

# BC ABAP Reporting Tutorial



**Release 4.6B**



## Copyright

© Copyright 2000 SAP AG. All rights reserved.

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

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

Microsoft®, WINDOWS®, NT®, EXCEL®, Word® and SQL Server® are registered trademarks of Microsoft Corporation.

IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, MVS/ESA®, RS/6000®, AIX®, S/390®, AS/400®, OS/390®, and OS/400® are registered trademarks of IBM Corporation.

ORACLE® is a registered trademark of ORACLE Corporation, California, USA.

INFORMIX®-OnLine for SAP and Informix® Dynamic Server™ are registered trademarks of Informix Software Incorporated.

UNIX®, X/Open®, OSF/1®, and Motif® are registered trademarks of The Open Group.







HTML, DHTML, XML, XHTML are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Laboratory for Computer Science NE43-358, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

JAVA® is a registered trademark of Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA.

JAVASCRIPT® is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

SAP, SAP Logo, mySAP.com, mySAP.com Marketplace, mySAP.com Workplace, mySAP.com Business Scenarios, mySAP.com Application Hosting, WebFlow, R/2, R/3, RIVA, ABAP, SAP Business Workflow, SAP EarlyWatch, SAP ArchiveLink, BAPI, SAPPHIRE, Management Cockpit, SEM, are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other products mentioned are trademarks or registered trademarks of their respective companies.

## Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax
	Tip

## Contents

<b>BC ABAP Reporting Tutorial</b> .....	<b>6</b>
<b>BC - ABAP Workbench: Reporting Tutorial</b> .....	<b>7</b>
<b>Note to the Reader</b> .....	<b>8</b>
<b>Overview</b> .....	<b>9</b>
<b>Requirements</b> .....	<b>10</b>
<b>Syntax conventions</b> .....	<b>11</b>
<b>Platform</b> .....	<b>12</b>
<b>Naming Conventions for SAP Objects</b> .....	<b>13</b>
<b>Further Documentation</b> .....	<b>14</b>
<b>ABAP Query</b> .....	<b>15</b>
<b>Overview of ABAP Query</b> .....	<b>16</b>
<b>User Groups</b> .....	<b>17</b>
<b>Functional Areas</b> .....	<b>18</b>
<b>Query</b> .....	<b>20</b>
<b>Basic Settings for a Query</b> .....	<b>21</b>
Creating a Basic List .....	23
Layout of a Basic List .....	24
Other presentations of a basic list .....	26
Creating Statistics .....	27
<b>Creating a Report Using ABAP Statements</b> .....	<b>29</b>
<b>Overview</b> .....	<b>30</b>
<b>The ABAP Programming Language</b> .....	<b>31</b>
<b>Creating an ABAP Program</b> .....	<b>32</b>
The ABAP Editor .....	34
Comment lines .....	35
ABAP Statements for Screen Display .....	36
Declaring data .....	38
Handling Variables .....	39
Text Layout .....	40
List Headers .....	42
Dynamic List Headings .....	43
Database Access .....	44
The SELECT Statement .....	45
Selection Criteria .....	47
PARAMETERS .....	48
SELECT-OPTIONS .....	50
Variants .....	52
Nested SELECT Statements .....	53
Performance of the SELECT Statement .....	55
<b>Sample Program: nested SELECT statements</b> .....	<b>56</b>
<b>Logical Databases and Events</b> .....	<b>59</b>
<b>Overview: logical databases</b> .....	<b>60</b>
<b>Creating a Report That Uses a Logical Database</b> .....	<b>61</b>
Retrieving Data Using a Logical Database .....	62

Sample program: reading SBOOK using a logical database .....	64
<b>Hierarchy of a Logical Database .....</b>	<b>65</b>
<b>Events .....</b>	<b>67</b>
GET <tablename> LATE .....	68
<b>Sample program: report using logical database .....</b>	<b>70</b>
<b>Structures and Internal Tables .....</b>	<b>72</b>
<b>Internal Tables .....</b>	<b>73</b>
<b>Structures and Internal Tables .....</b>	<b>74</b>
Declaring a structure .....	75
Declaring an internal table .....	77
Defining the Work Area .....	78
Filling internal tables .....	79
Working with Internal Tables .....	80
<b>IF statement .....</b>	<b>83</b>
<b>Sample program: internal tables .....</b>	<b>84</b>
<b>Processing control levels .....</b>	<b>87</b>
Control Level Processing .....	89
Calculating Totals .....	91
Sample program: processing control levels .....	92
<b>Interactive Reporting .....</b>	<b>95</b>
<b>Concept: interactive reporting .....</b>	<b>96</b>
New Event Keywords .....	97
Creating a Detail List .....	98
Valid Line Selection .....	100
Defining a User Interface .....	101
Sample program for interactive reporting .....	103

# BC ABAP Reporting Tutorial

## BC - ABAP Workbench: Reporting Tutorial

The *ABAP Reporting Tutorial* provides an introduction on how to create report lists.

Unit 1 presents the menu-supported tool for generating report lists.

Unit 2 introduces report programming, including the fundamental topics of reading data from a database and writing data to the screen.

Unit 3 explains what logical databases are and how to use them. It also describes the event concept on which ABAP programs are based.

Unit 4 introduces internal tables.

Unit 5 offers an introduction to interactive reporting. It includes the most important statements for creating details lists.

The tutorial presents only an overview of the extensive topic of reporting. You can find further information in the [ABAP User Manual \[Ext.\]](#).

---

**Note to the Reader**

## Note to the Reader

You should read the following remarks before starting to work with the *ABAP Workbench* documentation. Begin the *Reporting Tutorial*. This section discusses the following topics:

[Requirements \[Page 10\]](#)

[Syntax Conventions \[Page 11\]](#)

[Platforms \[Page 12\]](#)

[Naming Conventions for SAP Objects \[Page 13\]](#)

[Further Documentation \[Page 14\]](#)



## Overview

---

## Requirements

## Requirements

The *ABAP Workbench* documentation: *Reporting Tutorial* addresses readers who are familiar with the R/3 System.

To be able to work with this documentation, you must know how to log on to the R/3 System. If you read this documentation online, you are already logged on. Otherwise, ask your system administrator for your user name, client, and password. For more information on logging on to the R/3 System, see the documentation *Introduction to the R/3 System*.

To work with this tutorial, you need R/3 System Release 4.5 or higher. If you are not sure which version is running on your machine, ask your system administrator.

## Syntax conventions

The conventions for syntax statements in this documentation are as follows:

Key	Definition
<b>STATEMENT</b>	Keywords and options of statements are uppercase.
<b>&lt;variable&gt;</b>	Variables, or words that stand for values that you fill in, are in angle brackets. Do not include the angle brackets in the value you use (exception: field symbols).
<b>[ ]</b>	Square brackets indicate that you can use none, one, or more of the enclosed options. Do not include the brackets in your option.
<b> </b>	A bar between two options indicates that you can use either one or the other of the options.
<b>( )</b>	Parentheses are to be typed as part of the command.
<b>,</b>	The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.
<b>&lt;f<sub>1</sub>&gt; &lt;f<sub>2</sub>&gt;</b>	Variables with indices mean that you can list as many variables as you like. They must be separated with the same symbol as the first two.
<b>.....</b>	Dots mean that you can put anything here that is allowed in the context.

In syntax statements, keywords are in upper case, variables are in angle brackets. You can disregard case when you type keywords in your program. **WRITE** is the same as **write** is the same as **write**.

Output on the output screen is either shown as a screen shot or in the following format:

**Screen output.**

---

**Platform**

## Platform

The screen shots included in this documentation are taken under Windows NT. If you use another platform, for example Motif, the screens you see in your system may differ slightly from the shots shown here. However, this does not affect the functionality and handling of the system described in this tutorial.

## Naming Conventions for SAP Objects

In this tutorial, you will have to name a large number of new ABAP objects. To do this, you must know and adhere to two R/3 naming conventions.

1. All objects you create must have names beginning with **Y** or **Z**. This ensures that the names differ from those of the R/3 System versions.
2. The names you choose must be unique. The system warns you if a name already exists. Many development objects are used throughout the system. Therefore, this convention prevents data from being lost or modified unintentionally.

For clarity reasons, you should use the initials of your name within your program names. Thus you can avoid conflicts with other users who might work with the tutorial at the same time.

If, for example, Anne Jones should name an object in the form **Y<xx>ID**, she would choose **YAJID**. Her object then clearly differs from that of user Robert Smith: **YRSID**.



Non-observance may result in a loss of data! Therefore, memorize these two conventions before continuing with the tutorial.

## Further Documentation

The following documentation contain detailed information about the ABAP Workbench and how to program with ABAP:

- [ABAP Workbench Tutorial \[Ext.\]](#). This documentation introduces the most important tools of the ABAP Workbench. It teaches you how to program a simple ABAP application.
- [ABAP Workbench: Tools \[Ext.\]](#). This documentation contains a detailed description of the individual Workbench tools and tells you how to use them for programming.
- [ABAP User Manual \[Ext.\]](#). This documentation contains a detailed description of the individual parts of an ABAP program. It introduces you to ABAP basics, to report programming, transaction development, and to special techniques for advanced programmers.
- [BC - ABAP Query \[Ext.\]](#). This documentation contains a detailed description of the ABAP Query tool.

## ABAP Query

This unit introduces the ABAP Query. The ABAP Query is a tool that allows you to present data from database tables in report lists without having any programming knowledge.

[Overview on ABAP Query \[Page 16\]](#)

[User Groups \[Page 17\]](#)

[Functional Areas \[Page 18\]](#)

[Query \[Page 20\]](#)

[Basic Settings for a Query \[Page 21\]](#)

[Layout of a Basic List \[Page 24\]](#)

[Other Presentations of a Basic List \[Page 26\]](#)

[Creation of a Basic List \[Page 23\]](#)

[Creation of Statistics \[Page 27\]](#)

---

**Overview of ABAP Query**

## Overview of ABAP Query

This unit introduces the ABAP Query. You need not know anything about the ABAP programming language.

After completing this lesson, you

- understand the organization of ABAP Query
- can create basic lists
- can modify the layout of a basic list
- can create statistics.

## Organization of ABAP Query

ABAP Query is a tool that enables you to create lists by navigating through menus. It does not require programming knowledge.

An administrator must first make the required settings to allow the user to work with ABAP Query. The necessary steps are shown in the exercises of this unit.

ABAP Query consists of three components: queries, functional areas, and user groups. The functional areas provide the user with an initial set of data in accordance with the task to be accomplished. All users must be members of at least one user group. All members of one user group can access the same data as well as the same programs (queries) to create lists.

The exercises below introduce the components of ABAP Query. At the end of the unit, ABAP Query will be implemented completely for one task area and you will have used it to create an output report.



To be able to do the exercises below, you need the authorization for creating user groups, functional areas, and queries. If you are not sure whether or not you have it, ask your system administrator.



## User Groups


A user group is a collection of users that work with about the same data and carry out similar tasks.

The members of a user group can use all programs (queries) created by any user of the group. Changes to such a program are at once visible to all users. This ensures that all members of a user group use the same evaluation programs.

The exercise below shows you how to create a user group.



Create a user group and enter your own name as member.

1. Call the *ABAP Workbench*.
2. Go to the maintenance transaction for user groups with *Utilities* → *ABAP Query* → *User groups*.
3. Choose the possible entries button to see a list of all existing user groups.
4. Enter the name **TEST** in field *User group*. If user group TEST already exists, choose a different name.
5. Choose *Create*.
6. Enter a descriptive title in field *User group* and press **ENTER**.  
You go back to the initial screen of the maintenance transaction.
7. Choose *Assign users and functional areas*.
8. Enter your R/3 System user ID in table *User and Change Authorization for Queries*.
9. If the above steps were successful, you should now see this window:  [\[Ext.\]](#)
10. Save the user groups.
11. End the maintenance transaction for user groups.

Check your work.

In the exercise above, you created a user group. Now call the maintenance transaction for user groups and use **F4** to display the list of existing user groups. The list must include the user group you just created.

## Functional Areas

# Functional Areas

The purpose of ABAP Query is to present data from one or more database tables in lists. Depending on the complexity of the database tables, it may not be easy for the user to select the necessary data correctly.

By creating functional areas, you can initially select this data. This ensures that the data is presented to the ABAP Query user in a meaningful way to accomplish the task, and that only the data that the user may use is presented.

Within the functional areas, you should combine the data into [functional groups \[Ext.\]](#) to improve clarity.


The exercise below shows you how to create a functional area.



### Task description:



Create a functional area for the flight booking example. Let the users of the functional area evaluate data from database table SFLIGHT. Therefore, you must provide the information relating to the flight connections as well as the data relevant for the bookings. For more clarity, create the functional groups *Flight connection* and *Flight bookings*. Make the fields SFLIGHT-PLANETYPE and SFLIGHT-CURRENCY invisible to the users.

1. Call the *ABAP Workbench*.
2. Choose *Utilities* → *ABAP Query* → *Functional areas* to branch to the maintenance transaction for functional areas.
3. In the table of all functional areas, check if functional area DEMO\_QUERY was already created.
4. Enter the name **DEMO\_QUERY** in field *Functional area*. If a functional area DEMO\_QUERY already exists, choose a different name.
5. Choose *Create*.
6. Enter a descriptive title for the functional area in the field *Description*.
7. Enter database table SFLIGHT in field *Table/View/Structure* and select checkbox *Direct read*. This tells the system that the users of the functional area are allowed to access the database table SFLIGHT.

After completing the entries, your screen should look like this:  [\[Ext.\]](#)

8. Choose *Continue*.  
A list for maintaining functional areas appears.
9. Create the functional area *Airline* by selecting the create icon following the functional group entry in the list. In the next dialog box, enter **10** in the first column of the table and **Airline** as functional group name in the second column. The number '10' serves as identifier for the functional group. Also give functional area **Flight booking** identifier **20**.
10. Leave the dialog box for maintaining functional groups with **ENTER**.
11. Double-click on functional group **Airline**. This selects the functional group.

## Functional Areas

12. Click on the expand icon following table **SFLIGHT**. You get a list of all table fields.
13. Assign the following fields to functional group **Airline** by clicking on the icon following field names SFLIGHT-CARRID, SFLIGHT-CONNID and SFLIGHT-FLDATE.
14. Double-click on functional group **Flight booking**. This selects the functional group.
15. Assign the following fields to this functional group: SFLIGHT-PRICE, SFLIGHT-SEATSMAX, SFLIGHT-SEATSOCC, and SFLIGHT-PAYMENTSUM.
16. If you executed these steps successfully, the window looks as follows:  [\[Ext.\]](#)
17. Choose *Save*.
18. Choose *Generate* to generate the functional area.
19. Choose *Exit* to leave the maintenance screen and go to the initial screen of the maintenance transaction for functional areas (see 2.).
20. Make sure that the field *Functional area* contains the name of the functional area you just created. Choose *Assign to user groups*.
21. Select the user groups that should have access to the functional area. Specify your own user group as well. After completing this task, you get the following window  [\[Ext.\]](#) :
22. Choose *Save* to save the assignment of functional area to the user group and leave the maintenance transaction.

## Check your work

In the exercise above, you created a functional area and assigned it to a user group. Call the maintenance transaction for functional areas and check if the functional area you created appears in the table of all functional areas.

You have now made all the required settings for using ABAP Query. The next exercise will show you what a query is and how to use it.

---

## Query

### Query

In the last two exercises you became acquainted with functional areas and user groups. They form the environment for the queries.

In the exercises below you will create different lists using ABAP Query. You can save the list layout for each list. The element thus created is called query.

Provided the user has the appropriate authorizations, he can execute, modify, copy, and delete queries. The modifications to the queries always affect all users of a user group. That is, if one user deletes a query, it is lost for all other users as well. All changes or new creations are at once visible to all users.

The exercises below show you how to create queries that present basic lists, statistics or ranked lists.

## Basic Settings for a Query

Before you can actually begin creating your lists, you must make some basic settings. This includes selecting the functional area, relevant fields and selection criteria for data output. The exercise below shows how to do this.



This exercise defines the requirements for accessing the relevant data of table SFLIGHT. The following fields are evaluated in the exercises:


- Code of the airline
- Code of the flight connection
- Flight date
- Number of seats
- Seats occupied

Since you do not want to output the entire list of all flights each time, select by the following fields:

- Code of the airline
- Code of the flight connection
- Flight date

1. Call the *ABAP Workbench*.
2. Go to the maintenance transaction for queries with *Utilities* → *ABAP Query* → *Query*

Make sure to view the window *Query of User Group TE*. If you are not in the user group you created, choose *Edit* → *Other user group* to switch to your user group before continuing with the exercise. Then select your user group in the dialog box.

3. Since you only just created the user group, the table of existing queries does not yet contain an entry.
4. Enter the name **DEMO\_1** in the *Query* field.
5. Choose *Create*.
6. A dialog box  [\[Ext.\]](#) appears in which you must select the functional area.


Since only one functional area is assigned to your user group, you can select only this area by either

double-clicking on the name

or


clicking on the name and choosing *Choose*.


7. In the subsequent window, enter a descriptive title. To enter a more detailed

documentation of the query, use the *Notes* field. Choose *Next screen* by pressing  or with *Goto* → *Next screen*.

8. In the window *Select Functional Group*, select the functional groups you need for your task. If you are not sure which functional groups you need, repeat the exercise

**Basic Settings for a Query**

[Functional Groups \[Page 18\]](#). After your selection, the following window appears:  [\[Ext.\]](#)

9. Choose *Next screen*. Select the fields you need for your task. After your selection, the following window appears:  [\[Ext.\]](#)
10. Choose *Next screen* to go to the selection criteria. Select the fields you want to use as selection criteria according to your task. Maintain the selection texts for these fields. Enter short, but descriptive texts.

After this step, your window looks as follows:  [\[Ext.\]](#)

11. Save this basic setting with *Save*.
12. Leave the maintenance transaction for queries.

In this exercise you made the basic settings for the query. Based on these settings, you can now create the lists described in the exercises to follow.



You cannot yet execute this query, since no lists have been defined so far. The subsequent exercises tell you how to define lists.

**Check your work**

In the above exercise, you made the basic settings for a query. Call the maintenance transaction for queries. The query you created is now part of the list.


## Creating a Basic List

In the basic list, you can present data in many ways. You can easily modify the list layout, calculate totals on numeric fields, or sort by any fields. However, the number of possibilities goes beyond the limits of this tutorial. Therefore, you will output a simple list only.



Create a basic list. Output all flights of airline LH from database table SFLIGHT. As additional information, include the maximum number of seats and the number of occupied seats per flight.

1. Go to the maintenance transaction for queries.
2. Select the template query DEMO\_1 you created.
3. Choose *Change*.
4. Choose *Basic list*.
5. In the *Basic List Line Structure* window, select the fields you want to output in the basic list. In the *Line* column, enter the line on which you want the field to be displayed. In the *Sequence* column, specify the order in which you want the fields to appear by entering numbers.

After specifying the line structure, the window should look like this:  [\[Ext.\]](#)

6. Save the query.
7. Choose *Execute*.
8. A selection screen appears that allows you to select the data you want to output from the database table. You defined the selection criteria in the previous exercise.

For example, if you want to output only flights of the airline 'LH', enter **LH** in the field *Airline*. Choose *Execute* to end the selection and continue with the program.

9. The system displays a list of all flights that match the selection criteria you specified.
10. End the program and return to the *ABAP Workbench*.

### Check your work

In the exercise above you created a basic list. Call the maintenance transaction for queries. The query you created is now part of the list. Select it and execute it again. If you choose different functions on the selection screen, you can change the contents of the basic list.

## Layout of a Basic List

### Layout of a Basic List

In the last exercise, you created a simple list. ABAP Query enables you to present this list more clearly and to use additional functions.




This exercise tells you how to create control level totals. First sort the data by airlines, flight connections, and seats occupied. Then calculate the number of occupied seats for each flight connection of an airline. Separate the data of the different airlines by page breaks. Output the name of the corresponding airline after each page break.

First you will learn how to create lists that are sorted by certain fields:

1. Go to the maintenance transaction for queries and select query DEMO\_1.
2. Choose *Change*.
3. Choose *Basic list*. The [Window: Line Structure Basic List \[Ext.\]](#) should now appear.
4. So far you have not used the *Sort* column. It is used for the sort sequence of the fields. In the exercise, sort the data by airlines first, then by flight connections, and finally by the number of occupied seats. Specify the corresponding numbers in the *Sort* column.
5. Save your entries and choose *Execute*. The system displays a sorted list.

In a second step, calculate the number of seats occupied per flight connection. Proceed as follows:

1. Go to the [Window: Line Structure Basic List \[Ext.\]](#)
2. Choose *Next screen*. The window for control level processing appears  [\[Ext.\]](#). Only the fields used for sorting are displayed here. Following the field names, you find several checkboxes, some of whose functions you will now learn.
3. In this exercise, create a list with the number of seats occupied for each flight connection. To do this, select checkbox *Total* following the *Code of flight connection* field.
4. To tell the program for which numeric field to calculate the total for each flight connection, choose *Previous screen* and select checkbox *Total* following field *Occupied seats* in the *Basic List Line Structure* window .
5. Choose *Save* and execute the query. The system displays a list where the number of the occupied seats appears after each flight connection.
6. To make the list easier to read, you can select field *BlnkLn* for column *Code of flight connection* on the *Control Level* screen. When you execute the query, you will see that a blank line appears after each total..

To complete the exercise, all you must do now is separate the individual flight connections by page breaks and display the names of the airlines after each page break. Proceed as follows:

1. Go to the window *Control Levels*. In the line *Code of the airline* select *Text*.



### Layout of a Basic List

2. Execute the query. In the list, the name of the airline appears in front of the data concerning this airline.
3. In the maintenance transaction for queries, go to the window *Control Levels* and select *NewPg.* in the line *Code of the airline*.
4. Save the query and execute it. The list now contains a page break after outputting the data of an airline.

You can now try other options for displaying a basic list. If you encounter problems, press **F1** to see a description for each option.

---

Other presentations of a basic list

## Other presentations of a basic list

After executing a query, you can continue to modify the created list. You may, for example, present the list as table, create ABC analyses, or store the list as Word or Excel documents and process them with the corresponding programs. Or, you can present the data in graphical form.



The exercise below shows you how to convert a list to a table and how to sort the table and calculate totals.


1. Execute the query you created in the last exercise. The system displays a list of the airlines and their flight connections.
2. Choose *Display as table*.
3. Mark the column *Flight date* and choose *Sort ascending*. The system sorts the table by date in ascending order.
4. Mark the column *Flight date* again. Choose *Sum*. In addition to the previous presentation, the system includes the totals for the number of occupied seats and the maximum number of seats for each flight date.
5. Mark the lines that contain the totals and mark the column *Occupied*. Choose *List → Graphics*. The system displays a set of presentation graphics on the totals of the occupied seats.
6. If you like, you can try other possibilities of processing the created list. If you encounter problems, look into the extended help for information on the different topics.

## Creating Statistics

The previous exercise taught you how to easily create a basic list. It is just as easy to create statistics. Statistics usually present data in condensed form, that is, numeric values usually are totals.



In this exercise, create a set of statistics on the number of occupied seats for the flight connections of an airline. Determine the average number of occupied seats and the percentage the occupation of each flight connection presents of the overall occupation. Output the flight connections in descending order of their bookings.

1. Go to the maintenance transaction for queries.
2. Select a new name for the query and choose *Create*.
3. Define the basic settings for the query as in [Basic Settings of a Query \[Page 21\]](#).
4. Choose *Statistics*.
5. Maintain a title for the statistics in the *Title* field.
6. In the *No* field, specify the order in which to output the fields within the statistics. Since you want to create statistics for all flight connections, you should not output the date. In that case, the system would create a set of statistics for each individual flight.
7. In the *Srt* field, specify the criteria according to which you want to sort the set of statistics. If you want to sort by several fields, specify the sorting order by numbering the fields. To solve the exercise, sort by the field *Occupied seats*.
8. To output the flight connections in ascending order of occupation, select checkbox *De*.
9. To output the average value of the occupation of all flights of a flight connection, select checkbox *Av*.
10. To output the percentage of the overall occupation, select checkbox *%*.
11. If your entries are all correct, your window looks like this:  [\[Ext.\]](#)
12. Save the query.
13. Execute the query and enter **LH** in the *Airline* field of the selection screen.
14. The system displays a set of statistics that matches the problem.
15. End the program and return to the *ABAP Workbench*.

### Check your work

In this exercise, you created a set of statistics. Call the maintenance transaction for queries. The query you just created now appears in the list of existing queries. Select it and execute it again. If you like, you can try other options to modify the statistics further.

---

**Creating Statistics**

Instead of creating a new query to produce statistics, you can also use one query for both basic list and statistics.

## Creating a Report Using ABAP Statements

[Overview \[Page 30\]](#)

[The ABAP Programming Language \[Page 31\]](#)

[Creating an ABAP Program \[Page 32\]](#)

[The ABAP Editor \[Page 34\]](#)

[Comment Lines \[Page 35\]](#)

[ABAP Statements for Screen Display \[Page 36\]](#)

[Data Declarations \[Page 38\]](#)

[Handling Variables \[Page 39\]](#)

[Text Layout \[Page 40\]](#)

[List Headers \[Page 42\]](#)

[Dynamic List Headers \[Page 43\]](#)

[Database Access \[Page 44\]](#)

[The SELECT Statement \[Page 45\]](#)

[Selection Criteria \[Page 47\]](#)

[PARAMETERS \[Page 48\]](#)

[SELECT-OPTIONS \[Page 50\]](#)

[Variants \[Page 52\]](#)

[Nested SELECT Statement \[Page 53\]](#)

[Sample Program: Nested SELECT Statement \[Page 56\]](#)

---

**Overview**

## Overview

The last unit introduced the ABAP Query. ABAP Query is a tool for the end user to create lists, ranked lists, and statistics. You can also create lists by programming them using ABAP statements.

This unit gives you a short introduction to ABAP Reporting. After working through this unit, you

- can display and format lists on the screen
- understand the basics of the SAP type concept
- can read database tables
- know how to restrict the amount of data to be read by using selections.

## The ABAP Programming Language

Originally, SAP developed the ABAP (Advanced Business Application Programming) programming language for internal use, to offer the application developers better working conditions. ABAP is continually improved and adapted to the growing requirements of the business applications. Today, ABAP is the only tool for developing applications at SAP.

SAP customers use ABAP for their own enhancements or for modifying R/3 standard solutions according to their special requirements. The ABAP development environment contains all tools you need to create and maintain ABAP programs.

There are two types of ABAP programs: dialog programs and reports.

You use dialog programs to enter data into database tables or to output individual table fields. For an introduction to dialog programming, see [ABAP Workbench Tutorial \[Ext.\]](#).

You use reports to output large amounts of data in lists. These lists can either be displayed on the screen or be output on a printer.

## Creating an ABAP Program

## Creating an ABAP Program

This section tells you how to specify the attributes of a report program.



Creating a program can be divided roughly into the following steps:

1. Assigning a name
2. Specifying the program attributes
3. Writing the program code
4. Testing the program



In this exercise you create an ABAP program. You will also maintain the necessary attributes.

Assigning a name:

1. Choose *Tools* → *ABAP Workbench* on the screen on SAP level.
2. Choose *Repository Browser*. If you have not yet set markers in the Repository Browser, the system displays the initial screen of the *Repository Browser*. If you have set markers, use *Back* to return to the initial screen:  [\[Ext.\]](#)
3. Select the radio button for *Local objects* and choose *Display*.  
A list of all the objects you stored as local private objects is displayed.  [\[Ext.\]](#)
4. Double-click on the field *Development class object types*.  
The system displays a modal dialog box. Mark the radio button *Program objects* and choose *Continue*.
5. Enter a program name in the *Program name* field in the [Program objects \[Ext.\]](#) window. Note that the name must consist of at least one character and may be up to 30 characters long. Certain characters may not be used. These are: point, comma, blank, apostrophe, quotation mark, equality sign, asterisk, umlaut, 'ß', percentage sign and underlining. You must also adhere to the naming conventions for report names:  
  
Yaxxxxxx or Zaxxxxxx.  
  
Replace the **a** with the letter of the application area for which you write the report.  
Replace **x** with any valid character.
6. Make sure that the *Program* radio button is set and choose *Create*.
7. In the modal dialog box *Create Program*, deselect *With TOP INCL.* and choose *Continue*.  
  
If the program name does not exist in the system, the screen *ABAP: Program Attributes* appears. If the name exists, enter another name and choose *Create* again.
8. Enter a descriptive title for the program in the *Title* field.
9. Reports are always of *Type 1*. You must therefore enter *Type 1* in the field.
10. The status of a program is not a required entry field. Nevertheless, enter **T** in the *Status* field. This identifies the program as a test program.



## Creating an ABAP Program

11. Enter the application area in the *Application* field. If you do not know to which area to assign your program, place the cursor on the field and use the possible entries key **F4**. You get a list of all application areas in the system. You must choose one of them.
12. Choose *Enter*. The following [screen \[Ext.\]](#) appears.
13. Choose *Save*. The system stores the program as your local private object.  
You have maintained the required attributes and can now start writing the ABAP program. You must carry out the steps described above each time you create a new program.

### Check your work

You should know now how to create a program and maintain the most important attributes. You will repeat these steps in the lessons to come.

Return to the Object Browser. The list of local private objects now contains the node *Programs*. If you click on the + (plus sign) in front of it, the system displays the list of your programs. The program you just created is part of this list.


---

The ABAP Editor

## The ABAP Editor

You write your programs in the ABAP Editor. This tool offers any functionality needed for programming. For a detailed description of the functions see [ABAP Workbench: Tools \[Ext.\]](#).

### Calling the ABAP Editor:

- If you have just finished the last exercise and you are on the screen *ABAP: Program Attributes*, call the ABAP Editor by choosing *Source code*.
- If you are looking at the local private objects using the Repository Browser, you can display your programs. Select the desired program and choose *Change*. You go directly to the ABAP Editor  [\[Ext.\]](#).

The ABAP Editor has three different display modes. This tutorial uses the *PC mode with line numbering*. To change the mode, choose *Settings → Editor mode....* In the subsequent dialog box, you can choose the desired mode: Select the radio button *PC mode with line numbering* and choose *Continue*.

In addition, the ABAP Editor distinguishes between two statuses: *Display only* and *Editing*. Use *Display → Change* to change the status. To change your program text, you must be in the editing status.

### Important Commands in the ABAP Editor:

- Insert line Place the cursor at the end of a line and press ENTER.
- Delete line Position the cursor in the line you want to delete and choose *Edit → Delete*.

For further information about the ABAP Editor, see the documentation [ABAP Workbench: Tools \[Ext.\]](#).

If you create a new program, the system initializes the following line:

```
REPORT Zaxxxxxx.
```

This line marks the beginning of the program. You must never delete this line.

Zaxxxxxx is the program name you entered. All data declarations and program statements you use in your program must come after this line.

## Comment lines

Comments are explanatory texts inserted between program statements. They are intended to make the program easier to read. Therefore, even for small programs, use as many comments as possible.

To identify a whole program line as comment line, start the line with an asterisk (\*). To identify only part of the line as comment (to the right of the program text), precede the comment with quotation marks (").

When executing the program, the system ignores comments.



The entire line is a comment line:

```
* This is a comment line
```

Part of a line is a comment:

```
REPORT REAATME1. "Here starts the comment line
```

## ABAP Statements for Screen Display

## ABAP Statements for Screen Display

An ABAP program consists of individual records (statements). Each record has to be terminated by a period. The first word of a record usually is a keyword. When assigning values to variables, a variable may be the first word of a record. You must use at least one blank between two individual words.

However, one record may extend over several lines (ended only by the period).

### WRITE

A very important keyword in the reporting area is the WRITE statement. You use WRITE to output literals or fields on the screen or printer. Subsequent WRITE statements appear on the same output line, unless a line feed occurs. If a line is full, the system continues the output in the next line.

The simplest form of the WRITE statement outputs a literal. Remember always to include literals in quotation marks. A simple WRITE statement may be:

```
WRITE 'My first program'.
```



Copy this line now into your new program and find out the result of the statement. Proceed as follows:

1. Load your program into the ABAP Editor as described in the last exercise.
2. Include the above WRITE statement in your program.
3. Choose *Check*. The syntax is checked. If you made an input error in your program text, the system will highlight the error. If possible, the system also offers a solution. When you create programs, check the syntax regularly.
4. Choose *Activate*.
5. If your program does not contain a syntax error, choose *Program* → *Execute* to start it.

The output contains a standard page header, consisting of a list heading and an underline, and a list of one line. The list header corresponds to the text you defined as title in the unit [Creating an ABAP Program \[Page 32\]](#). The list corresponds to the character string in the WRITE statement.

6. Choose *Exit* to return to the ABAP Editor.

### NEW-PAGE

Use the NEW-PAGE statement to create a page break in your list. Page breaks improve the clarity of the page layout of your list.



```
NEW PAGE.
```

This statement creates a page break if a preceding statement wrote output to the screen or printer.

## SKIP

Use the SKIP statement to create a line feed. To create several line feeds, specify the desired number behind SKIP.



```
SKIP 5.
```

This line creates five line feeds.

## ULINE

Use the ULINE statement to create a line feed first and then draw a horizontal line. This statement also helps to make lists easier to read.



Use the statements NEW-PAGE and ULINE to modify the layout of your list so that a line is output before and after outputting 'My first program'. Insert a page break in your program at the end of the list.

1. Load the program created in the last exercise into the ABAP Editor.
2. Enter the ULINE statement in the lines before and after the WRITE statement. This creates one line before 'My first program' and one after.
3. At the end of the program, enter NEW-PAGE.
4. Check the syntax. Correct all errors and save the program.
5. Execute the program.

A line is now displayed before and after 'My first program'. The effect of the page break is not yet visible, since no further output occurs after the page break.

## Declaring data

### Declaring data

Since literals are usually too inflexible to be used in programs, you can define other data objects (variables, constants, tables, and so on). For each data object, the assigned data type describes the technical characteristics. Data types may be predefined or user-defined.

This section deals exclusively with predefined data types. Chapter [Structures and Internal Tables \[Page 72\]](#) tells you how to define your own data types.

The table below contains all data types predefined by SAP together with their characteristics:

Data type	Description	Initial value	Standard length
P	packed number	0	8
I	integer	0	platform-dependent
F	floating point number	0.000...	platform-dependent
N	numeric text	00....	1
C	text	blank	1
D	date YYYYMMDD	00000000	8
T	time HHMMSS	000000	6
X	hexadecimal number	X'00'	1



Suppose, you want to write the output of the last exercise not using a literal but a variable. Then you must define the variable first. The keyword for declaring variables is DATA, followed by the name of the variable. Then comes the keyword TYPE, followed by the data type. In our case, the data type is C, corresponding to a text:

```
DATA TEXT TYPE C.
```

This statement assigns to variable TEXT the standard length of type C which is one character. If you want to define a length different from the standard length for the variable, specify this length in brackets directly after the variable name:

```
DATA TEXT(30) TYPE C.
```

## Handling Variables

The last unit taught you how to declare a variable. Here you will learn how to handle a variable:

To assign texts to a variable, use one of the following:

```
MOVE 'Variable TEXT is filled' TO TEXT.
```

```
TEXT = 'Variable TEXT is filled'.
```



With this type of assignment, make sure that the assigned literal does not contain more characters than the length you specified when defining the variable. If you assign a longer character string to a variable, the system truncates the last part of the string and assigns only the first part to the variable. A syntax check does not find this "error".

To output a variable on the screen, use the WRITE statement. Specify the name of the variable after the WRITE statement, without using quotation marks:

```
WRITE TEXT.
```



In the exercise below, you create a new program. In this program, you define a text variable of 30 characters length and assign text to it. Then display the variable on the screen.

1. Create a new program. Proceed as in [Creating an ABAP Program \[Page 32\]](#).
2. Declare the variable TEXT with a length of 30.
3. Assign a text to the variable. Make sure that this text does not exceed 30 characters length.
4. Display the variable on the screen.

After completing the exercise, your program looks as follows:

```
REPORT RSAATME5.
```

```
DATA TEXT(30) TYPE C.
```

```
MOVE 'Variable TEXT is filled' TO TEXT.
```

```
WRITE TEXT.
```

## Text Layout

### Text Layout


You can modify the layout of text you output. An elegant way to achieve this is to use the statement structure.

The exercise below shows how to use statement structures. They help you create your program. The result of using a statement structure always consists of one or more lines of code.



At the end of the exercise below, your program will contain another output line, in which the text of variable TEXT is displayed in a new line and in a different color. Proceed as follows:

1. Load the program you created in the last exercise into the ABAP Editor.
2. Position the cursor on the line into which you want to insert the additional WRITE statement.
3. Choose *Structure*.
4. A dialog box appears, showing a selection of statements for which statement structures exist. Select the WRITE statement.
5. Choose *Continue*. You can enter your selections on the subsequent screen. First specify the name of the field you want to display and select *Output from fld*.
6. Select the *Color* field and place the cursor on the adjoining input field. Press the possible entries key **F4**. Select color *COL\_KEY* in the window that appears.
7. Select the *to new line* field. This tells the system to start a new line before outputting the variable.

After executing these steps, you see the following screen:  [Ext.]

8. Choose *Display* to check the layout.
9. Choose *Copy* to insert the statement into your program.

The following line appears in your program text:

```
WRITE / text COLOR COL_KEY.
```

The slash (/) indicates that the variable will be output on a new line. The option COLOR COL\_KEY determines the color of the output text.

10. Activate your program and execute it. The display now contains an additional line, which appears in a different color.

Using the statement structure you can generate any formatting of the WRITE statement.

If you like, you can now insert further output lines into your program to familiarize yourself with the options the WRITE statement offers. Use the statement structure again, look at the resulting lines of program text, and execute the program again and again to check the results on the screen.





If you want to output literals using statement structures, select *Output from fld* on the selection screen of the WRITE statement structure. Enter the literal in quotation marks in the adjoining input field and proceed as before.

## List Headers

### List Headers

The last exercise taught you how to modify the output format of a WRITE statement. Another step to improve the list layout is to use list headers.

To create or change a list header, proceed as follows:

1. Execute the program you created in the last exercise. If you start the program from within the ABAP Editor, activate it first.
2. Choose *System* → *List* → *List header*. The system displays five lines for input.
3. The first line is the header, displayed on each new page. By default the system uses the text you entered when you [created the attributes \[Page 32\]](#) of your program in the *Description* field. Enter a new header.
4. The lines below make up the column headers. You can enter further information you want to appear as list header in these lines. If, for example, you create a list that consists of several columns, you can name these columns in the list header. The position of the texts in this view corresponds to the output position in the list, which makes positioning easy and exact. Enter text in these lines.
5. Save the list headers. You are now in your list again. You cannot yet see the changes made to the list headers.
6. Choose *Exit* to end the list display and execute the program again. The new list now contains the headers you maintained.



- Before starting a program from within the ABAP Editor, save it. If you forget to save, the system only temporarily stores any changes to the list headers, and they are usually lost.
- Save the list header after each change.
- Note that you only see the changes when you create a new list.

## Dynamic List Headings

The last exercise taught you how to create static list headers. However, it may be useful to create list headers dynamically at program runtime. The system offers 10 text variables (**SY-TVAR0** to **SY-TVAR9**) for this purpose. In the list header itself, you can call the contents of these system variables using &0 to &9. The default output length is 2. To increase the output length, insert the appropriate number of periods (.):

&0..                      Outputs variable SY-TVAR0 with a length of 4.



The next exercise is to output the contents of variable TEXT in the list header. Proceed as follows:

1. Load your program into the ABAP Editor.
2. Assign the variable TEXT to the system variable SY-TVAR0 using this statement:

`MOVE TEXT TO SY-TVAR0.`

**Make sure that this statement appears before the first WRITE statement but below the value assignment to variable TEXT.**

3. Activate your program and execute it.
4. Choose *System* → *List* → *List header*.
5. Enter the following statement into the line of the list header:

&0.....

&0 tells the system to output variable SY-TVAR0. The periods define the output length of the variable text. The maximum length is 20 characters.

6. Save your changes.
7. Leave the list and execute the program again.
8. The system now outputs the contents of variable TEXT as list header.

---

**Database Access**

## Database Access

The last exercise taught you how to display data on the screen. You will now learn how to retrieve data from database tables.

To access databases in ABAP, you use OpenSQL statements. These statements are the same for all databases supported by the R/3 System. Therefore, you need not know which databases you may need to access when you create a program.

## The SELECT Statement

The keyword for retrieving data from databases is SELECT. This statement is used to read data from a database table. The data is stored in a special work area having the same name as the database table.



In this exercise you create a new program. This program reads data from database table SFLIGHT and displays it on the screen. Proceed as follows:

1. Create a new program as you learned in [Creating an ABAP Program \[Page 32\]](#) and go to the ABAP Editor.
2. To be able to read data from database table SFLIGHT, you must tell the system in the declaration part of your program that you want to use table SFLIGHT. Include the following statement in the declaration part of your program:

```
TABLES SFLIGHT.
```

This statement automatically creates a data structure that corresponds to the structure of the database table. This data structure serves as work area in your program. You can access the work area, for example, using a WRITE statement.

3. After declaring the database table, you can read the table using the SELECT statement. Use the following statement:

```
SELECT * FROM SFLIGHT.
```

```
ENDSELECT.
```

This statement corresponds to a loop that is scanned until the entire database table is read (full table scan). The work area SFLIGHT is filled with data from the database table SFLIGHT during each scan.

4. You can now output the data in a list. As in the [Text Layout \[Page 40\]](#) exercise, you should use the statement structure of the WRITE statement. Proceed as follows:
5. Place the cursor on an empty line between the SELECT and ENDSELECT statements.
6. Choose *Structure*.
7. In the *Insert Statement* dialog box, select the radio button before *WRITE* and select *Continue*.
8. In the window *Assemble a WRITE Statement*, select the radio button *Output to struct*. Enter the database table name SFLIGHT.
9. Choose *Select components*.
10. Enter the airline, flight connection, flight date, maximum number of seats, and seats occupied. Select the corresponding checkboxes and choose *Copy*.
11. Select the checkbox *to new line*.
12. Choose *Copy*.

---

**The SELECT Statement**

The WRITE statement is inserted in the program. Study the statement carefully. You will find that to output the fields of the work area SFLIGHT, the system prefixes their names with the name of the work area and a hyphen.

13. Include the ULINE and NEW-PAGE statements to improve the list layout.
14. Check the syntax of the program and save it. Execute the program. The resulting list contains all data from the database table SFLIGHT.
15. Maintain the list headers as described in [List Headers \[Page 42\]](#).

## Selection Criteria

The last unit taught you how to read an entire database table. However, you may not need all data. Therefore, you should specify selection criteria before reading the database table. The selection criteria can be used when reading from the database. In this case your list will contain only relevant data and the performance will improve, since the system must not pass all data from the database table to your program.

The easiest way to select data is to specify the selection criteria in the program. In the SELECT statement, you use the keyword WHERE to specify the selection criterion.



Change the program created in the last exercise to output only data of the airline 'LH'. Use table field CARRID as selection criterion. In a second step, use table field CONNID (specifying flight connections) as another selection criterion. Output only flight connection '0400'. At the end of the exercise, your list should look like this:

[\[Ext.\]](#)

1. Load the program created in the last exercise into the ABAP Editor.
2. Modify the SELECT statement in your program:  

```
SELECT * FROM SFLIGHT WHERE CARRID = 'LH'.
```

Make sure that the literal is enclosed in quotation marks and watch out for uppercase/lowercase.
3. Check the syntax and activate your program.
4. Execute the program. The resulting list displays data of the airline 'LH' only.
5. Now add the selection criterion CONNID = '0400' to the SELECT statement and combine the two selection criteria with the logical operation AND. The statement now looks like this:  

```
SELECT SFLIGHT WHERE CARRID = 'LH' AND CONNID = '0400'.
```
6. Check the syntax and activate your program.
7. Execute the program. The resulting list displays data on the flight connection 0400 of the airline 'LH' only.

Check your work

In this unit you learned how to select data. You also learned how to link several selection criteria using logical operations. Modify the selection criteria of your program further and study the results these modifications have on the list.

## PARAMETERS

## PARAMETERS

The last exercise taught you how to select data before reading the database table. However, this selection is static, so you must know which selections to make when writing the program.

You can make your program more flexible by creating a selection screen using the keyword **PARAMETERS**. This selection screen appears before the list is displayed, and allows you to enter the selection criteria during program runtime.



This exercise shows you how to use field **CARRID** of database table **SFLIGHT** as a dynamic selection.

1. Load the program created in the last exercise into the ABAP Editor.
2. Insert the following line into the declaration part of the program:

```
PARAMETERS ID LIKE SFLIGHT-CARRID.
```

This statement declares the variable **ID**. By specifying the keyword **LIKE** and the field **CARRID** of database table **SFLIGHT**, the variable automatically assumes the attributes of the corresponding database field.

3. Modify the **SELECT** statement:

```
SELECT * FROM SFLIGHT WHERE CARRID = ID AND CONNID = '0400'.
```

4. Check the syntax and activate your program. Execute the program.

The system displays a selection screen on which you can enter values for the parameter **ID**. If for example, you want to see only the flight connections of airline 'LH', enter **LH** and then choose *Continue*. The resulting list contains all flight connections '0400' of the airline whose ID you entered on the selection screen.

### Modify the selection text

By default, the system displays the name of the parameter as the selection text on the selection screen. Since this name usually is not very descriptive, you can store another text to be displayed. Proceed as follows:

1. Load your program into the ABAP Editor.
2. Choose *Goto* → *Text elements* → *Selection text*. The *ABAP Text Elements* screen appears.
3. You get a list of all selection parameters. In the adjacent column you can enter selection texts.

If the current selection parameter does not appear in the list, return to the ABAP Editor and save your program. Then go back to the selection texts dialog.

4. After entering the desired selection texts, choose *Save*. Use *Exit* to return to the ABAP Editor.



---

**PARAMETERS**

5. Activate the program and execute it. The selection text for the parameter, and not the parameter name, now appears on the selection screen.

Check your work.

This exercise taught you how to create a selection screen using the `PARAMETERS` statement. You also learned how to store texts that appear on the selection screen for the selection criteria.

## SELECT-OPTIONS

## SELECT-OPTIONS

If single values do not suffice as selection criteria, ABAP allows you to create complex selection criteria using the statement `SELECT-OPTIONS`. You can enter single values as well as intervals.

The `SELECT-OPTIONS` statement creates an internal table that is filled by the selections on the selection screen.



You have already created a program that outputs a list of flight connections. In the last exercise, you included a selection condition for the airline using the `PARAMETERS` statement. In the present exercise, you will include a selection condition for the flight connection that allows selecting intervals.

1. Load the program created in the last exercise into the ABAP Editor.
2. To allow intervals to be entered on the selection screen for the flight connection, include the following statement into the declaration part of your program:

```
SELECT-OPTIONS CID FOR SFLIGHT-CONNID.
```

This statement declares selection criterion `CID` for field `CONNID` of database table `SFLIGHT`.

3. Activate your program and execute it. The selection screen of the last exercise was enhanced by another element:


Airline    
CID  to  

1. You can also maintain a selection text for this selection criterion in the same way as described in the last exercise.
2. Change the `SELECT` statement in your program to include the new selection criterion. Note that database field `CONNID` is now compared to an internal table, resulting in a change in notation. You must change `"="` to `"IN"`.

```
SELECT * FROM SFLIGHT WHERE CARRID = ID AND CONNID IN CID.
```

3. Check the syntax and activate your program. Execute the program.

On the selection screen, you can now select the airline as well as the flight connection.

For the flight connection, you can specify single values or intervals. With  you can also enter several intervals or single values.

After this selection, the system outputs all data that matches one of the specified conditions.

Check your work.

The last exercise taught you how to specify several single values and intervals as selection criteria in one step using the `SELECT-OPTIONS` statement.

Familiarize yourself with the functionality of the selection criteria by testing the selection screen of your program.



## Variants

### Variants

Selection screens may contain many input fields. It is therefore reasonable to store certain selections that are used again and again with only slight changes in variants. You can then use such a variant at a later time instead of entering all the same selections again.

The first step will teach you how to create a variant for your program. In the second step, you will learn how to call this variant.



In this exercise you create a variant that selects all flights of airline 'LH' with a flight connection number between '0400' and '1000'.

1. Execute the program you created in the last exercise.
2. Enter the appropriate selections on the selection screen:

Airline	LH		
Flight connection	0400	to	1000

1. Choose *Goto* → *Variants* → *Save as variant...* or press
2. On the selection screen *ABAP: Save as Variant*, enter a name for this selection variant and enter a description of the selection criteria in field *Meaning*.
3. Save the variant. The selection screen appears again.
4. Continue to execute the program.



Now use the variant you just created to output a list of the flight connections '0400' to '2500' of the airline 'LH'.

1. Execute the program again.
2. On the selection screen, choose *Goto* → *Variants* → *Open...* or press
3. The selections you entered in the last exercise appear. All you must do now is change the upper limit of the interval for the flight connection from '1000' to '2500'.
4. Choose *Execute* to get the list of flight connections.

Check your work.

The last exercise taught you how to store frequently used selections as variants and how to call such a variant. Use variants for all selection screens that contain a large amount of input fields of which only few values change each time.


## Nested SELECT Statements

You often retrieve data from a database table that you need as selection criteria for accessing another database table. In the flight booking model, for example, you may first read all flights of an airline from database table SFLIGHT and then retrieve the booking data of these flights from database table SBOOK.



In this exercise, use all elements you have learned so far. You will also learn how to create a nested SELECT statement.

At the end of this exercise, create a program that reads data for one or more flight connections of an airline from table SFLIGHT. Then use this data to select which data to read from booking table SBOOK. In addition, output the flight connection data in the list heading.

Then output all bookings for each flight of a flight connection. Make the list look like this:  [\[Ext.\]](#)

Separate the data of the individual flights by page breaks.

If you have doubts at any point of the following exercise, refer to the previous exercises for help. Only use the [sample solution \[Page 56\]](#) if you cannot solve the exercise on your own.

Procedure:

1. Create a new program. Proceed as in the exercise [Creating an ABAP Program \[Page 32\]](#).
2. [Declare \[Page 38\]](#) tables SFLIGHT and SBOOK.
3. Create a [selection screen \[Page 47\]](#) for fields CARRID and CONNID of table SFLIGHT. Only define [single values \[Page 48\]](#) for CARRID, but you can also define intervals for [CONNID \[Page 50\]](#). Do not forget to create the [selection text \[Page 48\]](#).
4. Create the selection loop for reading table SFLIGHT. Use the statement structure for SELECT as a help. Proceed as follows:
  - a) Position the cursor in the program line where the SELECT statement should be created.
  - b) Choose *Structure*.
  - c) Select the radio button before *SELECT \* FROM* and enter the name of database table SFLIGHT.
  - d) Choose *Continue*.
  - e) A list of all table columns of SFLIGHT is displayed. Select the columns you want to use as selection criteria.
  - f) Choose *Copy*. Your program text now includes a new SELECT statement with a pre-generated WHERE condition and ENDSELECT statement.
  - g) Complete the WHERE condition similarly to the exercises [PARAMETERS \[Page 48\]](#) and [SELECT-OPTIONS \[Page 50\]](#) so that a selection is made with the PARAMETERS *ID*

**Nested SELECT Statements**

and the SELECT-OPTIONS *C/D*. Make sure to delete the pre-generated underscores ( \_ ).

5. Supply the system variables for the [list headers \[Page 43\]](#) with the data of fields CARRID, CONNID and FLDATE in table SFLIGHT.
6. Create a selection loop for table SBOOK using the statement structure. Use fields CARRID, CONNID, and FLDATE as selection criteria. The values used for comparison should be the current values obtained from table SFLIGHT.  
Make sure that the selection loop is within the selection loop for table SFLIGHT.
7. Use the statement structure of the WRITE statement to output the relevant booking data (see sample list).
8. Modify the text layout. Draw a horizontal line after outputting all bookings for one flight and start a new page for the next flight.
9. Check the syntax and activate your program.
10. Execute the program and create a [variant \[Page 52\]](#) on the selection screen.
11. Compare the list with the sample list and change the [list headers \[Page 43\]](#). Use the system variables to which you passed the data in step 5. Remember to adapt the output length of the variables. Save the list headers and start the program again. On the selection screen, use the variant you created before.
12. Improve the layout until it matches the sample list.



The overhead on the database for nested SELECT statements is very large. You therefore should never create productive reports with nested SELECT statements. Techniques for avoiding nested SELECT statements are described in [Minimizing the Search Overhead \[Ext.\]](#).

## Performance of the SELECT Statement

In the previous units, the SELECT statement always read all fields from the database table, even though not all contents were needed. To improve system performance in these cases, modify the SELECT statement.

SELECT on all fields:

```
SELECT * from <dbtable>
```

SELECT on fields f1, f2, ..., fn:

```
SELECT f1 f2 ... fn from <dbtable>
```

For more information see the [ABAP User Manual \[Ext.\]](#).

## Sample Program: nested SELECT statements

## Sample Program: nested SELECT statements

The sample program below is the solution for the problem stated in the last exercise. The first part contains the source code including many comments. The second part shows the list headings.

**Source code:**

```

*&-----
-----*

*& Report   RSAATME4
*

*&  nested SELECT
*

*&-----
-----*

*&  Output all booking data for certain flight connections
*

*&  Select the flight connections via selection screen
*

*&-----
-----*

REPORT   RSAATME4                .

*{ Declaration section
*Declare database tables used
TABLES: SFLIGHT, SBOOK.
*{ Create a selection screen
PARAMETERS:  ID LIKE SFLIGHT-CARRID.
SELECT-OPTIONS: CID FOR SFLIGHT-CONNID.
*} End of selection screen
*} End of declaration section

*{ Start reading database table SFLIGHT using the selection
* criteria ID and CID
SELECT          * FROM SFLIGHT
              WHERE CARRID      = ID

```



Sample Program: nested SELECT statements

```

        AND      CONNID      IN CID.

*{ Pass the variables for the list heading
    MOVE SFLIGHT-CARRID TO SY-TVAR0.      "Airline
    MOVE SFLIGHT-CONNID TO SY-TVAR1.      "Flight connection
    MOVE SFLIGHT-FLDATE TO SY-TVAR2.      "Flight date
*} End of passing data for the list heading

*{ Start reading database SBOOK using the selection criteria
* SFLIGHT-CARRID, SFLIGHT-CONNID, and SFLIGHT-FLDATE
    SELECT      * FROM SBOOK
        WHERE   CARRID      = SFLIGHT-CARRID
        AND     CONNID      = SFLIGHT-CONNID
        AND     FLDATE      = SFLIGHT-FLDATE.
*{ Output fields relevant for booking
    WRITE:/ SBOOK-BOOKID,
           SBOOK-CUSTOMID,
           SBOOK-LUGGWEIGHT,
           SBOOK-WUNIT,
           SBOOK-CLASS,
           SBOOK-ORDER_DATE.
*} End output of fields for booking

    ENDSELECT.

*{ Horizontal line and new page after outputting all booking
* data for one flight
    ULINE.
    NEW-PAGE.
*} End of formatting

*} End of SELECT loop for reading database table SBOOK

    ENDSELECT.
*} End of SELECT loop for reading database table SFLIGHT

```

## Sample Program: nested SELECT statements

## List heading:

Booking Data: Airline &0. for Flight connection &1..

Flight date: &2.....

Booking Number	Customer Number	Weight of	Booking Date
			luggage

## Logical Databases and Events

This unit introduces logical databases and how you can use them in your ABAP programs. It also describes events in ABAP, which you need to control the flow of your program.

[Overview: Logical Databases \[Page 60\]](#)

[Creating a Report That Uses a Logical Database \[Page 61\]](#)

[Retrieving Data Using a Logical Database \[Page 62\]](#)

[Example Program: Using a Logical Database to Read Table SBOOK \[Page 64\]](#)

[Events \[Page 67\]](#)

[GET <table> LATE \[Page 68\]](#)

[Example Program: Report With Logical Database \[Page 70\]](#)

## Overview: logical databases

ABAP/4 provides two basic means of retrieving data:

1. You use the SELECT statement in your program to read the data yourself, as described in the previous chapter.
2. You call an extra read program, called logical database, and let this program read the data you need and provide it to you in the correct order. You only write the statements that process the data.

The link within a program between the report processing and a logical database is the event keyword GET. The logical database triggers this event after reading the corresponding record. The report can access the data in the work area.


More advantages of the logical database:

- It offers an easy-to-use selection screen.
- You can modify the pre-generated selection screen to your needs.
- It offers check functions to check whether user input is complete, correct, and plausible.
- It offers reasonable data selections.
- It contains central authorization checks for database accesses.
- Enhancements such as improved performance immediately apply to all report programs that use the logical database.

## Creating a Report That Uses a Logical Database

In the [overview \[Page 60\]](#) section, you learned about the advantages of logical databases. To use these advantages for your report program, you must declare the logical database in the report attributes.

There is a logical database F1S that applies to the flight data model. Use this logical database for your program. To declare the database in your program:

1. Create a new program (see [Creating an ABAP Program \[Page 32\]](#)).
2. In the list of attributes, enter the name of the logical database in the corresponding field. The possible entries help provides you with an overview of all existing logical databases.
3. Save the attributes. The following screen appears:  [\[Ext.\]](#) [\[Ext.\]](#)

To start writing the source code of your program, choose *Source code*.

## Retrieving Data Using a Logical Database

## Retrieving Data Using a Logical Database

After you have specified the logical database in the report attributes, you can access the database in the program. In the declaration part of your program, declare the tables you want to access in the program using the TABLES statement, as described in the [SELECT Statement \[Page 45\]](#) section. This provides the work areas for passing the data from the logical database to the program. The system also configures the selection screen to include fields from the tables you specified.

The program of the logical database places the data from the database tables into the work areas created by the TABLES statement. The logical database then triggers an event. In your program, you catch this event using the keyword GET with the corresponding table name. If, for example, the logical database just filled the work area of table SBOOK, it triggers the event GET SBOOK in your program. The system then executes the statement block belonging to this event.

A statement block starts directly after the event keyword and ends at the next event keyword or at the end of the program.



In this exercise, you should display all booking data for flight number '0400' of airline 'LH'. Use the logical database F1S to retrieve the data. Output the booking number, the customer number, the luggage weight, the weight unit, the class, and the booking date in a list. Enter headings as well.

1. Open the program whose attributes you declared in the exercise [Creating a Report that Uses a Logical Database \[Page 61\]](#).
2. Declare the database table SBOOK. This contains the booking data that you want to read.
3. Include the GET event for table SBOOK in your program.
4. Use the statement pattern for the WRITE statement to output the table fields of table SBOOK.
5. Check the syntax of your program.
6. Activate the program.
7. Run the program. The following selection screen appears: [Ext.] [Ext.]  
This selection screen is generated automatically by the logical database. Enter 'LH' in the *Airline* field.
8. The selection criterion for the flight connection does not exist on this selection screen. To make a selection for this field as well, choose *Dynamic selection*.
9. A new selection screen appears, on which you can enter selections for the flight numbers. Enter '0400'.
10. Choose *Copy* to save the selections. This returns you to the first selection screen.
11. Save the selections as a variant.
12. Choose *Execute*. The resulting list contains the bookings for flight number '0400' of airline 'LH'.
13. Maintain the list headings.

---

Retrieving Data Using a Logical Database

Check your work.

You have just used the logical database F1S to read data from the database table SBOOK. The logical database saved you having to create your own selection screen. It also saved you having to formulate the database selection yourself.

Sample program: reading SBOOK using a logical database

## Sample program: reading SBOOK using a logical database

```
*&-----
*
*& Report  RSAATME8
*
*&
*
*&-----
*
*&  Read database table SBOOK using the logical database F1S
*
*&-----
*

REPORT  RSAATME8          .

*Declare the database table SBOOK
TABLES SBOOK.

*{ Begin of processing block for event GET SBOOK
GET SBOOK.

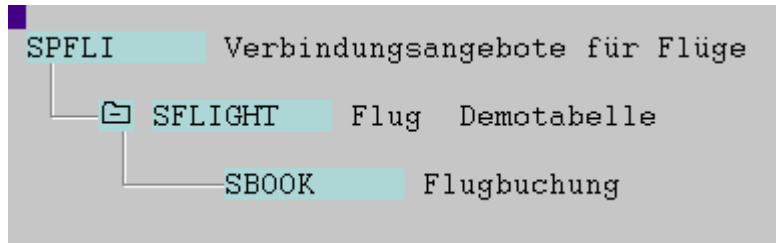
    WRITE:/ SBOOK-BOOKID,
            SBOOK-CUSTOMID,
            SBOOK-LUGGWEIGHT,
            SBOOK-WUNIT,
            SBOOK-CLASS,
            SBOOK-ORDER_DATE.
*} End of processing block for GET SBOOK
```



## Hierarchy of a Logical Database

Logical databases are programs that read data from database tables and pass it to other programs for processing. The order of reading the database tables is determined by a hierarchy.

Many tables in the R/3 System are linked using foreign key relationships. (For further information, refer to the [ABAP Dictionary \[Ext.\]](#) documentation.) Parts of these relationships form tree-like hierarchical structures. Logical databases allow you to read data easily from database tables that form parts of these structures. The logical database F1S has the following hierarchy:



When reading the tables, the system first reads one element of table SPFLI. Then, it reads the first element of the subordinate table SFLIGHT that, according to the foreign key relationship, belongs to the first element of table SPFLI. Then, it reads all elements of table SBOOK that belong to the first element read from table SFLIGHT. Next, it reads the second element of table SFLIGHT and all corresponding elements of table SBOOK. This step is repeated until the system has read all elements of table SFLIGHT that belong to the first element of table SPFLI. Then the system reads the second element of table SPFLI and the entire procedure starts again. This procedure is repeated until the entire hierarchy has been processed.

You can also restrict the data selection, as you have already seen in the exercise for [Retrieving Data Using a Logical Database \[Page 62\]](#). The effect of such a selection is that the logical database reads only those elements that match the selections, and passes only this data to your program. Consequently, you do not need to worry about restricting the data selection in your program.



To make this concept clearer, modify the program created in the last exercise. Create a list that outputs the airline, the flight connection, and the flight date before each flight of flight connection '0400' of the airline 'LH'. Use the event GET SFLIGHT.

1. In the ABAP Editor, open the program that you created in the last exercise.
2. Include the new event GET SFLIGHT. Remember to declare the database table SFLIGHT in the declaration section of your program.
3. In the statement block of GET SFLIGHT output the airline, the flight connection, and the flight date. Use the statement pattern for the WRITE statement.
4. Check the syntax and save your program.
5. Run the program.



The WRITE statement in the statement block of the event GET SFLIGHT is only triggered if the system reads a new record of the database table SFLIGHT.



## Events

In the last exercise, you learned about the event keyword GET. GET introduces a processing block that is executed at the event GET <tablename>. You can imagine such blocks as callable modules. In your program, these modules need not appear in any special order. The module belonging to a certain event starts with the event keyword and ends either with the next event keyword or at the end of the program.

As soon as you start a report, the system starts a second process (a control program), that calls these modules and controls the external program flow. This control program is the ABAP processor. The ABAP processor controls the interaction of ABAP reports, logical database programs, and other program modules (for example, dialog screens). It interprets the runtime objects of your ABAP program.

The interaction of ABAP processor and the different programs is controlled by the event keywords used in your program. The keywords in a report may be:

AT SELECTION SCREEN	Event after processing the user input on the selection screen, while the selection screen is still active.
START OF SELECTION	Event after processing the selection screen. The system sets this event keyword automatically when you specify no other event keyword (for example, in your program the last section).
GET <tablename>	Event at which the logical database in use offers a line of the database table <tablename>.
GET <tablename> LATE	Event after processing all tables that are hierarchically below the database table <tablename> in the structure of the logical database in use.
END-OF-SELECTION	Event after processing all lines offered by the logical database in use.
TOP-OF-PAGE	Event during list processing at the beginning of a new page.

## GET &lt;tablename&gt; LATE

## GET <tablename> LATE

The logical database triggers the event GET <tablename> LATE after processing all tables below the table <tablename> in the hierarchy structure. This allows you to output additional information or insert statements that change the list layout.



In the last exercise, you created a program that displays the booking data of database table SBOOK for the flight connections of the database table SFLIGHT. Now change your program so that the list has the same form as the one in the exercise for [nested SELECT statements \[Page 53\]](#). If you have any problems writing the program, you can refer to the [model solution \[Page 64\]](#).

The list should have the following form:  [\[Ext.\] \[Ext.\]](#)

1. Open the program that you created in the last exercise.
2. Delete the WRITE statement from the processing block GET SFLIGHT.
3. In the GET SFLIGHT event, pass the contents of the fields CONNID, CARRID, and FLDATE to the system variables for the list headings.
4. Check the syntax and save your program.
5. Execute the program and enter reasonable selections for the data you want to output. Save these selections as a variant.
6. Change the list headings to output the system variables as headings. Remember to save the list headings.
7. Run the program again.

You will see that page breaks and horizontal lines are missing. This is the reason why the system does not update the list headings for the different flights. Find out at which event you must include page breaks and underlines into the processing block.

8. Include the event GET SFLIGHT LATE. The logical database always triggers this event after reading all booking data from table SBOOK for a flight of table SFLIGHT and before reading a new flight from SFLIGHT.
9. Include an underline at the event GET SFLIGHT LATE. Save the program and execute it again.

After the booking data for one flight, an underline appears. However, the list heading is still not updated. To achieve this, include a page break at the appropriate event.

10. Include a page break at the right position into the processing block of GET SFLIGHT LATE. Save the program and execute it again.

Your list should now resemble the sample list.



The result of this exercise is the same as that from the unit on [nested SELECT statements \[Page 53\]](#). However, the program code is very different. While in the other exercise you must create the selection

**GET <tablename> LATE**

screen yourself and read the data from the database yourself, in this exercise the logical database does all this work for you. The advantage of this concept is that all your programs that use the logical database share the same access method. The selection screens are also the same for all programs.

Sample program: report using logical database

## Sample program: report using logical database

```
*&-----
*
*& Report  RSAATME8
*
*&
*
*&-----
*
*&  Read database tables SBOOK and SFLIGHT using the logical
*
*&  database F1S
*
*&-----
*

REPORT  RSAATME8          .

*Declare the database tables SBOOK and SFLIGHT
TABLES: SBOOK, SFLIGHT.

*{ Begin of processing block for the event GET SBOOK
GET SBOOK.

    WRITE:/ SBOOK-BOOKID,
           SBOOK-CUSTOMID,
           SBOOK-LUGGWEIGHT,
           SBOOK-WUNIT,
           SBOOK-CLASS,
           SBOOK-ORDER_DATE.
*} End of processing block of GET SBOOK

*{ Begin of processing block of GET SFLIGHT
GET SFLIGHT.

*{Pass variables for list headings
```

Sample program: report using logical database

```

MOVE SFLIGHT-CARRID TO SY-TVAR0.      "Airline
MOVE SFLIGHT-CONNID TO SY-TVAR1.      "Flight connection
MOVE SFLIGHT-FLDATE TO SY-TVAR2.      "Flight date
*} End of passing data for list headings
*} End of processing block of GET SFLIGHT

*} Begin of processing block of GET SFLIGHT LATE.
GET SFLIGHT LATE.
ULINE.
NEW-PAGE.
*} End of processing block of GET SFLIGHT LATE

```

## Structures and Internal Tables

This chapter provides an overview on internal tables. You will familiarize with every item from declaration to processing of control levels.

[Internal tables \[Page 73\]](#)

[Structures and internal tables \[Page 74\]](#)

[Declaring a structure \[Page 75\]](#)

[Declaring an internal table \[Page 77\]](#)

[Defining the work area \[Page 78\]](#)

[Filling internal tables \[Page 79\]](#)

[Working with internal tables \[Page 80\]](#)

[Sample program: internal tables \[Page 84\]](#)

[Processing control levels \[Page 87\]](#)

[Programming control level processing \[Page 89\]](#)

[Calculating totals \[Page 91\]](#)

[Sample program: processing control levels \[Page 92\]](#)



## Internal Tables

In ABAP, you work mainly with tables. Tables are the most important data structures in the R/3 System. Persistent data is stored in relational database tables.

However, you can also create internal tables, which are table objects that only exist for the runtime of the program. There are several ABAP statements for working with internal tables. You can, for example, append, insert, delete, or find lines.

The number of lines of an internal table is not fixed. Instead, the internal table is extended dynamically at runtime as required. That means, if you want to read a database table into an internal table, you do not need to know the size of the database table in advance. This makes internal tables easy to use.

You can use internal tables for table calculations on subsets of database tables. For example, you can read a part of one or more database tables into an internal table. You can use internal tables for calculating totals or for creating ranked lists.

They also allow you to reorganize their contents to suit the needs of your program. You can, for example, read particular entries from one or more large customer tables into an internal table, and then use them to create a list. When you run your program, you can access this data directly, instead of having to search for each record in the database.

After working through the exercises below, you will be able to:

- declare an internal table
- fill an internal table
- change, output, and delete individual elements
- sort tables by any fields
- understand and use control levels.

## Structures and Internal Tables

In the [Data Declarations \[Page 38\]](#) section, you learned about the main data objects in ABAP. As well as these objects, ABAP also allows you to use complex data objects.

An example of a complex data object is a structure. A structure is an object consisting of several elementary data objects in a particular order.

Another type of complex data object is an internal table. It consists of any number of data objects with the same structure.

To access the contents of an internal table, you use a work area. The work area must have the same structure as the internal table. You can access objects from the internal table by transferring them into the work area.

## Declaring a structure

Before you can use structures and internal tables in your program, you must declare them.



Declare a structure that contains one column for the departure city (CITYFROM), one for the arrival city (CITYTO), one for the airline (CARRID), and one for the flight connection (CONNID). Use parts of the structure of database table SPFLI.

1. Create a new program. When maintaining the attributes, make sure to use the logical database F1S for data retrieval.
2. Declare the structure using the statement pattern. Position the cursor in the declaration section of your program and choose *Pattern*.
3. On the dialog box, mark the radio button *Internal table*.



There is no extra pattern for structures. Since it takes only two modifications to change the declaration of an internal table into the declaration of a structure, we, nevertheless, use the statement pattern for internal tables here.

4. Mark *with LIKE fields fr.* and enter the name of the database table into the adjacent input field.

The item *with LIKE fields fr.* allows you to select parts of the structure of a database table. If you mark *with LIKE for struct. of* instead, you copy the entire structure of the database table.

5. Choose *Continue*.
6. Mark the fields which you need in the internal table.
7. Choose *Copy*.
8. Specify the name <structure> of the structure to be defined and choose *Continue*.
9. Replace the keyword DATA, used for the declaration of an internal table, by the keyword TYPES, used for structures, and delete the option OCCURS 0 from the pattern. The declaration then looks like this:

```
TYPES: BEGIN OF structure,
        CARRID LIKE SPFLI-CARRID,
        CONNID LIKE SPFLI-CONNID,
        CITYFROM LIKE SPFLI-CITYFROM,
        CITYTO LIKE SPFLI-CITYTO,
      END OF structure.
```

The statement block starts with the keyword TYPES. The beginning of the declaration of structure <structure> is marked by the statement BEGIN OF <structure>. Then, the individual fields of the structure are declared. The declaration ends with the line END OF <structure>. To declare additional fields for the structure, include the field declarations between BEGIN OF <structure> and END OF <structure>.

---

**Declaring a structure**

## Declaring an internal table

In the last exercise, you declared a structure. Now, declare an internal table, based on this structure.



Declare an internal table, based on the structure <structure> you declared in the last exercise.

1. Fetch the program created for the last exercise into the editor.
2. Declare the internal table in the declaration section of your program below the declaration of the structure. The statement you need is:

```
DATA itab TYPE structure OCCURS 0.
```

The statement starts with the keyword DATA, followed by the name of the internal table.

The keyword TYPE refers to the previously declared structure <structure>.

The keyword OCCURS defines the structure as internal table. The number behind OCCURS determines the number of lines the table has after initialization. As mentioned in [Internal tables \[Page 73\]](#), the size of an internal table is variable, that is, the system can expand the number of lines beyond the predefined size, if required.



If you forget to include the keyword OCCURS into the declaration, the result is a field string instead of an internal table. However, you use such a field string as work area for an internal table.

## Defining the Work Area

### Defining the Work Area

To change or output the contents of an internal table, you need a work area. When processing an internal table, the system always fills the work area with the contents of the current table line.

There are two ways of defining a work area:

1. Use an internal table with a header line: If you use an internal table with header line, the system automatically creates a work area (the header line) with the same name as the internal table.
2. Define your own structure with the same structure as the internal table. You can then use this as the work area for the internal table.

In the exercises below, you will work with internal tables without header lines. You will therefore need a structure for your work area.



You are going to define a structure as a work area for the internal table you defined in the last exercise.

1. Open the program that you created in the last exercise.
2. Include the following line to declare a work area <wa> for the internal table <itab>:  
DATA: wa TYPE structure.
3. Check the syntax and save your program.

You have now defined a work area <wa> with the same structure <structure> as the internal table <itab>. You can use this work area in the exercises below to display or change the contents of the internal table. You can access the fields of the work area as you would access fields in the work area of database tables. First, specify the name of the work area <wa>, then a hyphen, and finally the field name:

WA-CONNID is the field CONNID of work area WA.

## Filling internal tables

You declared an internal table without header line and a work area for the table. Now you can work with the table. You will now learn how to fill internal tables.

For filling internal tables, ABAP/4 provides the keyword APPEND. It appends the contents of work area <wa> to the end of the internal table <itab>:

```
APPEND wa TO itab.
```



Use the logical database to read the database table SPFLI. Insert the data into the internal table <itab>.

1. Fetch the program from the last exercise into the editor.
2. Declare the database table from which you want to read data.
3. Use the keyword MOVE to move the contents of table SPFLI at the event GET SPFLI to the corresponding fields of the work area <wa> of the internal table <itab> (see [Handling variables \[Page 39\]](#)).
4. Append the contents of the work area <wa> to the internal table.
5. Check the syntax and save your program.



Instead of moving each field contents individually from the work area of the database table to the work area of the internal table, you can use the MOVE-CORRESPONDING statement to move all fields with identical names in one go.

```
MOVE-CORRESPONDING SPFLI TO wa.
```

corresponds in our exercise to:

```
MOVE SPFLI-CARRID TO wa-CARRID.
```

```
MOVE SPFLI-CONNID TO wa-CONNID.
```

```
MOVE SPFLI-CITYFROM TO wa-CITYFROM.
```

```
MOVE SPFLI-CITYTO TO wa-CITYTO.
```

## Working with Internal Tables

### Working with Internal Tables

You normally process internal tables in loops. In this loop, all elements of the internal table are read, starting from the first line. The keyword that starts the loop is `LOOP AT`, followed by the name of the internal table.

If `<itab>` is an internal table without header line, such as the one you created in the exercise [Declaring an Internal Table \[Page 77\]](#), you need a work area `<wa>` into which you can place the contents of the current line. To specify this work area, use the addition `INTO <wa>`.

You can now write a statement block to process the individual lines of the internal table.

At the end of the block, close the loop using the keyword `ENDLOOP`.

The entire statement block looks like this:

```
LOOP AT itab INTO wa.
```

```
<statement block>
```

```
ENDLOOP.
```

You can find the program that is based on the following examples in the section [Example Program: Internal Tables \[Page 84\]](#). If you have any problems writing the program, you can look there for help.

### Displaying the Table Contents

You normally display the contents of an internal table within a `LOOP` structure using the `WRITE` statement. Remember that the contents of the current table line are in the work area, so you will need to specify the relevant fields of the work area `<wa>` in the `WRITE` statement.



You can use the statement pattern to create the `WRITE` statement. However, note that you cannot mark *Output to Struct.* in the window *Assemble a WRITE Statement*. Instead, use *Output from fld* for each field you want to output. Enter the entire field name into the adjacent input field and choose *Copy*.



In the last exercise, you wrote a program that filled the internal table without header line `<itab>` with values from the database table `SPFLI`, using a logical database. Now display the internal table.

1. Load your program into the ABAP Editor.
2. Find the correct event keyword at which to output the internal table. Since you want to display your list after the `GET SPFLI` event, the only possible event is [END-OF-SELECTION \[Page 67\]](#).
3. Add a `LOOP` to your program for the internal table `<itab>`. Remember to fill the work area `<wa>` with the values from the current line of the internal table.
4. Output the fields of the work area in the statement block of the loop.
5. Check the syntax and save your program.



## Working with Internal Tables

6. Run the program. On the selection screen, choose Airline = 'LH' and Flight connection = '0400' to '2000'. Use the *Dynamic selections*. Save the selection as a *Variant*.

The system displays a list of all flights that correspond to the selections entered on the selection screen.

**Changing Single Lines in an Internal Table**

To change a single line in an internal table, use the MODIFY statement. This modifies the current line of the LOOP statement.

However, before using the MODIFY statement, you must first make the required changes to the current line in the work area <wa> of the internal table. Then, you can assign the contents of the work area <wa> to the current table line using:

MODIFY <itab> FROM <wa>.



In the last exercise, you displayed the contents of the internal table. Now modify the flight number '0400' to the flight number '0401' after the first output. Then output the modified internal table using another LOOP statement.

1. Load your program into the ABAP Editor.
2. Insert an [IF statement \[Page 83\]](#) into the LOOP statement that is only true for flight connection '0400'.
3. Within the statement block of the IF statement, modify the contents of the work area according to the requirements.
4. Modify the internal table only for those lines that contain '0400'.
5. Include another loop into your program to output the modified internal table. Inset a page break to separate the first set of table contents from the second.
6. Check the syntax and save your program.
7. Run the program. On the selection screen, use the variant created in the last exercise.

The system displays the contents of both the original and the modified internal table in a list on the screen.

**Deleting Single Lines From an Internal Table**

In the last exercise, you modified individual lines of an internal table. You can also delete individual lines using the DELETE statement within a loop. The DELETE statement then deletes the current line of the internal table <itab>:

DELETE <itab>.



Modify your program to delete flight connection '0402' from the internal table after the last output. Then display the result using another LOOP statement.

1. Load your program into the ABAP Editor.

## Working with Internal Tables

2. Insert an [IF statement \[Page 83\]](#) into the second LOOP statement that is only true for flight connection '0402'.
3. Delete the contents of the lines that contain flight connection '0402' from the internal table.
4. Include another loop into your program to output the modified internal table. Inset a page break to separate the first set of table contents from the second.
5. Check the syntax and activate your program.
6. Run the program. On the selection screen, use the existing variant.

The system displays a list of table contents that has been modified again. The difference to the previous list is that flight connection '0402' was deleted.

## Sorting Internal Tables

To sort internal tables by one or more fields, use the keyword SORT. In the statement, you can specify the sort sequence and the sort direction. You use SORT outside a loop.

Its syntax is:

`SORT <itab> BY <field1> <field2> ...<field n>.`

With this statement, the system sorts the internal table by the table fields <field1> to <fieldn>, sorting the entries by <field1> first. All entries with the same contents in <field1> are then sorted by <field2>, and so on.



Sort the last list of flight connections of the last exercise by the arrival city.

1. Load your program into the ABAP Editor.
2. After the last LOOP statement, insert the SORT statement. Sort by the field CITYTO. The line you must include looks as follows:  
`SORT <itab> BY CITYTO.`
3. Include another loop in your program to output the modified internal table. Inset a page break to separate the first set of table contents from the second.
4. Execute the program. On the selection screen, use the existing variant.

At the end of the list, the system displays the contents of the internal table sorted by the arrival city.

## IF statement

You use the IF statement for case distinctions. Depending on whether the logical expression <logexp> is true or not, the system processes different branches within the processing block between IF and ENDIF.



```
IF <logexp>.  
    processing block 1  
ELSE.  
    processing block 2  
ENDIF.
```

In this example, the system executes 'processing block 1' if the logical expression <logexp> is true. If it is false, the system executes 'processing block 2'.



The ELSE statement is optional. If you do not need it, you can omit it.

## Sample program: internal tables

## Sample program: internal tables

```
*&-----
*
*& Report  RSAATME9
*
*&
*
*&-----
*
*& Create, output, modify an internal table and delete entries
*
*&
*
*&-----
*

REPORT  RSAATME9          .

TABLES: SPFLI.

*{ Declare structure STRUCTURE
TYPES: BEGIN OF STRUCTURE,
        CARRID LIKE SPFLI-CARRID,
        CONNID LIKE SPFLI-CONNID,
        CITYFROM LIKE SPFLI-CITYFROM,
        CITYTO LIKE SPFLI-CITYTO,
        END OF STRUCTURE.
*} End of declaration of structure STRUCTURE

*} Declare internal table ITAB
DATA: ITAB TYPE STRUCTURE OCCURS 0.

* Declare work area for the internal table ITAB
DATA: WA TYPE STRUCTURE.

*{ Event GET SPFLI, fill internal table
```

Sample program: internal tables

```

GET SPFLI.

  MOVE-CORRESPONDING SPFLI TO WA.

  APPEND WA TO ITAB.
*} End event GET SPFLI

*{Start processing internal table ITAB
END-OF-SELECTION.
*{ First LOOP for outputting and modifying the original
* internal table
  LOOP AT ITAB INTO WA.
    WRITE: / WA-CARRID,
            WA-CONNID,
            WA-CITYFROM,
            WA-CITYTO.

* Modify the current line of the internal table
    IF WA-CONNID = '0400'.
      WA-CONNID = '0401'.
      MODIFY ITAB FROM WA.
    ENDIF.
  ENDLOOP.
*} End of first LOOP

*{Page break to separate list output
  NEW-PAGE.

*{ Second LOOP for outputting the previously modified
* internal table and for deleting an element from table
  LOOP AT ITAB INTO WA.
    WRITE:/ WA-CARRID,
            WA-CONNID,
            WA-CITYFROM,
            WA-CITYTO.

* Delete current line from internal table
    IF WA-CONNID = '0402'.
      DELETE ITAB.

```

**Sample program: internal tables**

```
        ENDIF.
    ENDLOOP.
*} End of second LOOP

*{ Page break to separate the list output
    NEW-PAGE.

*{ Third LOOP for outputting the once again
* modified internal table
    LOOP AT ITAB INTO WA.
        WRITE:/ WA-CARRID,
                WA-CONNID,
                WA-CITYFROM,
                WA-CITYTO.
    ENDLOOP.
*} End of third LOOP

* Page break to separate the list output
NEW-PAGE.

* Sort internal table by table field CITYTO
SORT ITAB BY CITYTO.

*{ fourth LOOP for outputting the internal table
LOOP AT ITAB INTO WA.
    WRITE:/ WA-CARRID,
            WA-CONNID,
            WA-CITYFROM,
            WA-CITYTO.
ENDLOOP.
*} End of fourth LOOP
```

## Processing control levels

Processing control levels allow you to create statement blocks within a LOOP-ENDLOOP loop that are processed only for certain table lines.

Start such a statement block using the control statement AT and conclude it with the control statement ENDAT:

```
AT <line>.  
    <statement block>  
ENDAT.
```

The line condition <line>, at which the statement block within AT-ENDAT is processed, can be one of the following:

<line>	describes
FIRST	the first line of the internal table
LAST	the last line of the internal table
NEW <field>	the beginning of a group of lines with the same contents in <field> and in the superior fields
END OF <field>	the end of a group of lines with the same contents in <field> and in the superior fields

The order of the fields is determined by the sequence in which the table fields of the internal table are declared. A field superior to <field> is declared before <field>, a field inferior to <field> is declared after <field>.



When writing the control level processing, make sure that, depending on the sequence of the field declaration within the internal table, you adhere to the order of the individual control levels within the loop as shown below:

```
REPORT DEMO.  
  
...  
DATA: BEGIN OF ITAB OCCURS 0,  
      F1...,  
      F2...,  
      F3...,  
      END OF ITAB.  
DATA: AB LIKE ITAB.  
...  
SORT ITAB BY F1 F2.
```

**Processing control levels**

```
LOOP AT ITAB INTO AB.  
  AT FIRST. <anweisungen>. ENDAT.  
  AT NEW F1. <anweisungen>. ENDAT.  
    AT NEW F2. <anweisungen>. ENDAT.  
      <einzelsatzverarbeitung>  
    AT END OF F2. <anweisungen>. ENDAT.  
  AT END OF F1. <anweisungen>. ENDAT.  
  AT LAST. <anweisungen>. ENDAT.  
ENDLOOP.
```



If you want to work with control levels, you must keep this procedure in mind while declaring the internal table to be able to use the above convention. If you discover after completing the control level processing, that the procedure collides with the table field declaration, you must modify the field declaration.




## Control Level Processing

In this exercise, you will write a program to include control level processing.



Create a new program. Define an internal table and fill it with data from the database tables SPFLI and SFLIGHT, using the logical database F1S. Include the following fields in the internal table:

Departure city	CITYFROM (from SPFLI)
Arrival city	CITYTO (from SPFLI)
Airline	CONNID (from SPFLI)
Flight connection	CONNID (from SPFLI)
Flight date	FLDATE (from SFLIGHT)
Capacity	SEATSMAX (from SFLIGHT)
Number of occupied seats	SEATSOCC (from SFLIGHT)

Use control level processing to output a comment before starting to output data. Then create the following list:  [\[Ext.\]](#) [\[Ext.\]](#)

Use different colors for outputting the fields and use list headings. After displaying the last record, output another comment.

If you have problems writing the program, you can use the [model solution \[Page 92\]](#) for help.

1. Create a new program and enter logical database F1S in the attributes.
2. Declare the database tables.
3. Declare a structure containing the relevant fields.
4. Declare an internal table and a work area for that table.
5. At the event GET SPFLI, move the data from database table SPFLI into the corresponding fields of the work area of the internal table.
6. At the event GET SFLIGHT, move the data from database table SFLIGHT into the corresponding fields of the work area of the internal table.
7. Append the contents of the work area to the contents of the internal table.
8. Start processing the internal table at the event END-OF-SELECTION.
9. Start with the control level processing. Start the control level processing using the line condition AT FIRST and output a comment with the WRITE statement.
10. Use the line condition AT CITYFROM to output the field CITYFROM in the color COLOR\_KEY.
11. Use the line condition AT CITYTO to output the field CITYTO in the color COLOR\_KEY. Start the output at position 20.
12. Use the line condition AT CONNID to output the fields CARRID and CONNID in the color COL\_POSITIVE.
13. Process the individual records to output the fields FLDATE, SEATSOCC, and SEATSMAX.

---

**Control Level Processing**

14. Once you have displayed all of the data from the internal table, display another comment using the line condition AT LAST.
15. Check whether the order of the control level processing corresponds to the sequence in which the fields of the internal table are declared. If necessary, change the data declaration.
16. Check the syntax and activate your program.
17. Execute the program and create a variant on the selection screen.
18. Maintain the list headings and execute the program again.

If you like, you can include other line conditions into the program and check their effects. Always keep the order of the control level processing in mind.

## Calculating Totals

In the last exercise you learned how to output the number of occupied seats for the flights of an airline. In this exercise, you will learn how to calculate simple subtotals.



In this exercise, you will calculate the total number of occupied seats and the maximum number of seats that can be occupied for each flight connection. Then, you will calculate a grand total for the entire airline.

If you have any problems writing this program, you will find help in the [model solution \[Page 92\]](#).

1. Load your program from the last exercise into the ABAP Editor.
2. At the line condition AT END OF CONNID, use the keyword SUM to calculate the total numbers of seats occupied and of seats available.

SUM calculates totals for all numeric fields within the current control level and places them in the relevant fields of the work area.

3. Output the control level totals.
4. Output the totals of all seats occupied and of all seats available at the line condition AT END OF CARRID as well.
5. Check the syntax and save your program
6. Activate your program and execute it.

Sample program: processing control levels

## Sample program: processing control levels

```
*&-----
*
*& Report  RSAATMEA
*
*&
*
*&-----
*
*& Processing control levels of internal table ITAB
*
*&
*
*&-----
*

REPORT  RSAATMEA          .

* Declare database tables
TABLES: SFLIGHT, SPFLI.

* Declare structure STRUCTURE
TYPES: BEGIN OF STRUCTURE,
        CITYFROM LIKE SPFLI-CITYFROM,
        CITYTO LIKE SPFLI-CITYTO,
        CARRID LIKE SPFLI-CARRID,
        CONNID LIKE SPFLI-CONNID,
        FLDATE LIKE SFLIGHT-FLDATE,
        SEATSMAX LIKE SFLIGHT-SEATSMAX,
        SEATSOCC LIKE SFLIGHT-SEATSOCC,
        END OF STRUCTURE.

*} Declare internal table ITAB
DATA: ITAB TYPE STRUCTURE OCCURS 0.

* Declare work area for internal table ITAB
DATA: WA TYPE STRUCTURE.
```

Sample program: processing control levels

```

*Read records of SPFLI and move to work area for
*internal table
GET SPFLI.
    MOVE-CORRESPONDING SPFLI TO WA.

*Read records of SFLIGHT, move to work area for internal
*table and fill internal table
GET SFLIGHT.
    MOVE-CORRESPONDING SFLIGHT TO WA.
    APPEND WA TO ITAB.

* Actions after reading logical database
END-OF-SELECTION.

* Sort internal table
    SORT ITAB BY CITYFROM CITYTO CARRID CONNID SEATSOCC.

* Begin processing control levels
    LOOP AT ITAB INTO WA.
* Control level: start control level processing
        AT FIRST
            WRITE / 'Start output internal table'.
            ULINE.
        ENDAT.

* Control level: new contents in WA-CITYFROM
        AT NEW CITYFROM.
            WRITE / WA-CITYFROM COLOR COL_KEY.
        ENDAT.

* Control level: new contents in WA-CITYTO
        AT NEW CITYTO.
            WRITE /20 WA-CITYTO COLOR COL_HEADING.
        ENDAT.

```

## Sample program: processing control levels

```
* Control level: new contents in WA-CONNID
  AT NEW CONNID.
    WRITE /20 WA-CARRID COLOR COL_POSITIVE.
    WRITE 25 WA-CONNID COLOR COL_POSITIVE.
  ENDAT.

* Processing single records
  WRITE /20 WA-FLDATE COLOR COL_NORMAL.
  WRITE: WA-SEATSOCC COLOR COL_NORMAL,
         WA-SEATSMAX COLOR COL_NORMAL.

* Control level after output of all records for WA-CONNID
  AT END OF CONNID.
    SUM.
    WRITE:/ WA-SEATSOCC COLOR COL_TOTAL UNDER WA-SEATSOCC,
          WA-SEATSMAX COLOR COL_TOTAL.
    ULINE.
  ENDAT.

* Control level after output of all records for WA-CARRID
  AT END OF CARRID.
    SUM.
    WRITE:/ WA-SEATSOCC COLOR COL_TOTAL INTENSIFIED UNDER WA-
SEATSOCC,
          WA-SEATSMAX COLOR COL_TOTAL INTENSIFIED.
    ULINE.
  ENDAT.
ENDLOOP.
*} End of control level processing
```

## Interactive Reporting

Interactive reporting helps you to create easy-to-read lists. You can display an overview list first that contains general information and provide the user with the possibility of choosing detailed information that you display on further lists.

[Concept: interactive reporting \[Page 96\]](#)

[New event keywords \[Page 97\]](#)

[Creating a secondary list \[Page 98\]](#)

[Selecting valid lines only \[Page 100\]](#)

[Defining user interfaces \(GUI\) \[Page 101\]](#)

[Sample program for interactive reporting \[Page 103\]](#)

**Concept: interactive reporting**

## Concept: interactive reporting

With interactive reporting, the user can actively control data retrieval and display during the session. Instead of an extensive and detailed list, you create a basic list with condensed information from which the user can switch to detailed displays by positioning the cursor and entering commands ("interactively").

The detailed information appear in secondary lists. Secondary lists may either overlay the basic list completely or you can display them in an extra window on the screen. The secondary lists can themselves be interactive again.

After working through this chapter, you

- can create a basic list
- know several new event keywords
- can output additional information in secondary lists
- know how to create menu bars, standard and application toolbars
- know how to create the current screen title with dynamic contents.

In the exercises below, you will create a program that outputs a basic list. This basic list contains the flight connections with the number of seats occupied and the maximum number of seats. The list corresponds to the one created in [Programming control level processing \[Page 89\]](#).

Based on this list, you will create a secondary list that contains additional booking data for a flight.



## New Event Keywords

In the [Logical Databases and Events \[Page 59\]](#) section, you learned that ABAP programs are event-driven. In addition to the events introduced before, interactive reporting provides some new events:

Event keyword	Event
AT LINE-SELECTION	Event triggered by the user double-clicking a line or selecting it using F2
AT USER-COMMAND	Event triggered by the user pressing a function key
TOP-OF-PAGE DURING LINE-SELECTION	Event during list processing when a new page begins on a detail list

## Creating a Detail List

### Creating a Detail List

Detail lists allow you to present more information than is contained in the basic list.

The user can, for example, select a line of the basic list for which he or she wants to see more detailed information. You then display the extra information in a detail list.

This method requires that you have previously stored the contents of the selected line within the program.

To do this, you use the ABAP statement `HIDE`, which saves the field contents for the current line. When you start a detail list for a list line that has `HIDE` fields, the system places their values into the corresponding variables in the program.

In the program code, insert the `HIDE` statement directly after the `WRITE` statement for the current line.



```
HIDE: WA-CONNID, WA-CARRID.
```

This statement tells the system to store the fields `WA-CONNID` and `WA-CARRID` and to place their contents back into the fields on a detail list.

In addition to the `HIDE` statement, you need an event that occurs as soon as the user selects a line. The event keyword for a double-click is `AT LINE-SELECTION`. It is also triggered when the user presses `F2` or chooses a function with the function code `PICK`.



You are going to use the list that you created in the [control level processing \[Page 89\]](#) exercise as a basic list for this exercise. The detail list should contain booking data from database table `SBOOK` for the selected flight.

If you have problems writing the program, you can refer to the [example program for interactive reporting \[Page 103\]](#).

1. Open the program that you wrote in the [control level processing \[Page 89\]](#) exercise.
2. Insert a `HIDE` statement in the control level processing. You want to "hide" only those lines for which the flight date appears. Store the fields `CONNID`, `CARRID`, and `FLDATE` of the work area `<wa>` of the internal table `<itab>`.
3. Insert the event `AT LINE-SELECTION`.
4. Write the airline, the flight connection, and the flight date to the screen. Then insert a horizontal line.
5. Read the data from database table `SBOOK` that matches the flight selected from the basic list. Remember to declare `SBOOK` in your program.
6. Display the booking data `BOOKID`, `CUSTOMID`, and `ORDER_DATE` in a new line.
7. Check the syntax and activate your program.
8. Run the program. On the basic list, select a flight by double-clicking the corresponding line.

The system displays a list that contains the booking data for that flight.

## Creating a Detail List

9. Improve the layout of the detail list. If you like, you can output a list heading for this list. Note that there is no standard list header for a detail list. Instead, you must create it yourself using WRITE statements.



You can use the same functions on a detail list as you can on a basic list. From a detail list, the *Back* and *Cancel* functions return to the basic list. The *Exit* function returns you to the selection screen of the program.

## Valid Line Selection

### Valid Line Selection

In the last exercise, you learned how to create detail lists. You learned about the HIDE statement that allows you to store the current data of a list line.

You may have realized that the system still creates a new detail list when you select an invalid line, or select a line on the detail list itself. There are various ways of preventing this from happening, and in this exercise, you will learn one way of doing so by exploiting the characteristics of the HIDE statement.

At the end of the processing block END-OF-SELECTION, you will delete the contents of one or more fields you previously stored for valid lines using the HIDE statement. In the AT LINE-SELECTION event, check whether the work area is initial or whether the HIDE statement stored field contents there. If the work area is not initial, you can create a detail list, because you know that valid data has been stored there.

After processing the detail list, clear the work area again. This prevents the user from trying to create further detail lists from the current detail list.



This exercise tells you how to check whether the user selected a valid line for creating a detail list. Use and modify the program that you created in the last exercise.

If you have problems writing the program, you can refer to the [example program for interactive reporting \[Page 103\]](#).

1. Open your program from the last exercise in the ABAP Editor.
2. In the last line of the processing block of END-OF-SELECTION, delete the contents of the work area <wa> of the internal table <itab>. Use the CLEAR statement:  
`CLEAR <wa>.`
3. At the beginning of the statement block in the AT LINE-SELECTION event, check whether the field CARRID of the work area <wa> is initial. If it is, terminate the processing. Use the CHECK statement:  
`CHECK NOT <wa>-CARRID IS INITIAL.`
4. At the end of the AT LINE-SELECTION processing block, delete the contents of the work area <wa>. This ensures that no further detail lists can be created, even if the user tries to select a line from the current detail list.
5. Check the syntax and activate your program.
6. Execute the program and select an invalid line. No detail list should be displayed.

## Defining a User Interface

The system automatically creates a user interface for your lists, which contains the basic list functions such as save, print, and so on. However, if you want to add extra functions to your lists, you must define your own list status. This also allows you to change the window title.

To create a new status, use the Menu Painter. With the Menu Painter, you can create menus and application toolbars. You can also assign function keys to certain functions.




In this exercise, you will create a status and a GUI title for your basic list. If you have problems creating the program, you can refer to the [example program for interactive reporting \[Page 103\]](#).

1. Open the program that you created in the last exercise in the ABAP Editor.
2. At the beginning of the statement block of AT END-OF-SELECTION, activate the status of the basic list using the statement:

```
SET PF-STATUS 'STATUS'.
```

Note that you must specify the status name 'STATUS' in uppercase letters.

3. Check the syntax and save your program.
4. Double-click the name <STATUS>. A dialog box appears, asking you whether you want to create the status <STATUS>. Choose Yes.
5. Enter a short description for the status you want to create.
6. Choose *List* as status type. The system then automatically copies the standard list functions into your status.
7. Choose *Continue*. The *Maintain status* dialog box appears.
8. Expand the menu bar.
9. Choose *Display standards*.


The standard menus appear in the menu bar:  [Ext.] [Ext.]

10. Replace the name of menu <List> with **Flight data**.
11. Double-click the *Flight data* menu.

A list of the standard menu entries appears. To define a menu entry, enter a value in the *Func.* column.

12. Place the cursor into the top line of the menu and choose *Edit* → *Insert* → *Entry*.
13. Enter the menu entry PICK in the *Code* column.

Function code PICK triggers the event AT LINE-SELECTION.

14. Expand the function key setting section of the Menu Painter editor. Choose *Merge list functions*.
15. Expand the application toolbar section of the Menu Painter editor.  [Ext.]
16. Into the first input field of the application toolbar, enter function code PICK and press

ENTER.

---

**Defining a User Interface**

17. A dialog box appears. Assign the function to function key F2.  
The system automatically assigns an icon to this function, since in the *Recommended function key settings* the function code PICK is predefined with this icon.
18. Choose *Interface* → *Activate* to activate the status.  
You can now use the interface in your program.
19. Choose *Back*. This closes the Menu Painter and returns you to the ABAP Editor.
20. In the next line of the program, activate the GUI title using the statement:  
SET TITLEBAR 'TITLEBAR'.
21. Check the syntax and activate your program.
22. Double-click the name <TITLEBAR>. A dialog box appears, asking you whether you want to create the title. Choose Yes.
23. Enter a title for your list and choose *Save*. The system then returns you to the ABAP Editor.
24. Run the program. The system displays the basic list with the newly-defined GUI title and status. Test the functions you defined.

## Sample program for interactive reporting

```

*&-----
*
*& Report  RSAATME2
*
*&
*
*&-----
*
*&
*
*&
*
*&-----
*

REPORT  RSAATME2          .

* Declare database tables
TABLES: SFLIGHT, SPFLI, SBOOK.      "declare SBOOK

* Declare structure STRUCTURE
TYPES: BEGIN OF STRUCTURE,
        CITYFROM LIKE SPFLI-CITYFROM,
        CITYTO LIKE SPFLI-CITYTO,
        CARRID LIKE SPFLI-CARRID,
        CONNID LIKE SPFLI-CONNID,
        FLDATE LIKE SFLIGHT-FLDATE,
        SEATSMAX LIKE SFLIGHT-SEATSMAX,
        SEATSOCC LIKE SFLIGHT-SEATSOCC,
        END OF STRUCTURE.

*} Declare internal table ITAB
DATA: ITAB TYPE STRUCTURE OCCURS 0.

* Declare the work area for internal table ITAB
DATA: WA TYPE STRUCTURE.

```

---

Sample program for interactive reporting

```
* Read the records of SPFLI and move them to work area
* of internal table
GET SPFLI.
    MOVE-CORRESPONDING SPFLI TO WA.

* Read the records of SFLIGHT and move them to work area of
* internal table and fill internal table
GET SFLIGHT.
    MOVE-CORRESPONDING SFLIGHT TO WA.
    APPEND WA TO ITAB.

* Actions after reading logical database
END-OF-SELECTION.

* Define user interface
SET PF-STATUS 'GRUND'.
SET TITLEBAR 'GRU'.
* Sort internal table
    SORT ITAB BY CITYFROM CITYTO CARRID CONNID SEATSOCC.

* Start control level processing
    LOOP AT ITAB INTO WA.
* Control level: start control level processing
        AT FIRST.
            WRITE / 'Start output internal table'.
            ULINE.
        ENDAT.

* Control level, new contents in WA-CITYFROM
        AT NEW CITYFROM.
            WRITE / WA-CITYFROM COLOR COL_KEY.
        ENDAT.

* Control level, new contents in WA-CITYTO
```



Sample program for interactive reporting

```

    AT NEW CITYTO.
        WRITE /20 WA-CITYTO COLOR COL_HEADING.
    ENDAT.

* Control level, new contents in WA-CONNID
    AT NEW CONNID.
        WRITE /20 WA-CARRID COLOR COL_POSITIVE.
        WRITE 25 WA-CONNID COLOR COL_POSITIVE.
    ENDAT.

* Process single records
    WRITE /20 WA-FLDATE COLOR COL_NORMAL.
    WRITE: WA-SEATSOCC COLOR COL_NORMAL,
           WA-SEATSMAX COLOR COL_NORMAL.

* Fill the HIDE area with the field contents of
* CONNID, CARRID, and FLDATE from work area WA
    HIDE: WA-CONNID, WA-CARRID, WA-FLDATE.

* Control level after output of all records for WA-CONNID
    AT END OF CONNID.
        SUM.
        WRITE:/ WA-SEATSOCC COLOR COL_TOTAL UNDER WA-SEATSOCC,
              WA-SEATSMAX COLOR COL_TOTAL.
        ULINE.
    ENDAT.

* Control level after output of all records for WA-CARRID
    AT END OF CARRID.
        SUM.
        WRITE:/ WA-SEATSOCC COLOR COL_TOTAL INTENSIFIED UNDER WA-
SEATSOCC,
              WA-SEATSMAX COLOR COL_TOTAL INTENSIFIED.
        ULINE.
    ENDAT.

```

## Sample program for interactive reporting

```
        ENDLOOP.

*} End of control level processing

* Deleting the Workarea WA (Selecting valid lines)
    CLEAR WA.

*{ Start of secondary list after double-click
AT LINE-SELECTION.
    WRITE:/ 'Airline:           ',WA-CARRID,
           / 'Flight connection:',WA-CONNID,
           / 'Flight date:      ',WA-FLDATE.
    ULINE.

* Read database table SBOOK with WHERE condition
    SELECT      * FROM SBOOK
              WHERE CARRID      = WA-CARRID
              AND   CONNID      = WA-CONNID
              AND   FLDATE      = WA-FLDATE      .
    WRITE:/ SBOOK-BOOKID,
           SBOOK-CUSTOMID,
           SBOOK-ORDER_DATE.
    ENDSELECT.

* Deleting the Workarea WA (selecting valid lines)
    CLEAR WA.
```