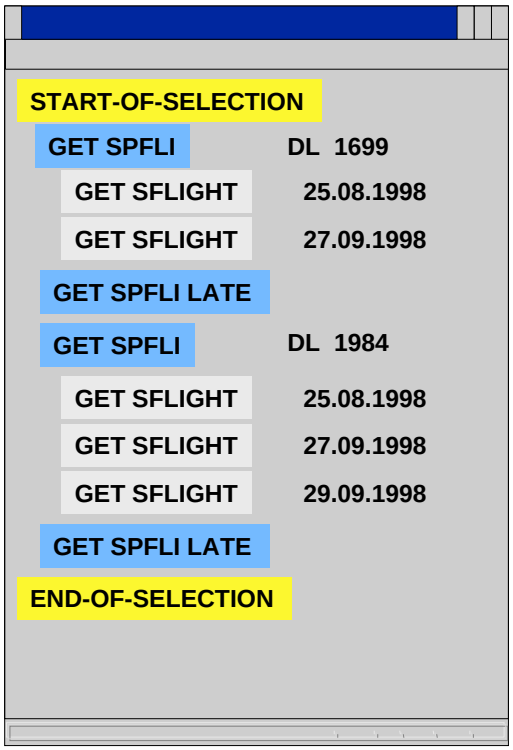
- Logical databases are included in type 1 programs as program attributes. Only one logical database can be attached per program.
- You can tell a logical database exactly which fields you need from the database using the **GET** addition **FIELDS**. If the logical database supports this action, then it will read only those fields specified from the database.
- If you need database table data for a list that is not supplied by your logical database, you can program any additional database access needed using **SELECT**.




- You can include a logical database in every type 1 program using the program attributes.
- Each node in the logical database's hierarchy also provides you with a **GET** event block (in addition to the other event blocks). (**GET SPFLI**, **GET SFLIGHT**, **GET SBOOK** in the example above).
- You can program individual record processing within these **GET** event blocks.
- At runtime the event blocks that create lists are processed, in the following order:
  - **START-OF SELECTION**.
  - **GET SPFLI** and **GET SFLIGHT** are called several times in nested **SELECT** logic according to the structure of the logical database.
  - **END-OF-SELECTION** is called after all **GET** events, and immediately before the list is sent to the presentation server.

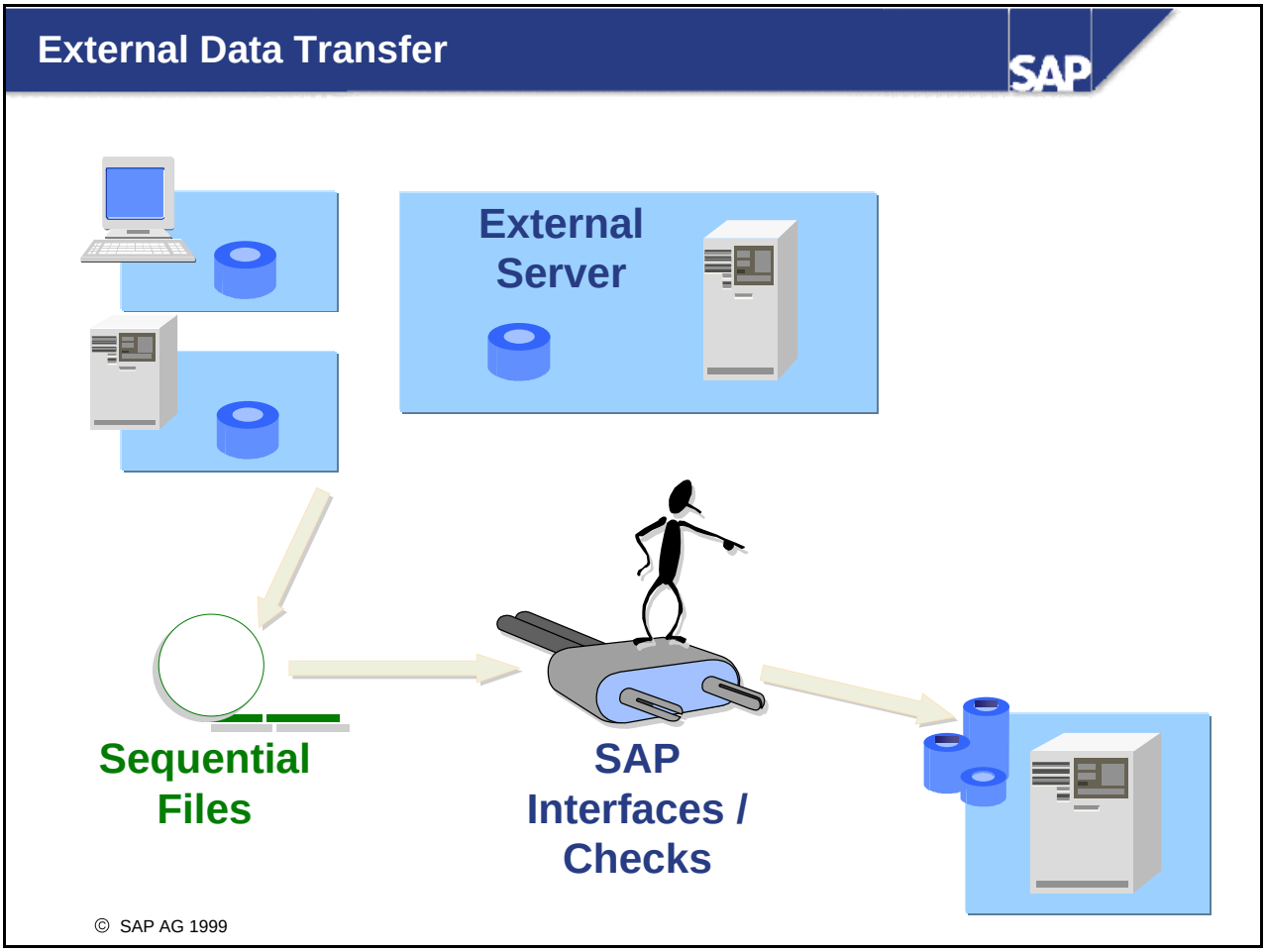
## Example: Event Sequencing

```
REPORT bc400d_logical_database.
NODES: spfli, sflight.

START-OF-SELECTION.
  WRITE: / 'START-OF-SELECTION'
             color 3.

GET spfli FIELDS carrid connid.
  WRITE: / 'GET SPFLI' color 1,
           spfli-carrid,
           spfli-connid.

GET sflight FIELDS fldate.
  WRITE: / 'GET SFLIGHT' color 2,
           sflight-fldate.

GET spfli LATE.
  WRITE :/ 'GET SPFLI LATE'.

END-OF-SELECTION.
  WRITE: 'END-OF-SELECTION'
             color 3.
```



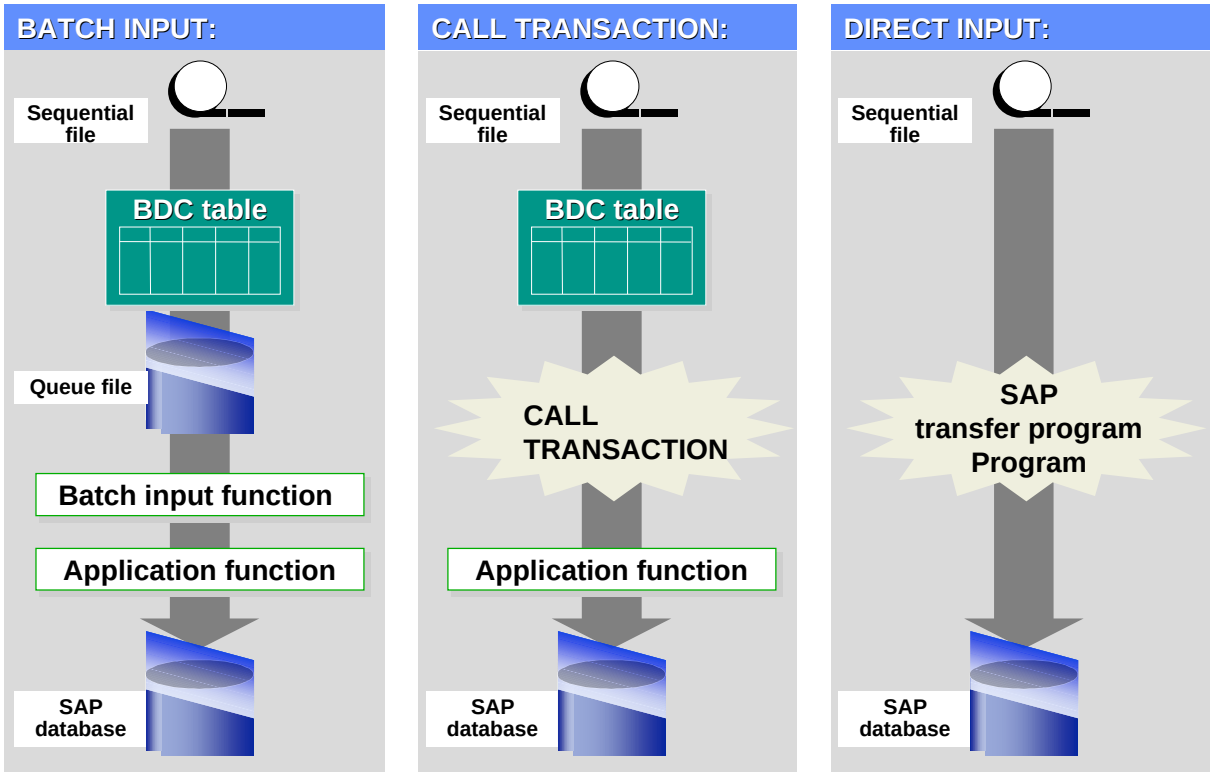



- At runtime the event blocks that create lists are processed in the following order:
  - **START-OF-SELECTION**.
  - **GET spfli**: the first data record from database table **SPFLI** that corresponds to the selection criteria is placed in work area **spfli** and the event block is processed.
  - **GET sflight:** the first data record from **SFLIGHT** that corresponds to the selection criteria as well as to the key of the current **SPFLI** record is placed in work area **sflight** and the event block is processed.
  - **GET sflight:** the next data record from database table **SFLIGHT** is placed in work area **sflight** and the event block is processed again.
  - **GET sflight:** is called again until no further corresponding data records are found.
  - **GET spfli LATE** is called before the next data record from **SPFLI** is placed in work area **spfli**.
  - **GET spfli:** The logical database places the next corresponding data record from **SPFLI** in work area **spfli**.
  - ...
  - **END-OF-SELECTION:** is called immediately before the list is sent.

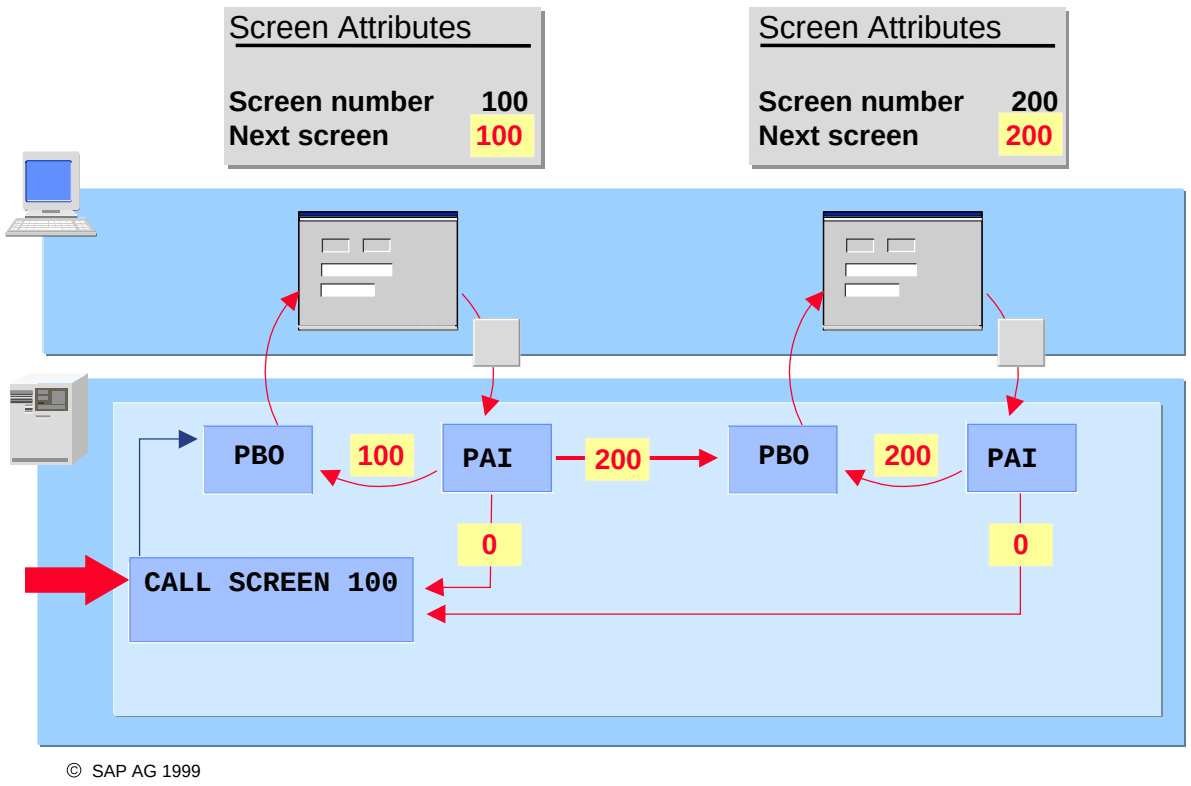



- When transferring data from another SAP or external system to your own SAP system it is important to ensure data integrity.
- Thus the necessity of subjecting this type of data transfer to the same checks as data transfer in dialog mode.
- Since dialog mode checks in transactions are comprehensive and partially cross-application, it is extremely difficult to program them yourself.
- Therefore, it is much easier to check transactions using the same checks as in the SAP dialog mode. Concretely, this means that SAP transactions are also used during data transfer.
- The techniques used for foreign data transfer are called batch input processes.
- SAP offers standardized foreign data transfer procedures for many areas within R/3. The procedures use the programming techniques batch input, call transaction, and direct input. You can access **SAP standard data transfer procedures** using the **Data Transfer Workbench (transaction SXDA).** If no SAP data transfer procedures are available, transfer can be programmed individually using batch input or call transaction.

# External Data Transfer



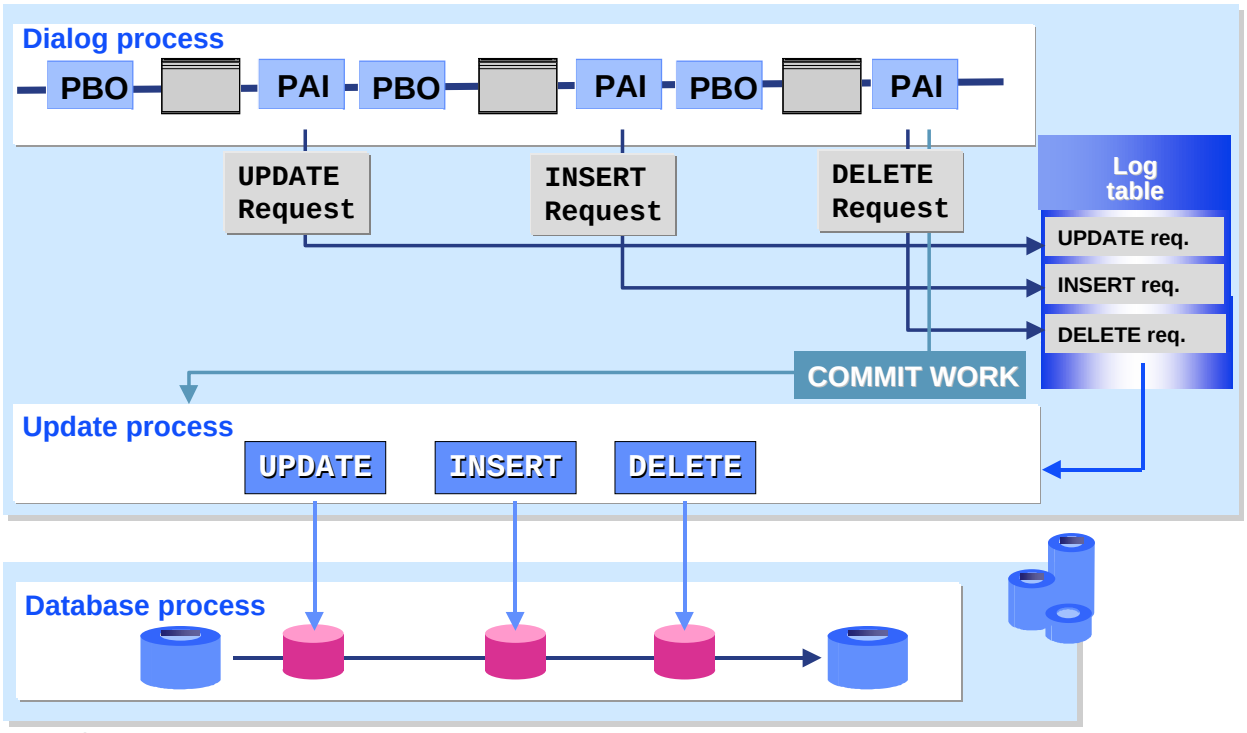
# Advanced Techniques: Dynamic Screen Sequencing






- You can use this technique to program clusters of integrated screens without having to continually return to the source code and call them using CALL SCREEN.

# Advanced Techniques: Update






- Another way of bundling database changes at the end of an SAP LUW is to use the update technique. Here, you do not pass updates directly to the database, but enter them as update requests in a log table instead.
- The dialog part of the SAP LUW ends when the system reaches the COMMIT WORK statement. The R/3 System then triggers a special work process called an update work process, which processes the update requests that you have registered in the log table. The SAP LUW ends when the update work process has finished the database update.
- The dialog and update parts of the SAP LUW can run either synchronously or asynchronously.
- The advantage of update in contrast to bundling using subroutines is that you can enter your update requests in the log table at any time instead of having to keep them in the program area. Its disadvantage is the impaired performance caused by using the log table.
- Use asynchronous update when response times are important and the database updates are complicated enough to justify the overheads involved in using the log table.
- Use synchronous update whenever you need the changed data immediately and when the database updates are complicated enough to justify the overheads involved in using the log table.