# TABC41 ABAP Development Workbench Basics 1/2

TABC41 1/2

R/3 System

Release 46B

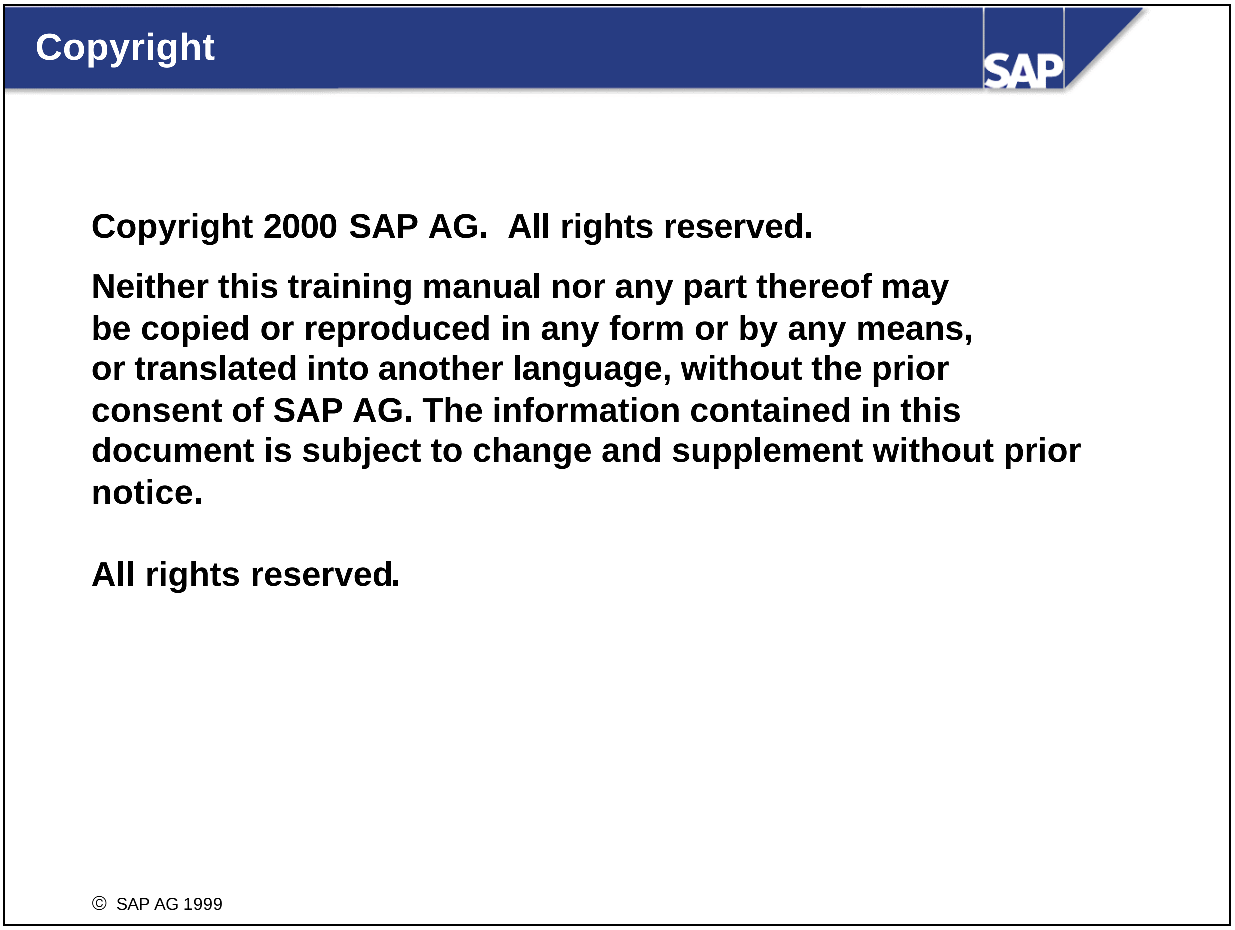
17.06.2000

# TABC41 1/2

## ABAP Development Workbench Basics

## Part 1 of 2

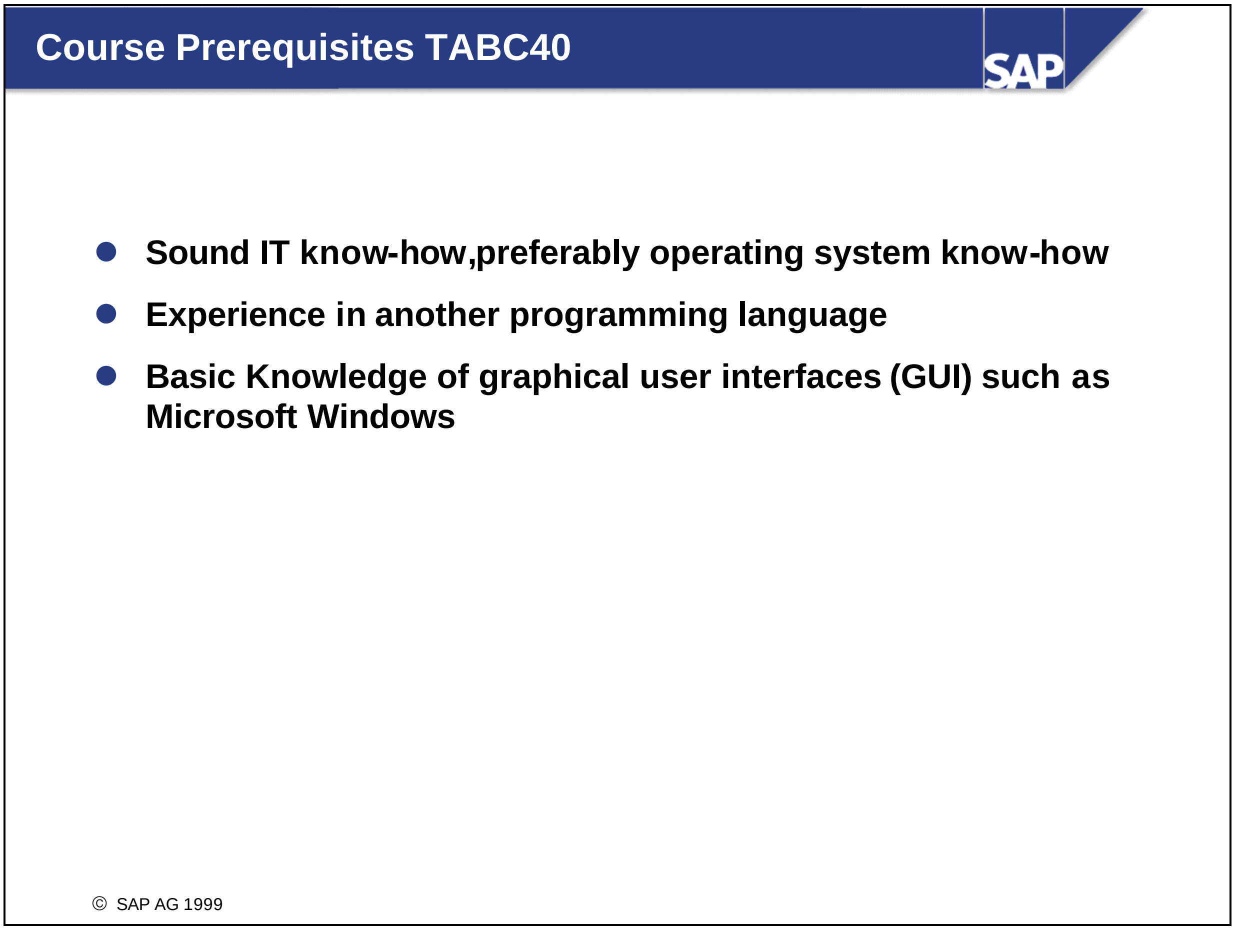
- R/3 System
- Release 4.6B
- May 2000
- Material number 50039583



## Copyright

# Target Group TABC40

- Audience:
  - future ABAP Consultants
  - Consultants from SAP Partners
  - Project team members from SAP Customers
- Duration: 5 weeks

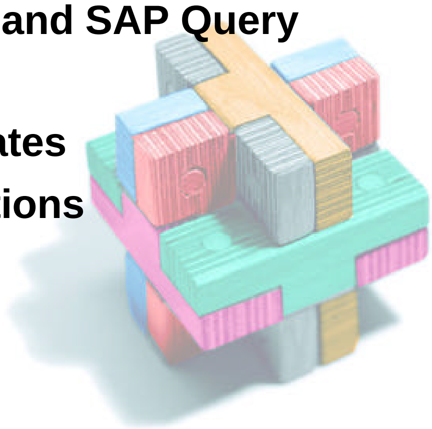


## Course Prerequisites TABC40

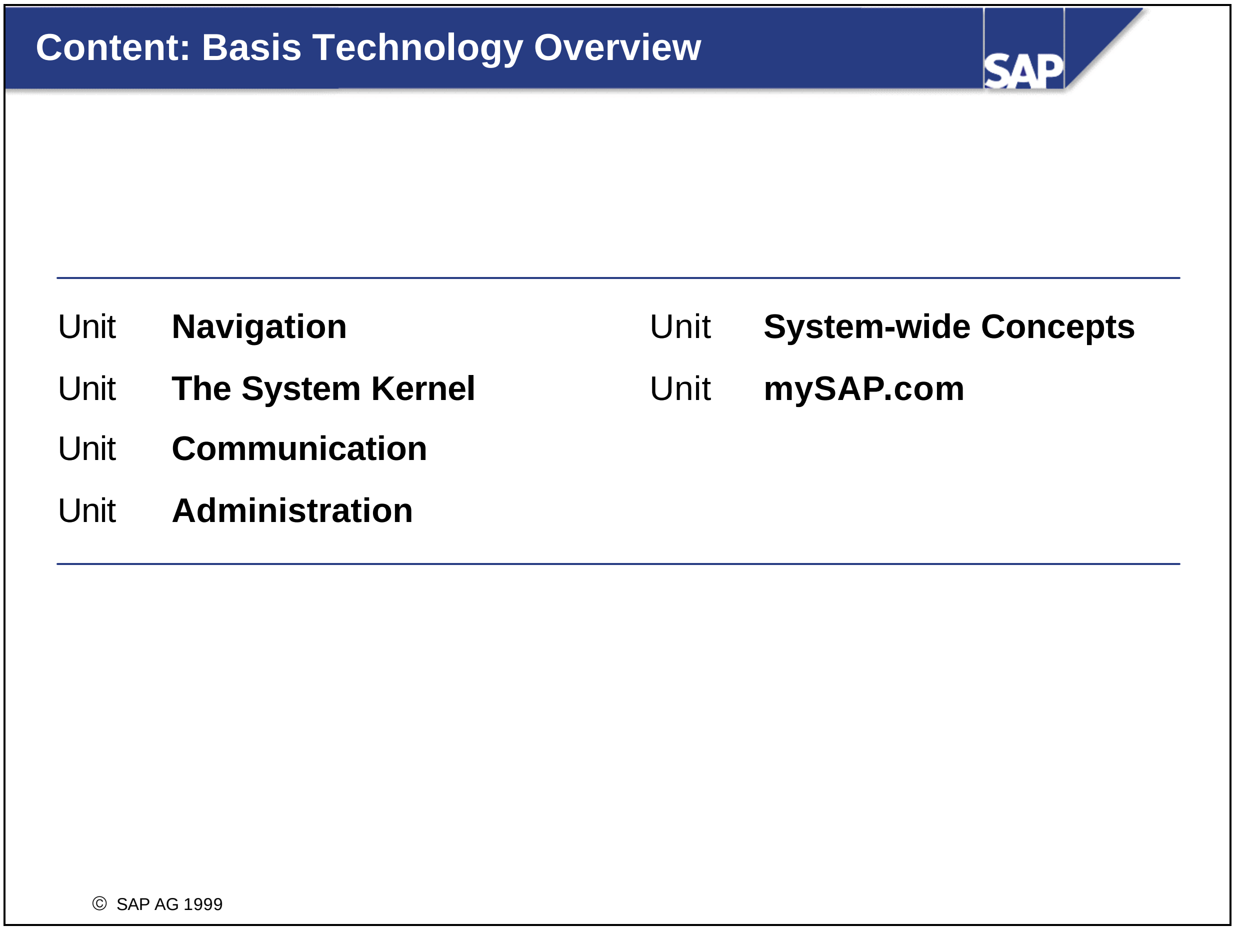
- Sound IT know-how,preferably operating system know-how
- Experience in another programming language
- Basic Knowledge of graphical user interfaces (GUI) such as Microsoft Windows

# Section Overview

| | |
|---|---|
| Section | **Basis Technology Overview** |
| Section | **ABAP Workbench Concepts and Tools** |
| Section | **Managing ABAP Development Projects** |
| Section | **ABAP Dictionary** |
| Section | **ABAP Programming Techniques** |
| Section | **Techniques for List Creation and SAP Query** |
| Section | **Transaction Programming** |
| Section | **Programming Database Updates** |
| Section | **Enhancements and Modifications** |
| Section | **Data Transfer** |



# Section: Basis Technology Overview



## Content: Basis Technology Overview

Unit **Navigation**

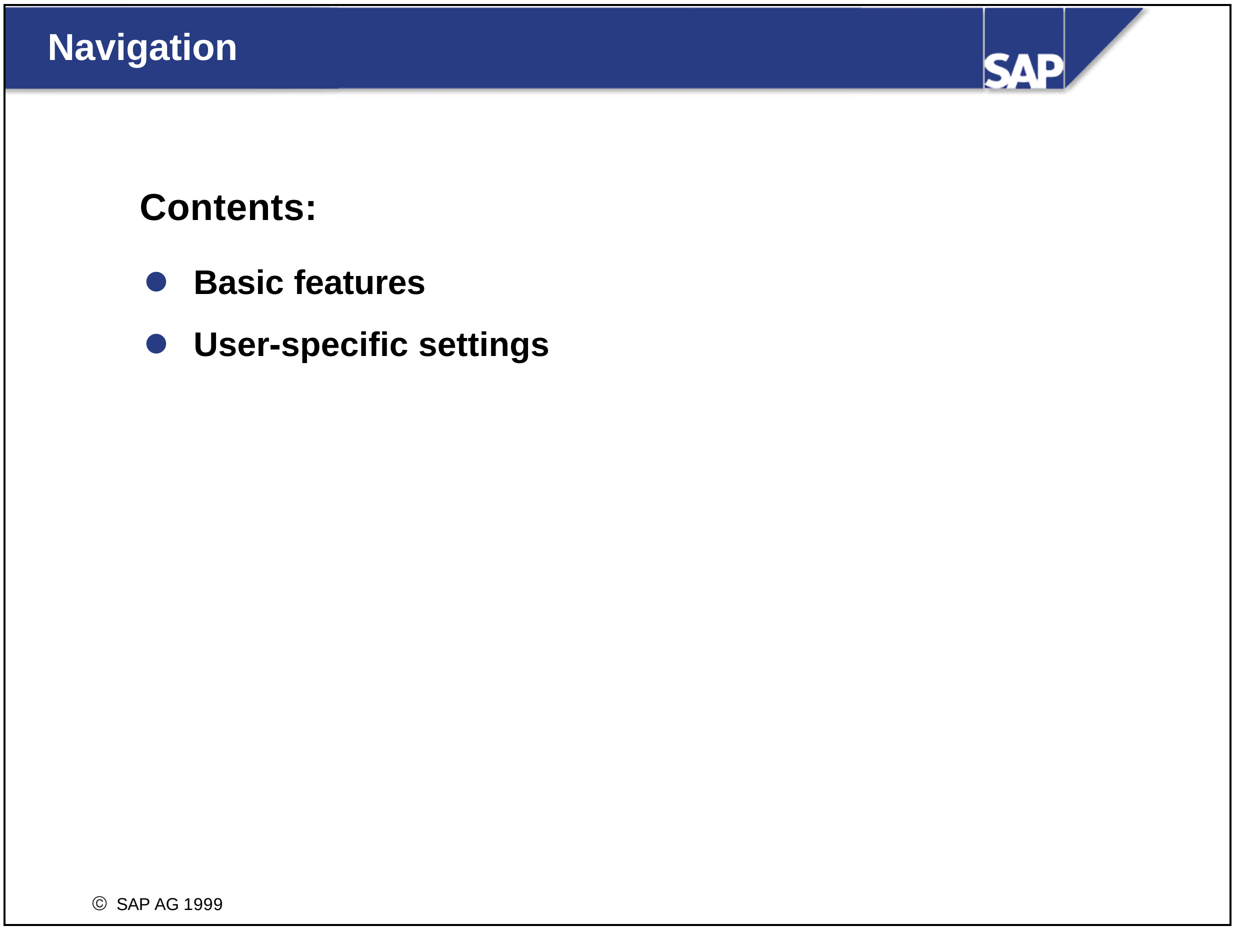
Unit **The System Kernel**

Unit **Communication**

Unit **Administration**

Unit **System-wide Concepts**

Unit **mySAP.com**



# Navigation

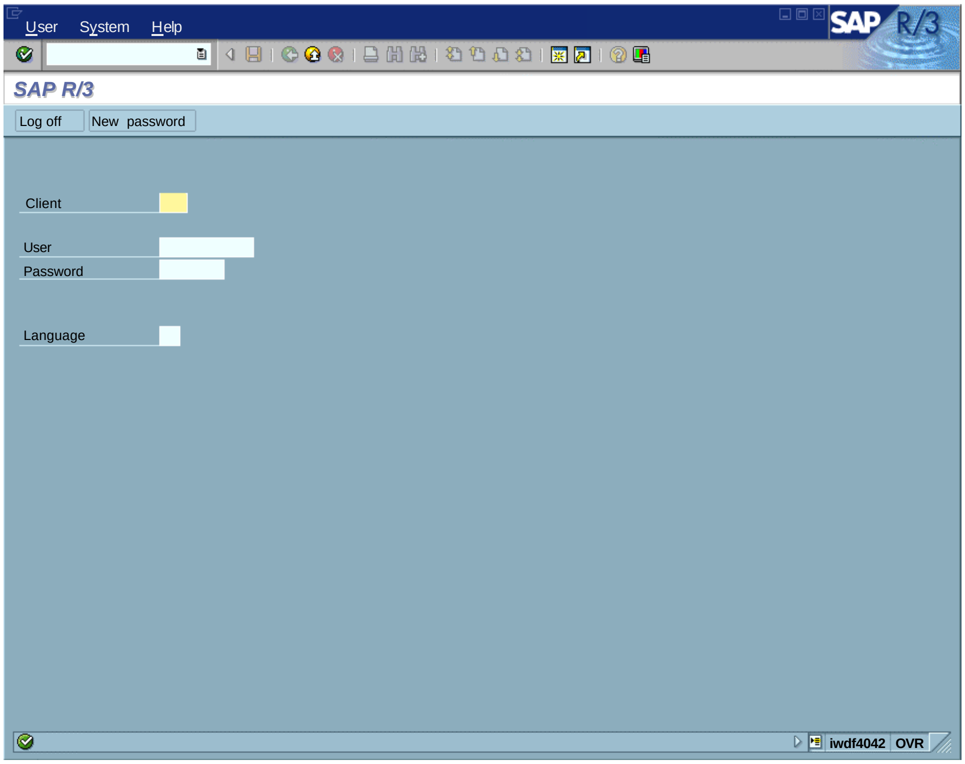
## Contents:

- Basic features
- User-specific settings

# Navigation: Business Scenario

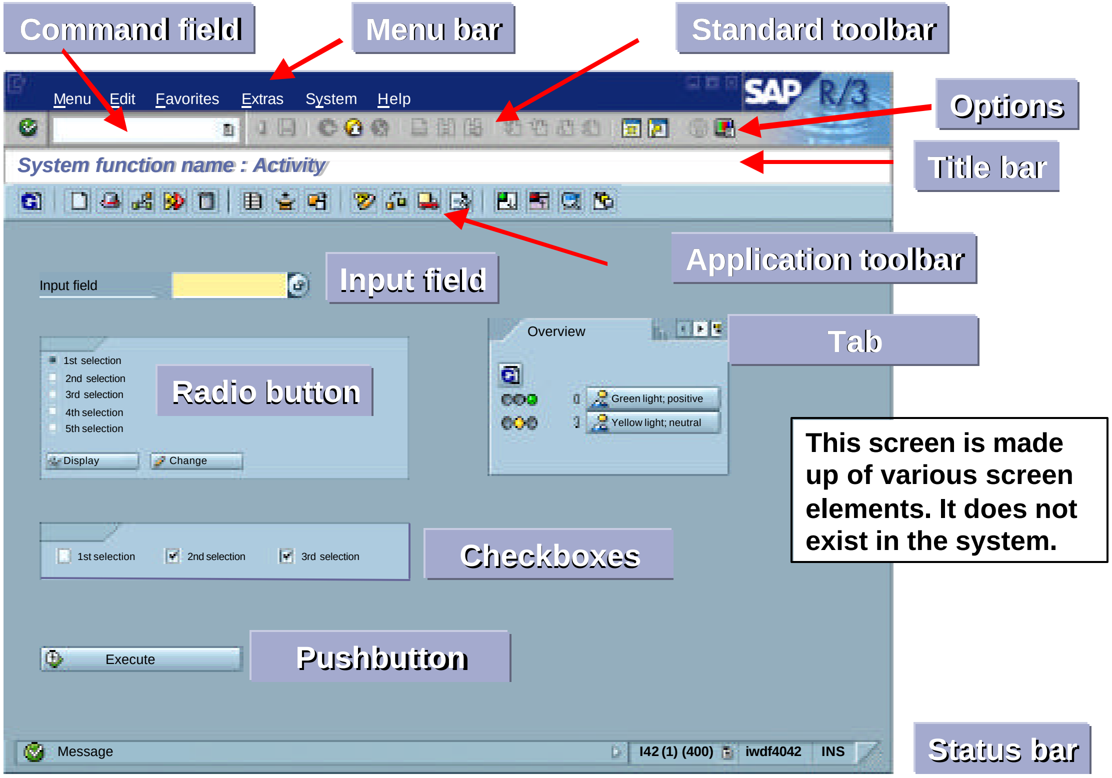
- New users need to familiarize themselves with the screens in the R/3 System and define their personal default settings

## Logging on to the R/3 System



- The R/3 System is a **client system**. The client concept enables the joint operation, in one system, of several enterprises that are independent of each other in business terms. During each user session you can only access the data of the client selected during the logon.
- A **client** is, in organizational terms, an independent unit in the R/3 System. Each client has its own data environment and therefore its own master data and transaction data, assigned user master records and charts of accounts, and specific customizing parameters.
- A user master record linked to the relevant client must be created for users to be able to log on to the system.
- To protect access, a password is required for logon.
  The password is hidden as you type (you only see asterisks).
- SAP systems are available in several languages. Use the *Language* input field to select the logon language for each session.
- Multiple logons are always logged in the system beginning with Release 4.6. This is for security as well as licensing reasons. A warning message appears if the same user attempts to log on twice or more. This message offers three options:
  - Continue with current logon and end any other logons in the system
  - Continue with current logon without ending any other logons in the system (logged in system)
  - Terminate current logon

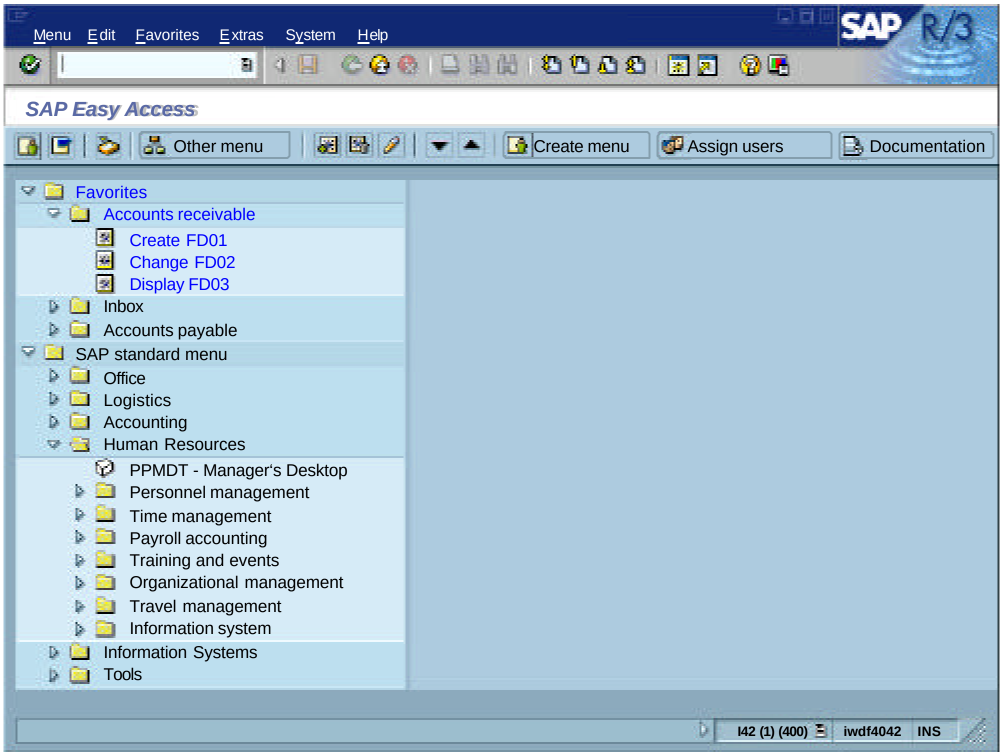
# Screen Elements



This screen is made up of various screen elements. It does not exist in the system.

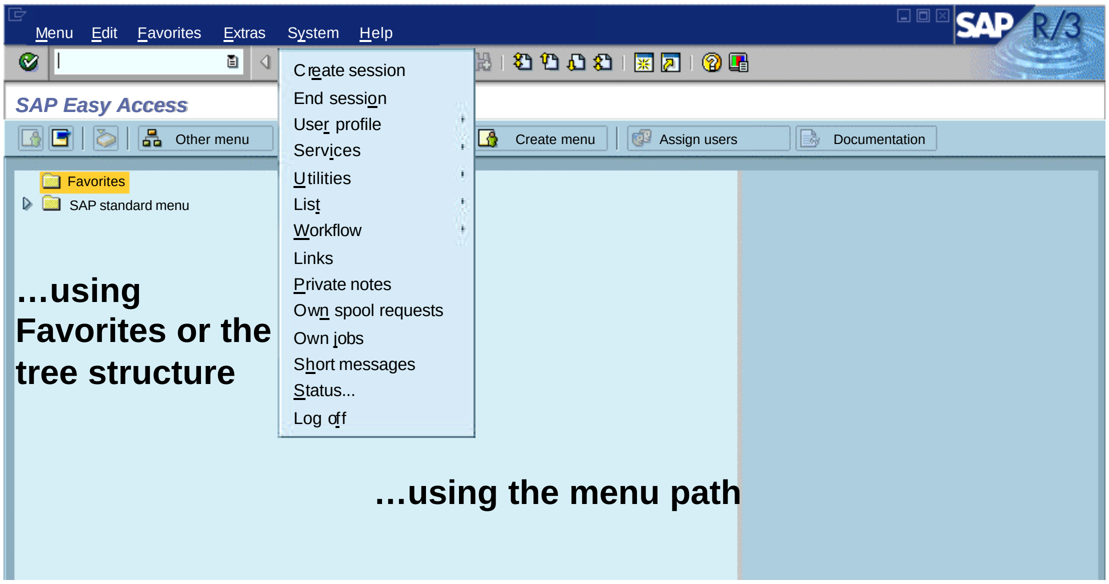
- **Command field:** You can use the command field to go to applications directly by entering the transaction code. You can find the transaction code either in the SAP Easy Access menu tree (see next slide) or in the relevant application under *System® Status*.
- **Menu bar:** The menus shown here depend on which application you are working in. These menus contain cascading menu options.
- **Standard toolbar:** The icons in the system function bar are available on all R/3 screens. Any icons that you cannot use on a particular screen are dimmed. If you leave the cursor on an icon for a moment, a small flag will appear with the name (or function) of that icon. You will also see the corresponding function key. The **application toolbar** shows you which functions are available in the current application.
- **Title bar:** The title bar displays your current position and activity in the system.
- **Check boxes:** Checkboxes allow you to select several options simultaneously within a group.
- **Radio buttons:** Radio buttons allow you to select one option only.
- **Status bar:** The status bar displays information on the current system status, for example, warning and error messages.
- A **tab** provides a clearer overview of several information screens.
- **Options:** You can set your font size, list colors, and so on here.

# SAP Easy Access - Standard



- **SAP Easy Access** is the standard entry screen displayed after logon. Using the menu path *Extras*® *Set start transaction* you can select a transaction of your choice to be the default entry screen after logon.
- You navigate through the system using a compact tree structure that you can adapt to your own specific requirements. Use the menu path *Extras*® *Settings* to change your view of the tree structure. You can use this to display technical names (transaction codes).
- You can also create a **Favorites** list of the transactions, reports, files and Web sites you use most.
- You can add items to your favorites list using the *Favorites* menu option or by simply dragging & dropping them with the mouse.

## Selecting Functions...



...using Favorites or the tree structure

...using the menu path

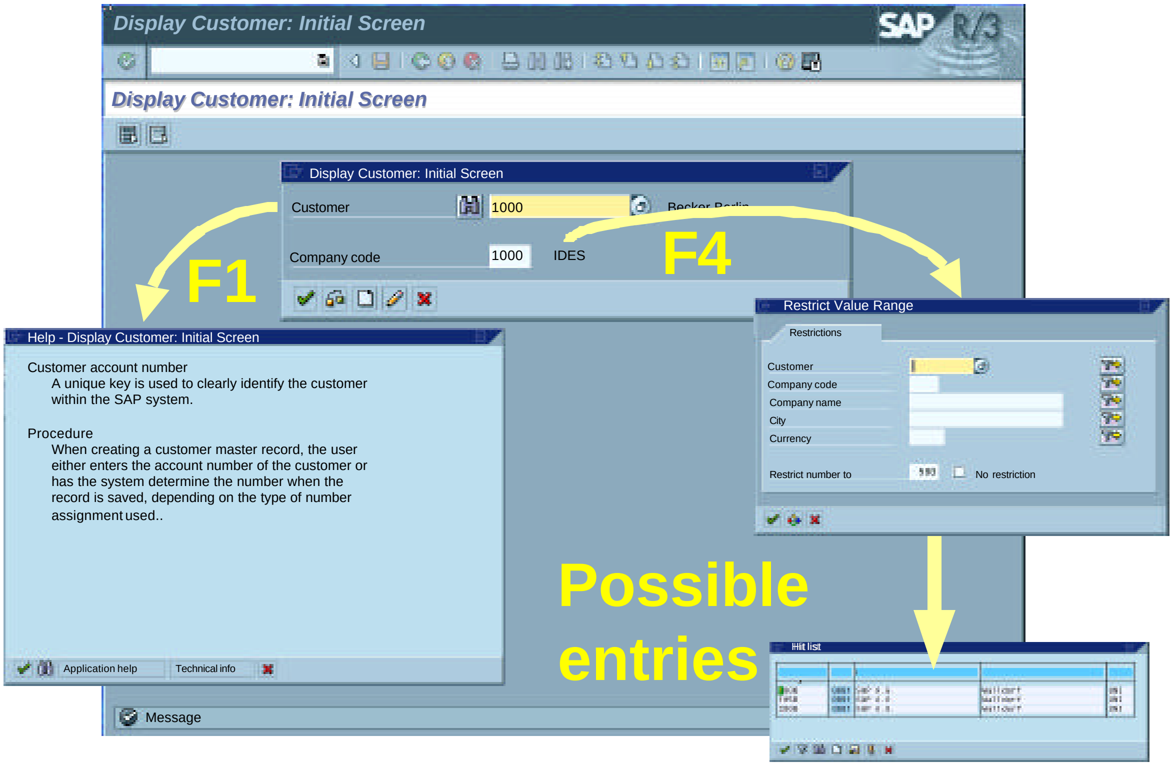
/nFD03

...using the technical name (transaction codes)

You can select system functions in the following ways:

- Use the mouse to choose
  - Menu options
  - Favorites
  - Other options in the tree structure (tree control)
- Use the keyboard (ALT + the underlined letter of the relevant menu option)
- Enter a **transaction code in the command field**:
  - A transaction code (T-Code) is assigned to each function in R/3 (not each screen).
  - You can access the assigned transaction code from any screen in the R/3 System.
  - You can find the transaction code for the function you are working in under the *Status* option of the *System* menu.
  - For example, to display Accounts receivable master data, enter “/n” and the appropriate transaction code (in this case “/nfd03”).
  - Other possible entries:
    “/n” ends the current transaction. “/i” ends the current session.
    “/osm04” creates a new session and goes to the transaction specified (SM04).
  - You can also use the keyboard to get to the command field. Use the CTRL + TAB key combination to make the cursor move from one (input) field group to the next. Use TAB to move between fields within a group.

# Field Help - F1, F4



Display Customer: Initial Screen

Display Customer: Initial Screen

Customer 1000 Becker Berlin

Company code 1000 IDES

Help - Display Customer: Initial Screen

Customer account number

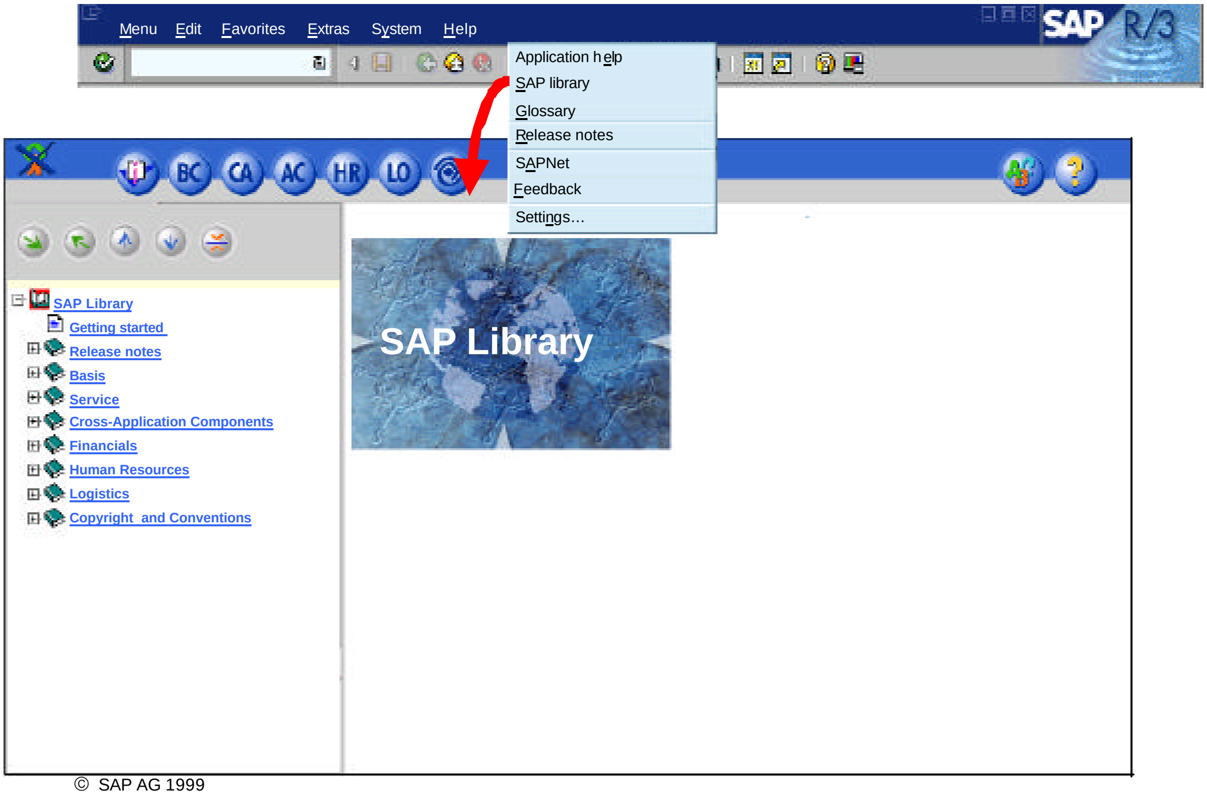
A unique key is used to clearly identify the customer within the SAP system.

Procedure

When creating a customer master record, the user either enters the account number of the customer or has the system determine the number when the record is saved, depending on the type of number assignment used..

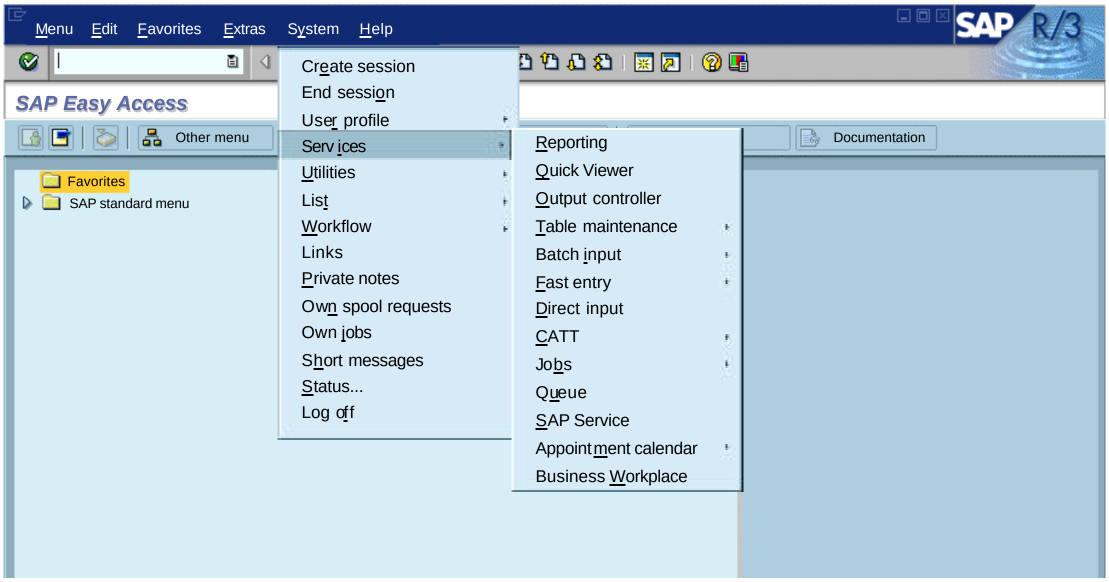
- Use **F1** for help on fields, menus, functions and messages.
- F1 help also provides technical information on the relevant field. This includes, for example, the parameter ID, which you can use to assign values to the field for your user.
- Use **F4** for information on what values you can enter. You can also access F4 help for a selected field using the button immediately to the right of that field.
- If input fields are marked with a small icon with a checkmark, then you can only continue in that application by entering a permitted value.
  - You can flag many fields in an application to make them either required entry fields or optional entry fields. You can also hide fields using transaction or screen variants or Customizing.

# SAP Online Help



- The R/3 System provides comprehensive online help. You can display the help from any screen in the system. You can always request help using the *Help* menu or using the relevant icon.
- The *Help* menu contains the following options:
  - Application help: Displays comprehensive help on the current application. Selecting this menu option in the initial screen displays help on getting started with R/3.
  - SAP Library: This is where all online documentation can be found.
  - Glossary: Enables you to search for definitions of terms.
  - Release notes: Displays notes which describe functional changes that occur between R/3 releases.
  - SAPNet: Enables you to log on to SAPNet.
  - Feedback: Enables you to send a message to the SAPNet R/3 Frontend, SAP’s service system.
  - Settings: Enables you to select settings for help.

# System Functions - Services



- The *System* menu contains, among others, the following options:
  - Create/end session: Enables you to create and end sessions. You can work with up to 6 sessions at a time.
  - User profile: This is where you can enter user-specific settings.
  - Services: Takes you to important service functions (see below).
  - List: Contains important list functions, such as searching for character strings, saving in PC files, printing, and so on.
  - Status: Enables you to display important user and system data.
  - Log off: Ends the SAP R/3 session with a confirmation prompt.
- The *System* ® *Services* menu contains, among others, the following options:
  - Reporting: Starts reports (ABAP programs).
  - Output controller: This is where you manage user-specific print requests.
  - Table maintenance: This is where you process tables and views.
  - Batch input: Administers batch input sessions and data transfer.
  - Jobs: This is where you can administer jobs that are processed in the background.
  - SAP Service: Enables you to log on to SAP's SAPNet R/3 Frontend.

- Use the menu option *System® User profile® Own data* to set your own personal profile. You can choose between the *Address, Defaults* and *Parameters* tabs.
  - Address: You can create and maintain personal data here, for example, name,function, room number, telephone number, e-mail addresses and so on.
  - Defaults: Defaults include the date display format, the decimal notation format, the default printer, the logon language, and so on.
  - Parameters: Use this to assign entries to commonly-used fields. This is only available for input fields that have been allocated a parameter ID.

    **Procedure for finding out a field's Parameter ID**: Go to the input field to which you want to assign a value. Choose F1, then the "Technical info" pushbutton. This opens a window that displays the corresponding parameter ID (if one has been allocated to the field) in the "Field data" section.
- The *User profile* menu also contains, among others, the following options:
  - *Hold data*, *Set data*, *Delete data*. Use *Hold data* to keep data values that you have entered in fields in an application for the duration of a user session. When you call up the application again, you can overwrite these values. Once you have *Set data*, you can no longer overwrite these values and have to use *Delete data* if you want to enter different values.

- Use the *Table Settings* function to change, in the table control, the individual basic table settings that are supplied with the system. This is particularly useful for tables where you do not need all the columns. You can use the mouse to drag & drop column positions and widths, or even make the column disappear.
- Save the changed table settings as a *variant*. The number of different variants you can create per table is not restricted.
- The first variant is called the *basic setting*; the SAP System defines this setting. You cannot delete the basic setting (you can delete the variants you define yourself).
- The table settings are stored with your user name. The system uses the variant currently valid until you exit the relevant application. If you then select the application again, the system will use the standard settings valid for this table.
- **Note:** you can change table settings wherever you see the table control icon in the top right-hand corner of a table.

- The R/3 System provides numerous options for settings and adjustments:
  - Define default values for input fields
  - Hide screen elements
  - Deactivate screen elements (shaded out).
    You can do this by, for example, defining transaction variants.
    If you preallocate all necessary parameters for parameter transactions, you do not need to go through the initial screen.
    These functions have been available in R/3 for several releases.
- SAP now also includes the **GuiXT**. In addition to all the above functions, you can now:
  - Include graphics
  - Convert fields and add pushbuttons and text
  - Change input fields (or their F4 help results) into radio buttons
- The **GuiXT** scripts are stored on the frontend. In accordance with local scripts, the GuiXT scripts determine how data sent from the application server is displayed. These scripts can be standard throughout a company, or they can be different for each frontend.
- As of Release 4.6, **GuiXT** is part of the SAP standard system.

## Unit: Navigation
## Topic: Basic features

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

- Log on to a given R/3 System
- Find transaction codes
- Access the SAP Library
- Use F1 help to find field information
- Use F4 help to search for possible field entries

As a new user of the R/3 System, you begin to navigate the system using the menu paths and transaction codes. You also begin to access various online help and discover the kinds of information each provides.

1-1 Logging on to the R/3 System

Select the appropriate R/3 System for this course. Use the client, user name, initial password and logon language specified by the instructor. The first time you log on, you will get a prompt in which you must enter your new password twice. Make a note of the following:

Client: _ _ _ User: _ _ _ _ _ _ _ _ _ Password: ____________ Language: _ _

1-2 What is the maximum number of sessions you can have open simultaneously? __

1-3 Identify the screen names and find the transaction codes that correspond to the following menu paths?

1-3-1 ***Tools*** ® ***Administration*** ® ***Monitor*** ® ***System Monitoring*** ® ***User Overview***

Name of screen: ________________________________________

Transaction: ___________

1-3-2 ***Accounting*** ® ***Financial Accounting*** ® ***Accounts receivable*** ® ***Master records*** ® ***Display***

Enter *Customer* **1000** and *Company code* **1000** to get to the next screen.

Name of screen: ________________________________________

Transaction: __________

1-4 Help

1-4-1 If you choose *Application help* in the *SAP Easy Access* screen, which area of the SAP Library does it take you to ?

__________________________________________________________

To answer the questions below, you will need to go to the *Display Customer: Initial Screen*

1-4-2 Use **F4** help on the *Customer* field to find the customer number for Becker ##.

Note: ## corresponds to your assigned group number.

__________________________________________________________

1-4-3 Use **F1** help on the *Customer* field. What is the use of this field? Please write a brief summary of the business-related information.

__________________________________________________________

1-4-4 Use **F1** help on the *Company code* field. If you choose the *Application help* button from the F1 help screen, which area of the SAP Library does it take you to?

______________________________________________

1-4-5 Which pushbutton do you need to use on the F1 help screen to find the parameter ID for the *Company code* field?

______________________________________________

## Unit: Navigation

## Topic: User-specific settings

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

- Set a user parameter for a field
- Set various user defaults such as language, date format, and decimal notation
- Create folders and add transactions to your Favorites
- Select a start transaction of your choice as the default displayed after logging on (optional)

You begin to set various user-specific settings to personalize the system to your liking.

Exercises marked * are optional.

2-1 Setting user parameters.

2-1-1 Assign a parameter value for the *Company code* field to your user profile.

Note: The instructor will tell you what parameter value to enter.

Parameter ID: ___ ___ ___

Parameter value: ___ ___ ___ ___

2-2 Setting user defaults.

2-2-1 In your user profile, set your logon language to the value used for the course.

2-2-2 In your user profile, select the decimal notation and date format that you desire.

2-3 Defining favorites of your choice.

2-3-1 Insert at least one new folder under the Favorites folder.

2-3-2 Add any two of your “favorite” transactions to the corresponding folder(s).

2-3-3 Add the Internet address “http://www.sap.com” under the text “SAP Homepage”.

2-3-4 Add the Internet address for the online evaluation (the instructor will tell you the URL) under the text “Online Evaluation”.

*2-4 [illegible]

2-4-1 Enter a transaction of your choice as the initial transaction. You will then need to log off and on again for the change to take effect.

Note: If desired, you can change the initial transaction back to the system default (SAP Easy Access).

## Unit: Navigation

## Topic: Basic features

1-1 Log on to the system specified by the instructor and change your initial password.

1-2 You can open and close sessions using ***System* ® *Create session*** (or using the appropriate icon) and ***System* ® *End session***.
The maximum number of sessions you can have open simultaneously is 6.

1-3 To find the transaction code, select ***System* ® *Status***. These screen names and transaction codes correspond to the menu paths:

1-3-1 Transaction: SM04 for Screen Name: *User list*

1-3-2 Transaction: FD03 for Screen Name: *Display Customer: General data*

1-4 Help

1-4-1 The entire SAP Library is available including Getting Started.

***Help* ® *Application help***

1-4-2 T-CO05A## (## corresponds to your assigned group number)

When you select **F4** in the *Customer* field, the *Restrict Value Range* window appears. You can explore the various tabs to see the different search criteria available. Find a tab that includes the *Name* field and enter the following:

| **Field Name** | **Values** |
|---|---|
| *Name* | *Becker ##* |

Select the *Continue Enter* pushbutton. A window now appears listing the customer account numbers that match your search criteria. Select the line that corresponds to *Becker ##*, then select the *Copy Enter* pushbutton. This automatically copies the customer account number into the *Customer* field.

1-4-3 Suggestion: The customer is a unique key (account number) used to clearly identify the customer within the system.

1-4-4 FI – Accounts Receivable and Accounts Payable

1-4-5 Use the *Technical Info* pushbutton to find the *Parameter Id*: BUK.

## Unit: Navigation

## Topic: User-specific settings

2-1 Setting user parameters.

2-1-1 To assign a parameter value to a field you will need the parameter ID of the field. First you need to select a transaction that contains this field. For example, *Company code* can be found in transaction **FD03**. Next, place the cursor on that field (just click on it with the mouse). Accessing:

***F1* ® *Technical Info* ® *Parameter ID***

gives you the required information. For the *Company code* field, the parameter ID is **BUK**.

Finally, you enter the parameter ID and desired value in your user profile:

***System* ® *User profile* ® *Own data***

On the *Parameter* tab you enter the parameter ID and value that you want to be entered into the field. *Save* your entries.

2-2 Setting user defaults.

2-2-1 To set the logon language, go to your user profile:

***System* ® *User profile* ® *Own data***

On the *Defaults* tab, enter the language of your choice in the *Logon language* field.

2-2-2 To set the decimal notation and date format, remain on the *Defaults* tab in your user profile. Select the indicator adjacent to the notation and format you desire. *Save* your selections.

2-3 Defining favorites of your choice.

2-3-1 ***Favorites* ® *Insert folder***

Type any name for the folder then select *Enter*. You can add as many folders as you desire. Once created, folders can be dragged and dropped to position them where you want.

2-3-2 To create favorites, select specific applications (transactions) that you need as favorites for your daily work from the menu tree of the SAP standard menu. Add them to your Favorites list by selecting them and choosing *Favorites* ® *Add* from the menu bar. Alternatively, use the mouse to drag & drop favorites to a folder. You can also use the menu path ***Favorites* ® *Insert transaction*** to add using a transaction code.. Finally, you can move existing favorites to different folders later using ***Favorites* ® *Move*** or using drag & drop.

2-3-3 Create Internet addresses using ***Favorites*** ® ***Add Web address or file****.* When you select SAP Homepage from your favorites, an Internet browser will open and you will be connected to SAP's homepage.

2-3-4 ***Favorites*** ® ***Add Web address or file***

You will use this link at the end of the course to fill out the course evaluation.

2-4 Setting a start transaction.

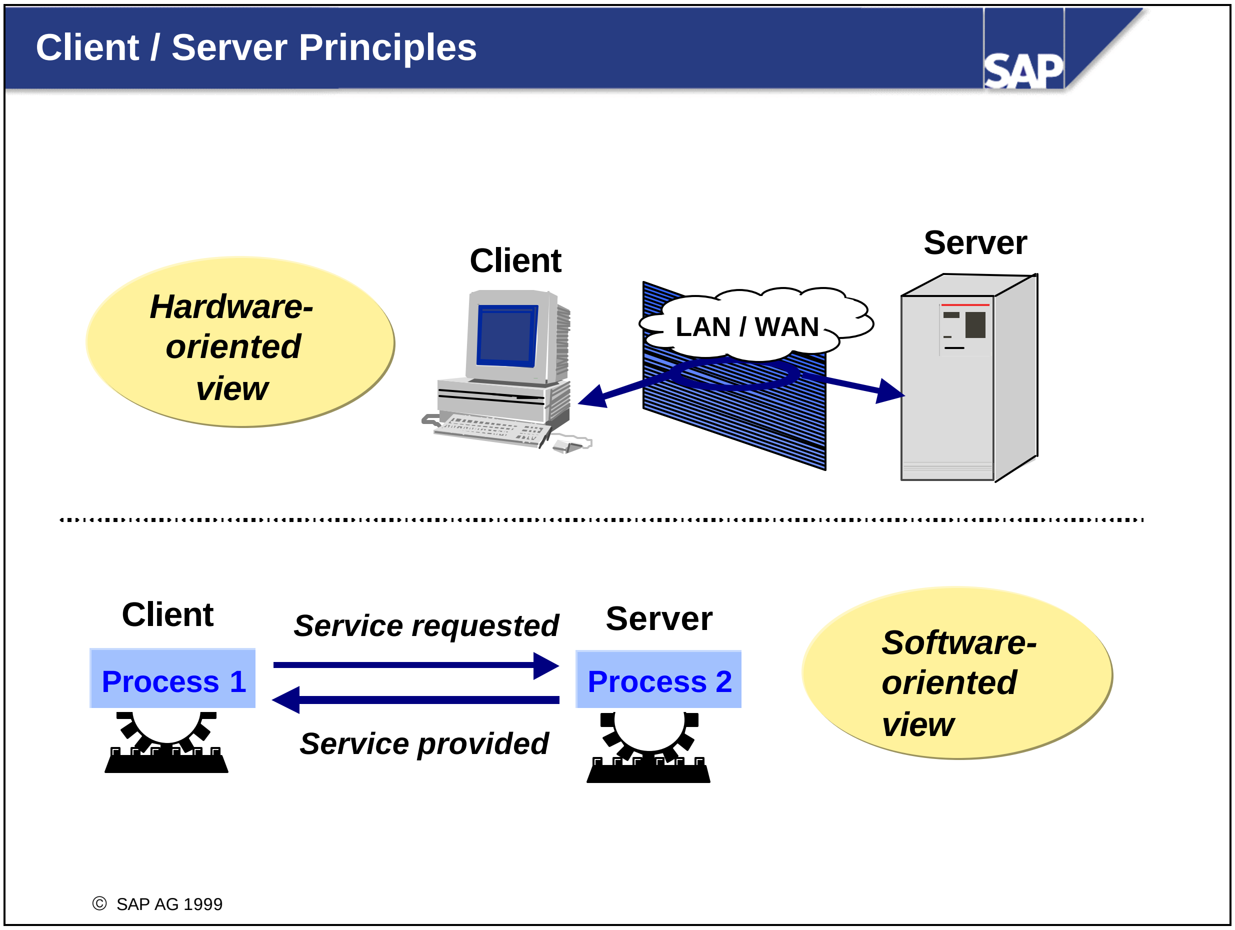
2-4-1 ***Extras*** ® ***Set start transaction***

Enter a transaction of your choice then select the *Enter* pushbutton. Notice the system message on the status bar indicates that your selected transaction has been set as the start transaction. The next time you log on, the system will go directly to your start transaction.

Note: To change back to SAP Easy Access as the initial screen, follow the menu path again, delete the transaction code and select *Enter*. The next time you log on, SAP Easy Access will be the initial screen.

# System Kernel: Contents

- Flow of user requests through the system
- Communication between the application layer and the database
- The processes on the frontend and application layers
- Asynchronous update
- Background processing and the spool system



## Client / Server Principles

Hardware-oriented view

Client

LAN / WAN

Server

Client

Process 1

Service requested

Service provided

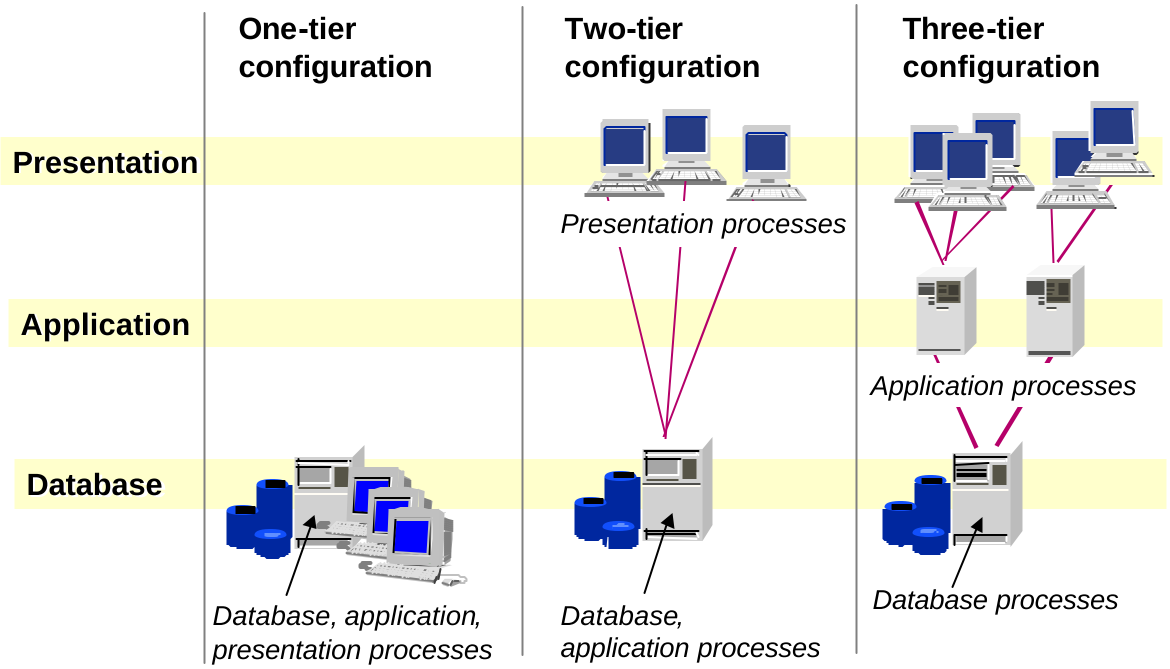
Server

Process 2

Software-oriented view

- In SAP terminology, a *service* means a service provided by a software component (software-oriented view). This component can consist of a process (compare work process) or a group of processes (compare application server) and is then called a server for that service.
- Software components that use this service are called clients. At the same time, clients can also be servers for specific services.
- A server often also means a computer (host) on which software components that provide specific services are running (hardware-oriented view).

## R/3 System Client / Server Configurations



One-tier configuration

Two-tier configuration

Three-tier configuration

Presentation

Application

Database

Presentation processes

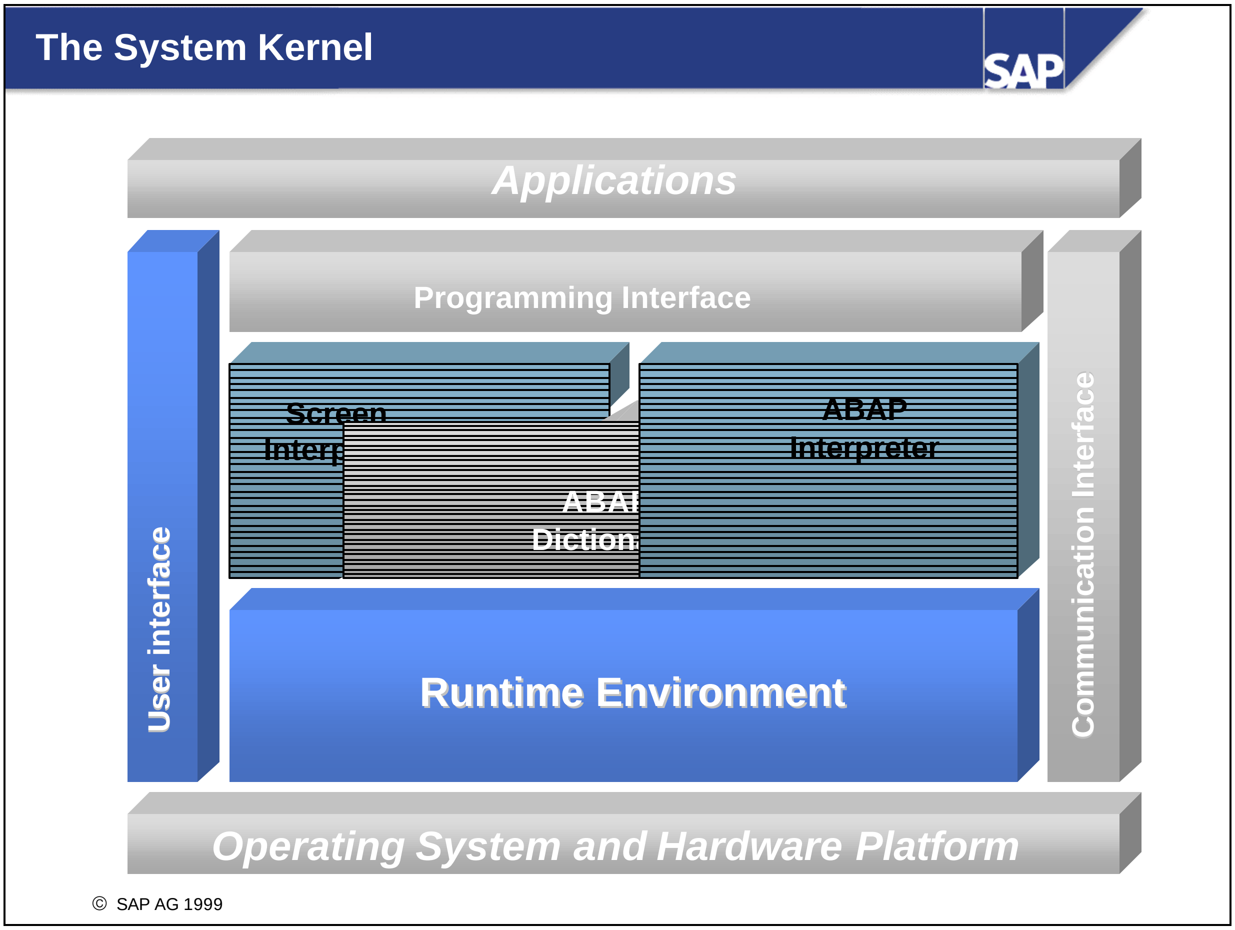
Application processes

Database, application, presentation processes

Database, application processes

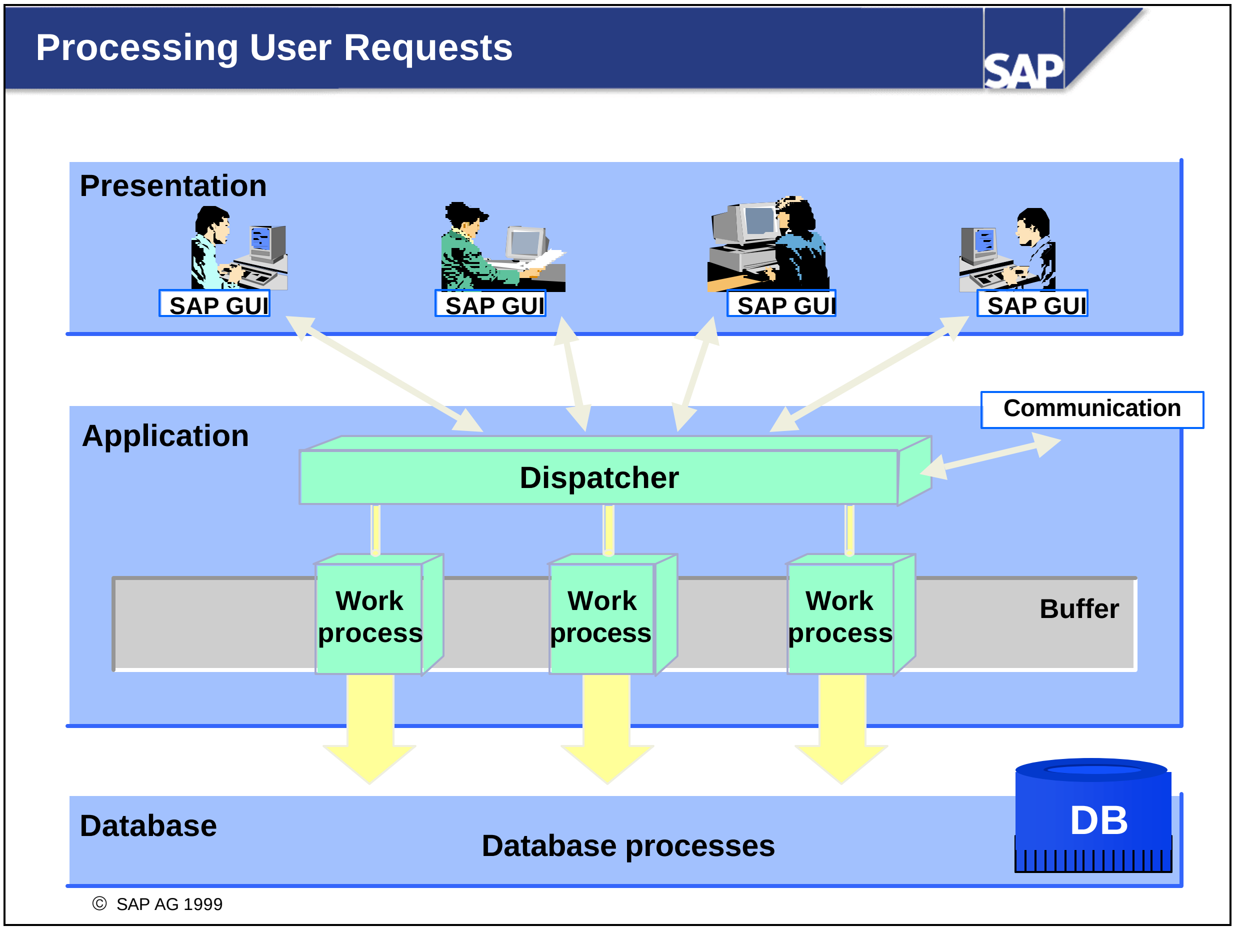
Database processes

- The fundamental services in a business application system are presentation services, application services, and database services.
- In a one-tier R/3 System configuration, all processing tasks are performed on one server, as in classic mainframe processing.
- Two-tier R/3 System configurations are usually implemented using special presentation servers that are responsible solely for formatting the graphical user interface. Many R/3 System users use Windows PCs for example as presentation servers. An alternative two-tier configuration (not shown) is to install powerful desktop systems and to use these for presentation and applications also (two-tier client/server). This type of configuration is particularly useful for processing-intensive applications (such as simulations) or for software developers, but due to the additional administration requirements is usually used for test purposes only.
- In a three-tier configuration, separate servers are used for each tier. Using data from the database server, several different application servers can operate at the same time. To ensure that the load on individual servers is as even as possible and to achieve optimal performance, you can use special application servers for individual application areas such as distribution or financial accounting (logon and load balancing).



## The System Kernel

- In this unit, we discuss the central processes of the R/3 Basis System. This includes an explanation of how a user request is sent to and processed by the application layer, and which process types are involved in processing the request. Data entered by the user is sent through the user interface (the SAP GUI) to the dispatcher, which coordinates further processing. The work processes used are those that map to the same source code as the dispatcher and whose substructures such as Screen Interpreter and ABAP Interpreter are presented here. Another topic is data exchange with the database.



## Processing User Requests

Presentation

SAP GUI

SAP GUI

SAP GUI

SAP GUI

Application

Communication

Dispatcher

Work process

Work process

Work process

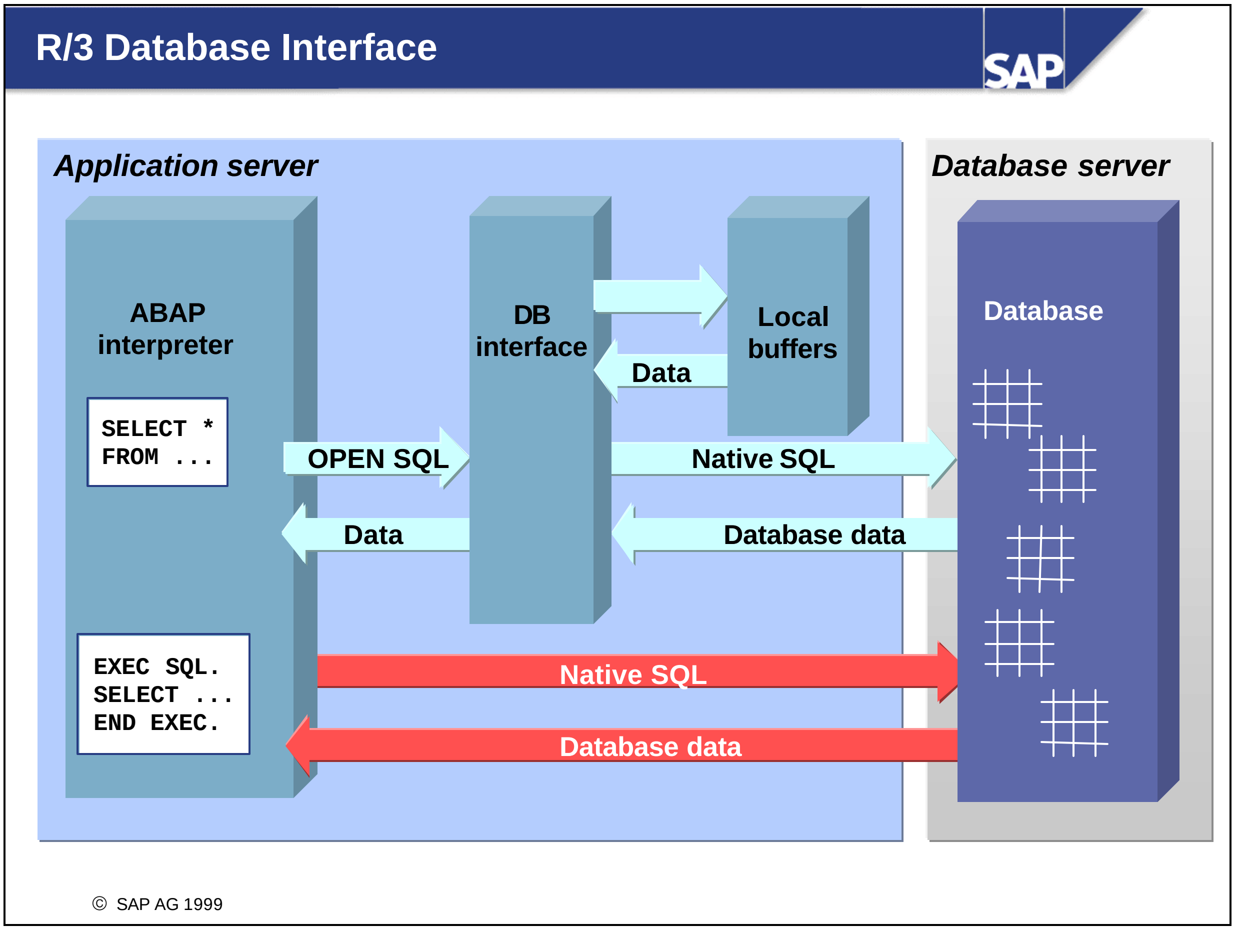
Buffer

Database

Database processes

DB

- The central process in the R/3 application layer is the dispatcher. Together with the operating system, the dispatcher controls the resources for the R/3 applications. The main tasks of the dispatcher include distributing transaction load to the work processes, connecting to the presentation layer, and organizing communication.
- User screen input is received by the SAP presentation program SAP GUI, converted into its own format, and then sent to the dispatcher. The processing requests are then saved by the dispatcher in request queues and processed according to “first in / first out”.
- The dispatcher distributes (dispatches) the requests one after the other to the available work processes. Data is actually processed in the work process. The user that sent the request through the SAP GUI is usually not assigned the same work process, because there is no fixed assignment of work processes to users.
- Once the data has been processed, the processing result from the work process is sent through the dispatcher back to the SAP GUI. The SAP GUI interprets this data and generates the output screen for the user with the help of the operating system on the frontend computer.
- During initialization of the R/3 System, the dispatcher executes the following actions, among others: it reads the system profile parameters, starts work processes, and logs onto the message server (this service will be explained later).



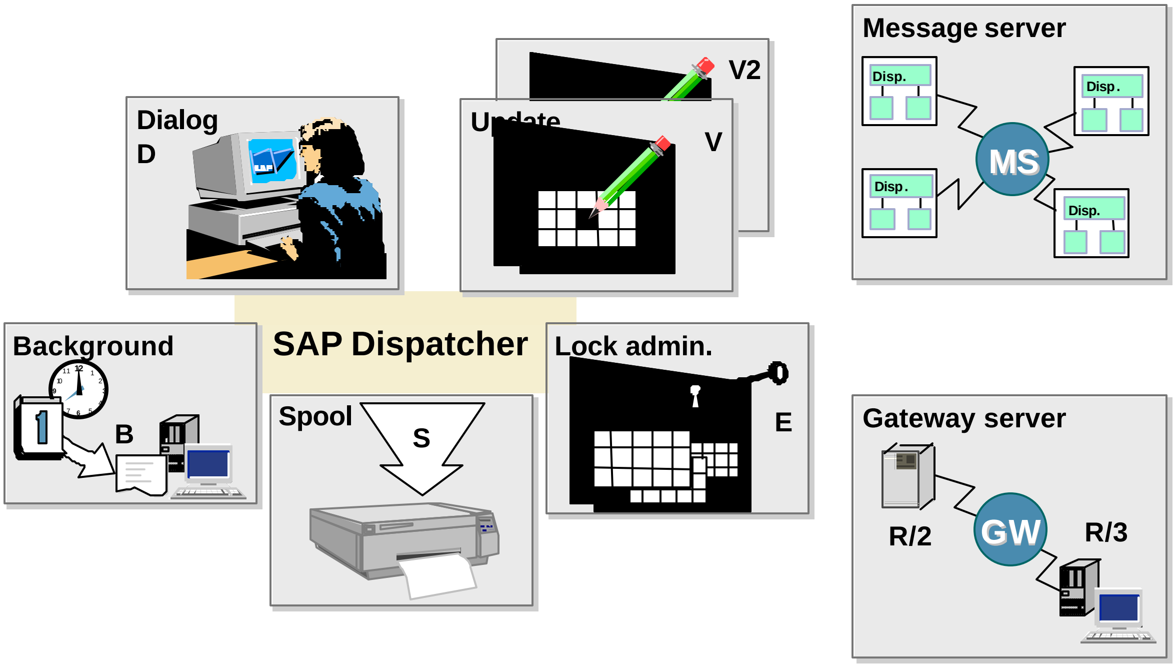
## R/3 Database Interface

```
SELECT *
FROM ...
```

```
EXEC SQL.
SELECT ...
END EXEC.
```

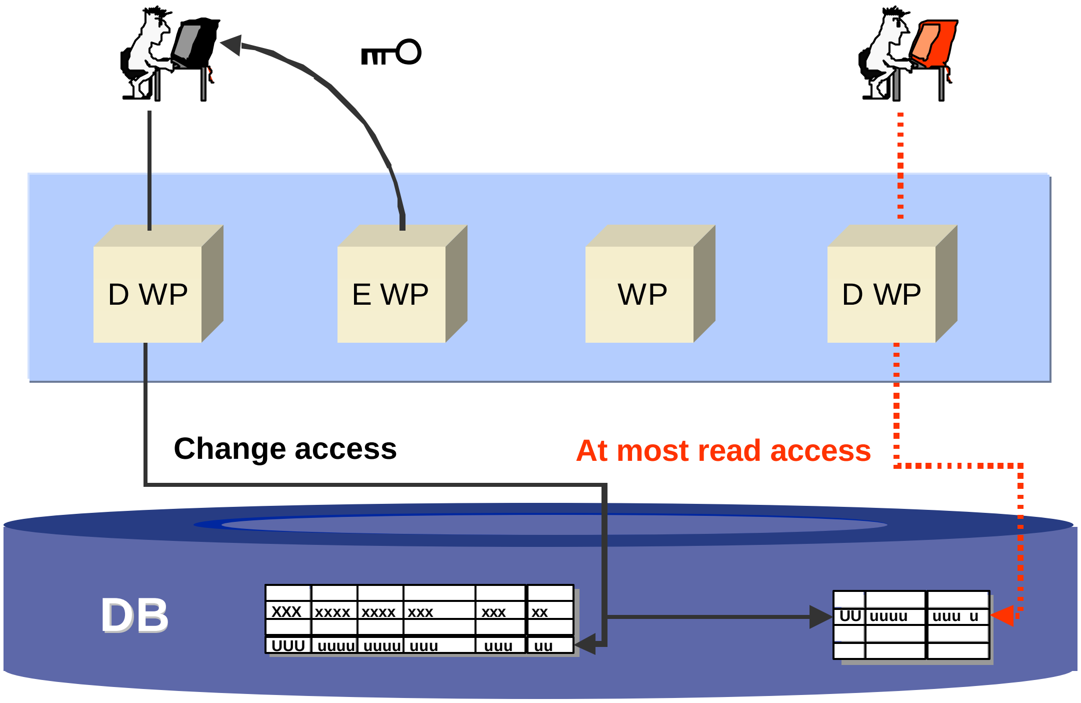
- Today, large amounts of data are usually administered using relational database management systems (RDBMS). These systems store the data and the relationships between the data in two-dimensional tables, which are known for their logical simplicity. The definitions of the data, tables, and table relationships are stored in the data dictionary of the RDBMS.
- Within ABAP, SAP OPEN SQL is used to access application data in the database, independent of the corresponding RDBMS. The R/3 database interface converts the open SQL statements from the ABAP statements into the corresponding database statements. This means that application programs written in ABAP are database-independent. Native SQL commands can be used in ABAP.
- When interpreting open SQL statements, the R/3 database interface checks the syntax of these statements and automatically ensures the local SAP buffers in the shared memory of the application server are utilized optimally. Data frequently required by the applications is stored in these buffers so that the system does not have to access the database server to read this data. In particular, all technical data such as ABAP programs, screens, and ABAP Dictionary information, as well as some business process parameters usually remain unchanged in a running system, making them ideal buffering candidates. The same applies to certain business application data, which is accessed as read-only.

# R/3 Application Services



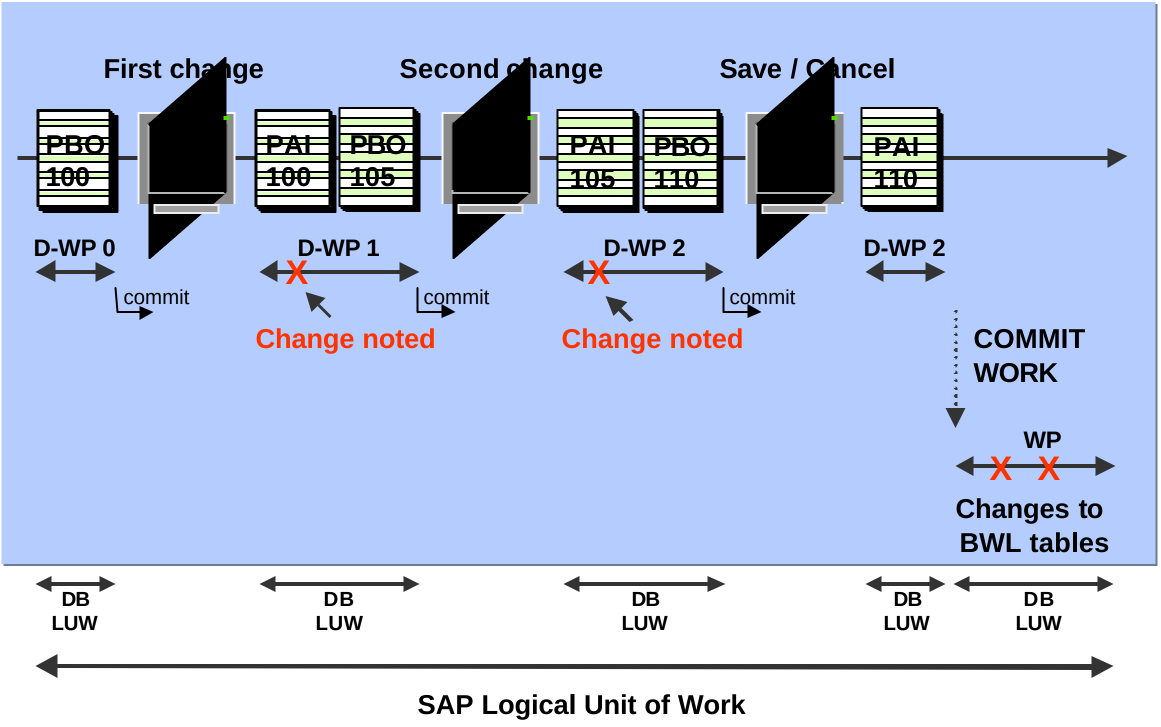
- The operating system views the R/3 runtime system as a group of parallel, cooperating processes. On each application server these processes include the dispatcher as well as work processes; the number of work processes depends on the available resources. Work processes may be installed for dialog processing, update, dialog free background processing and spooling.
- In addition to these work process types (dialog processing (D), update (V: for the German “Verbuchung”),  lock management (E), background processing (B), spool (S), the R/3 runtime system provides two additional services for internal and external communication (below are the restrictions on the number of work processes):
  - The message server (MS or M) communicates between the distributed dispatchers within the R/3 System and is therefore the prerequisite for scalability using several parallel-processing application servers.
  - The gateway server (GW or G) allows communication between R/3, R/2 and external application systems.
  - Dialog: Every dispatcher requires at least two dialog work processes
  - Spool: At least one for each R/3 System (more than one allowed for each dispatcher)
  - Update: At least one for each R/3 System (more than one allowed for each dispatcher)
  - Background processing: At least two for each R/3 System (more than one allowed for each dispatcher)
  - Enqueue: Only one enqueue work process is needed for each system

# Locks in R/3 at the Business Process Level



- The lock mechanisms in today’s relational database systems are usually not able to handle business data objects (such as customer orders) that affect several database tables. To coordinate several applications simultaneously accessing the same business object, the SAP System provides its own lock management, controlled by the enqueue work process.
- In order for the system to execute lock requests, you must first define a lock object in the ABAP Dictionary. The lock object contains tables whose entries are to be locked. A lock object consists of a primary table. You can also define additional secondary tables using foreign key relationships (the name of a user-defined lock object must begin with "EY" or "EZ").
- You can specify the lock mode ("S”: shared lock or "E”: exclusive lock) for a lock object. An exclusive lock (mode "E") can only be set if no other user has set a lock (“E” or “S”) on the data record. The same user can request additional "E" or "S" locks within a program call sequence (call chain).
- If a lock object is activated, the system generates an ENQUEUE and a DEQUEUE function module. These function modules are called ENQUEUE_<object_name> and DEQUEUE_<object_name> and are used in ABAP code to lock and unlock data.

## Asynchronous Update



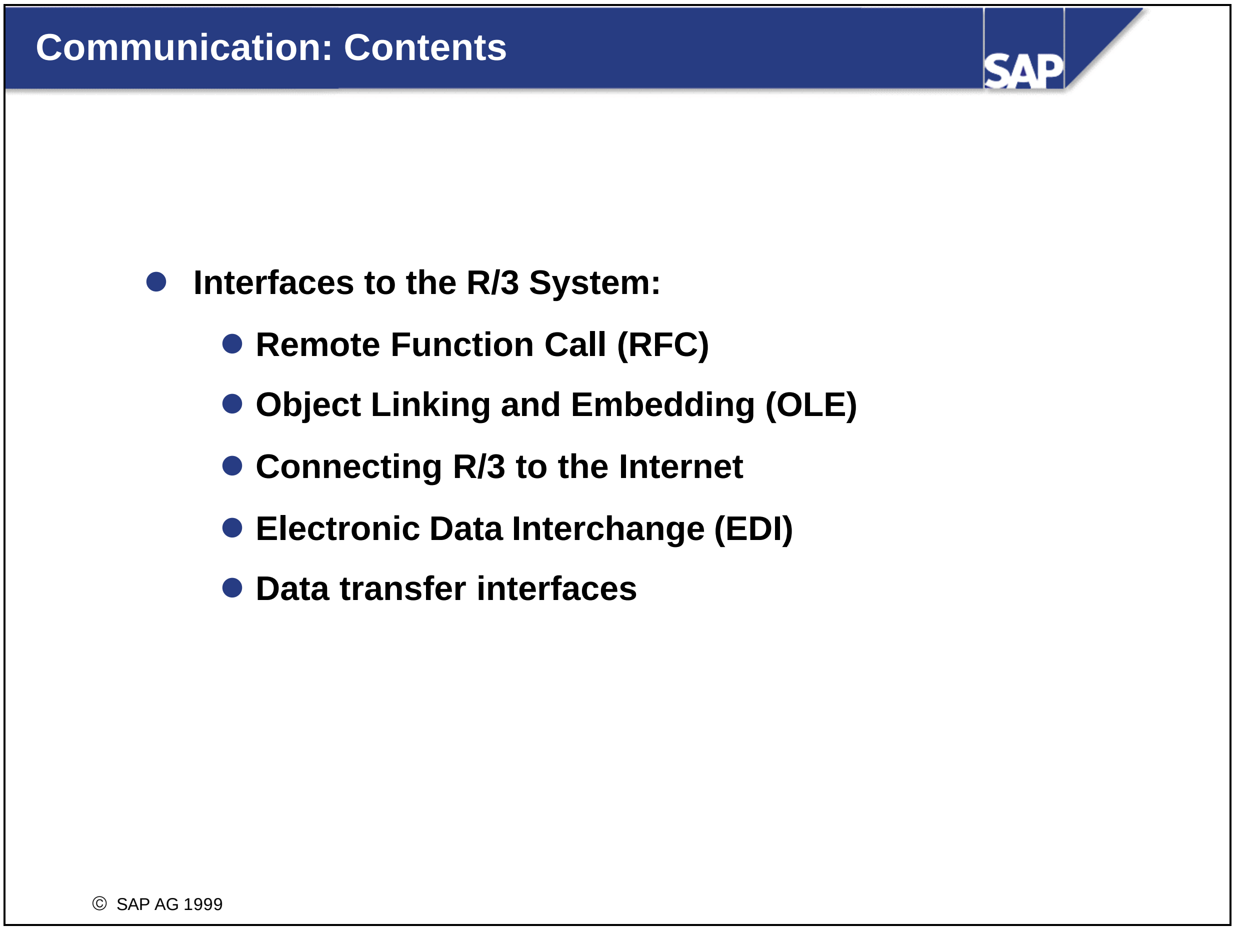
- A transaction corresponds to a logical unit of work (LUW).
- However, as today’s database systems do not support cross-process transaction flow, we must differentiate between the elementary processing steps (LUWs) in the SAP System and those in the database system (SAP - LUW / DB - LUW). A DB - LUW is either committed or not updated (rollback). The DB - LUW moves the database from one consistent state to the next. This means that the data must be logical and correct before as well as after the LUW; this applies to both DB - LUW and SAP - LUW.
- The start of an SAP transaction is also the start of an SAP - LUW. SAP - LUWs are completed either by a "COMMIT WORK" in the ABAP code, or by the completion of the corresponding asynchronous update (second part of the SAP - LUW). As explained previously, each dialog step in an SAP - LUW is processed by one work process, as is the case for the DB - LUW. Each database change is executed in its own DB-LUW.
- The asynchronous updating usually used in an SAP - LUW allows the system to temporarily collect changes made by users and then, at the end of the dialog phase (in the second part of the SAP - LUW), make the necessary changes to the database in a separate update work process. To ensure data consistency, the resulting database change (which includes every “dialog step change”) is executed in only *one* final DB - LUW.

- Dialog work processes should not be loaded down with long-running dialog steps, as these work processes would then not be available to other users. The remaining dialog work processes would have to handle many more users, thus considerably increasing response times.
- This is the reason for the parameter rdisp/max_wprun_time (default setting: 300 seconds), which sets the maximum time a dialog step is allowed to remain in a dialog work process. If this time is exceeded by more than double, the dialog step is terminated and the started transaction terminates with an error. This allows the administrator to ensure that users execute long-running actions only in the background work processes, which are designed for these types of long-running actions.

- Background work processes are designed for periodic tasks such as reorganization or the automatic transfer of data from an external system to the R/3 System.
- Background processing is scheduled in the form of jobs. Each job consists of one or more steps (ABAP reports, external programs, or other operating system calls), that are processed sequentially. You can also set priorities (from "C" to "A") so that certain jobs are prioritized.
- Job processing is not generally triggered immediately (immediate start). Instead you specify a start date and time when you schedule the job. It may also be necessary to start jobs periodically, for example, system control jobs repeated on a fixed cycle. Using the program SAPEVT, you can trigger a job start at the operating system level.
- The background scheduler is responsible for automatically triggering the job at the specified time. The background scheduler is an ABAP program that regularly looks in the scheduling table for jobs to be executed and then ensures that they are executed (RDISP/BTCTIME, default 60 s).

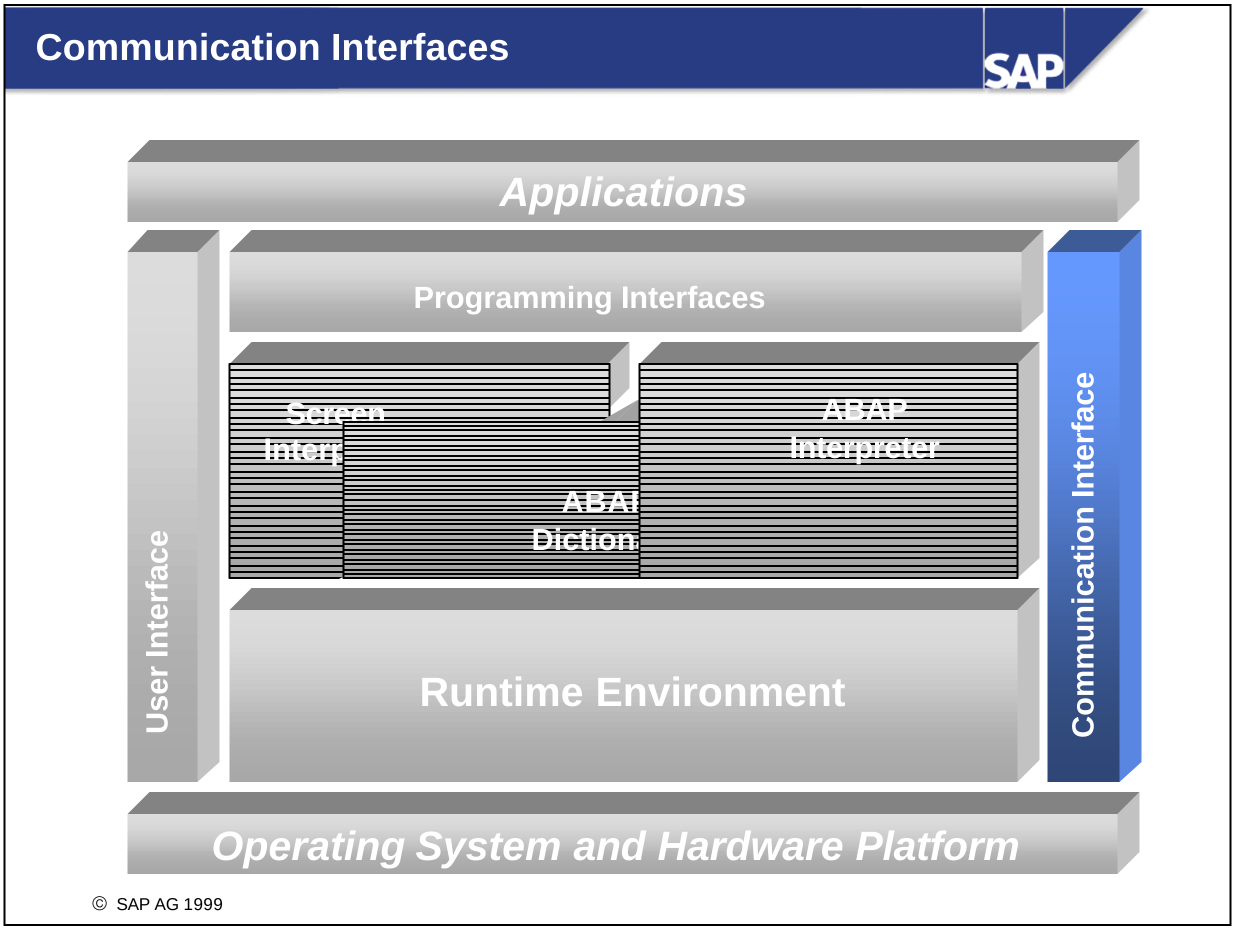
- Spooling refers to the buffered transfer of data to output devices such as printers, fax devices, and so on. In distributed systems, networked administration is necessary for this output.
- The R/3 System spool mechanism can supply print requests to printers and external spoolers both within a local network as well as across wide-area networks (WANs). The spool mechanism works with the local spool system on each server.
- Spool requests are generated in dialog mode or during background processing and are then set in the spool database with details about the printer and the print format. The data itself is stored in the TemSe (TEMporary SEquential object) database.
- When data is to be printed, a print request is generated for a spool request. This print request is processed by a spool work process.
- Once the spool work process has formatted the data for output, it returns the print request to the operating system spool system.
- The operating system spool takes over queue management and ensures that the required data is passed to the output device.

- An instance is an administrative unit that combines R/3 System components providing one or more services. The services provided by an instance are started or stopped together. You use a common instance profile to set parameters for all the components of an instance.
- A central R/3 System consists of a single instance providing all the necessary R/3 System services.
- Each instance has its own SAP buffer areas.
- The example illustrates how an additional background processing server (a) and dialog server (b) are set up. These instances, which provide specific services, generally run on separate servers, but can also run on the same server, if needed.
- The message server provides the application servers with a central message service for internal communication (for example, trigger update, request and remove locks, trigger background requests).
- The dispatchers for the individual application servers communicate through the message server, which is installed once in each R/3 System (it is configured in the R/3 System profile files).
- Presentation servers can also log on to an application server through the message server. This means that you can use the message server performance database for automatic load distribution (logon load balancing).



## Communication: Contents

- Interfaces to the R/3 System:
  - Remote Function Call (RFC)
  - Object Linking and Embedding (OLE)
  - Connecting R/3 to the Internet
  - Electronic Data Interchange (EDI)
  - Data transfer interfaces



## Communication Interfaces

Applications

User Interface

Programming Interfaces

Screen Interp

ABAP Diction

ABAP Interpreter

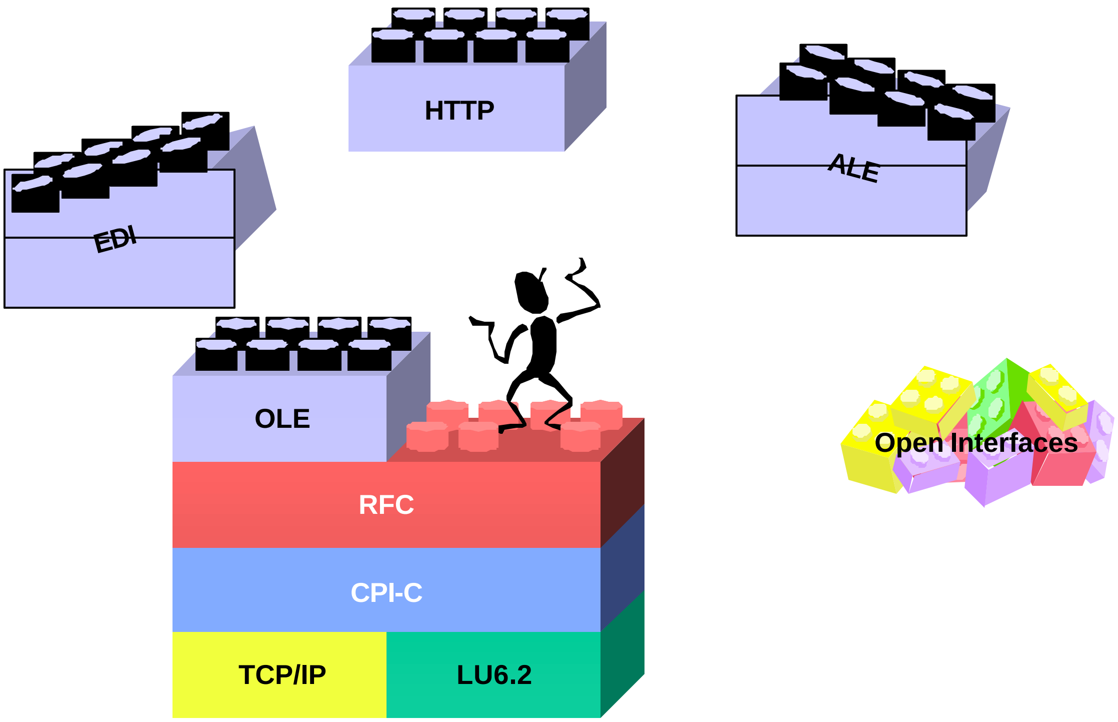
Runtime Environment

Communication Interface

Operating System and Hardware Platform

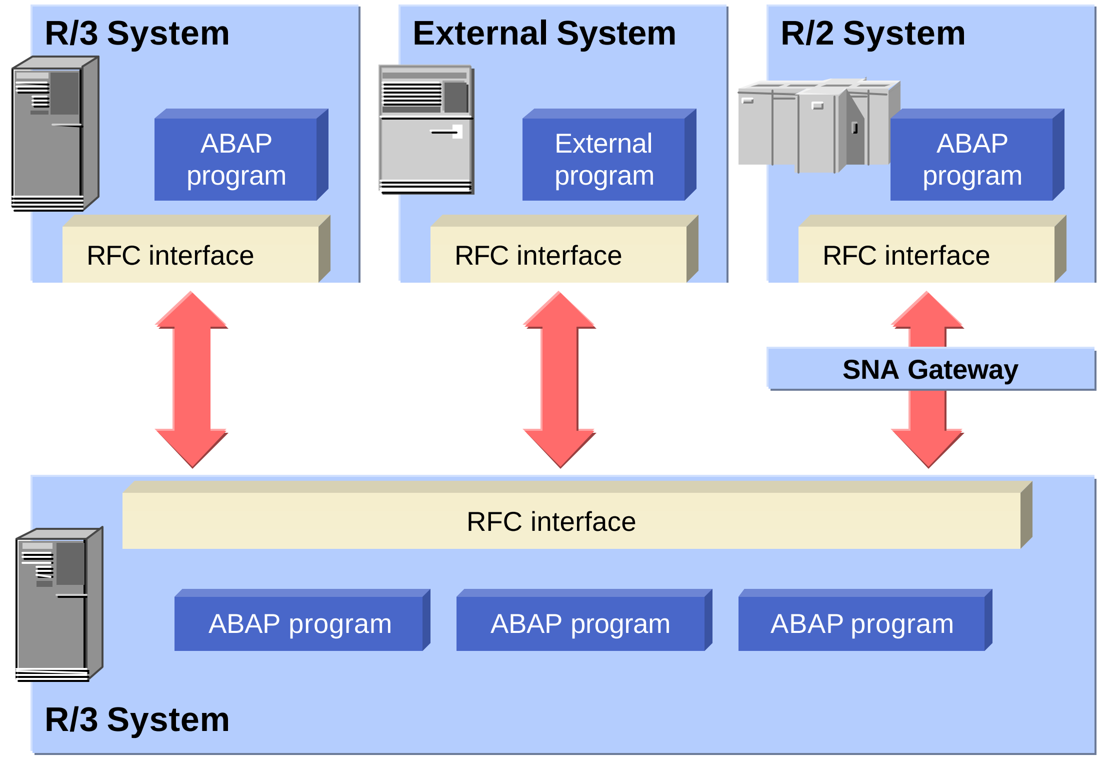
- The R/3 System ensures portability by using industry-standard interfaces that support the interaction of applications, data, and user interfaces. The system can interact with various operating systems, databases, and networks. The R/3 System uses open industry standards, such as TCP/IP, EDI, OLE, and open interfaces.

## Communication: R/3 is an Open System



- The R/3 System is an **open system**. It supports a variety of network communication protocols. Information can be exchanged between R/3 Systems and other R/3, R2, or non-SAP systems across a network.
- SAP supports the Transmission Control Protocol / Internet Protocol (TCP/IP) and System Network Architecture: Logical Unit 6.2 (SNA LU6.2) protocols. Communication within the R/3 System uses the standard protocol TCP/IP. LU6.2 was developed by IBM and is used to communicate with mainframe-based R/2 Systems.
- R/3 application programming supports the following communication interfaces: common programming interface communication (CPI-C), remote function call (RFC), and object linking and embedding (OLE) automation.
- For more information about communication, see the online documentation. You can also order an “Interface Adviser” Knowledge CD from SAP that uses many practical examples to explain communication in the R/3 System. SAPNet also contains additional information, such as under the alias /*int-adviser*.

# Remote Function Call



R/3 System

ABAP program

RFC interface

External System

External program

RFC interface

R/2 System

ABAP program

RFC interface

SNA Gateway

RFC interface

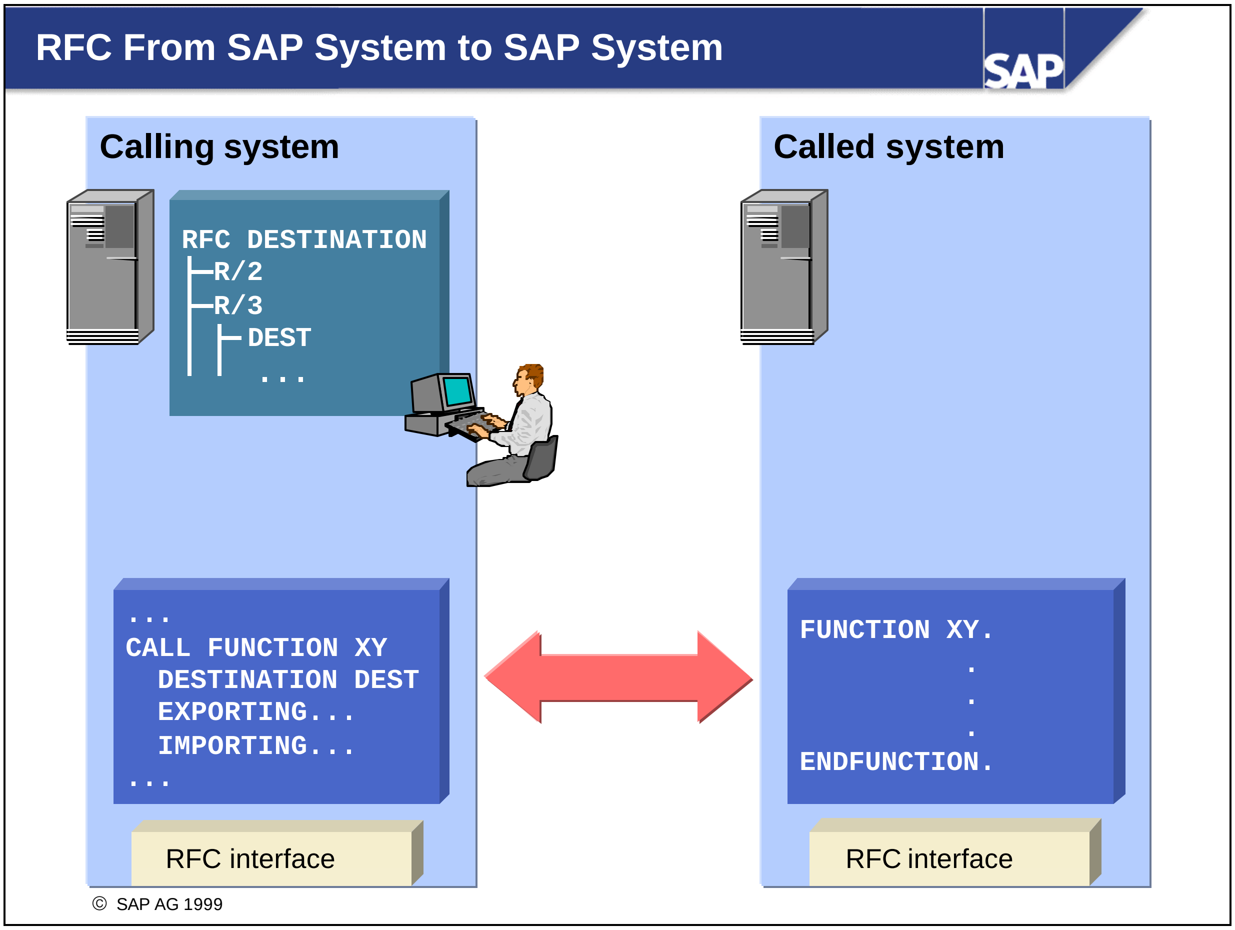
ABAP program

ABAP program

ABAP program

R/3 System

- Remote Function Call (RFC) is a communications interface based on CPI-C, but with more functions and easier for application programmers to use. You can use R/3 and R/2 Systems as well as external applications as RFC communication partners.
- For communicating with R/2 Systems, additional software (SNA gateway) is required on at least one applic ation server. See also R/3 Note 13903.
- RFC is the protocol for calling special subroutines (function modules) over the network. Function modules are comparable with C functions or PASCAL procedures. They have a defined interface through which data, tables and return codes can be exchanged. Function modules are managed in the R/3 System in their own function library, called the Function Builder.
- The Function Builder (transaction SM37) provides application programmers with a useful environment for programming, documenting and testing function modules that can be called locally as well as remotely. The R/3 System automatically generates the additional code (RFC stub) needed for remote calls.
- You maintain the parameters for RFC connections using transaction SM59. The R/3 System is also delivered with an RFC-SDK (Software Development Kit) that uses extensive C libraries to allow external programs to be connected to the R/3 System.



## RFC From SAP System to SAP System

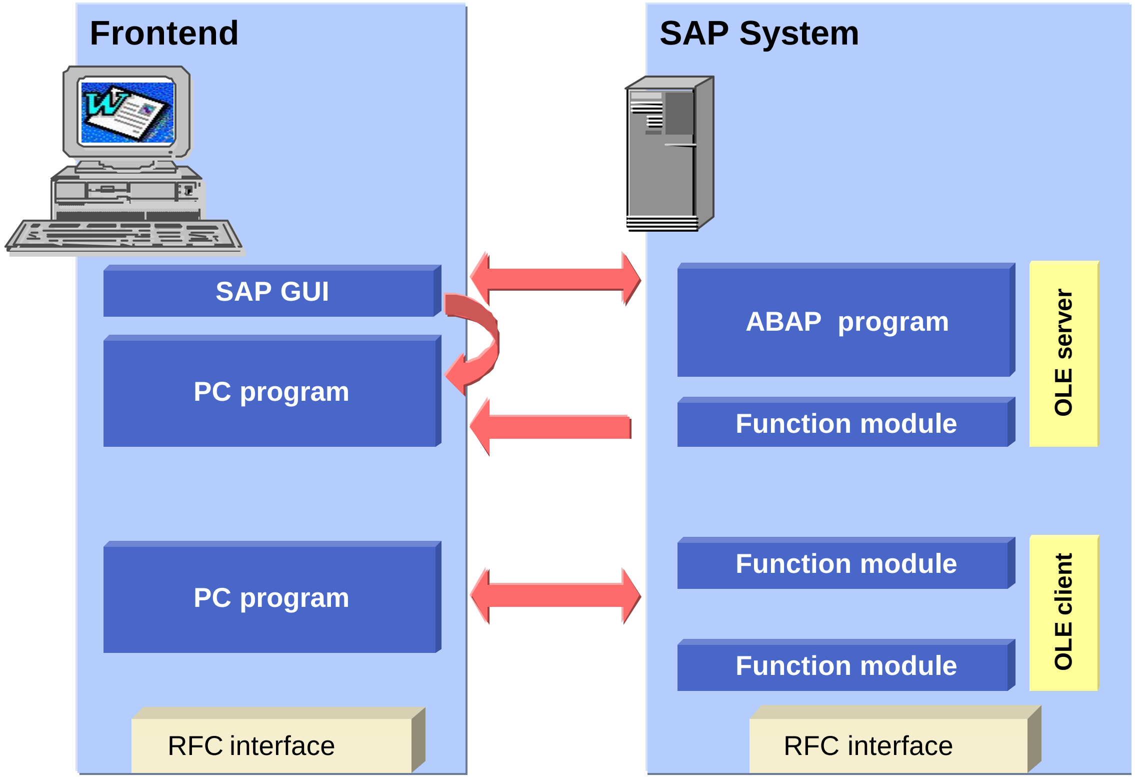
```
RFC DESTINATION
├R/2
├R/3
│├DEST
│  ...
```

```
...
CALL FUNCTION XY
  DESTINATION DEST
  EXPORTING...
  IMPORTING...
...
```

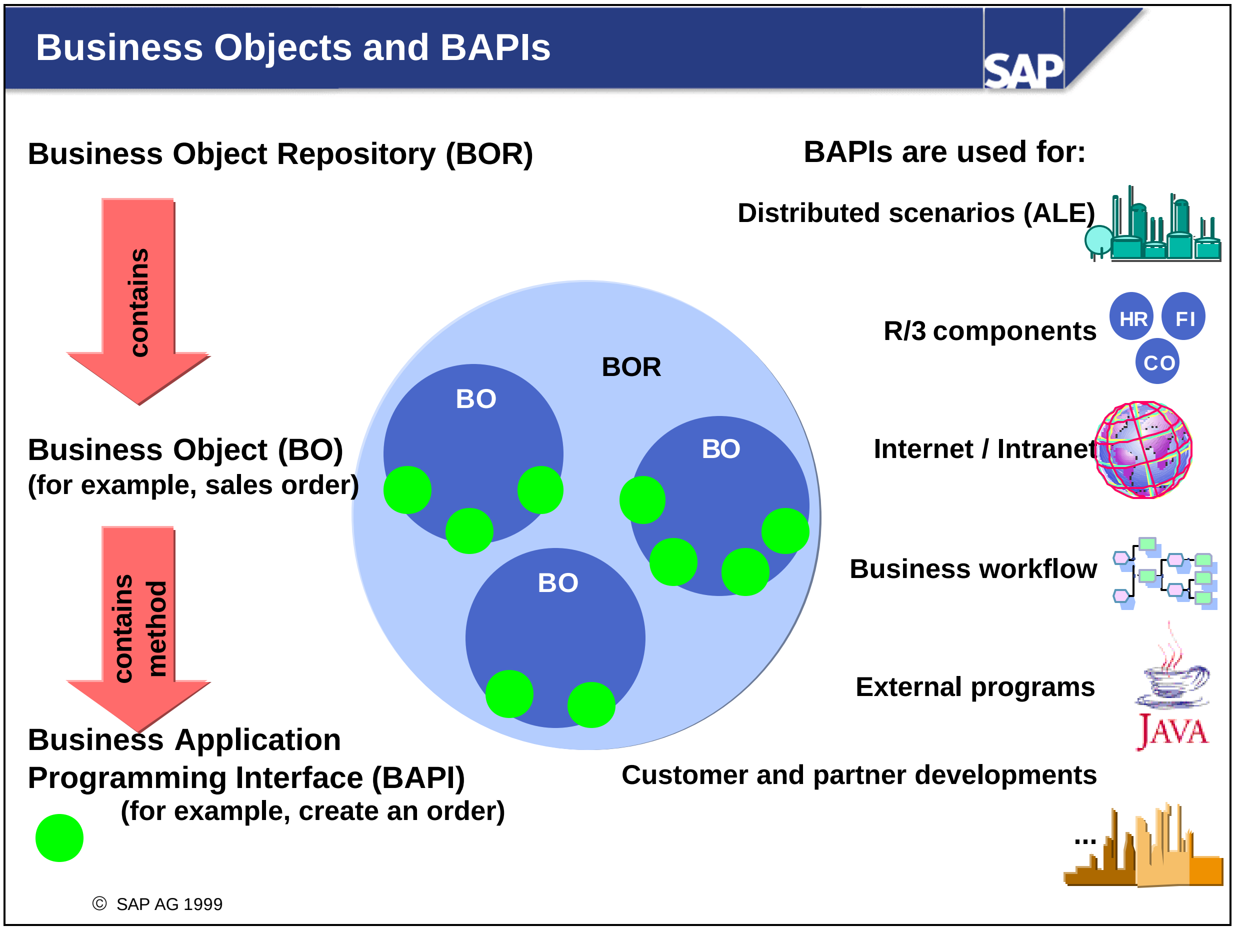
```
FUNCTION XY.
          .
          .
          .
ENDFUNCTION.
```

- The only difference between a remote call of a function module to another server and a local call is a special parameter (destination) that specifies the target server on which the program is to be executed.
- There are three types of RFC calls:
  - **Synchronous RFC call**: The calling program stops until the function module has been processed on the target server  and any results have been returned to the caller. Only then does the calling program continue processing.
  - **Asynchronous RFC call**: The calling program runs parallel to and independently of function module processing in the target system. Programmers are responsible for the processing of the results. In addition, the target system must also be available at the time of the RFC call.
  - **Transactional RFC call**: Several function modules can be grouped into one transaction. They are processed only once in the target system, within an LUW, and in the sequence in which they were called. In the case of an error, a message is sent to the calling system that you can analyze using transaction SM58. For transactional RFC, the target system does not have to be available at the time of the RFC call. In addition, you can configure the frequency and intervals of individual queries.

## Office Integration Using OLE



- Object linking and embedding (OLE) is an object-oriented method for program-to-program communication. You can connect **office applications** that support OLE2 automation (for example, Word and Excel) to the R/3 System. In this way, users can use the R/3 functions within their usual desktop environment.
- The office programs' OLE functions are specified in the R/3 System in the type information. This information contains a description of the methods, attributes and parameters. Type information can be language-independent.
- When using OLE, the R/3 System can play two separate roles:
  - If the R/3 System is acting as an **OLE client**, then the user calls the desktop program from the ABAP application. OLE commands are transferred from the ABAP code as remote function calls (RFC) through the SAP GUI to the PC. The SAP GUI maps RFC calls to OLE commands for the PC application.
  - If the R/3 System is acting as an **OLE server**, R/3 functions can be called from the desktop application. OLE commands are sent to the SAP automation server. The server converts them into RFC calls and passes them on to the R/3 System. In the R/3 System, function calls and BAPIs are triggered by business objects. After the data is processed successfully, the business object sends the data back to the desktop program through the SAP automation server.



## Business Objects and BAPIs

Business Object Repository (BOR)

Business Object (BO)
(for example, sales order)

Business Application Programming Interface (BAPI)
(for example, create an order)

BAPIs are used for:

Distributed scenarios (ALE)

R/3 components

Internet / Intranet

Business workflow

External programs

Customer and partner developments

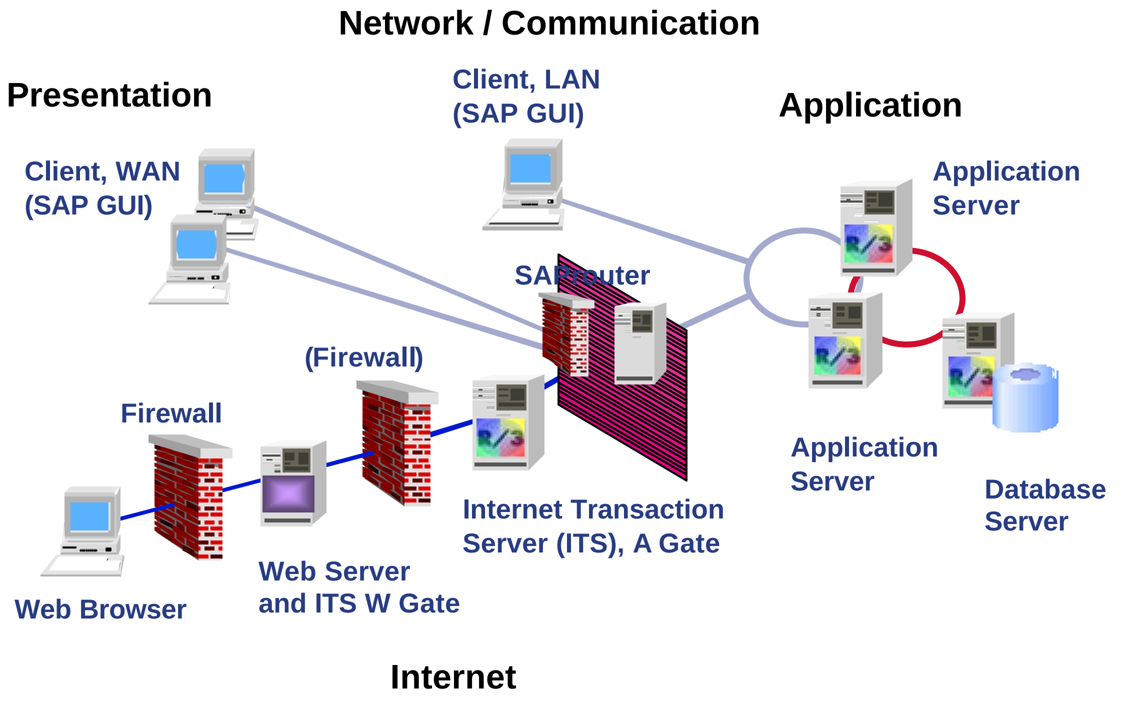
...

- **Business objects** form the basis for communicating on high (user-friendly) network layers. For example, they enable the R/3 System to support the Internet, and desktop programs to be connected. The goal of SAP's object-oriented strategy is to integrate objects at a business level rather than on a purely technical level.
- Business objects:
  - Form the basis of well-defined communication between client / server systems.
  - Are business-oriented: there are objects such as "customer", "order" or "employee".
  - Provide business functions (methods). For a "customer" object, for example, there are "Create customer" and "View customer" methods. These names support clear and therefore error-free programming.
  - Are managed centrally in the R/3 System in the Business Object Repository (BOR).
- Business Application Programming Interfaces (**BAPIs**) are functional interfaces. They use the business methods from the business objects. BAPIs may be addresses within or outside the R/3 System.
- For specifications and more information about BAPIs, see the alias "*bapi*" in SAPNet.

# Administration: Contents

- Security concepts in the R/3 System
- Important administration functions
- The Computing Center Management System (CCMS)
- SAPNet and SAPNet - R/3 Frontend

# Security in the R/3 System



Network / Communication

Presentation

Client, LAN
(SAP GUI)

Application

Client, WAN
(SAP GUI)

Application
Server

SAProuter

(Firewall)

Firewall

Application
Server

Database
Server

Internet Transaction
Server (ITS), A Gate

Web Server
and ITS W Gate

Web Browser

Internet

R/3 System technology integrates security mechanisms on several levels:

•Presentation: The SAP GUI software uses check sums to check for integrity each time the R/3 frontend is started. This also recognizes any computer viruses.

•Network / Communication: A firewall and the SAProuter protect the internal network. You can also use additional security mechanisms by integrating external security products such as SECUDE (or Kerberos).

•Application: The authorization concept prevents unauthorized access to data and transactions. Users must authenticate themselves using their user ID and password. The lock mechanism within the R/3 System also prevents users from making changes to the same data simultaneously.

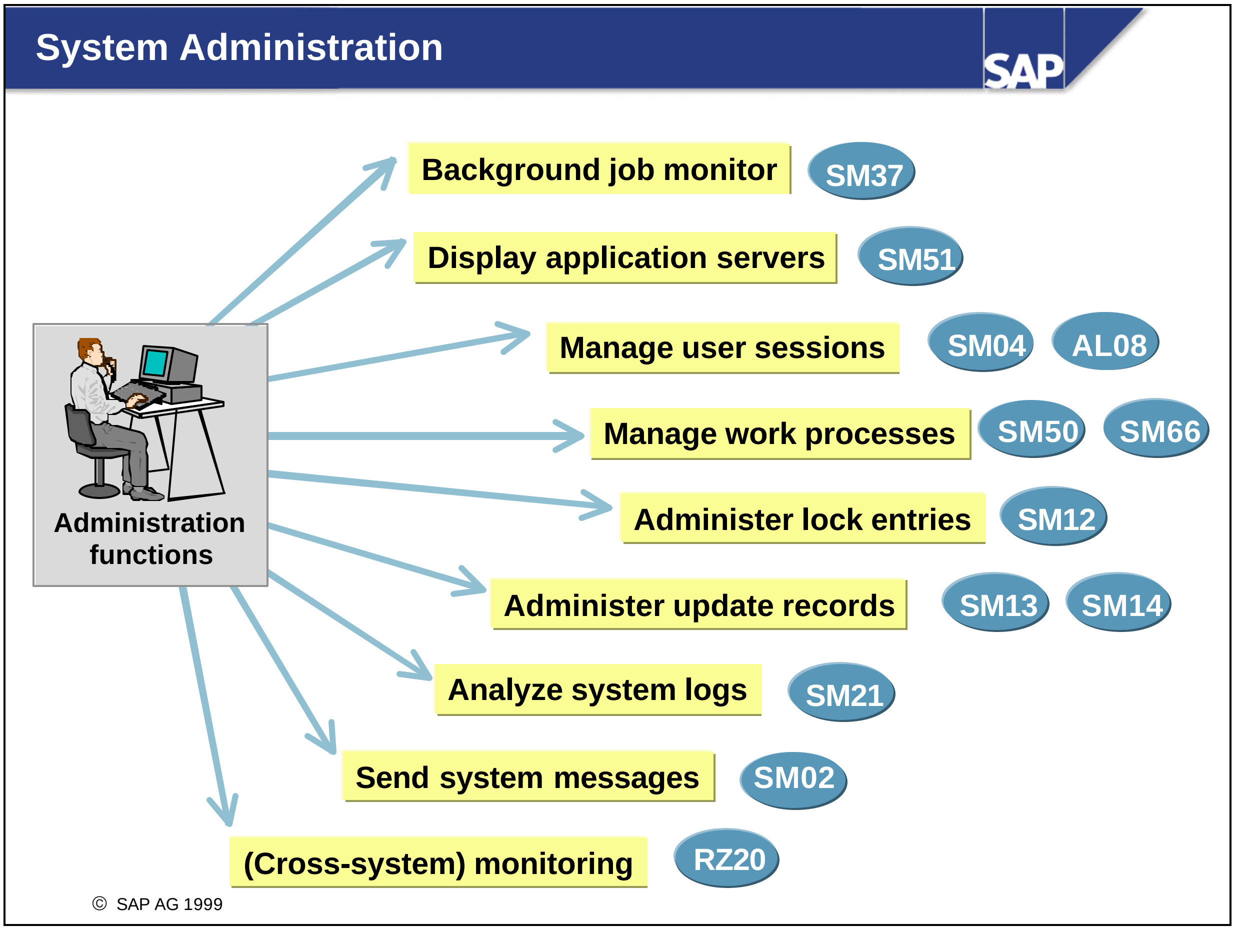
•Internet: The R/3 System supports current Internet security standards, such as HTTPS.

•Database: Only database administrators can access data in the R/3 database from outside the R/3 System. The database manufacturer's security mechanism is active here.

•Passwords: Preconfigured users exist in clients 000 and 001 after the R/3 System has been installed. These default users, DDIC and SAP*, have comprehensive authorizations. You should, therefore, change their initial passwords as soon as possible.

•For more information about security, see the SAP Notes, the installation guide, the online documentation, and the security guide.

# System Administration



Administration functions

Background job monitor SM37

Display application servers SM51

Manage user sessions SM04 AL08

Manage work processes SM50 SM66

Administer lock entries SM12

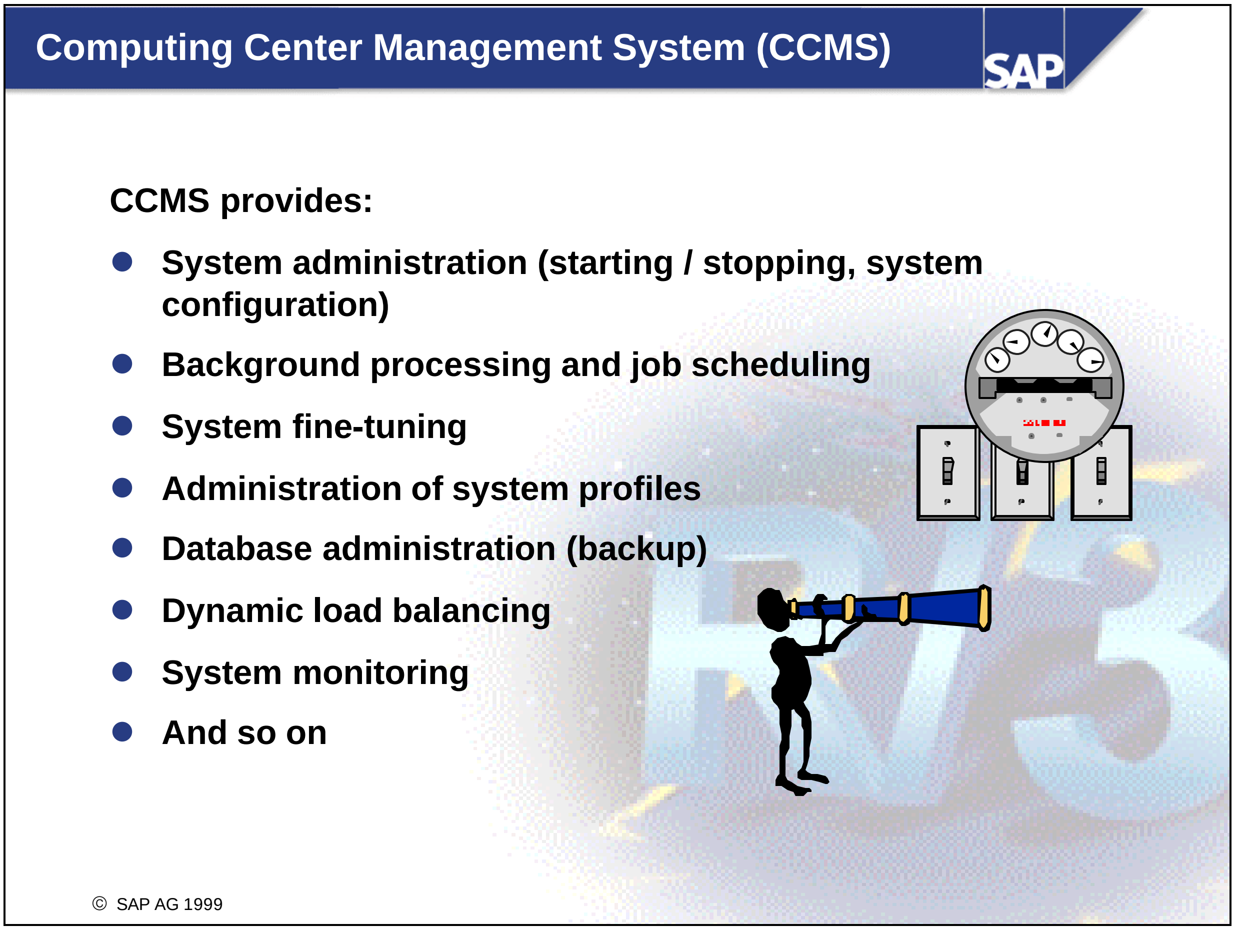
Administer update records SM13 SM14

Analyze system logs SM21

Send system messages SM02

(Cross-system) monitoring RZ20

- The R/3 System provides system administrators with a number of powerful tools to perform their daily tasks.
  You can find the following functions in some of the transactions listed above:
  - Display server, user, work process, and background job overviews
  - Manage locks and updates
  - Lock transaction codes
  - Create system messages
  - Monitor system and cross-system components (see related information)



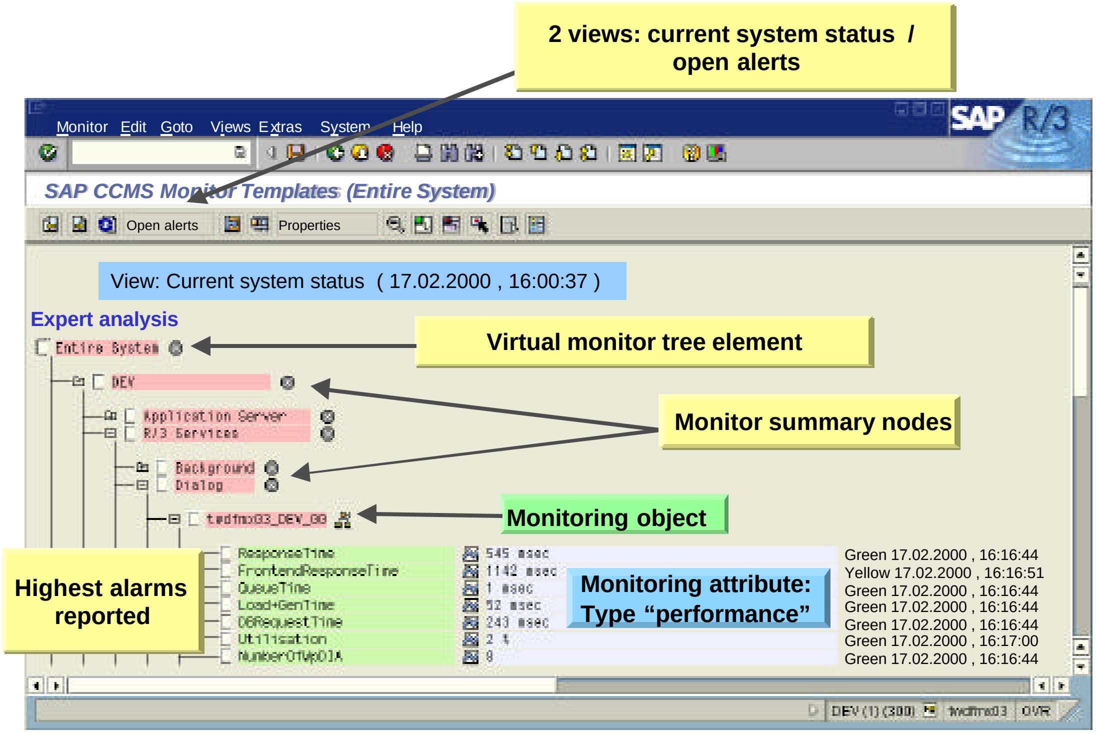
## Computing Center Management System (CCMS)

CCMS provides:

- System administration (starting / stopping, system configuration)
- Background processing and job scheduling
- System fine-tuning
- Administration of system profiles
- Database administration (backup)
- Dynamic load balancing
- System monitoring
- And so on

- Using the Computing Center Management System (CCMS) you can monitor, control and configure an R/3 System. You can use the tools to analyze system load and determine the resource consumption of various system components, among other tasks.
- CCMS provides you with a number of graphical monitors and administration functions:
  - Starting and stopping the R/3 System
  - R/3 System monitoring and analysis
  - Automatic reporting of system alerts
  - Dynamic user distribution
  - R/3 System configuration: Editing system profiles (not authorization profiles)
  - Processing and controlling background jobs, scheduling database backups

# Monitoring Architecture



2 views: current system status / open alerts

SAP CCMS Monitor Templates (Entire System)

View: Current system status ( 17.02.2000 , 16:00:37 )

Expert analysis

```
Entire System
  DEV
    Application Server
    R/3 Services
      Background
      Dialog
        twdfmx03_DEV_00
          ResponseTime             545 msec    Green 17.02.2000 , 16:16:44
          FrontendResponseTime     1142 msec   Yellow 17.02.2000 , 16:16:51
          QueueTime                1 msec      Green 17.02.2000 , 16:16:44
          Load+GenTime             52 msec     Green 17.02.2000 , 16:16:44
          DBRequestTime            243 msec    Green 17.02.2000 , 16:16:44
          Utilisation              2 %         Green 17.02.2000 , 16:17:00
          NumberOfWpDIA            8           Green 17.02.2000 , 16:16:44
```

Virtual monitor tree element

Monitor summary nodes

Monitoring object

Highest alarms reported

Monitoring attribute: Type “performance”

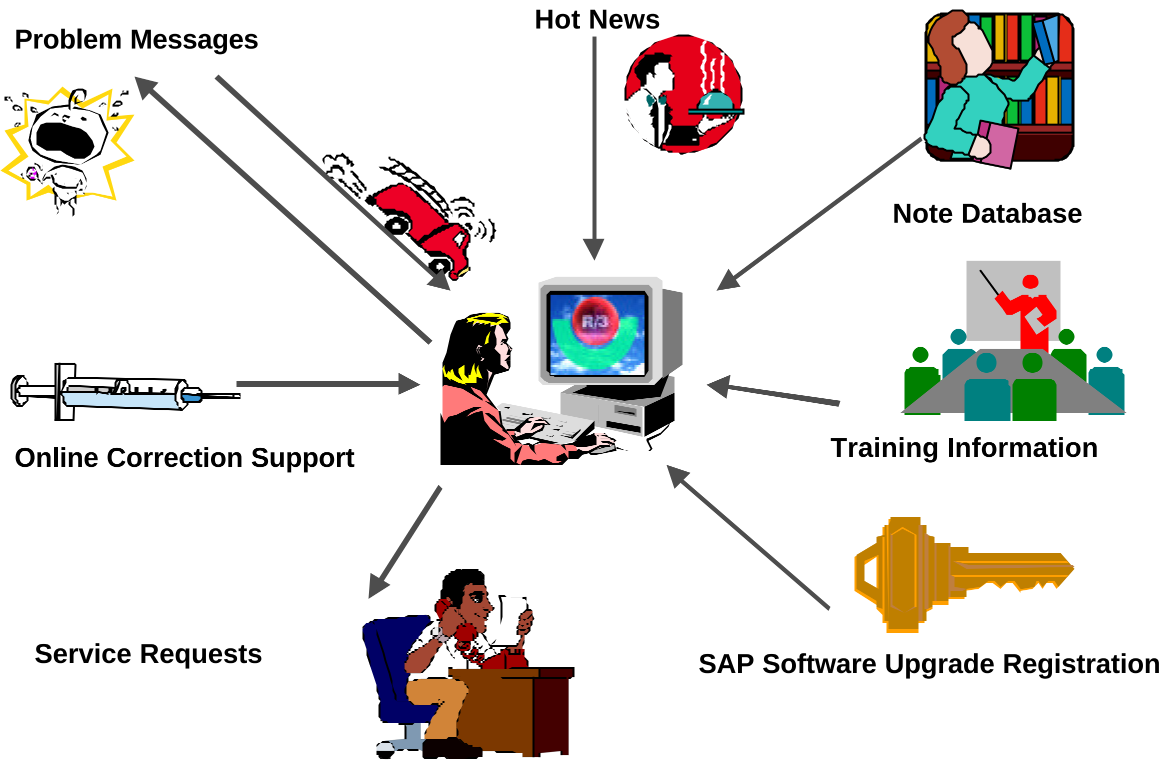
- Transaction RZ20 provides a system monitoring structure allowing centralized monitoring of many system parameters and includes links to other analysis tools.
- Open interfaces allow the incorporation of other system monitoring tools (including non-SAP tools). Several R/3 Systems can be monitored provided an RFC connection with the other system is possible and configured.
- You can create your own system monitor views. These can be used to provide specific people with only those alerts they are interested in.
- All threshold values can be easily changed.
- The average dialog response time in the last 15 minutes is an example of a typical monitoring attribute.

## Remote Services Provided by SAP

- ➔ SAPNet (incl.SAP Note database)
- ➔ GoingLive and EarlyWatch services
- ➔ Remote consulting
- ➔ Other services

- SAPNet - R/3 Frontend
  - SAPNet - R/3 Frontend provides SAP’s extensive database of notes, which users can consult if they have any questions or if any problems occur, before they create a problem message.
  - In SAPNet you can find current messages, documentation, tools (QuickSizer) and much more. You can also use the discussion forums to offer and search for information. Our goal is to make the vast store of customer knowledge available to a wide audience.
- GoingLive and EarlyWatch:
  - The GoingLive check occurs shortly before an R/3 System is used in production. This test checks that the system meets the requirements set. An EarlyWatch session recognizes performance bottlenecks in an R/3 System before they become a problem, and proposes suitable solutions.
- Remote consulting:
  - During a remote consulting session, an SAP consultant accesses your SAP System at a time you specify and attempts to analyze and solve the problem in your system from their workstation.
- Additional services: Remote upgrade, remote archiving, conversion, migration, security, and euro services.

## SAPNet



Problem Messages

Hot News

Note Database

Online Correction Support

Training Information

Service Requests

SAP Software Upgrade Registration

In SAPNet, you can:

- Write system problem messages to SAP
- Search the Note database for help
- Read SAP HotNews, which contains information about Support Packages or new SAPNet functions
- Request developer keys for developers and for SAP standard objects
- Use Support Packages to install corrections in your R/3 System
- Find up-to-date SAP training information
- Allow an SAP employee to dial in to your system for fast problem solving (through a service connection).

# Administration: Unit Summary

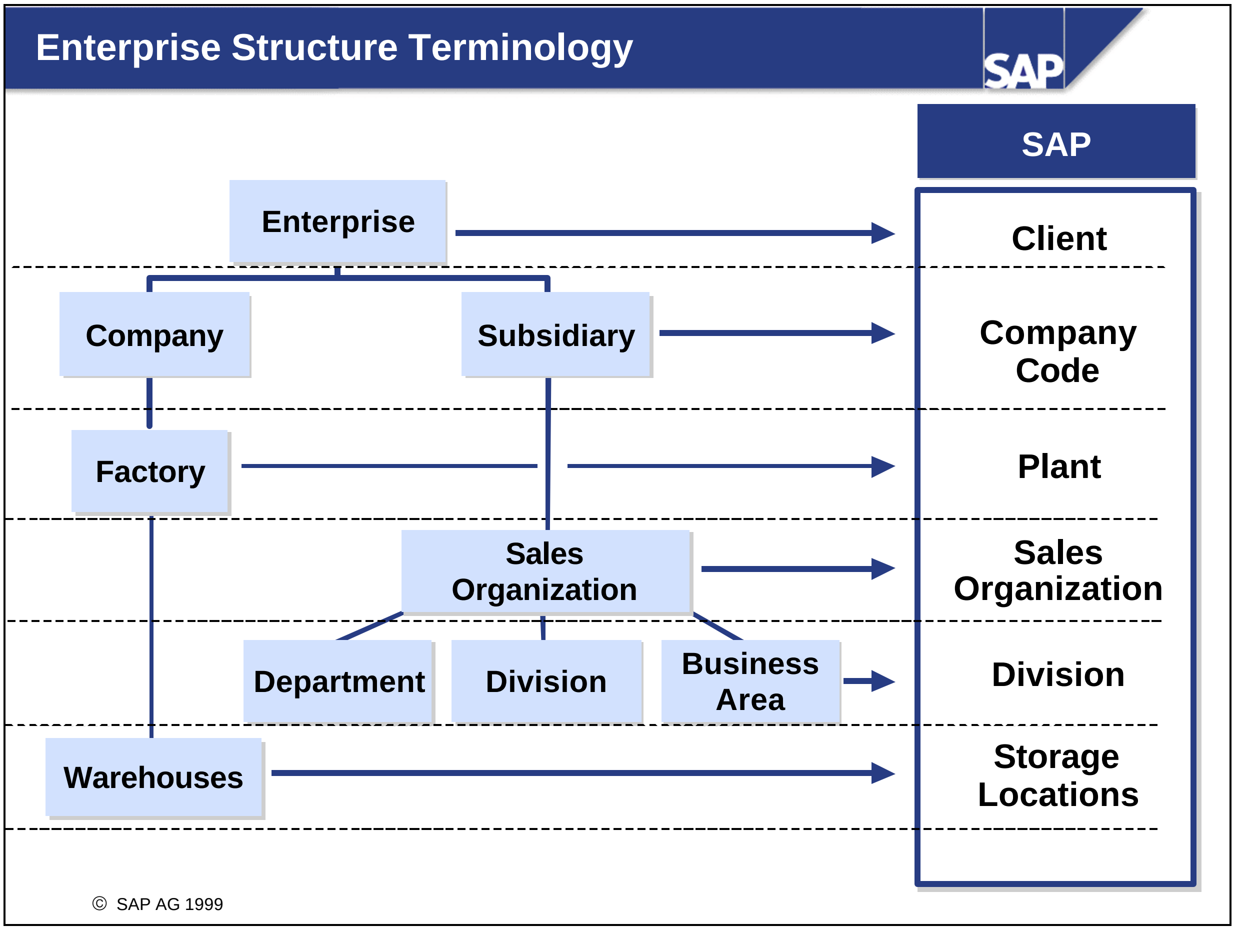
You are now able to:

- Name some of the security aspects of the R/3 System
- Name some basic administration functions
- Use SAPNet as an information source

# System-wide Concepts

## Contents:

- Organizational Units and Master Data
- Transactions and Documents
- Authorizations
- Analysis and Reports



## Enterprise Structure Terminology

SAP

Enterprise → Client

Company

Subsidiary → Company Code

Factory → Plant

Sales Organization → Sales Organization

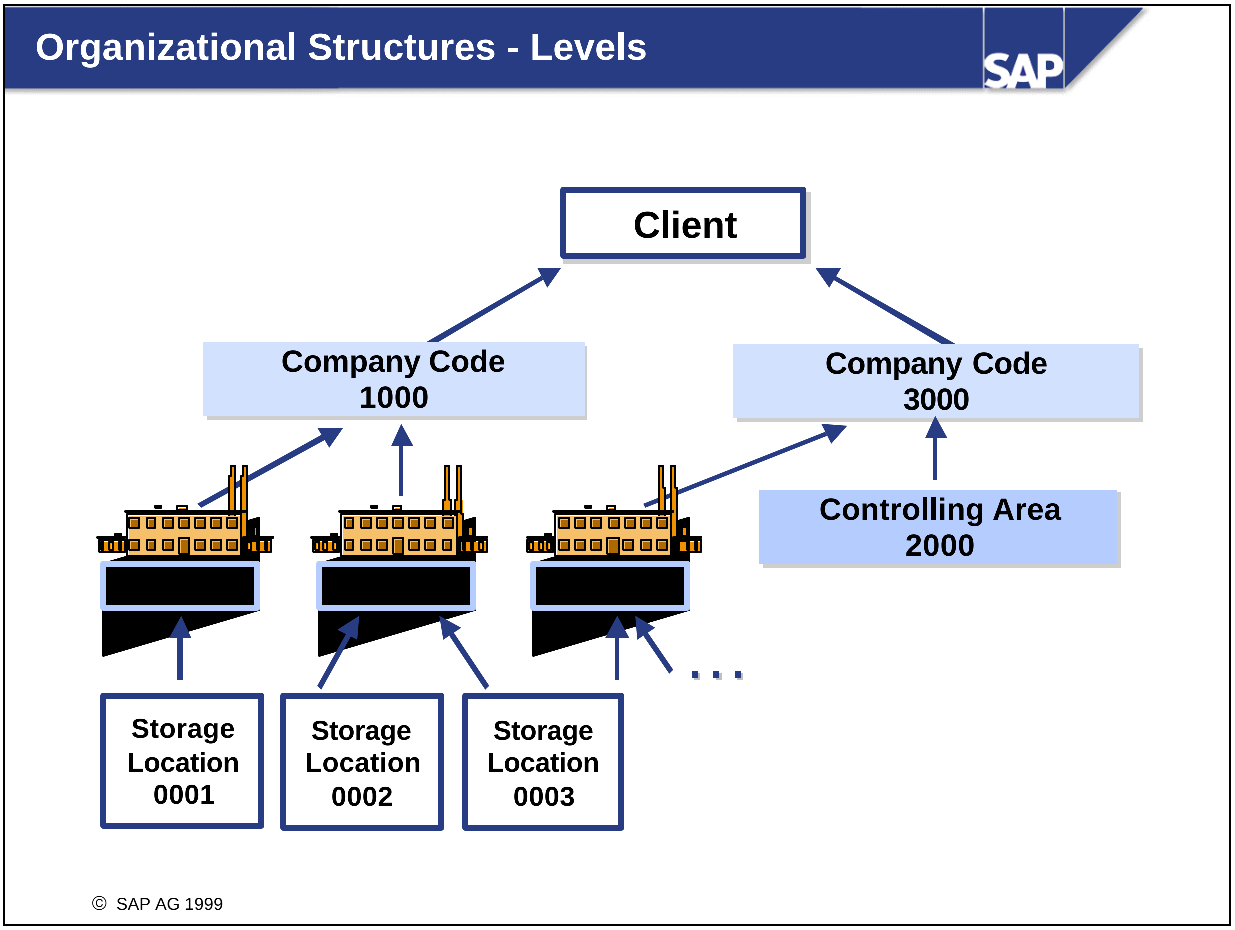
Department

Division

Business Area → Division

Warehouses → Storage Locations

- An enterprise structure is mapped to SAP applications using organizational units. Organizational units handle specific business functions.
- Organizational units may be assigned to a single application (such as a sales organization assigned to Sales and Distribution, or to several applications (such as a plant assigned to Materials Management and Production Planning).



## Organizational Structures - Levels

Client

Company Code 1000

Company Code 3000

Controlling Area 2000

Storage Location 0001

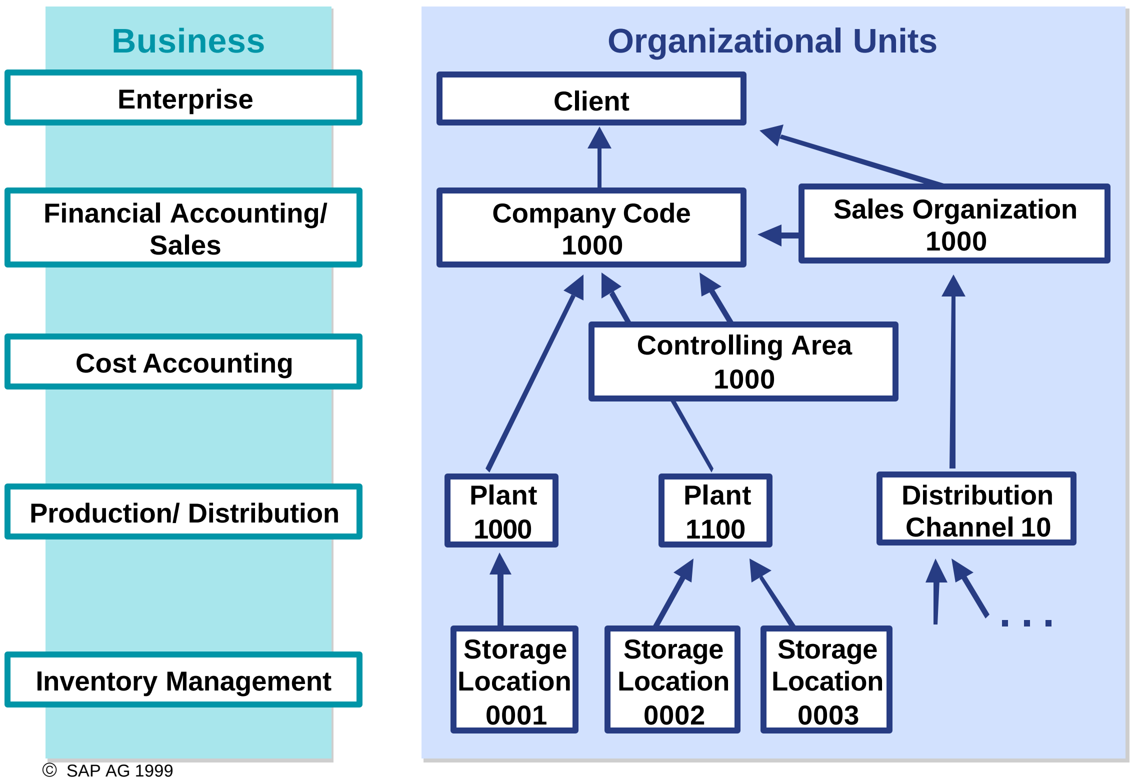
Storage Location 0002

Storage Location 0003

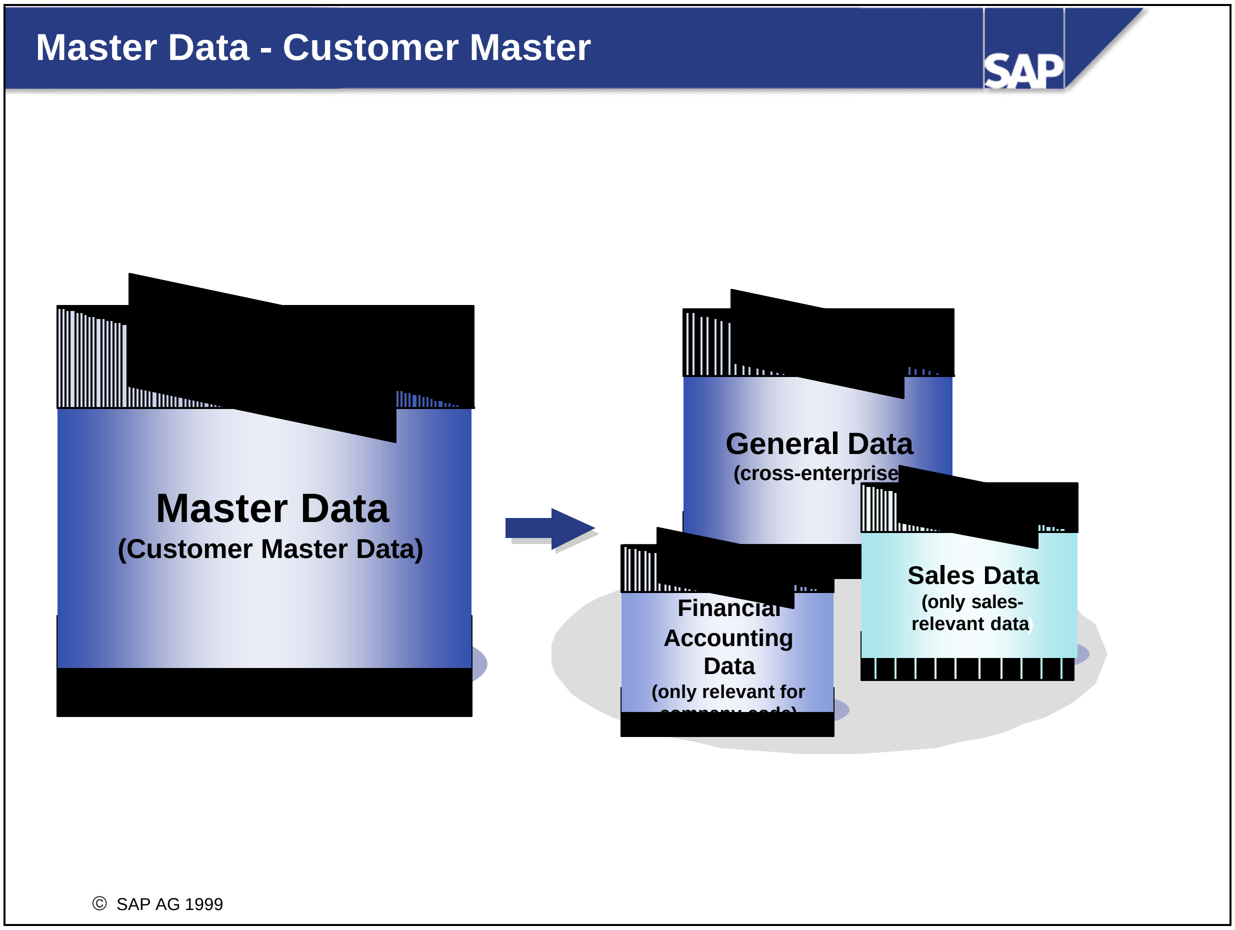
. . .

- **The highest-level element of all organizational units is the client.** The client can be an enterprise group with several subsidiaries. All of the enterprise data in an R/3 System implementation is split into at least the client area, and usually into lower level organizational structures as well.
- Flexible organizational units in the R/3 System enable more complex enterprise structures to be represented. If there are many organizational units, the legal and organizational structure of an enterprise can be presented in different views.
- By linking the organizational units, the separate enterprise areas can be integrated and the structure of the whole enterprise represented in the R/3 System.

## Organizational Structures - Business Functions

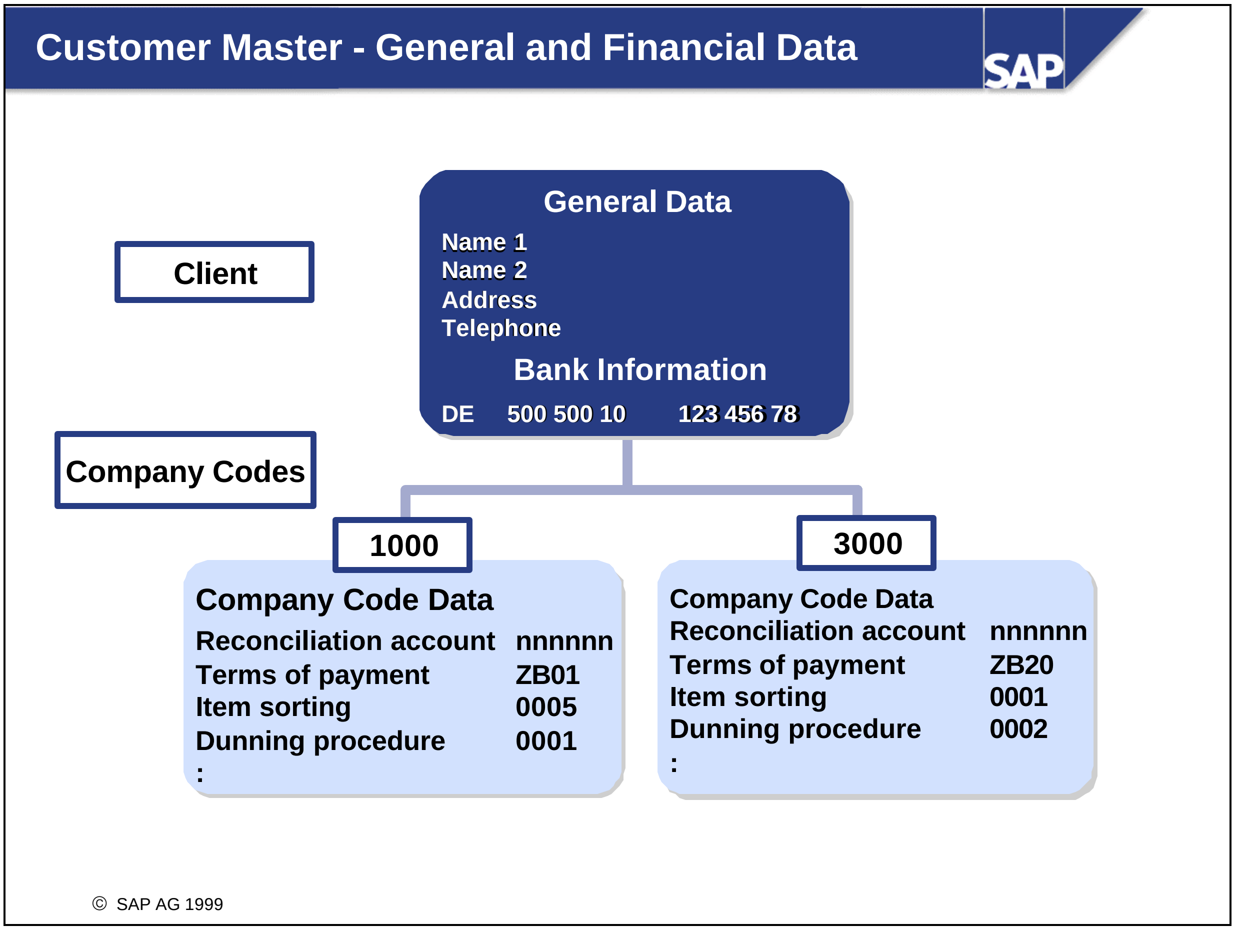


- An enterprise is structured in the SAP R/3 System according to business functions that must correspond to the functionality assigned to the organizational units.
- Examples:
  - A Company Code is a unit included in the balance sheet of a legally-independent enterprise. It is the central organizational element of Financial Accounting.
  - The Controlling Area is the business unit where Cost Accounting is carried out. Usually there is a 1:1 relationship between the controlling area and the company code. For the purpose of company-wide cost accounting, one controlling area can handle cost accounting for several company codes in one enterprise.
  - In the context of Sales and Distribution, the Sales Organization is central organizational element that controls the terms of sale to the customer. Distribution Channel is the element that describes through what channel goods and/or services will be distributed to the customer.
  - In the context of Production Planning and Control, the Plant is the central organizational unit. A plant is the place of production or simply a collection of several locations of material stocks in close physical proximity.
  - A Storage Location is a storage area comprising warehouses in close proximity. Material stocks can be differentiated within one plant according to storage location (inventory management).



## Master Data - Customer Master

- Data records that remain in the database for a long period of time are called master data. Master data includes creditors, vendors, materials, accounts, and so on.
- Master data is created centrally and can be used in all applications.
  Example:
  - A customer is master data that can be used in customer requests, deliveries, invoices, and payments.
- Master data also has an organizational aspect as it is assigned to organizational units.
- Master data has cross-component usage
  Examples:
  - Customer master data uses the same data for financial accounting and sales
  - Customer master records can be assigned to the following organizational units:
    - company code
    - sales organization
    - distribution channel
    - division



## Customer Master - General and Financial Data

Client

General Data

Name 1
Name 2
Address
Telephone

Bank Information

| | | |
|---|---|---|
| DE | 500 500 10 | 123 456 78 |

Company Codes

1000

Company Code Data

| | |
|---|---|
| Reconciliation account | nnnnnn |
| Terms of payment | ZB01 |
| Item sorting | 0005 |
| Dunning procedure | 0001 |
| : | |

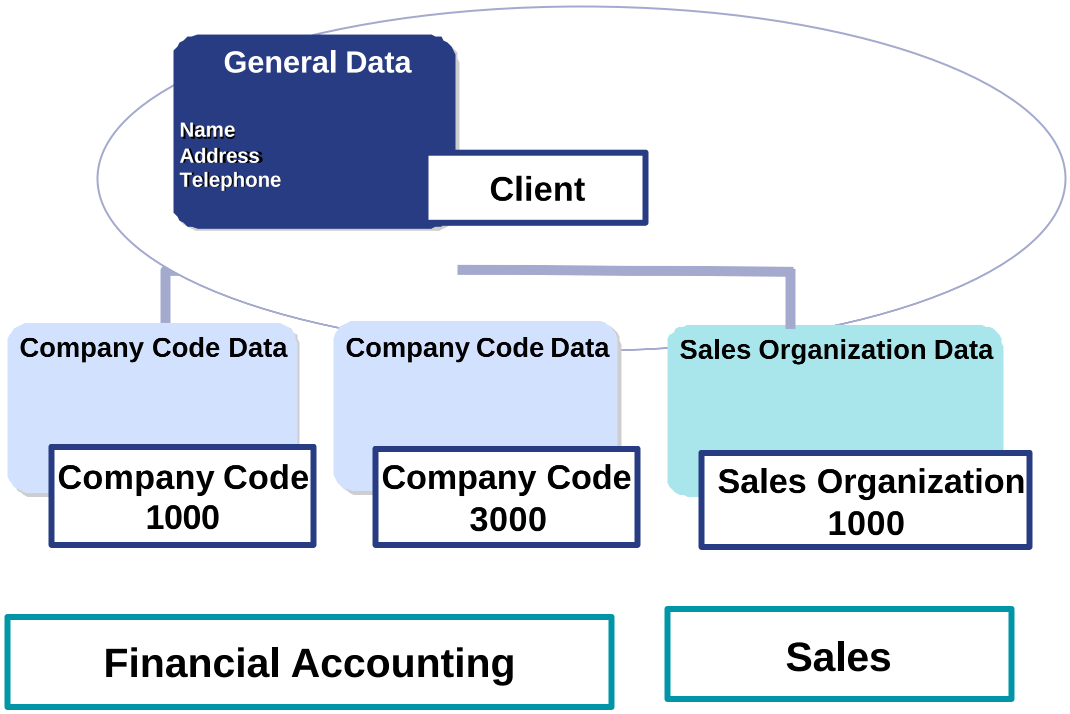
3000

Company Code Data

| | |
|---|---|
| Reconciliation account | nnnnnn |
| Terms of payment | ZB20 |
| Item sorting | 0001 |
| Dunning procedure | 0002 |
| : | |

- When creating a customer master record, you enter:
  - Shared data on the client level
  - Company code-specific data for each company code
- Data on the client level can be used by all company codes. The customer account number is assigned on this level. That means, the same customer has an explicit accounts receivable number in all company codes from a financial view.

## Customer Master - Sales Data



General Data

Name
Address
Telephone

Client

Company Code Data

Company Code 1000

Company Code Data

Company Code 3000

Sales Organization Data

Sales Organization 1000

Financial Accounting

Sales

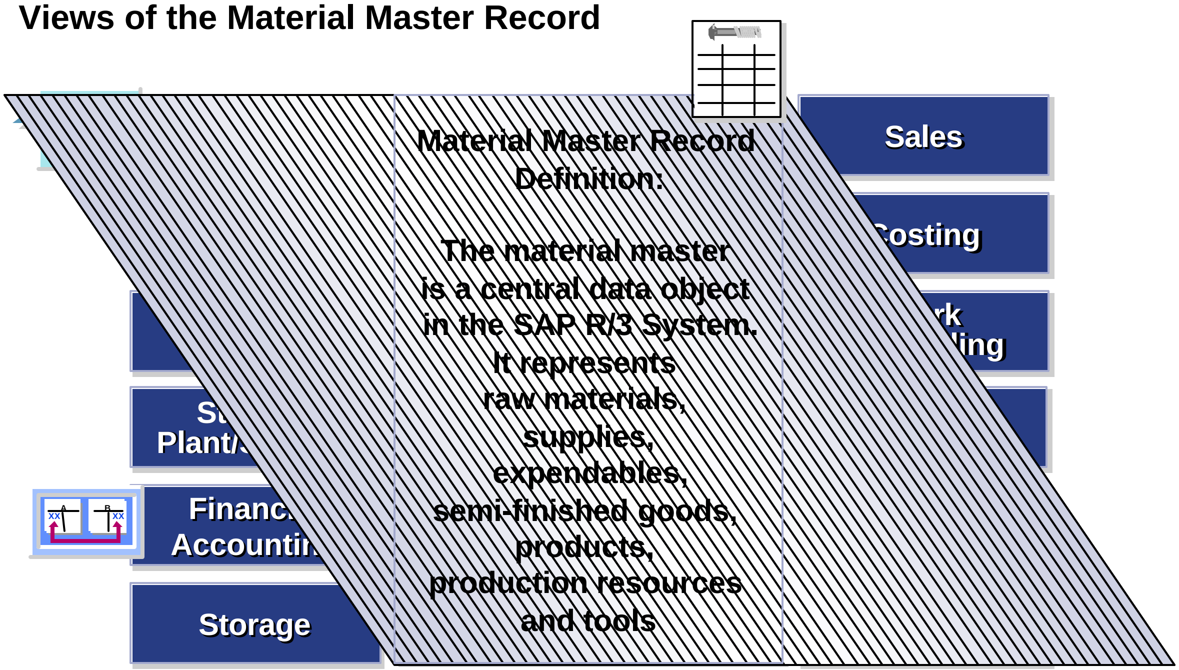
- If you also have SAP Sales and Distribution implemented, there are additional fields you can maintain. These fields contain information and control data that are necessary for processing the business activ ities in the Sales area.
- Fields for customer master data are divided into Accounting and Sales areas. Address data is used from both areas. In the Sales area, information recorded can be accessed by Financial Accounting and vice versa.

  Note:

  The structural logic for customer master records/accounts receivable is also valid for vendor master records/accounts payable.

## Master Data - Material Master

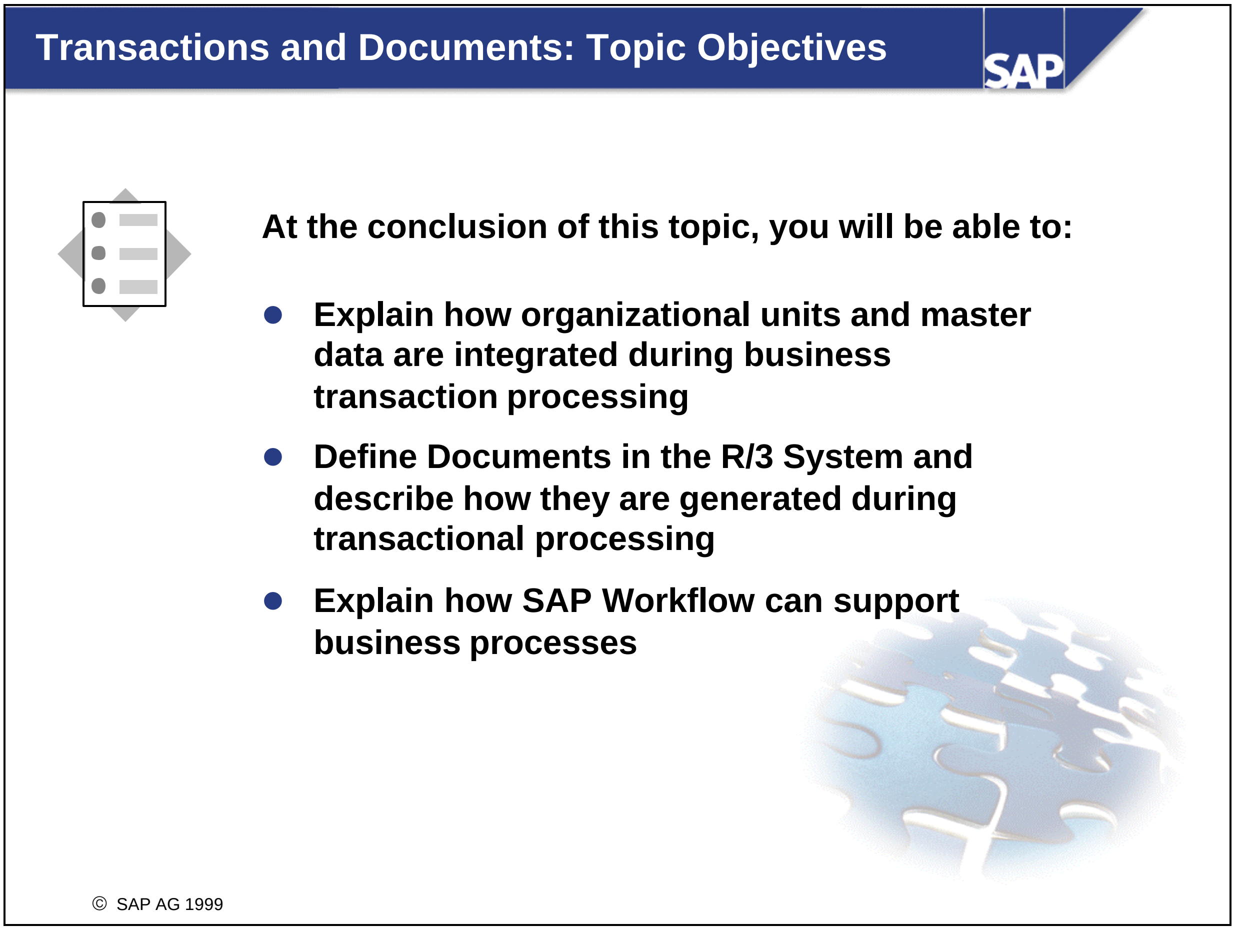
### Views of the Material Master Record



Material Master Record Definition:

The material master is a central data object in the SAP R/3 System. It represents raw materials, supplies, expendables, semi-finished goods, products, production resources and tools

- The material master represents the central source for releasing material-specific data. It is used by all of the SAP Logistics components in the R/3 System.
- Integrating all of the material data in one single database object means that the problem of data redundancy is not an issue. The stored data can be used by all areas, such as purchasing, inventory management, materials planning, invoice verification, and so on.
- The data contained in the material master is required, for example, by the following functions in the SAP Logistics component:
  - Ordering in Purchasing
  - Updating movement of goods and managing the physical inventory in Inventory Management
  - Posting invoices in Invoice Verification
  - Processing sales orders in Sales
  - Planning requirements, scheduling work in Production Planning and Control
- The structural logic that applies to vendors and customers is also valid for material master records.



# Transactions and Documents: Topic Objectives

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

- Explain how organizational units and master data are integrated during business transaction processing
- Define Documents in the R/3 System and describe how they are generated during transactional processing
- Explain how SAP Workflow can support business processes

- When creating an order for a customer, you must take transport agreements, delivery and payment conditions, and so on, with business partners into consideration. To avoid re-entering this information each time for every activity related to these business partners, relevant data for the activity from the master record of the business partner is simply copied.
- In the same way, the material master record stores information, such as the price per unit of quantity, and stock per storage location that is processed during order entry. This concept is valid for processing data for each master record included in the activity.

- When performing each transaction, applicable organizational units must be assigned. Assignments to the enterprise structure in the document are generated in addition to the information stored for the customer and material.
- The document generated by the transaction contains all relevant pre-defined information from the master data and organizational units.
- A document is generated for each transaction carried out in the R/3 System.

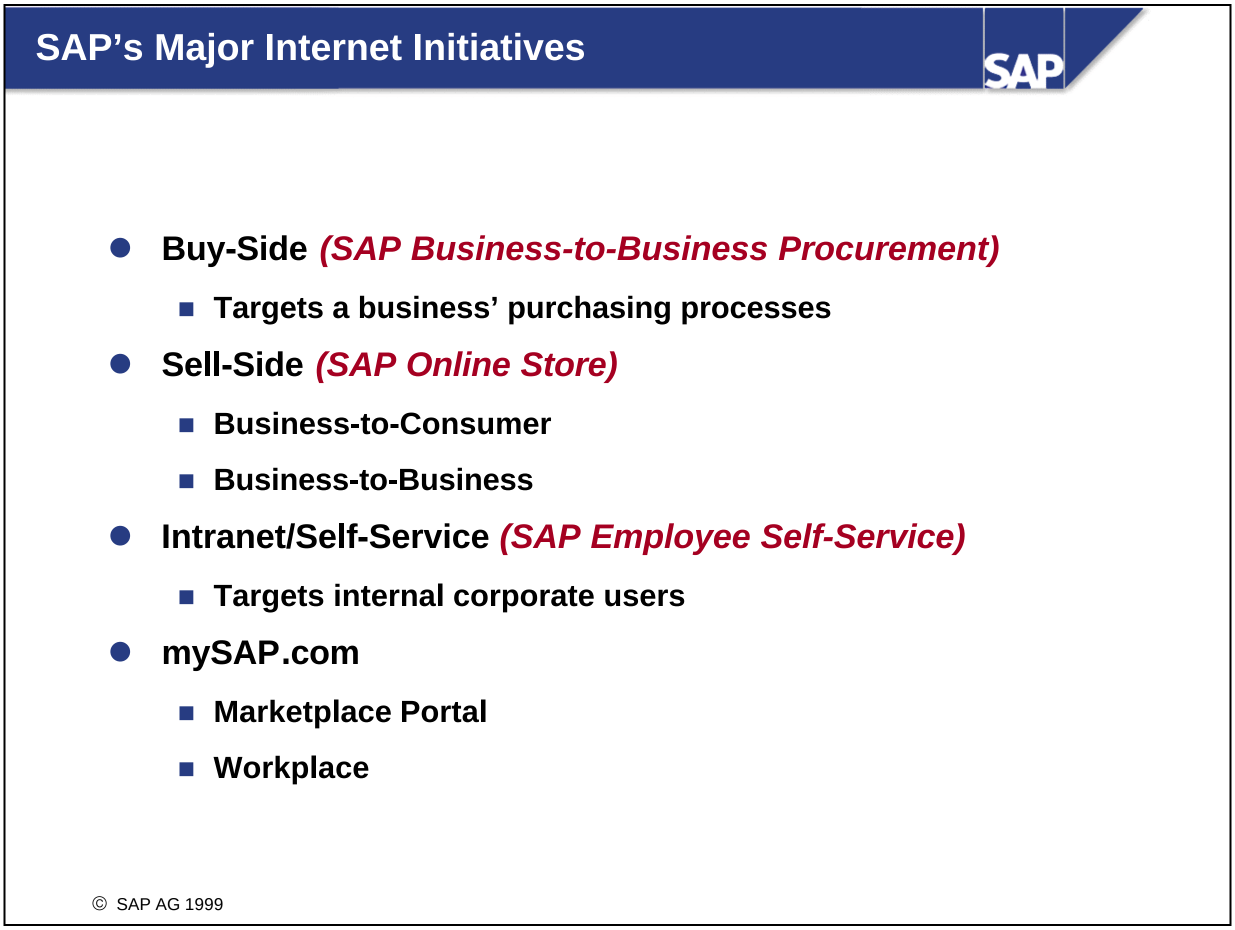
- Each time you save a quotation request, order, outline agreement, delivery note, production papers and so on, an output format is generated by the document. This output format represents a message (message type). The message is then placed in the message queue and it can be released for printing or output via EDI, for example. Messages can be released either automatically or manually using a message control program.
- You can release individual messages during processing using different communication media, provided that the relevant message types and communication media have been pre-defined. You can define when and how messages are sent for each document type.
- A form can be defined for each message type that contains the format for the message.
- Message control can be a default value from the master record of the business partner.

- The SAP Business Workflow is a support tool that can be used to optimize the execution of activities. Work steps carried out consecutively can be automated to coordinate flow of information. Workflows enable the electronic workflow management of structured flows that:
  - Cover a sequence of activities
  - Always occur in the same or similar form
  - Involve several departments or people
- Workflows control the information process flow according to a predefined model and are especially suited for structured organizations divided into departments/divisions.
- Workflows bring the “right” work in the “right” order at the “right” time to the “right” people. This can be achieved through automated mail or through a workflow item.
- A workflow item is a workflow task that has itself been generated from a Workflow and appears in the inbox of the office component. If human/manual intervention is required, employees can handle the messages from their electronic mail boxes.
- What Workflows are not: E-mail, EDI, ALE, or screen sequence management within a transaction. Workflows use all these processes.

# mySAP.com

## Contents:

- mySAP.com strategy
- Workplace
- Marketplace
- mySAP.com Business Scenarios
- Application Hosting

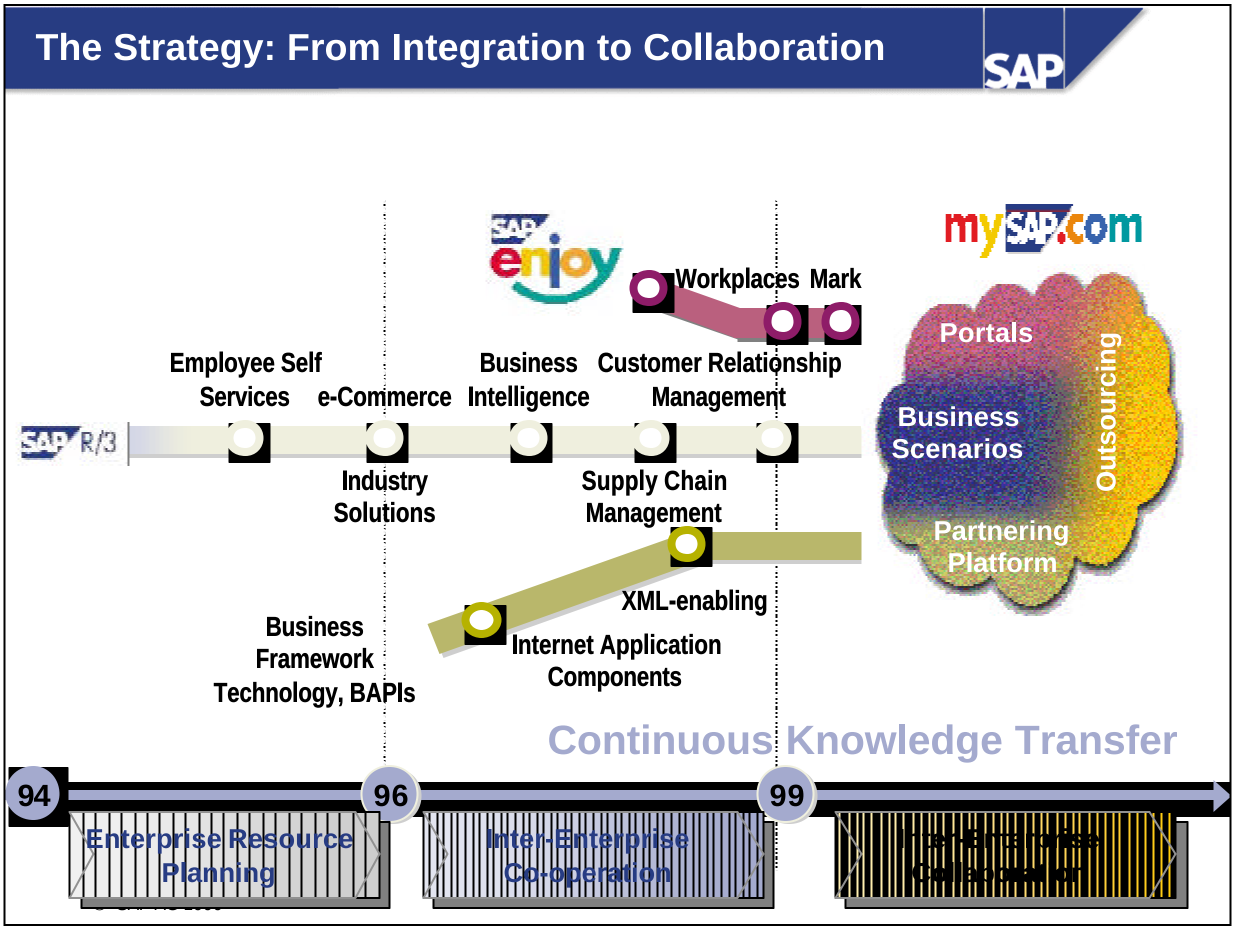


## SAP’s Major Internet Initiatives

- Buy-Side *(SAP Business-to-Business Procurement)*
  - Targets a business’ purchasing processes
- Sell-Side *(SAP Online Store)*
  - Business-to-Consumer
  - Business-to-Business
- Intranet/Self-Service *(SAP Employee Self-Service)*
  - Targets internal corporate users
- mySAP.com
  - Marketplace Portal
  - Workplace

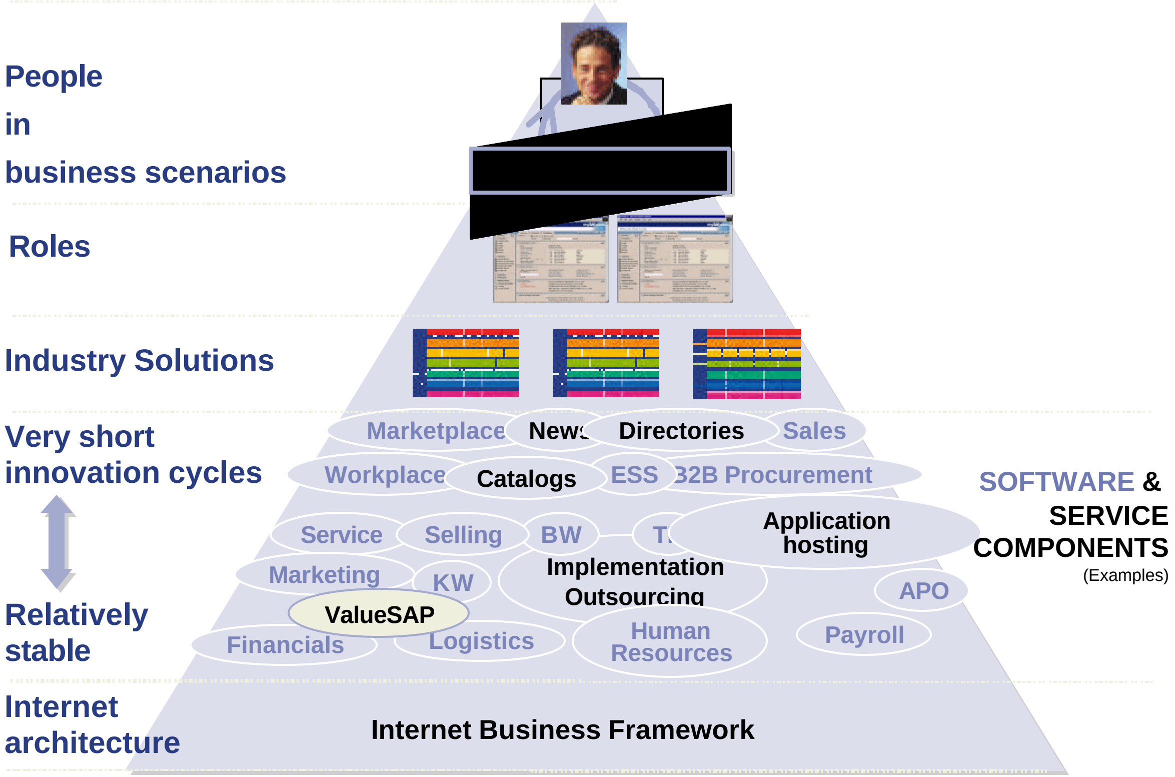
- SAP Business-to-Business Procurement is a solution for the entire procurement cycle for maintenance, repair and operations (MRO) items and services.
- SAP Online Store lets customers market their products and services on the Internet. To tap the full benefits of SAP Online Store, customers need only a standard Web browser, and the online store provider must implement the functionality of R/3 Sales and Distribution. The business-to-consumer sector represents the classic playground for this type of sales front-end.
- Employee Self Service (ESS) functionality gives employees complete control of their own data - they can request vacation at their PCs, enter trip costs, and record working hours using the browser of their company's intranet system.
- mySAP.com is a comprehensive, open, e-business solutions environment comprising of portals, industry-specific enterprise applications, Internet applications and services, as well as XML-based technology - all of which combine to enable companies to participate in the Internet economy.

## The Strategy: From Integration to Collaboration


mySAP.com
SAP enjoy
Workplaces
Mark
Employee Self Services
e-Commerce
Business Intelligence
Customer Relationship Management
SAP R/3
Industry Solutions
Supply Chain Management
Portals
Business Scenarios
Outsourcing
Partnering Platform
XML-enabling
Business Framework Technology, BAPIs
Internet Application Components
Continuous Knowledge Transfer
94
96
99
Enterprise Resource Planning
Inter-Enterprise Co-operation
Inter-Enterprise Collaboration

- Integration in the “old” economy meant business process integration.
  - ERP made SAP R/3 a worldwide standard system
  - Since 1996 SAP R/3 has been e-commerce capable
  - SAP products incorporated business technology for the future allowing customers to be ready for the future without system change
- Integration in the “new” economy requires integration of processes between enterprises.
  - Collaboration
  - More than working together
  - Processes, where many users participate, can be executed simultaneously as one-step-business

## People on Top



People in business scenarios

Roles

Industry Solutions

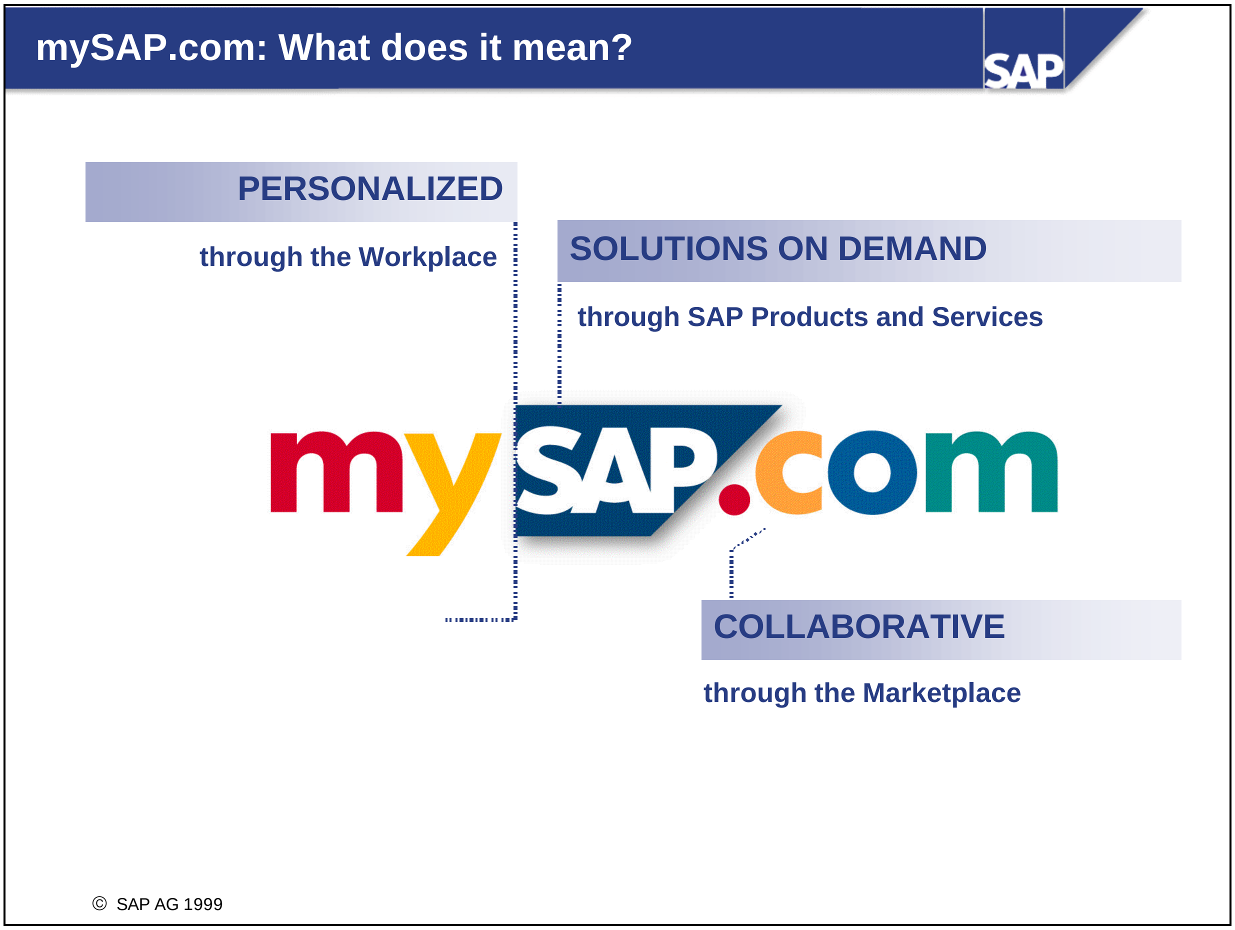
Very short innovation cycles

Relatively stable

Internet architecture

SOFTWARE & SERVICE COMPONENTS (Examples)

- All products offered by SAP today will be made available with the mySAP.com system. The focal point is the role-specific menus that are used to select the required function. In practice, it looks like this: A user is assigned to one or more roles. Business Scenarios contain various roles. The Business Scenarios reflect functionality used in the different industries.In addition to the R/3 System, there are the New Dimension Products and industry-specific solutions. All the functions provided on the Solution Map are organized according to Business Scenarios (for example, Purchasing or Employee Self-Service via the Internet).



## mySAP.com: What does it mean?

PERSONALIZED

through the Workplace

SOLUTIONS ON DEMAND

through SAP Products and Services

mySAP.com

COLLABORATIVE

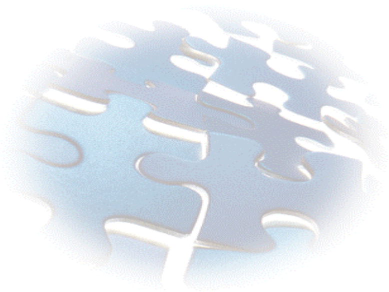
through the Marketplace

- mySAP.com places the Internet at the center of SAP's activities. It leverages all of SAP's key assets, including its extensive product portfolio, customer base, partner community, and expertise in integrating business processes.
- mySAP.com is the collaborative environment providing personalized business solutions on demand.

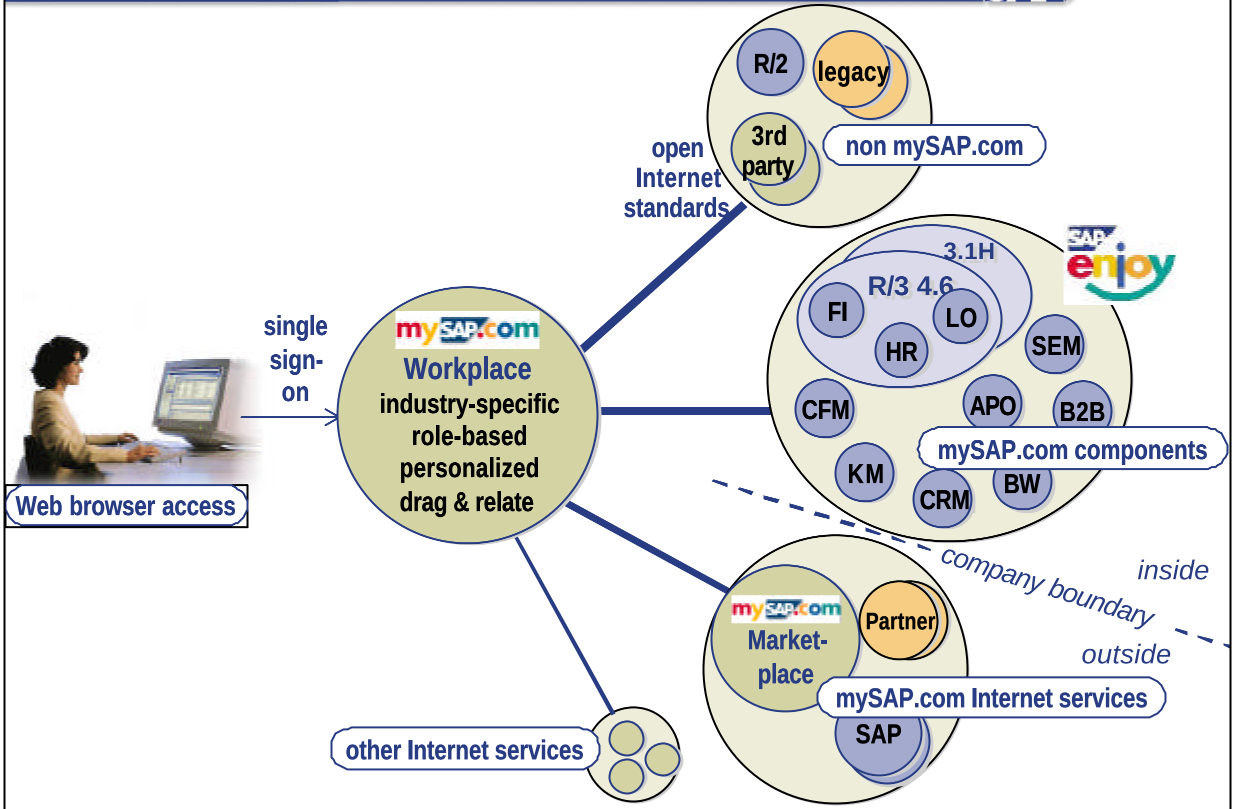
# Workplace: Topic Objectives

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

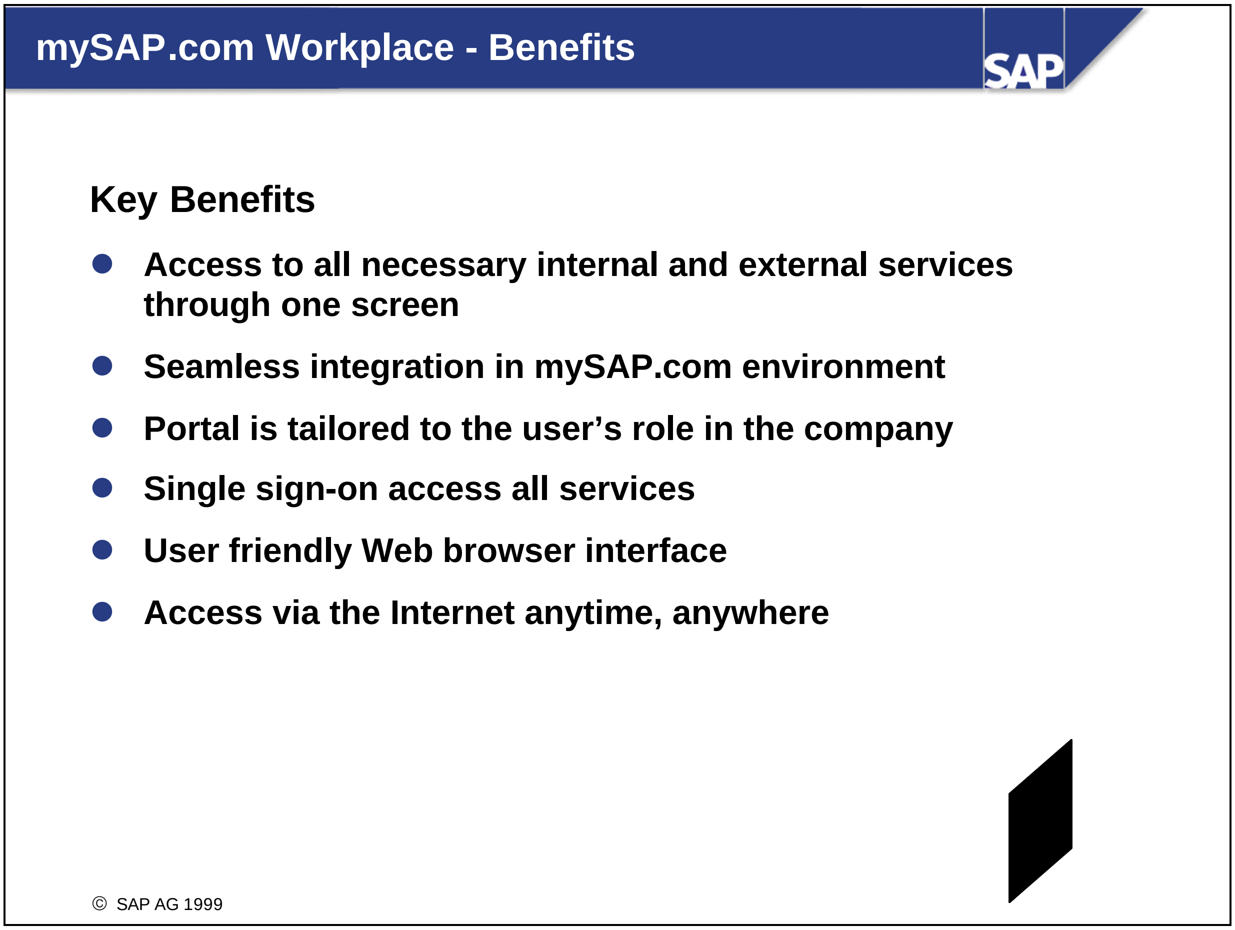
- Briefly discuss the concepts of the Workplace



## Workplace Internet Business Framework



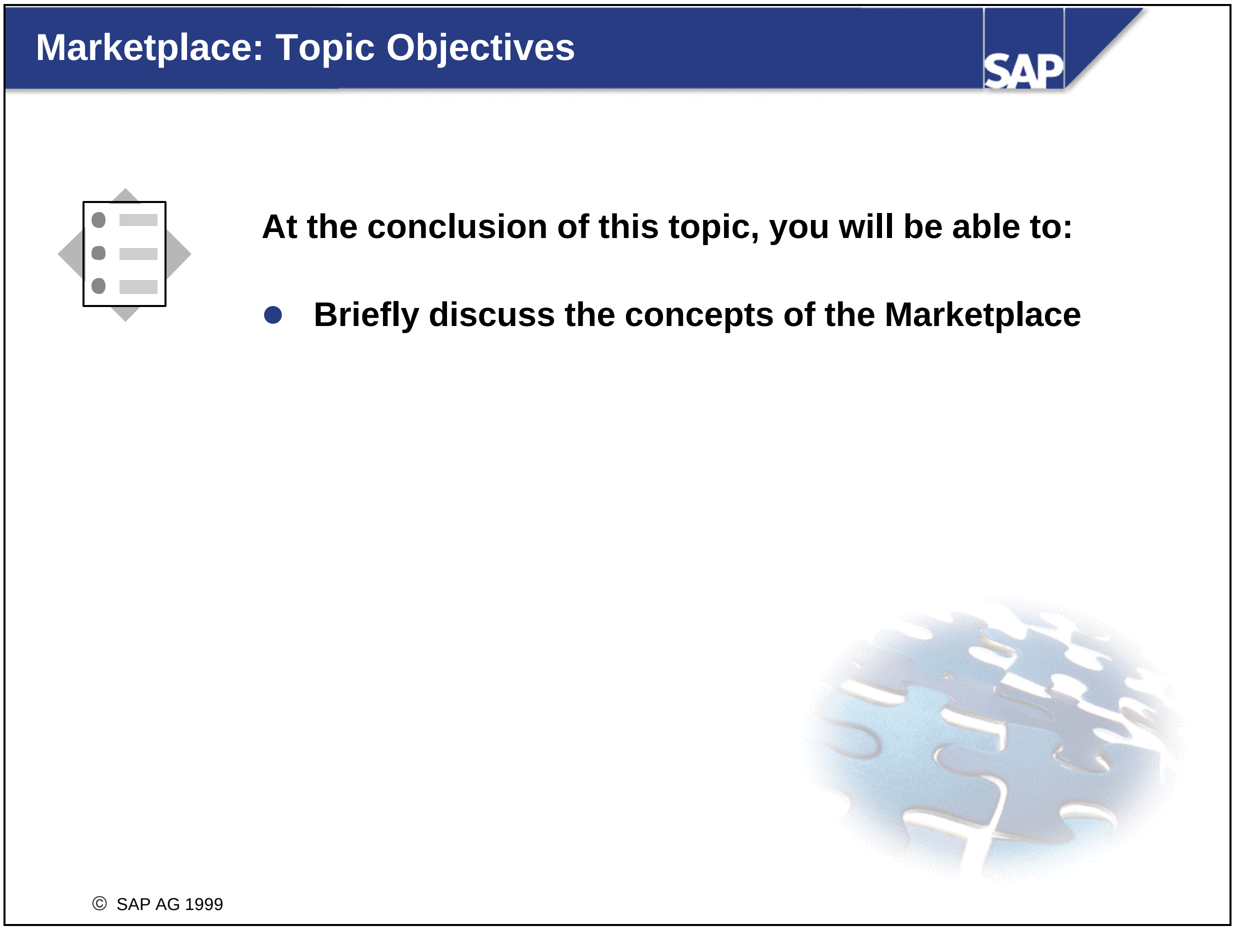
- The Workplace contains links inside and outside a company's boundaries. Links can be made to:
  - Non mySAP.com components: External systems using open internet standards
  - mySAP.com components: Classical and new web-based R/3 transactions (R/3 Standard System, New Dimensions, industry solutions) , Reports (for example, Business Information Warehouse reports with BW 2.0a) , Knowledge Warehouse contents
  - mySAP.com Internet services: my.SAP.com Marketplace
  - Any Internet or intranet web sites



# mySAP.com Workplace - Benefits

## Key Benefits

- Access to all necessary internal and external services through one screen
- Seamless integration in mySAP.com environment
- Portal is tailored to the user’s role in the company
- Single sign-on access all services
- User friendly Web browser interface
- Access via the Internet anytime, anywhere



# Marketplace: Topic Objectives

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

- Briefly discuss the concepts of the Marketplace

- The mySAP.com Marketplace consists of the four majoe elements:
  - Community
  - Content
  - Commerce
  - Collaboration

- Marketplace Portal is a place on the Web where communities can exchange goods and services electronically.
- Workplace is an application on a users desktop that cooperates with a Web browser and provides a personalized, role-specific view on the entire business world. This business world includes marketplaces, applications, services, and content provided by a company over the Intranet or other companies via the Internet.
- The business objectives of mySAP.com are to empower people, create value, and enable one-step business transactions. mySAP.com places the Internet at the center of SAP's activities. It leverages all of SAP's key assets, including its extensive product portfolio, customer base, partner community, and expertise in integrating business processes.

- The definitions above describe the Application Hosting options available.

# Section: ABAP Workbench Concepts and Tools

# Content: ABAP Workbench Concepts and Tools

Unit **Program Flow in an ABAP Program**

Unit **ABAP Workbench**

Unit **ABAP Statements and Data Declarations**

Unit **Database Dialogs I (Reading from the Database)**

Unit **Internal Program Modularization**

Unit **User Dialogs: Lists**

Unit **User Dialogs: Selection Screen**

Unit **User Dialogs: Screen**

Unit **Reuse Components**

# Main Business Scenario

Departure Airport

Destination

Departure City

Destination City

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

# Program Flow in an ABAP Program

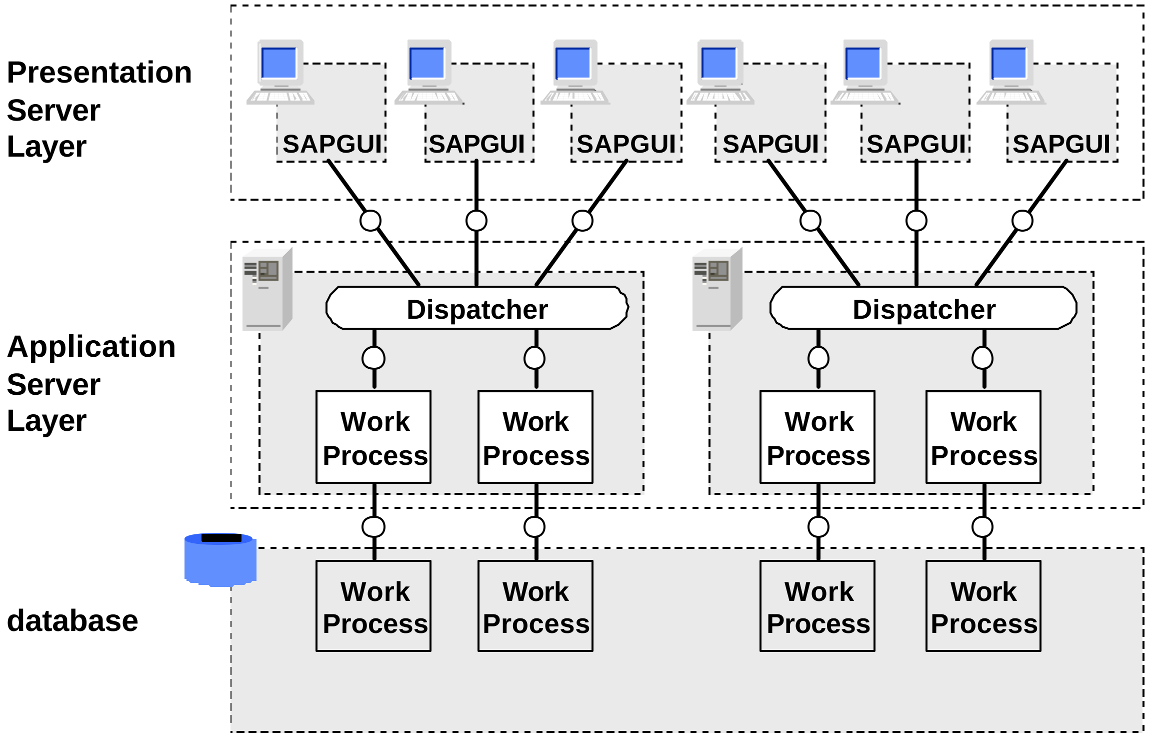
## Contents:

- 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?

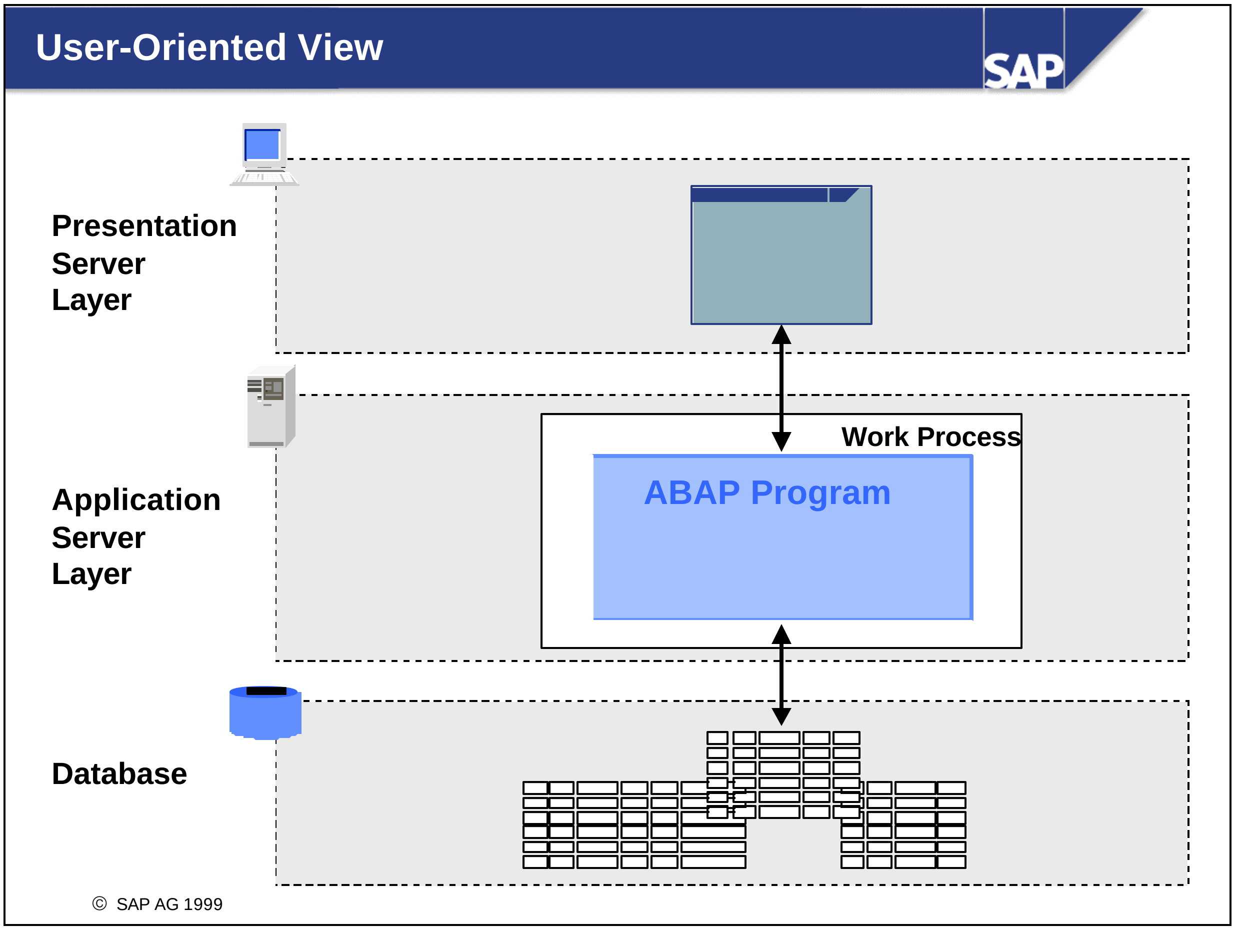
# Overview

- 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.



## User-Oriented View

Presentation Server Layer

Application Server Layer

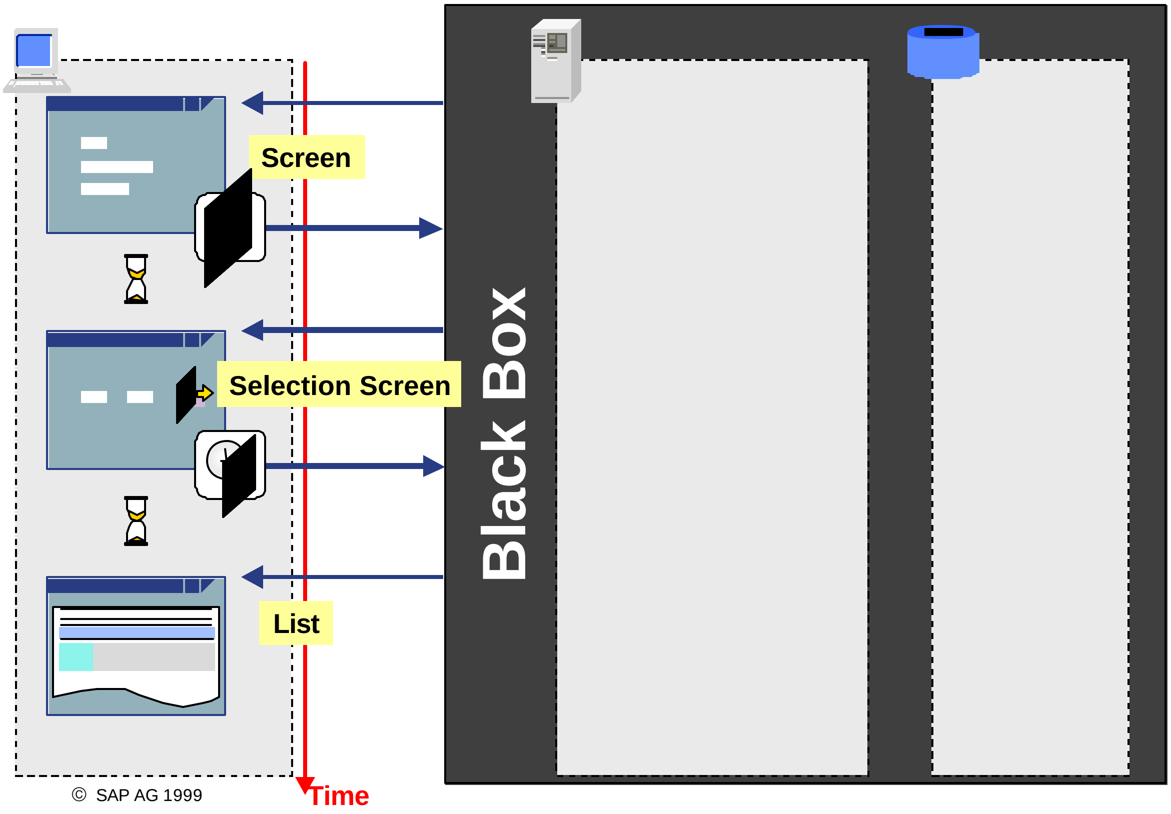
Work Process

ABAP Program

Database

- 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.

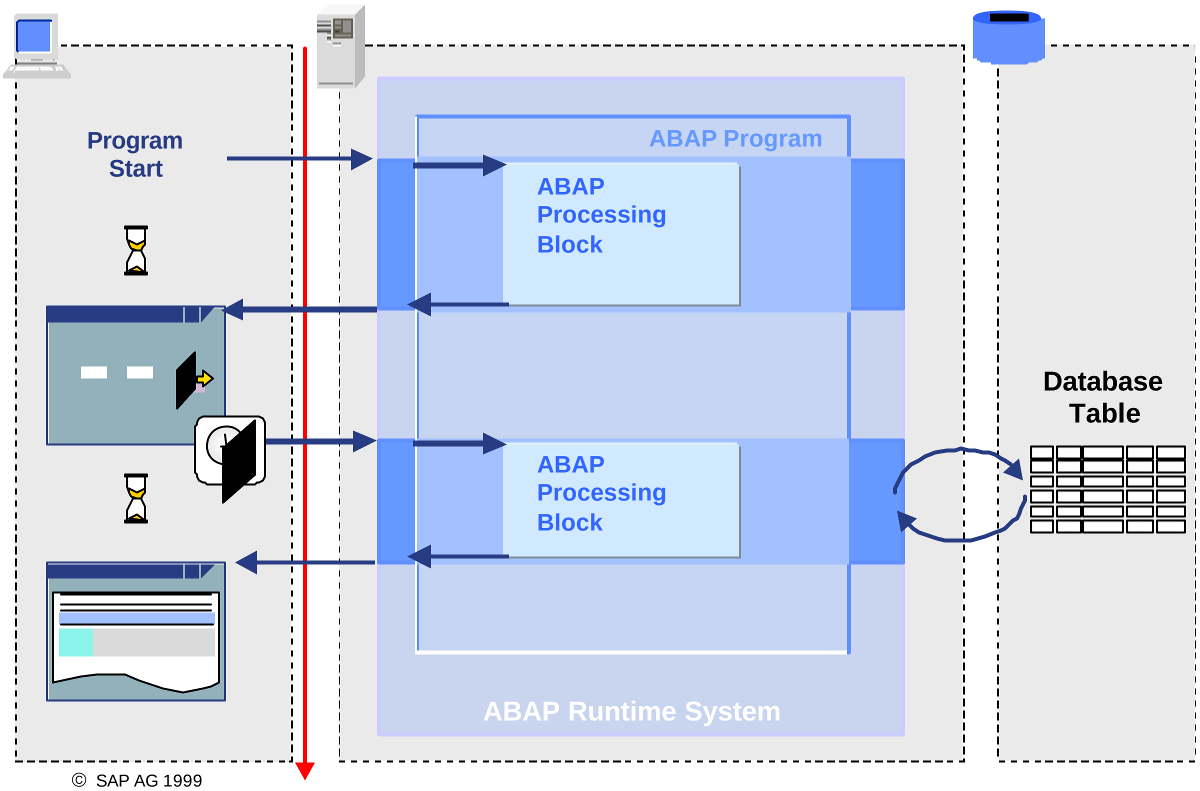
## Program Flow: What the User Sees



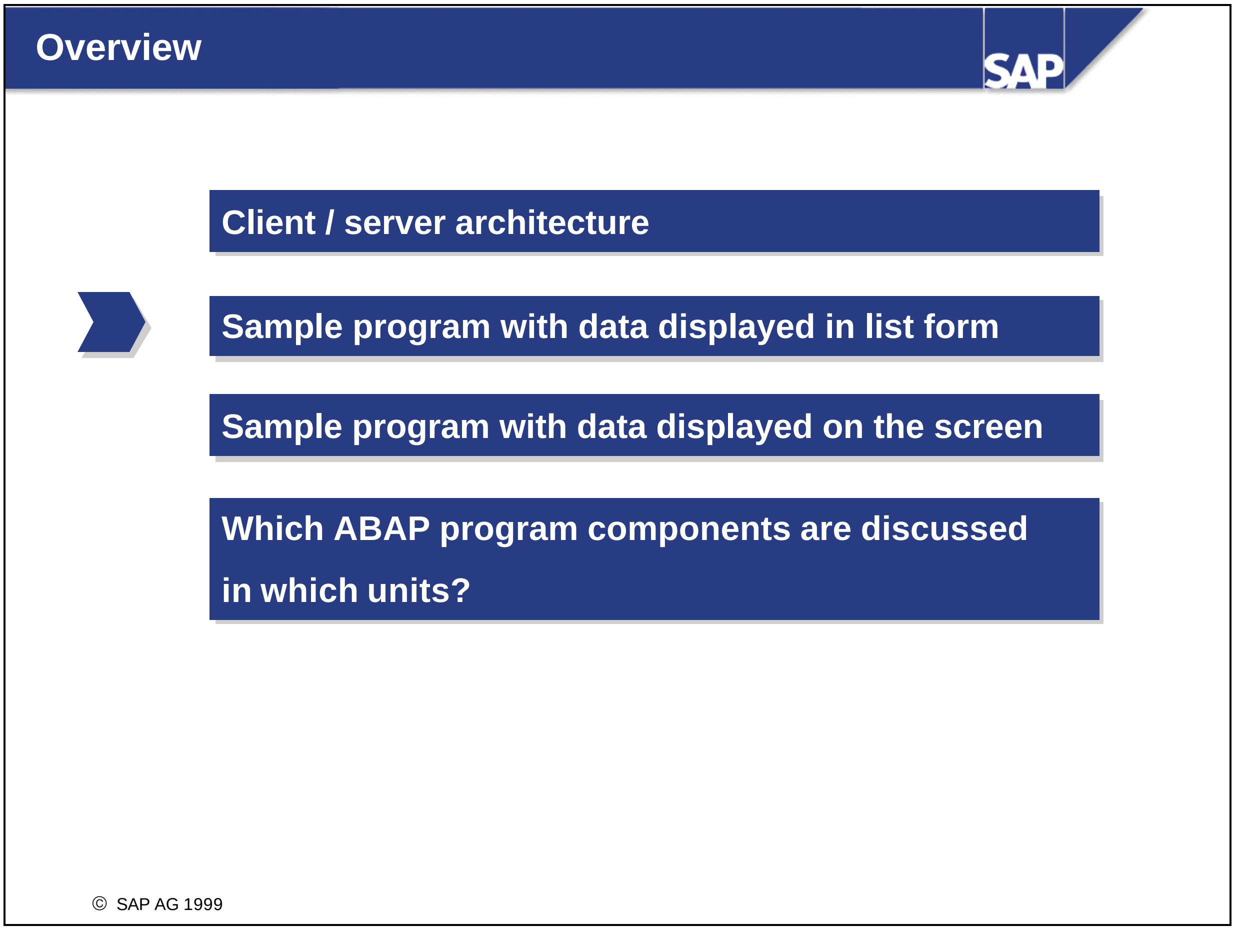
© SAP AG 1999

- 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.

## Interaction Between Server Layers



- 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.

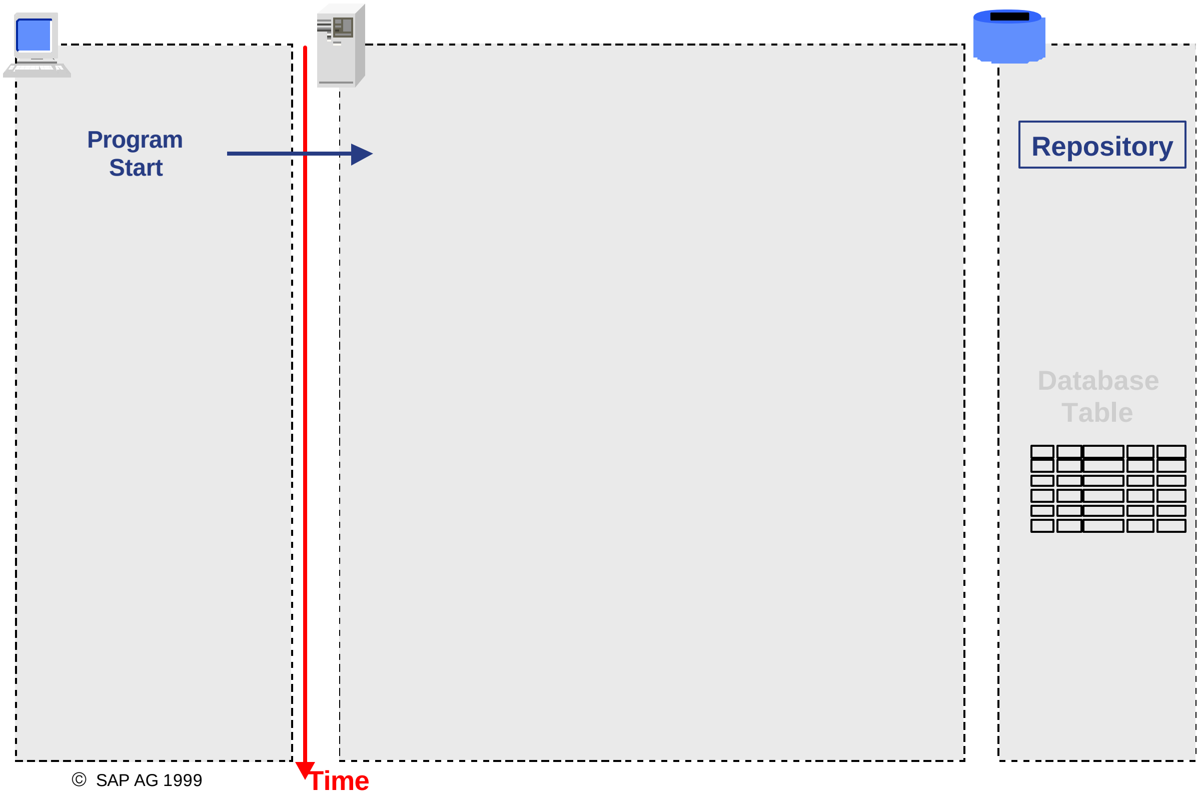


## Overview

- 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?

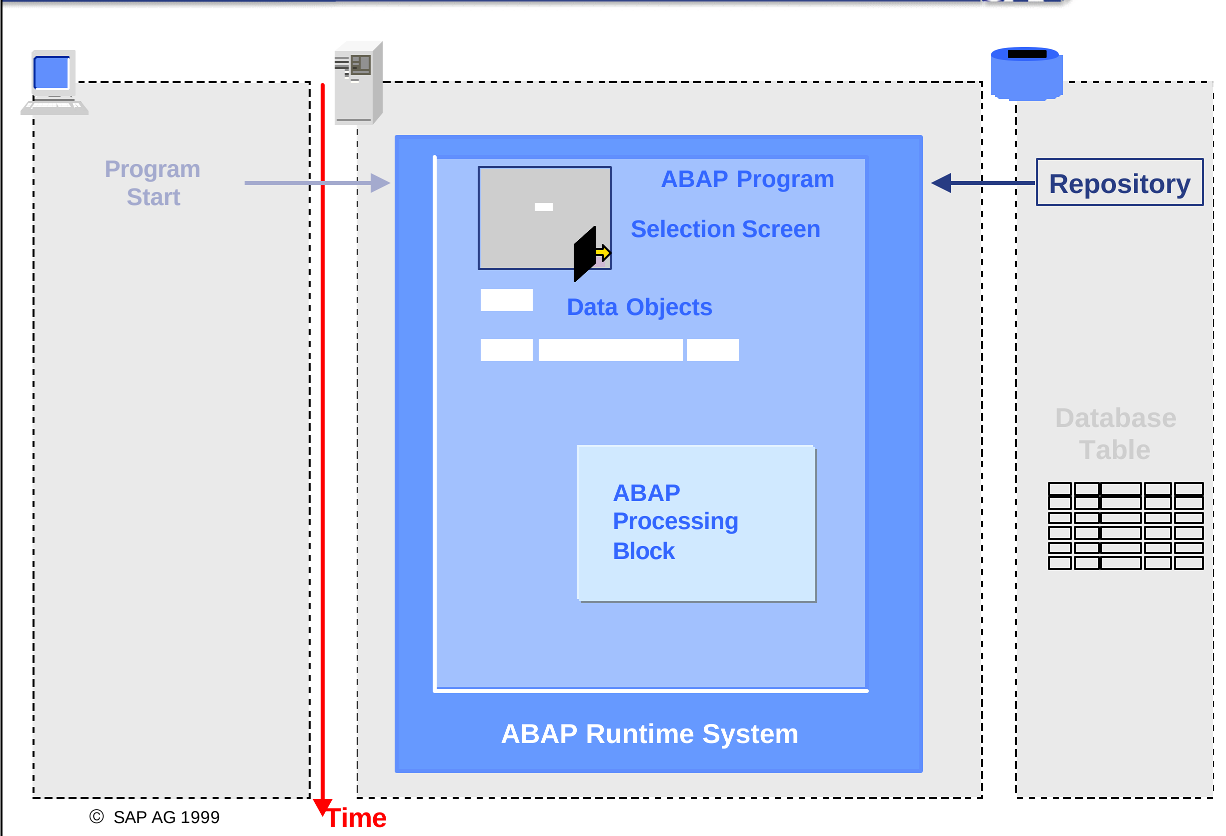
- In this part of the unit, the user has chosen to start a program where an airline ID can be entered on the initial selection screen. The program subsequently uses this information to retrieve the 'Long name of airline' and the 'Local currency of airline' from the database and display them for the user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

# Sample Program 1: Program Start



- 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.

## System Loads Program Context



- 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.

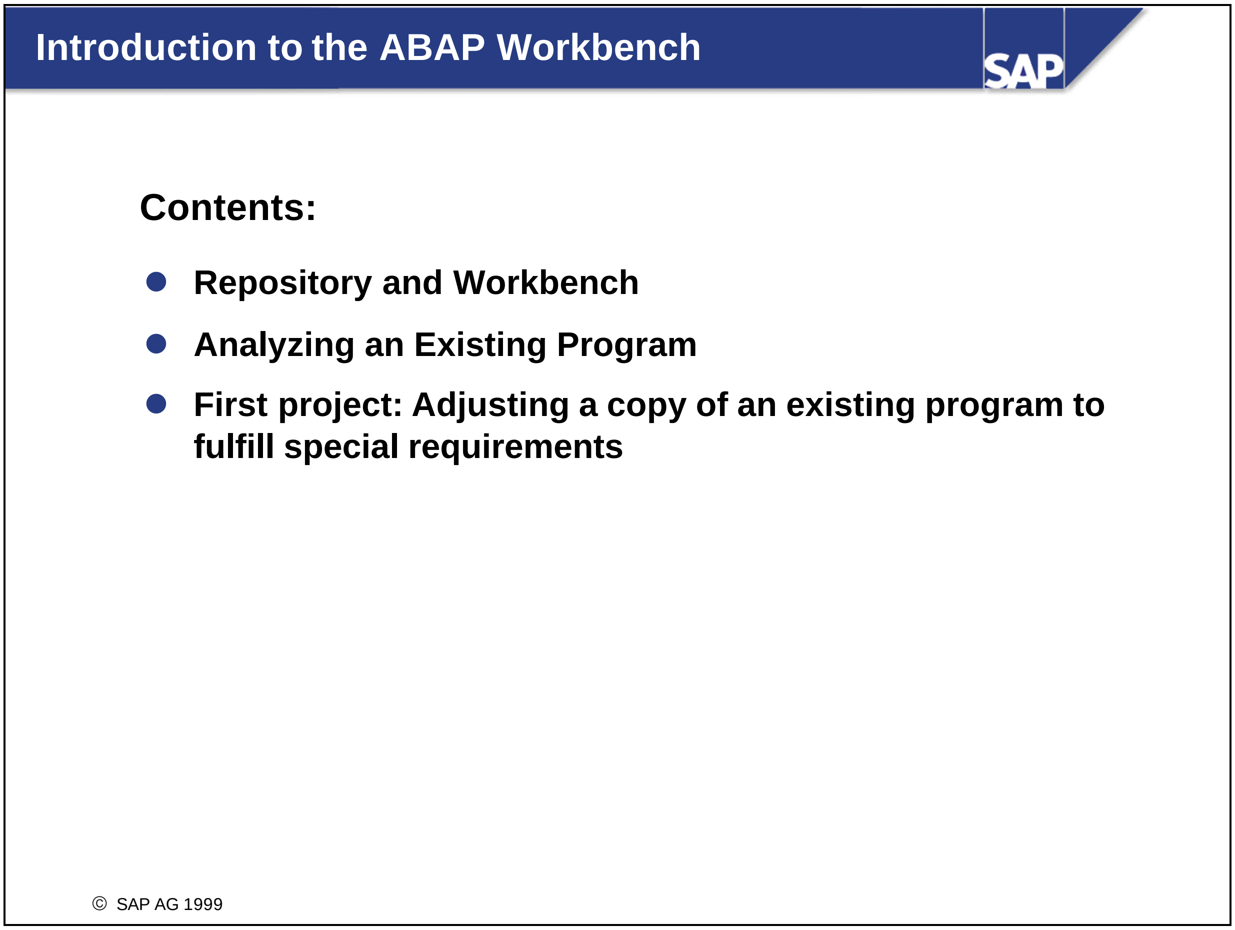
- The database returns the requested data record to the program and the runtime system ensures that this data is stored in the appropriate data objects. Normally a structure is the target field when a single record is accessed. The structure contains variables for all fields requested from the database.

- 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.



## 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

- The database contains, along with the Repository, 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.

- 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 according to 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 according to various criteria.

- 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.

- 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.

- 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 table definitions and retrieving global types
- **Menu Painter** for designing the user interface (menu bar, standard toolbar, application toolbar, function key assignment)
(see *Interfaces)*
- **Screen Painter** for designing screens (**dynamic programs**) for user dialogs
- **Function Builder** for displaying and processing function modules (routines 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:
  - An area for displaying an object list as a hierarchy
  - The object window, in which objects can be displayed and edited.
- You can hide the hierarchy area using the 'Close browser' pushbutton.
- You can display the object list for the object currently displayed in the object window using the 'Object list' icon.
- 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 hierarchy area. The object list is then displayed in the hierarchy 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 use the context menu to display objects from an object list. The system then automatically selects the correct tool for processing the object selected.
- If the object you require from the object list is not available in the system, you can create it by double-clicking. This is called **forward navigation**.

- 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*.

- You can display an overview of the program objects using the program object list in the Object Navigator.
- 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.

- If you start a program from the Object Navigator object list using the context menu, then you have two options.
    - Choose *Execute* -> *Direct* to execute the program directly.
    - Choose *Execute* -> *Debugging* to execute the program in the debugging mode.

- Starting the program in the 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 on the field name in the code displayed.

- 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.

- 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.
- 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.

- You can display detailed information on single objects in the Editor by double-clicking:
  - **Double-clicking** on 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** on a **field name** displays the part of the program source code where the data object is defined.
  - **Double-clicking** on 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.
- You can also set a **breakpoint** in any line of source code in the Editor. Then start the program without selecting the debugging mode. The program will now be executed up to the point where the breakpoint is defined. At this point, the debugging mode is started.

- 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 displa ys a dialog box offering you various views of the keyword documentation.

- If you need more precise information on parts of the source code, you can analyze the source code. 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. They are normally used as an 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.

- The data supplied by the database is put into 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_sbc400**.

- 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`** declares a structure object that serves as an interface for the screen. All data from this structure is automatically inserted into its corresponding screen fields when the screen is called by the **`CALL SCREEN`** statement. 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).

- 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.

- 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.

- 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 transportable development class (not $TMP).
- After development has ended, Repository objects are transported into the test systems or 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.

- 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: C11K900001). <sid> is the system number.
- Next, the Workbench Organizer (WBO) creates a **task** for each employee assigned to the change request. From now on whenever an employee allocates a Repository object to that change request, the Repository object will automatically be filed away in 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 change request 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 the operating system level. Afterwards, the developers check the import log.

- 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.

- Assign the program to a development class, in order to be able to 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. Assign 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 'Own requests' pushbutton.
- For more information about project organization from the project management point of view (including creating tasks), refer to the unit on software logistics.

- You can adjust the short text (= 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 adjust the title.
- The altered title appears as short text next to the program name in the Object Navigator object list.

- In order to adapt the source code, navigate to the Editor (context menu).
- To adapt the list, supplement 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 and then 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 that is defined in the ABAP Dictionary. 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.

- A syntax check started from the Editor always relates to the current contents of the Editor.
- As soon as you have saved the program, this source code is visible throughout the system. You can use the context menu to carry out a syntax check that encompasses all program components. Starting the program from the object list context menu ensures that the active version is started.
- As soon as you have activated the program, the active version can be executed throughout the system.
- You can run an extended program check for activated programs using the context menu or the menu option. These checks are considerably more extensive than the syntax check.

## 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 on a list.

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

**Program:** SAPBC400WBT_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 dynpro screen 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? To find out the names, use the F1 help for each field then choose *Technical info.*

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. How do you create a line break?

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 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: 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_sbc400** 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. The symbol / after the **WRITE** statement creates a line break.

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 fields **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_sbc400 TYPE sbc400_carrier.
PARAMETERS: pa_car TYPE scarr-carrid.



START-OF-SELECTION.
* Select all fields of one dataset from database table SCARR
  SELECT SINGLE * FROM scarr INTO CORRESPONDING FIELDS OF wa_sbc400
                  WHERE carrid = pa_car.
* At least one record could be selected
  IF sy-subrc = 0.
* Copy fields with corresponding names
    MOVE-CORRESPONDING wa_sbc400 TO sbc400_carrier.
    CALL SCREEN 100.
* Copy fields with corresponding names back
    MOVE-CORRESPONDING sbc400_carrier TO wa_sbc400.

* Write data on list
    WRITE: / wa_sbc400-carrid COLOR COL_KEY,
             wa_sbc400-carrname,
             wa_sbc400-currcode.
```

```
* add an empty line
    SKIP.
* add a horizontal line
    ULINE.
* write username, time and date on list
    WRITE: / wa_sbc400-uname,
             wa_sbc400-uzeit,
             wa_sbc400-datum.
  ENDIF.
```

## Screen 100:

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

# ABAP Statements and Data Declarations

## Contents:

- Types
- Data Objects
  - Elementary Data Objects
  - Structures
  - Internal Tables
- ABAP Statement Attributes

- The contents of this unit concentrate on the definition of data objects and selected ABAP statements.

- Types describe the attributes of
  - Input and output fields on screens,
  - Data objects and
  - 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.

- **Local types** are used in programs
  - If only technical attributes are needed and no semantic attributes are required, and
  - If the types are only used locally within a program.
- **Global types** ( = ABAP Dictionary types) are used
  - If you intend to use the types externally
    (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.

- 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 on the output field next to the data element label
    - for **local types** in programs or **data objects**: By double-clicking on the type
- Technical types and technical domains may be directly assigned to data elements. If you want more information on other data elements referencing the same domain, you can navigate to the domain from the data element by double-clicking on its name and then executing the function *Where-used list*.

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

- When defining simple types or variables, you can refer to a pre-defined type. For more information refer to the keyword documentation on **TYPES** or **DATA.**
  - **C** Character
  - **N** Numeric Text
  - **D** Date (YYYYMMDD)
  - **T** Time (HHMMSS)
  - **X** Byte (heXadecimal)
  - **I** Integer
  - **P** Packed Number
  - **F** Floating Point Number
  - **STRING** Character String
  - **XSTRING** String of Bytes (X String)

- 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.

- 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.

- Rules for naming data objects:
  - A name can consist of 30 characters maximum (letters, numbers, or symbols).
  - The following symbols ARE NOT allowed: ( ) + . , :
  - SPACE is a predefined field.
- For compatibility reasons, it is still possible  to 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 may enter a length (in bytes) 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 no type is entered, then the field is automatically a type **C** field.

- You define **constants** using the ABAP keyword **`CONSTANTS`**. The **`VALUE`** addition is required for constants. It defines their value.
- ABAP recognizes two types of literals: **number literals** and **text literals**. The latter is always enclosed in inverted commas (**`'`**).
- Whole numbers (with preceding minus sign if they are negative) are stored as number literals with either type **I** (up to and including nine digits) or type **`P`** (ten or more digits).
- 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.
- You can create **translatable text literals,** or ***text symbols***, for all ABAP programs. Use the *Program object types* dialog box to get to the maintenance screen for the text symbols.

- 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 a start value for an elementary field yourself using the **`VALUE`** addition. After **`VALUE`** 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`**.

- 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`**.

- **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 about **IF**.
  - 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.

- 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 choosing the 'Change' icon.

- 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-dependent. 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 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 need to be used as interface structures for screens.

- Fields of a structure are always addressed with `<Structure>-<Field_name>`.

- 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 names also occur 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.

- 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.

- You must define the following information in order to fully specify a table type:
  - **Line Type:** You can store 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 table kinds: standard tables, sorted tables and hashed tables. The estimated **access type** is mainly used to choose the table type.
- **Access type** defines how ABAP 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.

- Another internal table attribute is the table type. Internal tables can be divided into three table types according to the way they access data:
  - **Standard tables** maintain a linear index internally. This kind of table can be accessed using either the table index or keys.
  - **Sorted tables** are sorted according to key and saved. Here too, a linear index is maintained internally. This kind of table can also be accessed using either the table index or keys.
  - **Hashed tables** do not maintain a linear 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**.

- 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 global table type or a local table type.
- You can find detailed information on declaring table types in the documentation or on Course BC402.

- You can perform the following operations on single records in internal tables:
  - **APPEND** appends the contents of a structure having the same type as the line to an internal table. This operation can only be used with standard tables.
  - **INSERT** inserts the contents of a structure having the same type as the line 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 of the internal table to a structure having the same type as the line.
  - **MODIFY** overwrites a line of the internal table with the contents of a structure having the same type as the line.
  - **DELETE** deletes a line of the internal table.
  - **COLLECT** inserts the contents of a structure having the same type as the line 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 for identical keys.
- You can find detailed information about the ABAP statements described here in the keyword documentation for the relevant ABAP keywords.

- You can perform the following operations on sets of records in internal tables:
  - **LOOP … ENDLOOP** The **LOOP** places the lines of the internal table in the structure specified in the **INTO** clause one-by-one. The structure must have the same type as the line of the internal table. All single-record operations can be executed within the loop. In this case the system provides the information about the line to be edited in the single-record operation.
  - **DELETE** deletes the lines of the internal table that satisfy the condition <condition>.
  - **INSERT** copies the contents of several lines of an internal table to another internal table.
  - **APPEND** appends the contents of several lines 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.

- You can perform the following operations on internal tables:
  - **SORT** You can sort tables by any column or columns in ascending or descending order . 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 allocated to the table.

- You can add lines to a standard table by first filling a structure with the required values and then appending it to the internal table with the **APPEND** statement. This statement is only meaningful with standard tables.
- Use the **INSERT** statement to insert lines in sorted and hashed tables.
- **INSERT** works like an **APPEND** in standard tables.

- You can read and edit the contents of an internal table with a **LOOP** statement. In this example, one line is copied each time from internal table it_flightinfo to structure wa_flightinfo. The fields of the structure can then be edited. A list is built here from the fields with 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.

- With the **INDEX** addition you can restric t access to certain line numbers. 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.

- With the **`WHERE`** addition you can restrict access to lines with certain values in key fields. 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 editing that 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 of 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 already permitted key access to standard tables prior to Release 4.0, when it was not yet possible to define key columns explicitly.

- 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 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 avoid confusion, 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.

- A number of ABAP statements support a return code. Various exceptions are detected, depending on the statement. 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. One of the fields of this structure is field **`subrc`**. You therefore do not have to create a data object for the return code.
- In this example a line should be read 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.

- There is a special dialog type called the user message for error situations. Messages are triggered with the **`MESSAGE`** statement.
- Messages can be found in table T100. They are organized according to language, a two-digit ID, and a three-digit number.
- Messages can contain up to 4 variables, identified as **`&1`**, **`&2`**, **`&3`**, and **`&4`**. If you want to output the character & and do not want to use it as a variable, double it, for example: 'This is a message with an **`&&`**'.
- In message long texts use **`&v1&`**, **`&v2&`**, **`&v3&`**, and **`&v4&`** for their corresponding variable.
- You can create your own messages using ID numbers that begin with Y (for the head office) or Z (for branch offices).

- Send messages with the **MESSAGE** statement. The language for messages in table T100 is automatically set to the user's logon language. You can define the message ID following the parameter **MESSAGE-ID** in the **REPORT** statement. The message ID is now set for the entire report. Enter the message number at the **MESSAGE** statement.
- Enter the message type directly in front of the three-digit message number; this letter determines how the report user reacts to dialog messages (see next slide).
- Set values for variables (up to a maximum of four) following the parameter **WITH**. Fields and literals are also allowed. The field at level i thus replaces message variable &i. If the variables in the message are identified with & or $, these placeholders are supplied with values independent of the position of the fields of the message statement.
- In addition to using message ID with the **REPORT** statement, you can also add a different message ID to the command **MESSAGE** by entering the **ID** in parenthesis directly after the message number. This deviant message ID is only valid for a single message, however. Example: **MESSAGE E004(UD).**
- Use the following syntax, whenever you want to send a dynamic message: **MESSAGE ID <mid> TYPE <mtype> NUMBER <mnr> WITH <field1> <field2> <field3> <field4>.**
- System fields **SY-MSGID**, **SY-MSGTY** and **SY-MSGNO** are supplied with the message ID, message type, and message number respectively and system fields **SY-MSGV1** to **SY-MSGV4** with the fields for the placeholders.

- There are six different types of message: **`A`**, **`X`**, **`E`**, **`I`**, **`S`** or **`W`**. The runtime behavior of the messages depends on the context. 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 is continued 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.*

## Unit: Data Objects and Statements
## Theme: 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 compone nts of the structure **WA_SBC400** are filled in the **SELECT** statement? What is the value of system field **SY-SUBRC** after the statement?

1-4 Now observe how fields are copied from **WA_SBC400** 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_SBC400**.

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_SBC400`** changes. Each time the program stops, use the ‘Continue’ (F8) function in the Debugger to carry on processing.

## Unit: Data Objects and Statements

## Theme: 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

## Theme: Structures and Assigning Values

1-3 Which components of the structure **SBC400_CARRID** are filled in the **SELECT** statement?

**MANDT, CARRID, CARRNAME, CURRCODE**
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

### Theme: 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 record 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 record 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.
```

# Database Dialogs I

## Contents:

- Information about Database Tables
- Reading Database Tables
- Authorization Checks
- Reading Multiple Database Tables

- **Database tables** 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 couple of different ways in which you can 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, or
  - Navigate directly to the ABAP Dictionary from the ABAP Editor while editing the program: This can be done by double-clicking on the name of the transparent table in the **`FROM`** clause of the **`SELECT`** statement.

- 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 branch 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.** and then navigate to to the ABAP Dictionary by double-clicking on 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 on the **data element** and then use the **where-used list** function to search for transparent tables according to the field type.

- 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 least expensive 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 forth. ?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.

- All pieces of information that are logically dependent on each other contain entities: In the ABAP flight data model there are individual entit ies for:
  - All cities,
  - All airports,
  - All airline carriers,
  - All flight routes,
  - All flights.
- These entities all relate to each other in certain pre-determined 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 **BC_DATAMODEL**.
- The following is a list of the flight data model tables most frequently used in ABAP training courses:
  - **SPFLI**: Flight connections table
  - **SFLIGHT**: Flights table
  - **SBOOK**: Bookings table

- 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.
- The following information is of interest for enhancing the performance of database accesses:
  - **Key Fields:** If the lines requested from the database are being retrieved according to key fields, the Database Optimizer can perform access using a primary index. Checkboxes are on 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 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.
    The fields that 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.

- **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 SQL statements by the **database interface**.

- The **SELECT SINGLE*** statement allows you to read a single line from a database table. To ensure that you read a unique entry, all of the key fields must be filled by the **WHERE** clause. The informs the database interface that all columns in that line of the database table should be read. If only a specific cross-section of columns is desired, a structure can be inserted 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.

- If you do not use the addition **SINGLE** 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.
- The number of **lines** to be read can be restricted using the **WHERE** clause. The restrictions contained in the **WHERE** clause should either be made according to the database table's key fields or according to a secondary index. Further information about key fields and secondary indexes can be found in the ABAP Dictionary. For example, double-clicking on the database table included in the **FROM** clause will take you directly to the Dictionary.
- You may only enter the names of the database table fields you want to be read in the **WHERE** clause. The name of the database table you want to access is found 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 packages. 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**.
- **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. After the **ENDSELECT** statement, it contains the total number of lines read.

- 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 called an **array fetch**.
- Since an array fetch is not logically a loop, no **`ENDSELECT`** statement is used.
- **`SY-SUBRC`** = 0 if 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 of this course.

- The program must contain a data object with a suitable type for each column that is required from a database table. For reasons of program maintenance, 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 structure field names are disregarded.
  - **Single data objects:** You enter a set of data objects in the **INTO** clause. For 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 ...
```

- 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 and does not need to be left-justified
  - 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`. The runtime may therefore be longer.
- 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>`**.

- The SAP authorization concept recognizes a large number of different authorizations. These are all managed centrally in the user master record for every user.
- Authorizations are not directly assigned to users, but stored in work center descriptions (profiles).
- These profiles are generated using the Profile Generator, which administers the profiles as activity groups.
- Users can belong to one or more activity groups and are then assigned the authorizations contained in those activity groups.
- Release 4.6 contains a large number of pre-defined activity groups. You can use these as is or copy and tailor them to your specific needs.

- 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 check 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 you should display a message and take the appropriate action.
- 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.

- 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 the user is authorized for the object **`S_CARRID`** which has the value **`'LH'`** in the field **`CARRID`** (Airline carrier) and the value **`'02'`** for 'Change' in the field **`ACTVT`** (Activity). The abbreviations for the possible 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 in **`SY-SUBRC`**. When checking this return code, you can specify the consequences of a missing authorization (for example: terminate the program or display a message and skip some lines of code).

- 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.For example: When calling a transaction to change flight data, it makes sense to check whether 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.
- The keyword documentation for **AUTHORITY-CHECK** contains a complete list of return codes.
- 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.

- 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 are 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.

- Views are application-specific views of different ABAP Dictionary tables. Views can contain a sele ction of fields from a single very large table or fields from several different tables.
- Views allow you to gather information from the fields of different tables and present it to users in the form they require when working with the R/3 System.
- Views are mainly used for programming with ABAP and for F4 online help.
- If there are no components available that are suitable for your purposes, you can carry out complex database access using ABAP-OPEN- SQL statements. To do this you have to compare the merits of various techniques, as using an unsuitable technique can result in considerable performance problems. You can find more detailed information on optimum-performance database access in the documentation or on course **BC490** ***ABAP Performance Tuning.***

## Unit: Database Dialogs 1

## 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 displays all of the flights of a selected airline in a list. In the program, you should also calculate the percentage occupancy of each flight and display this as well.
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 Restrict the lines of the database selection according to the primary key. To find out the key fields, look at the ABAP Dictionary definition of **`SFLIGHT`**. The client field is filled automatically by the system. Program a selection screen on which the user can enter a value for the second key field (**`CARRID`**).

1-4 Read all flights from database table **`SFLIGHT`** that correspond to the airline entered on the selection screen by the user. 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. Ensure that you specify the key fields in the selection, so that the database can use the primary index.

1-5 Within the **`SELECT`** loop, calculate the percentage occupancy using the corresponding field of the work area. Assign the result to the **`PERCENTAGE`** field in

1-6 Create a list displaying the information you read from the database and the percentage occupancy of each flight.

## Unit: Database Dialogs 1

## 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** to the name **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: Database Dialogs I

## 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:

3-2 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.

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

## Unit: Database Dialogs 1

## Topic: SELECT Loops

## 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 records 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 COLOR COL_KEY,
             wa_flight-connid COLOR COL_KEY,
             wa_flight-fldate COLOR COL_KEY,
             wa_flight-seatsocc,
             wa_flight-seatsmax,
             wa_flight-percentage,'%'.
  ENDSELECT.
```

## Unit: Database Dialogs 1

## 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 records 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
    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 COLOR COL_KEY,
             wa_flight-connid COLOR COL_KEY,
             wa_flight-fldate COLOR COL_KEY,
             wa_flight-seatsocc,
             wa_flight-seatsmax,
             wa_flight-percentage,'%'.
ENDLOOP.
```

## Unit: Database Dialogs 1

## 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 sflight-carrid.



*----------------------------------------------------------------
* Optional:
* Array Fetch to fill the first 5 columns 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 record 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 =
      MODIFY it_flight from wa_flight.
    ENDLOOP.

    SORT it_flight by percentage.

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

      WRITE: / wa_flight-carrid COLOR COL_KEY,
               wa_flight-connid COLOR COL_KEY,
               wa_flight-fldate COLOR COL_KEY,
               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 sflight-carrid.

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.
    WRITE: / 'Authority-Check Error'(001).
ENDCASE.
```

# Internal Program Modularization

## Contents:

- ABAP event blocks
- Subroutines

- 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.

- 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 generate d 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 are processing blocks that are called by the ABAP runtime system. The sequence in 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.

- 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 generating a list. It is called by the ABAP runtime system as soon as you have pressed the execute button.
- **`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 by the user.

- Subroutines are processing blocks with a defined interface that can be called from any processing block using the ABAP statement. Subroutines provide internal program encapsulation.

- You can navigate from the program object list to the subroutines.
- The where-used list for a subroutine displays all the program lines that call the subroutine.

- Ideally, all you need to do to determine the functional scope of the subroutine is to examine the subroutine name, the interface and the comments. If the subroutine contains the functionality you require, then you need the following information to be able to call the subroutine:
    - **Subroutine name**
    - **Interface parameters it accesses (read-only):** the parameters are listed after the `USING` addition. The type and sequence of the interface parameters is important.
    - **Interface parameters it changes:** the parameters are listed after the `CHANGING` addition. The type and sequence of the interface parameters is important.

- When a subroutine is called, all the interface parameters have to be filled with values. A distinction is made between the following parameters:
  - After **USING**, the parameters that the subroutine only needs to read are listed.
  - After **CHANGING**, the parameters that are changed in the subroutine are listed.
- If the subroutine is called from the ABAP processing block by a **PERFORM** statement, the system interrupts the processing block to process the subroutine sequentially. When the last line of the subroutine (**ENDFORM**.) is reached, the system carries processing after the **PERFORM** statement.
- You can track runtime behavior in the debugging mode. This gives you various options:
  - You can go through the entire program, including the subroutine, line by line, using **Single Step**
  - You can go through a processing block line by line using **Execute**. Subroutines are then executed as a whole
  - You can leave single-step processing of a subroutine and return to the calling program using **Return**

- The method used for calling the interface parameters is set in the subroutine interface. The parameters can be called either by reference or by value.
- **Calling by reference:** The address of the actual parameter is called. Within the subroutine, the variable is addressed using the formal parameter name. Changes have an immediate effect on the global variable. If only the formal parameter name is specified in the subroutine interface, then the parameter is called by reference.
- **Calling by value:** When the subroutine is called, a local variant is created with the formal parameter name and the actual parameter value is copied to the formal parameter. There are two types of call by value:
  - **Calling by value:** the formal parameter is listed in the interface after the **`USING`** clause with the addition **`VALUE( <parameter name>).`** When the subroutine is called, the actual parameter is copied to the formal parameter. Changes made to the formal parameter only affect the local copy, not the actual parameter.
  - **Calling by value and result:** the formal parameter is listed in the interface after the **`CHANGING`** clause with the addition **`VALUE( <parameter name>)`**. When the subroutine is called, the actual parameter is copied to the formal parameter. Changes made to the formal parameter initially only affect the local copy. When the **`ENDFORM`** statement is reached, the formal parameter value is copied back to the actual parameter.

- The parameters in the interface are called **formal parameters** , and the parameters that you pass to the subroutine are called **actual parameters** .
- You must have the same number of actual parameters as formal parameters. You cannot have optional parameters. Parameters are assigned in the sequence in which they are listed.
- When you call a subroutine using **`PERFORM,`** the system checks whether the types of the actual parameters in the **`PERFORM`** statement are compatible with the formal parameters. Different kinds of checks are performed for different types:
- **Complete type checks:**
  - **`TYPE D, F, I, T`** or **`<dictionary type>`**. These types are fully specified. The system checks to see if the data type of the actual parameter is identical to the type of the formal parameter in its entirety.
- **Partial type checks of generic types**
  - **`TYPE C, N, P`** or **`X`**. The system checks whether the actual parameter has the type **`C, N, P`** or **`X`**. The length of the parameter and the number of decimal places in the **`DECIMALS`** addition (type **`P`**) are passed from the actual parameter to the formal parameter.
  - **`TYPE <generic dictionary type>`** all unspecified information from generic Dictionary types is inherited by the formal parameter from an actual parameter.
- The interface is defined in the **`FORM`** routine. **`USING`** and **`CHANGING`** in the **`PERFORM`** statement are purely documentary.

## 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…”. (## 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:

1 Airline code: Data element **S_CARR_ID**

2 Return code:
System field **SY-SUBRC**

3 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 **Optional:**

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_CARRID  text
*      -->P_LD_ACTVT  text
*      <--P_RETURN  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

- 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 on 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

- 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 on the desktop, in a report tree, or to the Office
  - 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 *Dialogs: Interfaces*.

- Each list can have a **list header** and up to four lines of **column headers**. There are two different ways to go about using these tools:
- 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 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.

- **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 detemines in which language text elements will be displayed.
- Text symbols are another kind of **text element**. These are special text literal data objects. Compared to normal text literals, text symbols have the advantage that they can be translated into different languages without having to change a program's source code. 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 character string three characters long)
  - **`'<text>'(<xxx>)`** (where **`xxx`** is a character string three characters long)

- In executable programs (type 1), lists are automatically displayed after their corresponding event blocks have been processed. These processing blocks must, however, contain a list creation statement. These are **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 detail lists.
- The ABAP runtime system:
  - Displays the basic list 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`** each time you double-click on an entry. If you are using a standard status, this happens automatically every time you choose the Choose icon, the *Choose* menu item in the Edit menu, or the function key F2.
  - Displays detail lists after the **`AT LINE-SELECTION`** event block has been processed and increases the value contained in **`sy-lsind`** by one.
  - Displays the detail list from the previous level in the list hierarchy (n-1) every time you choose the green arrow icon from the current detail list (n).

- The lists in the example program should function as follows:
  - The basic list should display the text 'Basic List' and system field **`sy-lsind`**.
  - The user should be able to call the initial detail list using a double-click or by choosing its corresponding icon from the application toolbar or its menu entry or by using the function key F2. Then the 'Detail list' appears and the system field **`sy-lsind`** has the value 1.
  - Repeating this action should call the second detail list, where system field **`sy-lsind`** contains the value 2 (representing the current detail list level).
  - Repeating this action increases the **`sy-lsind`** value by one every time up to a value of twenty (the total number of detail lists supported).
  - Choosing the green arrow takes the user back a single detail list level at a time until the basic list is reached.

- A detail 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 detail lists. Whenever you use **`WRITE`**, **`SKIP`**, or **`ULINE`** with this event block, you fill the detail list buffer for the next level (the detail list buffer with a level value one greater than the level on which the user performed his or her action).
  - You can pre-determine navigation between detail lists by querying system field **`sy-lsind`** at the event block **`AT LINE-SELECTION`**.

- We will now write a program using both basic lists and detail lists:
- The basic list in your program should contain flight data such as carrier ID flight number, departure city and airport, as well as departure and arrival times. This data can be found in the database table **SPFLI**.
- The user should be able to access information about any particular flight by double-clicking on a line with a carrier ID and flight number. Flight date and occupancy should be displayed. This data can be found in the database table **SFLIGHT**. You must use the **SPFLI** key fields in this detail list in order to read the appropriate data in **SFLIGHT**. The following slides demonstrate how this is done.
- The sample program is named **SAPBC400UDD_EXAMPLE_2** and is part of development class **BC400**.

- When the event **AT LINE-SELECTION** is processed, a program's data objects contain the same values as they did before the 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 detail list. You can predetermine which information should be classified by its line position when you are creating a 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.

- As soon as the 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 the Choose icon, the values for this line stored in the `HIDE` area are inserted into their corresponding data objects.

- You create a detail list by filling the detail 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.

## 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). (**HIDE**)

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. Use a structure to display the following fields of the database table **SBOOK** on the detail list:

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

**CLASS,**
**ORDER_DATE,**
**SMOKER,**
**CANCELLED,**
**LOCCURAM,**
**LOCCURKEY**.

1-6 Display the fields **BOOKID** and **CUSTOMID** in the color **COL_KEY**.

1-7 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.
```

## Unit: User Dialogs: Lists

## Topic: Detail lists

### Sample solution 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 sflight-carrid.



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 COLOR COL_KEY,
                 wa_flight-connid COLOR COL_KEY,
                 wa_flight-fldate COLOR COL_KEY,
                 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.
```

```
* 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 data objects
    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,
               wa_sbook-loccuram CURRENCY wa_sbook-loccurkey,
               wa_sbook-loccurkey.
    ENDSELECT.
  ENDIF.
```

# 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 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 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.

- Selection screens are designed to present users with an input template allowing them to enter selections, which reduce the amount of data that has to be read from the database. The following possibilities are available to the user:
  - Entries to single fields
  - **Complex entries:** Intervals, operations, patterns
  - Saving selections fields filled with values as **variants**
  - Input help and search helps are available 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. Functions of selection options programming include:
  - 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 you refer an input field to an ABAP Dictionary object to which a search help is assigned, the system automatically provides the corresponding possible values help.
- You can adapt the possible values help to meet your own requirements by defining a search help in 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.

- You can define and store 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 end users 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*.

- In an executable program, a single **PARAMETERS** statement is sufficient to generate a standard selection screen.
- The **PARAMETERS <name> TYPE <type>** statement and the **PARAMETERS <name> LIKE <data object>** statement both 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.

- If you have used the **PARAMETERS** statement to program an input field as a key field for a database table, you can use a **WHERE** clause at the **SELECT** statement to limit data selection to this value.
- In the example above only those data records are read from database table **SPFLI** whose key field **CARRID** have the same value as is contained in data object **pa_car** at runtime.

- 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 declared. This data object serves as a target field during database selection, the selection table as a pool of possible values. A special version of the **`WHERE`** clause exists for database selection. It determines whether or not the database contains the corresponding field within its pool of **possible values**.

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

- 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 ).**

- 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 event block **INITIALIZATION.**
- Each time the user presses Enter, a pushbutton, a function key, or chooses 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 action:
  - 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.
- For more detailed information on the **MESSAGE** statement, refer to the keyword documentation as well.
- Additional information can be found in the keyword documentation for **AT SELECTION-SCREEN.**

- As an example of an additional input check with error dialog, an input field for the airline ID needs to be added to the program:
- An authorization check is carried out 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.

## 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`**.

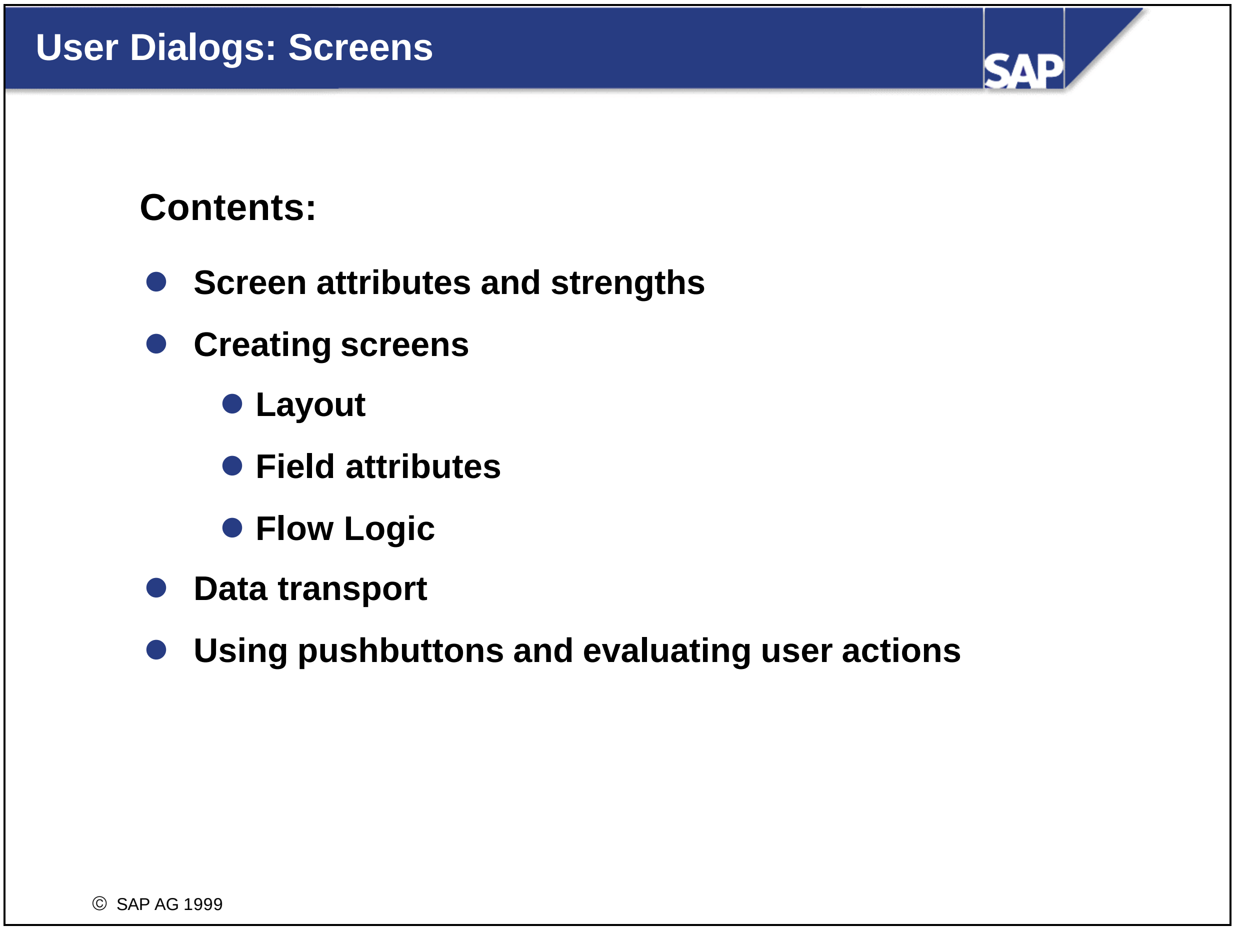
1-4 Change your program so that the user cannot progress from the selection screen if the authorization check against the authorization object **`S_CARRID`** fails. If the authorization check fails, display a suitable error message from message class BC400, and allow the user to enter a different value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

## 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 sflight-carrid.
* Data field for complex restrictions applied to connection id
SELECT-OPTIONS: so_con FOR wa_flight-connid.



* First event processed after leaving the selection screen
AT SELECTION-SCREEN.
  AUTHORITY-CHECK OBJECT 'S_CARRID'
    ID 'CARRID' FIELD pa_car
    ID 'ACTVT'  FIELD actvt_display.
  IF sy-subrc <> 0.
* Return to selection screen again and display message in status *
bar
    MESSAGE e045(bc400) 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 COLOR COL_KEY,
             wa_flight-connid COLOR COL_KEY,
             wa_flight-fldate COLOR COL_KEY,
             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 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 in what order checks are then performed.
- When an error is detected, the corresponding field is called and displayed ready for input. 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 in which direction 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.

- You can call screens from any ABAP processing block that you want.
- You can link several screens to one another and then call them from within a program by simply calling the first screen.
  - Some ABAP programs are made up exclusively of screens and their correponding ABAP processing blocks. In this case the first screen is called directly using a transaction code.

- In the following units you will develop a program that 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 carrier. 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 II* for more about this process.

- The major steps in creating a screen:
  - specifying its properties (*Screen Attributes*)
  - specifying its layout (in Fullscreen Editor)
  - defining attributes for the elements on the screen (*Field List)*
  - programming its flow logic

- 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.

- 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 *Next screen* input field, the 0 is suppressed, since it is the same as the initial value of the field.
- 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. In order 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.

- You can also edit screen field attributes by choosing *Field list*.
- The field list is then displayed as a tab.
- This same function can also be accessed in a different format from within the Graphical Screen Painter.

- 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

- The statement **TABLES** declares an internal data object that serves as an interface for the screen. **TABLES** always refers to a structure that is 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 helps to avoid errors as well.

- 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 the work area designated as the screen interface and program data objects.

- On the last development level, the program should allow the user to change data in the database. The user should be able to change the fields **FLTIME** and **DEPTIME**. To enable the user to change data for several airlines, a basic list of the airlines for which the user is allowed to change data should be displayed. The user reaches the change screen by double-clicking. Once the changes have been made successfully, the user returns to the basic list. However, a new basic list is not created. 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**.
- Advantages of this method:
  - 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 find more detailed information on this topic in the *Database Dialogs II* unit.
  - The program can be extended: Additional information from the data record can be displayed on the screen without having to make many changes.

- 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.

- 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:
  - 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 can be changed.
  - By choosing the 'Back' pushbutton, the user 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.
  - Choose 'Save' to write all of your changes to the database. We will take a closer look at this step in the unit *Database Dialogs II*. The pushbutton is already prepared in the following section. The user should return to the basic list after the pushbutton is chosen and a message should be displayed in the status bar.
  - After pressing Enter, the screen is redisplayed.
- Changes to the database are discussed in the unit *Database Dialogs II.*
- 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.

- In order to define functions for specific pushbuttons, these pushbuttons must be assigned function codes. You can do this either on the attributes screen or in the field list in the graphical Layout Editor.

- The **`OK_CODE`** field is a data object into which corresponding function codes are fed after every user action.
- The name **`OK_CODE`** must be inserted as the last line in each screen's field list.
- 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.** You will learn more about this in step three.
- 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.

- 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.

- You can use the ABAP statement `SET SCREEN <nnnn>` within a PAI module to override dynamically the value set in the *Next screen* attribute.
- 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.

- With the help of the OK_CODE field, different program logic can now be processed by the PAI modules depending on what the user inputs.
- If an OK_CODE field is not initialized, errors can occur since not every pushbutton is required to have a function code. There are two ways of doing this:
  - Initialize the **OK_CODE** 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.

- 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 on 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 statement `MODULE ... OUTPUT`, whereas modules defined with `MODULE ... INPUT`, can only be called at PAI events.

- 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 and message 047 is issued, or, if it has been filled, a detail list is displayed. Message 047 appears in the status bar of the screen subsequently displayed.
    - Choosing `'SAVE'` causes an S message to be displayed and the user then branches to a basic list or a detail list, the same as when `'BACK'` is chosen.

## Unit: Screen

## Topic: Creating Screens

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

- Create screens
- Call existing from the program

Program **SAPBC400UDT_DYNPRO_1** 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_1**

**Template:** **SAPBC400UDT_DYNPRO_1**

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: Screen

## 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_2** |

2-1 Extend your program, **ZBC400_##_DYNPRO**, or copy the relevant model solution **SAPBC400UDS_DYNPRO_1** and give it the name **ZBC400_##_DYNPRO_2**. 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 The Program should later be extended 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 on 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.

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: Screen

## 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_3`**

3-1 Extend your program, **`ZBC400_##_DYNPRO`**, or copy the relevant model solution **`SAPBC400UDS_DYNPRO_2`** and give it the name **`ZBC400_##_DYNPRO_3`**. 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>oder icon<br>ICON_SYSTEM_SAVE | **SAVE** |

3-3 Name the **`OK_CODE`** field on the screen and declare 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:** | List |

| Function code | **Action** | Next screen |
|---|---|---|
|  | Information message No. 060(BC400) |  |
| Other | None | Screen 100 |

3-5 Ensure that pressing 'Enter' always displays screen 100, regardless of the navigation history. Do this using either of the two methods for initializing the **`OK_CODE`**.

## Unit: Screen

## Topic: Creating Screens

### Model solution: Program SAPBC400UDS_DYNPRO_1

```
*&---------------------------------------------------------------*
*& Report        SAPBC400UDS_DYNPRO_1                            *
*&                                                               *
*&---------------------------------------------------------------*

REPORT  sapbc400uds_dynpro_1.
CONSTANTS: actvt_display TYPE activ_auth VALUE '03',
           actvt_change TYPE activ_auth VALUE '02'.

PARAMETERS: pa_agnum TYPE s_agncynum.
DATA: wa_cust TYPE sbc400cust,
      it_cust TYPE sbc400_t_sbc400cust.
DATA: wa_sbook TYPE sbook.
```

```
START-OF-SELECTION.
  SELECT id name FROM scustom INTO TABLE it_cust.
  SELECT carrid connid fldate bookid customid
         FROM sbook INTO CORRESPONDING FIELDS OF wa_sbook
         WHERE agencynum = pa_agnum.
  AUTHORITY-CHECK OBJECT 'S_CARRID'
```

```
          ID 'CARRID' FIELD wa_sbook-carrid
          ID 'ACTVT'  FIELD actvt_display.
   IF sy-subrc = 0.
    READ TABLE it_cust INTO wa_cust
         WITH TABLE KEY id = wa_sbook-customid.
    WRITE: / wa_sbook-carrid COLOR COL_KEY,
             wa_sbook-connid COLOR COL_KEY,
             wa_sbook-fldate COLOR COL_KEY,
             wa_sbook-bookid COLOR COL_KEY,
             wa_cust-name COLOR COL_KEY.
   ENDIF.
  ENDSELECT.

* Program continues here after having selected a booking on the   *
basic list.
AT LINE-SELECTION.
 IF sy-lsind = 1.
    AUTHORITY-CHECK OBJECT 'S_CARRID'
          ID 'CARRID' FIELD wa_sbook-carrid
          ID 'ACTVT' FIELD actvt_change.
    IF sy-subrc = 0.
        CALL SCREEN 100.
    ELSE.
        MESSAGE s047(bc400) WITH wa_sbook-carrid.
    ENDIF.
 ENDIF.
```

**Unit: Screen**

**Topic: Data transport**

## Model solution: Program SAPBC400UDS_DYNPRO_2

```
*&---------------------------------------------------------------*
*& Report         SAPBC400UDS_DYNPRO_2                          *
*&                                                              *
*&---------------------------------------------------------------*

REPORT  sapbc400uds_dynpro_2.
CONSTANTS: actvt_display TYPE activ_auth VALUE '03',
           actvt_change TYPE activ_auth VALUE '02'.

TABLES: sdyn_book.
PARAMETERS: pa_agnum TYPE s_agncynum.
DATA: wa_cust TYPE sbc400cust,
      it_cust TYPE sbc400_t_sbc400cust.
DATA: wa_sbook TYPE sbook.
```

```
START-OF-SELECTION.
  SELECT id name FROM scustom INTO TABLE it_cust.
  SELECT carrid connid fldate bookid customid
         FROM sbook INTO CORRESPONDING FIELDS OF wa_sbook
         WHERE agencynum = pa_agnum.
  AUTHORITY-CHECK OBJECT 'S_CARRID'
          ID 'CARRID' FIELD wa_sbook-carrid
          ID 'ACTVT'  FIELD actvt_display.
   IF sy-subrc = 0.
     READ TABLE it_cust INTO wa_cust
         WITH TABLE KEY id = wa_sbook-customid.
     WRITE: / wa_sbook-carrid COLOR COL_KEY,
              wa_sbook-connid COLOR COL_KEY,
              wa_sbook-fldate COLOR COL_KEY,
              wa_sbook-bookid COLOR COL_KEY,
              wa_cust-name COLOR COL_KEY.
* Hide key fields of database table SBOOK and customer name
    HIDE: wa_sbook-carrid, wa_sbook-connid, wa_sbook-fldate,
          wa_sbook-bookid, wa_cust-name.
   ENDIF.
  ENDSELECT.
 CLEAR wa_sbook.
```

```
AT LINE-SELECTION.
 IF sy-lsind = 1.
    AUTHORITY-CHECK OBJECT 'S_CARRID'
          ID 'CARRID' FIELD wa_sbook-carrid
```

```
         ID 'ACTVT' FIELD actvt_change.
   IF sy-subrc = 0.
     SELECT SINGLE * FROM sbook INTO wa_sbook
         WHERE carrid   = wa_sbook-carrid
          AND  connid   = wa_sbook-connid
          AND  fldate   = wa_sbook-fldate
          AND  bookid   = wa_sbook-bookid.

     IF sy-subrc <> 0.
         MESSAGE i176(bc400).
     ELSE.
         MOVE-CORRESPONDING wa_sbook TO sdyn_book.
         MOVE wa_scust-name TO sdyn_book-name.
         CALL SCREEN 100.
     ENDIF.
   ELSE.
       MESSAGE s047(bc400) WITH wa_sbook-carrid.
   ENDIF.
ENDIF.
CLEAR wa_sbook.
```

## Unit: Screen

## Topic: Field Transports and Next Screen Processing

### Model solution: Program SAPBC400UDS_DYNPRO_3

```
*&---------------------------------------------------------------*
*& Report         SAPBC400UDS_DYNPRO_3                          *
*&                                                              *
*&---------------------------------------------------------------*

REPORT  sapbc400uds_dynpro_3.
CONSTANTS: actvt_display TYPE activ_auth VALUE '03',
           actvt_change TYPE activ_auth VALUE '02'.

TABLES: sdyn_book.
PARAMETERS: pa_agnum TYPE s_agncynum.
DATA: wa_cust TYPE sbc400cust,
      it_cust TYPE sbc400_t_sbc400cust.
DATA: wa_sbook TYPE sbook.
DATA: ok_code LIKE sy-ucomm, save_ok LIKE ok_code.
```

```
START-OF-SELECTION.
  SELECT id name FROM scustom INTO TABLE it_cust.
  SELECT carrid connid fldate bookid customid
         FROM sbook INTO CORRESPONDING FIELDS OF wa_sbook
         WHERE agencynum = pa_agnum.
  AUTHORITY-CHECK OBJECT 'S_CARRID'
          ID 'CARRID' FIELD wa_sbook-carrid
          ID 'ACTVT'  FIELD actvt_display.
   IF sy-subrc = 0.
    READ TABLE it_cust INTO wa_cust
         WITH TABLE KEY id = wa_sbook-customid.
    WRITE: / wa_sbook-carrid COLOR COL_KEY,
             wa_sbook-connid COLOR COL_KEY,
             wa_sbook-fldate COLOR COL_KEY,
             wa_sbook-bookid COLOR COL_KEY,
             wa_cust-name COLOR COL_KEY.
* Hide key fields of database table SBOOK and customer name
    HIDE: wa_sbook-carrid, wa_sbook-connid, wa_sbook-fldate,
          wa_sbook-bookid, wa_cust-name.
   ENDIF.
  ENDSELECT.
 CLEAR wa_sbook.
```

```
AT LINE-SELECTION.
 IF sy-lsind = 1.
    AUTHORITY-CHECK OBJECT 'S_CARRID'
          ID 'CARRID' FIELD wa_sbook-carrid
          ID 'ACTVT' FIELD actvt_change.
    IF sy-subrc = 0.
      SELECT SINGLE * FROM sbook INTO wa_sbook
          WHERE carrid   = wa_sbook-carrid
```

```
           AND  connid   = wa_sbook-connid
           AND  fldate   = wa_sbook-fldate
           AND  bookid   = wa_sbook-bookid.

      IF sy-subrc <> 0.
          MESSAGE i176(bc400).
      ELSE.
          MOVE-CORRESPONDING wa_sbook TO sdyn_book.
          MOVE wa_scust-name TO sdyn_book-name.
          CALL SCREEN 100.
      ENDIF.
    ELSE.
        MESSAGE s047(bc400) WITH wa_sbook-carrid.
    ENDIF.
 ENDIF.
 CLEAR wa_sbook.
```

```
*&--------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&--------------------------------------------------------------*
*       dynamical screen flow depending on user action
*---------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  save_ok = ok_code .
* Clear OK-Code Field in order to have it initialized on next     *
screen
  CLEAR ok_code .
  CASE save_ok.
    WHEN 'BACK'.
      SET SCREEN 0.
    WHEN 'SAVE'.
      MOVE-CORRESPONDING sdyn_book TO wa_sbook.
* Saving the changed dataset will be implemented later
      message i060(bc400).
      SET SCREEN 0.
    WHEN OTHERS.
      SET SCREEN 100.
  ENDCASE.
ENDMODULE.                             " USER_COMMAND_0100  INPUT
```

# Reuse Components

## Contents:

- Function groups and function modules
- Objects and methods
- Business objects and BAPIs
- Logical databases

- 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 methods of business objects, which are made available using the Business Object Repository (BOR).
- **Logical databases** are data collection programs that can be coupled with executable programs. In a logical database, the data are transferred using logical hierarchy structures. Logical databases also make selection screens available and contain authorization checks.

- 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 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 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. Changing parameters can be optional.
  - **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.

- The global data in the function group remain 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.

- 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 could avoid programming this dialog box, if you could 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.
- 1. First method: In the Debugger, set a breakpoint at **`CALL SCREEN`**. If successful, the actual processing block (subroutine, function module or event block) will be listed under '**`CALLS'`** in the Debugging mode. You can then examine the call and the parameters passed to the interface.
- 2. Second method: In the Debugger, set a breakpoint at **`CALL FUNCTION`**. If successful, the actual processing block (subroutine, function module or event block) will be listed under '**`CALLS'`** in the Debugging mode. You can then examine the call and the parameters passed to the interface.
- 3. Third method: Click a text field in the standatrd 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.
- 4. Fourth method: In the *Save* dialog box, display the F1 help and then *Technical info*. Navigate to the screen, examine the flow logic and its modules.
- 5. Fifth method: In the component hierarchy, mark the component that you are interested in (in this case, Basis Services), select it, navigate to the Repository Information System, look under *Programming* -> *Function Builder* -> *Function modules* and select *Only released*.

- 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 how to handle 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 Function Builder test function to test function modules with table parameters.
- You can create test sequences.

- 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 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, the system displays the names of the interface parameters, while on the right, it displays the program's data objects.

- To do this, use a statement pattern in the ABAP Editor (the *Pattern* button), 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 parameter values are assigned explicitly by the name of the actual parameter. From the point of view of 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.

- **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 behaviour that have already been defined in one or more superclasses.

- 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.
- 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 the flight 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 "booking list" and "maximum number of free seats" private attributes.
- 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 added to 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.

- 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.

- 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. In this statement, a special method called **`CONSTRUCTOR`** is called implicitly. Any parameters required by the constructor must be specified in the **`CREATE OBJECT`** statement. 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.

- 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 AddIns:**
  An object-oriented enhancements technology, which combines the advantages of existing technologies. If Business AddIns 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 frontend. This task is performed by the Central Control Framework.
- **Pilot projects** are already object-oriented.

- 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 frontend.
- If Custom Controls are to be included on the frontend, 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 display 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 use the 'Subtotal' function to structure control level lists: select the You can use the 'Subtotal' function to structure control level lists: select the columns (non-numeric columns only) that you want to use and the corresponding control lFor '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. Saving variants: see 'Advanced Techniques'.

- 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_CUSTOMER_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 **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.

- **CL_GUI_ALV_GRID** contains many methods. To display an internal table 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**. 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 frontend control already knows the row type of the internal table and reference fields.

- 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, this area behaves similarly to a subscreen area.
- 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 attrbutes using *Min. row* and *Min. column*. You determine the maxium size of the area when you create it.

- 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`**.

- The **CREATE OBJECT** 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 number of the container screen and the program name.
- 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.

- 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.

- A BAPI is a point of entry to the R/3 System - that is, a point at which the R/3 System provides access to business data and processes.
- Each object in the BOR can have many methods, one or more of which can be implemented as BAPIs.
- BAPIs can have various functions:
  - Creating an object
  - Retrieving the attributes of an object
  - Changing the attributes of an object

- 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.

- An example business object from the training course data model is called FlightBooking. It contains a booking. Each booking is uniquely identifiable from its key information: **`AirlineCarrier`** , **`ConnectionNumber`** (flight number), **`DateOfFlight`**, **`BookingNumber`**. The following methods are available for this object:
  - **`FlightBooking.GetDetail`** returns detailed information on a booking
  - **`FlightBooking.CreateFromData`** creates a booking
  - **`FlightBooking.Cancel`**
  - **`FlightBooking.GetList`** returns a list containing details of all the bookings for that flight. Generally displays fewer details for each booking than **`GetDetail`**.

- You can display more information on business objects and the BAPIs that belong to them using BAPI Explorer Information. The screen is in 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).

- Symbols are used to indicate business objects, key attributes, and BAPIs. You can display the key to these symbols using the *Display Legend* button.

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

- 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.

- BAPIs with standardized names contain standard methods. Three of the most important are listed here.

- In Release 4.6 BAPIs have been 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 detailed information display window, choose the *Tools* tab.
    - Choose *Function Builder:* The system displays the name of the function module.
    - Choose *Display*

- 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 .

- Every BAPI contains an interface parameter, **RETURN**, which contains information about errors that occur. This parameter is always of an ABAP Dictionary type. This means that you must include a structure of an identical type in your program.

- 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.

- 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.

- 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).

- 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.

- 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.

- 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**.

## 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_5** |

1-1 your program, **ZBC400_##_DYNPRO**, or copy the relevant model solution **SAPBC400UDS_DYNPRO_**4 and give it the name **ZBC400_##_DYNPRO_5**. 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. Make the system react as follows to the user’s input:

- 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 programthat 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 For the class. **CL_GUI_CUSTOM_CONTAINER**, which parameters of the method **CONSTRUCTOR** are mandatory?

2-4-2 For the class. **CL_GUI_ALV_GRID**, which parameters of the method **CONSTRUCTOR** are mandatory?

2-5 Create a container control area on the screen. Make sure you give the area a name.

2-7 Define two reference variables, one for the **CL_GUI_CUSTOM_CONTAINER** class and one for the **CL_GUI_ALV_GRID** class.

2-8 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.

2-9 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**

2-10 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** (Typ SY-CUCOL) : First column of the dialog window
**START_ROW** (Typ 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
                title     = text-002
           IMPORTING
                answer    = answer.
      case answer.
        when 'N'.
        leave to screen 100.
        when 'J'.
        leave to screen 0.
     endcase.
    WHEN 'SAVE'.
      MESSAGE i060(bc400).
      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:**

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. Call the 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
```